K-trivial, K-low and MLR-low sequences: a tutorial

as explained by L. Bienvenu[*], notes by A. Shen[†]

Abstract

A remarkable achievement in algorithmic randomness and algorithmic information theory was the discovery of the notions of K-trivial, K-low and Martin-Löf-random-low sets: three different definitions turns out to be equivalent for very non-trivial reasons. This paper, based on the course taught by one of the authors (L.B.) in Poncelet laboratory (CNRS, Moscow) in 2014, provides an exposition of the proof of this equivalence and some related results.

We assume that the reader is familiar with basic notions of algorithmic information theory (see, e.g., [3] for introduction and [4] for more detailed exposition). More information about the subject and its history can be found in [2, 1].

1 K-trivial sets: definition and existence

Consider an infinite bit sequence and complexities of its prefixes. If they are small, the sequence is computable or almost computable; if they are big, the sequence looks random. This idea goes back to 1960s and appears in the algorithmic information theory in different forms (Schnorr–Levin criterion of randomness in terms of complexities of prefixes, the notion of algorithmic Hausdorff dimension). The notion of K-triviality is on the low end of this spectrum: we consider sequences that have prefixes of minimal possible (prefix) complexity:

Definition. *A bit sequence $a_0a_1a_2\ldots$, is called* K-trivial *if it has minimal possible prefix complexity of its prefixes, i.e., if*

$$K(a_0a_1\ldots a_{n-1}) = K(n) + O(1).$$

- Note that n can be reconstructed from $a_0\ldots a_{n-1}$, so $K(a_0\ldots a_{n-1})$ cannot be smaller than $K(n) - O(1)$.
- Every computable sequence is K-trivial (since $a_0\ldots a_{n-1}$ can be computed given n).

[*]Poncelet laboratory, CNRS, Moscow, laurent.bienvenu@computability.fr

[†]LIRMM, Montpellier, CNRS, UM2, on leave from IITP RAS, Moscow, alexander.shen@lirmm.fr

- Similar definition for plain complexity has no sense, since this would imply that sequence A is computable (it is enough to have $C(a_0a_1\dots a_{n-1}) \le \log n + O(1)$ for computability, see, e.g.,[4, problems 48 and 49]).

With prefix complexity we have a weaker property:

Theorem. *Every K-trivial sequence is* $\mathbf{0}'$*-computable.*

Here $\mathbf{0}'$ is the oracle for the halting problem.

Proof. Assume that $K((a)_n) = K(n) + O(1)$. Recall that $(a)_n = a_0a_1\dots a_{n-1}$ is equivalent to $(n,(a)_n)$, and use the formula for the complexity of a pair:

$$K((a)_n) = K(n,(a)_n) = K(n) + K((a)_n \mid n, K(n))$$

(with $O(1)$-precision); this means that

$$K((a)_n \mid n, K(n)) = O(1).$$

So $(a)_n$ belongs to a $\mathbf{0}'$-computable (given n) list of n-bit strings that has size $O(1)$. Therefore, a is a path in a $\mathbf{0}'$-computable tree of bounded width and is therefore $\mathbf{0}'$-computable. (Indeed, assume that the tree has k infinite paths that diverge before some level N. At levels after N we can identify all the paths, since all other vertices have finite subtrees above them, and we may wait until only k candidates remain.) □

The existence of (non-computable) K-trivial sets is not obvious, but not very difficult, even if we additionally require the set to be enumerable.

Theorem. *There exists an enumerable undecidable K-trivial set.*

The existence of noncomputable K-trivial sequences was shown by Solovay in 1970s (answering Chaitin's question).

Proof. We want $K((a)_n)$ to be (almost) equal to $K(n)$. In terms of a priori discrete probability, we want to match $\mathbf{m}(n)$ at the vertex $(a)_n$. In game terms: when opponent (who "lower semicomputes" $\mathbf{m}$) increases the weight of n (the sum of his total weights never exceeds 1), we should increase the weight of some vertex (=string) of length n, achieving the same (up to $O(1)$-factor) weight. Moreover, all these vertices should lie on a tree path that corresponds to a enumerable set, and the sum of our weights should be bounded by some constant.

It would be trivial for computable a: just fix some computable a and then place at $(a)_n$ the same weight as the opponent uses for n. But we want a to be non-computable. It is convenient to ensure that a is a characteristic function of a simple set (in Post's sense).

Recall that a simple set A is a set with infinite complement that has non-empty intersection with every W_n that is infinite (here by W_n we denote n-th enumerable set in some natural numbering of all c.e. sets). As in the classical Post's construction, we want for every n to put in A some element of W_n greater

than $2n$, and then forget about W_n. (The bound $2n$ is needed to guarantee that the set has infinite complement.) Post did it without reservations (as soon as some element greater than $2n$ is discovered in W_n, it is added to A), but now, when such an element is found, we have to pay something for it: when we add some number u to A, the corresponding path a, the characteristic sequence for A, changes; the weight used on $(a)_u$ is lost, and should be recreated at the new place. Moreover, all the weights put at the extensions of $(a)_u$ are lost, too (since now these vertices are not on the path, and the new weights should be placed along the new path). This lost amount can be called the *cost* of the action.

Now we can explain the construction. Initially our set A is empty, and the corresponding path a is all zeros. While observing the growth of the discrete a priori probability $\mathbf{m}(n)$, we replicate the corresponding values along the path a. We also enumerate all W_n in parallel, and add some element $u \in W_n$ to A, if three conditions are satisfied:

- $u > 2n$;
- W_n was not satisfied before;
- the cost of this action is small (say, less than 2^{-n}, so the total cost for all n is bounded).

Here the cost of the action is the total weight placed at the vertices in a starting from level u (this weight is lost and should be replicated along the new path).

In this way the total weight used by us is bounded: the lost weight is bounded by $\sum 2^{-n}$, and the replicated weight is bounded by $\sum \mathbf{m}(n)$.

If W_n is infinite, it contains arbitrarily large elements, and the cost of adding u is bounded by

$$\mathbf{m}(u) + \mathbf{m}(u+1) + \mathbf{m}(u+2) + \ldots,$$

which is guaranteed to go below threshold for large u. So every infinite W_n will be served at some moment. □

This proof can be represented in a game form. In such a simple case this looks like an overkill, but the same technique is useful in more complicated cases, so it is instructive to look at this version of the proof. The game field consists of the set of the natural numbers (*lengths*), the full binary tree, and sets $W_1, W_2, \ldots$ (of natural numbers). The opponent increases the weights assigned to lengths: each length has some weight that is initially zero and can be increased by the opponent at any moment by any rational non-negative number; the only restriction is that the total weight (of all lengths) should not exceed 1. Also the opponent may add new elements to any of the sets W_i (initially they are empty). We construct a path a in the binary tree that is a characteristic sequence of some set A (initially empty) by adding elements to A; we also increase the weights of vertices of the binary tree (in the same way as the opponent does for lengths); our total weight should not exceed 2.

One should also specify when the players can make moves; it is not important (the rules of the game always allow each player to postpone moves), but let us

agree that the players make their moves in turns and every move is finite (finitely many weights of lengths and vertices are increased by some rational numbers, and finitely many new elements are added to W_i and A). This is the game with full information, the moves of one player are visible to the other one.

The game is infinite, and the winner is determined in the limit (assuming that both player obey the weight restrictions). Namely, we win if

- for the limit path a our weight of $(a)_n$ is not less than the opponent's weight of n;
- for each n, if W_n is infinite, then W_n has a common element with A.

The winning strategy is as described: we match the opponent's weight along the current path, and also we add some u to A (changing the path and matching the opponent's weights along the new path) if u belongs to W_n, is greater than $2n$ and the loss (our total weight along the current path above u) does not exceed 2^{-n}.

This is a computable winning strategy. Indeed, the limit weights of all lengths form a converging series, so if W_n is infinite, it has some element that is greater than $2n$ and for which the loss (bounded by the tail of this series) is less than 2^{-n}.

Imagine now that we use this computable winning strategy against the "blind" computable opponent that ignores our moves and just enumerates from below the a priori probability (as lengths' weights) and the sets W_i (the list contains all enumerable sets). Then the game is computable, our limit A is an enumerable simple set, and our weights for the prefixes of a (and therefore $\mathbf{m}((a)_n)$, since the limit weights form a lower semicomputable semimeasure) match $\mathbf{m}(n)$ up to $O(1)$-factor.

2 *K*-trivial and *K*-low sequences

Now we know that non-computable K-trivial sequences do exist. Our next goal is to show that they are "almost computable". Namely, they are K-low in the sense of the following definition.

Consider a bit sequence a; one can relativize the definition of prefix complexity using a as a oracle (the decompressor may use the values of a_i in its computation). For every oracle this relativized complexity K^a does not exceed (up to $O(1)$ additive term) the non-relativized prefix complexity, since the decompressor may ignore the oracle. But it can be smaller or not, depending on the oracle.

Definition. *A sequence a is K-low if $K^a(x) = K(x) + O(1)$.*

In other words, K-low oracles are useless for compression (better to say, decompression) purposes.

Obviously, computable oracles are low; the question is whether there exist non-computable low oracles. One can note that *"classical" undecidable sets,*

like the halting problem, are not low: with oracle $\mathbf{0}'$ the table of complexities of all n-bit strings has complexity $O(\log n)$, but its non-relativized complexity is $n - O(1)$. (One can also consider the relativized and non-relativized complexities of the prefixes of Chaitin's Ω-numbers.) Note also that *K-low oracles are K-trivial* (since $K^a((a)_n) = K^a(n) + O(1)$: the sequence a is computable in the presence of oracle a).

It turns out that the reverse implication is true, and this is quite surprising. For example, one may note that *the notion of a K-low sequence is Turing-invariant* (i.e., depends only on the computational power of the sequence) but for K-triviality there are no reasons to expect this (since the definition deals with prefixes).

On the other hand, it is easy to see that *if a and b are two K-trivial sequences, then their join* (the sequence $a_0b_0a_1b_1a_2b_2\ldots$) *is also K-trivial.* Indeed, as we have mentioned, the K-triviality of a sequence a means that $K((a)_n \mid n, K(n)) = O(1)$; if also $K((b)_n \mid n, K(n)) = O(1)$, then (bound for the complexity of a pair) $K((a)_n, (b)_n \mid n, K(n)) = O(1)$, so $K(a_0b_0a_1b_1\ldots a_{n-1}b_{n-1} \mid n, K(n)) = O(1)$, and therefore $K(a_0b_0a_1b_1\ldots a_{n-1}b_{n-1}) = K(n) + O(1)$. It remains to note that $K(n) = K(2n)$ and that we can extend the equality to sequences of odd length, since adding one bit changes the complexity of the sequence and its length only by $O(1)$. The similar result for low sequences is not obvious: if each of the sequences a and b separately do not change the complexity function, why their join is also powerless in this regard? Not obvious at all.

The proof of the equivalence (every K-trivial sequence is K-low) requires a rather complicated combinatorial construction. It may be easier to start with a weaker statement and show that every K-trivial sequence is weaker (as an oracle) than the halting problem. (It follows from the equivalence statement, since $\mathbf{0}'$ is not K-trivial, see above.) This argument is given in the next section. On the other hand, the full proof is not so complicated, so the reader may also skip the next section and to read the equivalence proof without this training.

3 *K*-trivial sequence cannot compute $0'$

In this section we prove that a K-trivial sequence cannot compute $\mathbf{0}'$, in the following equivalent version:

Theorem. *No K-trivial sequence can compute all enumerable sets.*

(Note that together with the existence result proved above it gives a solution of the Post problem, the existence of non-complete enumerable undecidable sets.)

Proof. The proof consists of several steps.

Game reformulation.

We use the game argument and consider the following game with full information. The opponent approximates some sequence A by changing the values of

Boolean variables $a_0, a_1, a_2, \ldots$ (so $A(i)$ is the limit value of a_i). He also assigns increasing weights to strings; the total weight should not exceed 1. We assume that initially all weights are zeros (and that the initial values of a_i are also zeros—just to be specific).

We assign increasing weights to integers (lengths); the sum of our weights is also bounded by 1. Also we construct some set W by (irreversibly) adding elements to it.

The opponent wins the game if

- each variable a_i is changed only finitely many times (so some limit sequence A appears);
- the (opponent's) limit weight of $(A)_i$, the i-bit prefix of A, is greater than (ours) limit weight of i, up to some multiplicative constant;
- the set W is Turing-reducible to A.

It is enough to construct a computable winning strategy in this game. Indeed, assume that some K-trivial A computes $\mathbf{0}'$. We know that then A is limit computable, so the opponent can computably approximate it, and at the same time approximate from below the a priori probabilities for all strings (ignoring our moves, as usual). Our strategy will them behave computably, generating some lower semicomputable semimeasure on lengths, and some enumerable set W. Then, according to the definition of the game, either this semimeasure is not matched by $\mathbf{m}((A)_i)$, or W is not Turing-reducible to A. In the first case A is not K-trivial; in the second case A is not Turing-complete.

Reduction to a game with fixed machine and constant.

How can we win computably this game? First we consider a simpler game when the opponent has to declare in advance the constant c that relates the two semimeasures, and the machine Γ that reduces W to A. Imagine that we can win this game: for each c and Γ we have a (uniformly) computable strategy that wins in the c-Γ-game (defined in a natural way). Since the constant c in the definition of c-Γ-game is arbitrary, we may use c^2 instead of c and assume by scaling that we can force the opponent to spend more than 1 while using only $1/c$ total weight and allowing him to match our moves up to factor c.

Now we mix the strategies for different c and Γ into one strategy. Note that two strategies that simultaneously increase weights of some lengths, can only help each other, so we need only to ensure that the sum of the increases made by all strategies is bounded by 1. More care is needed for the other part: each of the strategies constructs its own W, so we should isolate them. For example, to mix two strategies, we split $\mathbb{N}$ into two parts N_1 and N_2, say, odd and even numbers, and let the first/second strategy construct a subset of N_1/N_2. Of course, each strategy should then beat not the machine Γ itself, but its restriction on N_i (the composition of Γ and the embedding of N_i into $\mathbb{N}$). In a similar way we can mix countably many strategies (splitting $\mathbb{N}$ into countably many countable sets in a computable way).

It remains to consider some computable sequence $c_i \to \infty$ such that $\sum 1/c_i \le 1$ and a computable sequence Γ_i that includes every machine Γ infinitely many times. (The latter is needed because we want every Γ to be beaten with arbitrarily large constant c.) Combining the strategies for these games as described, we get a computable winning strategy for the full game.

It is convenient to scale the c-Γ-game and require the opponent to match our weights exactly (without any factor) but allow him to use total weight c (instead of 1). We will prove the existence of the winning strategy by induction (the strategy for some c will be used to construct a strategy for bigger c). To start, let us first construct the strategy for $c < 2$.

Winning a game with $c < 2$: strong strings

This winning strategy deals with some fixed machine Γ and ensures $\Gamma^A \ne W$ at one fixed point (say, 0); the other points are not used. Informally, we wait until the opponent puts a lot of weight on strings that imply $0 \notin \Gamma^A$. If this never happens, we win in one way; if this happens, we then add 0 to W and win in another way.

Let us explain this more formally. We say that a string u is *strong* if it (as a prefix of A) forces $\Gamma^A(0)$ to be equal to 0 (i.e., Γ outputs 0 on input 0 using only oracle queries in u). Simulating the behavior of Γ for different oracles, we can enumerate all strong strings. During the game we look at the following quantity:

> *the total weight that our opponent has put on all (known) strong strings.*

This quantity may increase because the opponent puts more weight or because we discover new strong strings, but never decreases. We try to make the opponent to increase this quantity (see below about how it is done). As we shall see, if he refuses, he loses the game (and the element 0 remains outside W). If this quantity becomes close to 1, we change our mind and add 1 into W. After that all the weight put on strong strings is lost: they cannot be the prefixes of A such that $\Gamma^A = W$, if $1 \in W$. So we can make our total weight equal to 1 in an arbitrary way (adding weight somewhere if the total weight was smaller than 1), and the opponent needs to use additional weight 1 along some final A that avoids all strong strings, therefore his weight becomes close to 2.

Winning a game with $c = 1$: gradual increase

So our goal is to force the opponent to increase the total weight of strong strings. Let us describe our strategy as a set of substrategies (processes) that run in parallel. For each strong vertex x there is a process P_x; we start it when we discover that x is strong. This process tries to force the opponent to increase the weight of some vertex above x (this vertex is automatically strong); we want to make the lengths of these strings different, so for every string x we fix some number l_x that is greater than $|x|$, the length of x.

Process P_x is activated when the current path goes through vertex x and sleeps otherwise (for example, it may never become active). So—when running—it always sees that the current path goes through x. The process gradually increases the weight of length l_x: it adds some small δ_x to the weight of l_x and waits until the opponent matches[1] this weight along the current path (whatever this path is), then increases the weight again by δ_x, etc. The values of δ_x are fixed for each x in such a way that $\sum_x \delta_x$ is small. The process repeats this increase until it gets a termination signal from the supervisor (see below when this happens). Note that at any moment the current path can change in such a way that x is no more on it; then P_x is put to sleep until x becomes on the current path again (and this may never happen).

The supervisor sends the termination signal to all the processes when (and if) the total weight they have used comes close to 1. After that the strategy adds 0 to W, as explained above.

Let us show that it is indeed the winning strategy. Consider a game where it is used. By construction, we do not violate the weight restriction. If the opponent has no limit path, he loses. If there is a limit path, but it does not has strong vertices on it, he also loses ($\Gamma^A(0) \neq 0$ for the limit A while $0 \notin W$). If x is a strong vertex on the limit path, the process P_x is active starting from some moment, so it can use all the weight (unless other processes do it earlier); in both cases the termination signal is sent. It remains to note that at this moment (when the termination signal is sent) the total weight put on strong string is close to 1: indeed, all the weight put by us is matched by the opponent, except for one portion of size δ_x for each vertex x; the used weight is close to 1, and only small amount ($\sum_x \delta_x$) can be lost.

This argument shows how we can win the c-Γ-game for $c < 2$.

Induction statement

The idea of the induction step is simple: instead of forcing the weight increase for (some extension of) a strong vertex u directly, we can recursively call the described strategy at the subtree rooted at u (adding or not adding some other element to W instead of 0), and save our weight (almost twice, since we know how to win the game with c close to 2). In this way we can win the game for arbitrary $c < 3$, and so on.

To be more formal, we consider a recursively defined process $P(k, x, \alpha, L, M)$ with the following parameters:

- $k > 0$ is a rational number, the required coefficient of weight increase;
- x is the root of the subtree where the process operates;
- $\alpha > 0$ is also a rational number, our "budget" (how much weight we are allowed to use);

[1] A technical remark: note that in our description of the game we have required that the opponent's weight along the path is strictly greater than our weight: if this is true in the limit, it happens on some finite stage.

- L is an infinite set of integers (lengths where our process may increase weight);[2]
- M is and infinite set of integers (numbers that our process may add to W).

The process can be started (or waked up) only when x is a prefix of the current path A, and is put to sleep when A changes and this is no more true, until x becomes the path of the current path again. It is guaranteed that P never violates the rules (about α, L, and M). Running in parallel with other processes (as a part of the game) and assuming that other processes do not touch lengths in L and numbers in M, the process $P(k, x, \alpha, L, M)$ guarantees (if not put into sleep forever or terminated externally) that one of the following possibilities happens:

- either limit path A does not exist,
- or $W \neq \Gamma^A$ for limit A,
- or the opponent never matches some weight put on some length in L,
- or the opponent spends more than $k\alpha$ weight on vertices above x with lengths in L.

Induction base

Now we can adapt the construction of the previous section and construct a process $P(k, x, \alpha, L, M)$ with these properties for arbitrary $k < 2$. For each y above x we select some $l_y \in L$ (different for different y), and select some positive δ_y such that $\sum \delta_y$ is small compared to the budget α. We choose some $m \in M$ and consider y (a vertex above x) as strong if it guarantees that $\Gamma^A(m) = 0$. Then for all strong y we start the process P_y that is activated when y is in the current path and increases the weight of l_y in δ_y-steps waiting until the opponent matches it. We terminate all the processes when (and if) the total weight used becomes close to α, and then add m to W (making all the weight placed by the opponent on strong vertices useless).

The restrictions are satisfied by the construction. Let us check the promised property. If there is no limit path, there is nothing to check. If the limit path does not go through x, we have no obligations (the process is put into sleep forever). Assume that the limit path goes through x and the process was not terminated externally. If there is no strong vertex on the limit path A, then $m \notin W$, but $\Gamma^A(m)$ is not 0, so $W \neq \Gamma^A$. If there is a strong vertex y on the limit path, then the process P_y was started and worked without interruptions (starting from some moment), so either some of the δ_y-increases was not matched (third possibility) or the termination signal was sent. In the latter case the total

[2]To use infinite sets as parameters, we should restrict ourselves to some class of infinite sets. For example, we may consider infinite decidable sets and represent them by programs enumerating their elements in the increasing order.

weight used was close to α, and after adding m to W it is lost, so either our weight is not matched or almost 2α is spent by the opponent above x (recall that all processes are active only when the current path goes through x).

Induction step

The induction step is similar: we construct the process $P(k, x, \alpha, L, M)$ in the same way as for the induction base. The difference is that instead of δ_y-increase in the weight of l_y the process P_y now recursively calls

$$P(k', y, \delta_y, L', M')$$

for vertex y with some smaller k' (one can take $k' = k - 0.5$), the budget δ_y and some $L' \subset L$ and $M' \subset M$. If the started process forces the opponent to spend more than $k'\delta_y$ on the vertices above y with lengths in L', then a new process

$$P(k', y, \delta_y, L'', M'')$$

is started for some other $L'' \subset L$ and $M'' \subset M$, etc. All the subsets $L', L'', \ldots$ should be disjoint (and also disjoint for different y), as well as $M', M'', \ldots$. So we should first of all split L into a sum of disjoint infinite subsets L_y parametrized by y and then split each L_y into $L'_y + L''_y + \ldots$ (for the first, second, etc. recursive calls). The same is done for M, but here we in addition to the sets M_y select some m outside all M_y. This m is used to define strong vertices as those that guarantee $\Gamma^A(m) = 0$.

We start processes P_y (as described above: each of them makes a potentially infinite sequence of recursive calls with the same $k' = k - 0.5$ and budget δ_y) for all discovered strong vertices y (putting them into sleep when y is not on the current path). We take note of the total weight used by all P_y (for all y) and sent a termination signal to all P_y when this weight comes close to the threshold α (so it never crosses this threshold).

Let us show that we achieve the declared goal (assuming that the recursive calls achieve it). First, the restrictions about L, M and α are guaranteed by the construction. If there is no limit path, we have no other obligations. If the limit path exists but does not go through x, our process will be externally put to sleep, and again we have no obligations. So we may assume that the limit path goes through x and that our process is not terminated externally. If the limit path does not go through any strong vertex (defined using m), then $W \neq \Gamma^A$ for the limit path A, since $m \notin W$ and $\Gamma^A(m)$ does not output 0. If the limit path goes through some strong y, the process P_y will be active starting from some point, and makes recursive calls $P(k', y, \delta_y, L', M')$,..., $P(k', y, \delta_y, L'', M'')$, etc. Now we use the inductive assumption and assume that these calls achieve their declared goals. Consider the first call. If it succeeds by achieving one of three first alternatives, then we are done. If it succeeds by achieving the fourth alternative, i.e., by spending more than $k'\delta_y$ on the weights from L', then the second call is made, and again either we are done or the opponent spends more than $k'\delta_y$ on the weights from L''. And so on: at some point we either succeed

globally, or exhaust the budget and our main process sends the termination signal to all P_y. Then all the weight spent, except for the δ_y's for the last call at each vertex, is matched by the opponent with factor k', and on the final path the opponent has to match it with factor 1, so we are done (assuming that $k < k' + 1$ and $\sum \delta_x$ is small enough).

This finishes the induction step, so we can win every c-Γ-game by calling the recursive process at the root. We have proven that K-trivial sets do not compute $\mathbf{0}'$. □

4 *K*-trivial sequences are *K*-low

Now we want to prove the promised stronger result:

Theorem. *All K-trivial sequences are K-low.*

Proof. Let us start with some preparations.

First, we already know that K-trivial sequences are $\mathbf{0}'$-computable. So it is enough to show that every K-trivial $\mathbf{0}'$-computable sequence is K-low.

Second, let us provide a convenient way to represent the a priori probability $\mathbf{m}^A(\cdot)$ with oracle A. (We need to show that it is not significantly bigger than $\mathbf{m}(\cdot)$.) We may describe $\mathbf{m}^A$ in the following way. The sequence A is a path in a full binary tree. Imagine that at every vertex of a tree there is a label of the form (i, η) where i is an integer, and η is a non-negative rational number. This label is read as "please add η to the weight of i". We assume that the labeling is computable. We also require that for every path in the tree the sum of all rational numbers along the path does not exceed 1. Having such a labeling, and a path A, we can obey all the labels along the path, and get a semimeasure on integers. This semimeasure is semicomputable with oracle A.

In fact, this construction is quite general: for every machine M with oracle generating a lower semicomputable discrete semimeasure, we can find a labeling that gives the same semimeasure (in the way described) for every oracle. Indeed, we may simulate the behavior of M for different oracles A, and look which part of A was read when some increase in the output semimeasure happens. This can be used to create a label at some tree vertex. We need to make the labeling computable; also, according to our assumption, each vertex adds weight only to one object, but both requirements can be easily fulfilled by postponing the weight increase (we push the queue of postponed requests up the tree). If the sum of the increase requests along some path becomes greater than 1, this means that for A extending this path we do not get a semimeasure. As usual, we can trim the requests and guarantee that we get semimeasures along all paths, not changing the existing semimeasures.

We may assume now that some computable labeling is fixed such that for every path A the resulting semimeasure (obtained by fulfilling all the requests along the path) is $\mathbf{m}^A$.

Game description

We prove this theorem by providing a winning strategy in some game. (If you have read the previous section, note that the part related to K-triviality is the same.) The opponent approximates some sequence A by changing the values of Boolean variables $a_0, a_1, a_2, \ldots$ and assigns increasing weights to strings; the total weight should not exceed 1. We assume that initially all weights are zeros (and that the initial values of a_i are also zeros).

We assign increasing weights to integers (*lengths*); the sum of our weights is also bounded by 1. This semimeasure will be compared to the opponent's weights along his path. We also assign increasing weights to another type of integers, called *objects*: on them we compare our semimeasure with the semimeasure $\mathbf{m}^A$ (determined by the opponent's limit path A)[3].

The opponent wins the game if the following three conditions are satisfied:

- the limit sequence A exists;
- opponent's semimeasure along the path $*$-exceeds our semimeasure on lengths, i.e., there exists some $c > 0$ such that for all i the opponent's weight of $(A)_i$ is greater than (our weight of i)$/c$;
- our semimeasure on objects does not $*$-exceed $\mathbf{m}^A$.

It is enough to construct a computable winning strategy in this game. Indeed, then we can play it against $\mathbf{0}'$-computable sequence A and the universal semimeasure on strings. Then our moves are computable, and we generate semimeasures on lengths and objects. The winning condition guarantees now that (assuming the limit path A exists) either the opponent's semimeasure along the path is not maximal (so A is not K-trivial), or that $\mathbf{m}^A$ is $*$-bounded by $\mathbf{m}$ (so A is K-low).

As in the previous theorem, we consider an easier (for us) version of the game where the opponent starts the game by declaring some constant c that he plans to achieve (in the second condition), and we need to beat only this c. If we can win this game with $c = 2^{2k}$, then we can win the game with $c = 2^k$ using only 2^{-k} total weight, so we can combine these strategies for all k (we also assume that the total weight on objects for kth strategy is also bounded by 2^{-k}, but this is for free, since we need only to $*$-exceed $\mathbf{m}^A$ without any restrictions on the constant). So it remains to win the game for each c. It is convenient to scale that game and assume that the opponent needs to match our weights on length exactly (not up to $1/c$-factor), but his total weight is bounded by c (not 1).

Winning the game for $c < 2$

For $c = 1$ the game is trivial, since we require that the opponent's weight along the path is strictly greater than our weight on lengths, so it is enough to assign

[3]Formally speaking, we construct two semimeasures on integers; to avoid confusion, it is convenient to call their arguments "lengths" and "objects".

weight 1 to some length. We start our proof by explaining the strategy for the case $c < 2$.

The idea can be explained as follows. The naïve strategy is to believe all the time that the current path A is final, and just assign the weights to objects according to $\mathbf{m}^A$, computed based on the current path A. (In fact, at each moment we look at some finite prefix of A and follow the labels that appear on this prefix.) If indeed A never changes, this is a valid strategy: we achieve $\mathbf{m}^A$, and never exceed the total weight 1 due to the assumption about the labels. But if the path suddenly changes, then all the weight placed because of vertices on old path outside the new path, is lost. If we now follow all the labels on the new path, then our total weight on objects may exceed 1 (it was bounded only along every path, and now we have a part of the old path plus the new path).

There is some partial remedy: we can match the weights only up to some constant (say, use only 1% of what the label asks). This is enough since the game allows us to match the measure with arbitrary constant factor. In this way we can tolerate up to hundred changes in the path (each new path generates new weight at most 0.01), but this does not really help, since the number of changes (of course) is not bounded. Not a surprise, since we did not use the other part of the game, and without this part there is no hope (not all $\mathbf{0}'$-computable A are low).

How can we discourage the opponent to change the path? We can assign a non-zero weight to some length and wait until the opponent matches this weight along the current path. This is a kind of incentive for the opponent not to leave the vertex where he has put some weight: if the opponent leaves it, he would be forced to waste this weight (and put the same weight along the final path for the second time). After that we could implement a request (label) at this vertex: at least we know that if later the path changes and we lose some weight (by implementing the request not on the limit path), the opponent loses some weight, too. This helps if we are careful enough.

Let us explain the details. It would be convenient to represent the strategy as the set of parallel processes: for each vertex x we have a process P_x that is awake when x is a prefix of the current path (and sleeps when x is not). When awake, the process P_x tries to create the incentive for the opponent not to leave x, by forcing him to increase the weight of some vertex above x. To make the processes more independent (and the analysis simpler), let us assume that for every vertex x some length $l_x \geq |x|$ is chosen, lengths assigned to different vertices are different, and P_x increases only the weight of l_x.

Now we are ready to describe the process P_x. Assume that x carries the request (i, η) that asks to add η to the weight of object i. The process P_x increases the weight of l_x (adding small portions to it and waiting after each portion until the opponent matches this increase along the current path, in some vertex above x). When (and if) the weight of l_x reaches $\varepsilon\eta$ (where ε is some small positive constant; the choice of ε depends on c, see below), the process increases the weight of object i also by $\varepsilon\eta$ and terminates.

The processes P_x for different vertices x run in parallel independently, except for one thing: if the total weight spent by all the processes reaches 1, we

terminate all of them (so the increase that would make the total weight greater than 1 is blocked); after that our strategy stops working (in the hope that the opponent would be unable to match already existing weights not crossing the threshold c).

About the small portions: for each vertex x we choose in advance the size δ_x of the portions used by P_x, in such a way that $\sum_x \delta_x < \varepsilon$. (We use here the same small ε as above.) In this way we guarantee that the total loss (last portions that were not matched because the opponent changes the path instead and does not return, so the process is not resumed) is bounded by ε.

It remains to prove that this strategy wins c-game for c close to 2, assuming that ε is small enough. First note two properties that are true by construction:

- the sum of our weights for all lengths does not exceed 1;
- at every moment the sum of (our) weights for all objects does not exceed by the sum of (our) weights for all lengths.

(Indeed, we stop the strategy preventing the violation of the first requirement, and the second is guaranteed for each x-process and therefore for the entire strategy.)

If there is no limit path, the strategy wins the game by definition. So assume that limit path A exists. Now we count separately the weights used by processes P_x for x is on the limit path A, and for x outside A. Since the weights for x are limited by the request in x (with factor ε) and sum of the all requests along A is at most 1, the sum of the weights along A is bounded by ε. Now there are two possibilities: either the strategy was stopped (when trying to cross the threshold) or it runs indefinitely.

In the first case the total weight is close to 1 (at least $1-\varepsilon$, since the next increase will cross 1, and all the portions δ_x are less than ε). So the weight used by processes outside A is at least $1-2\varepsilon$, and if we do not count the last (unmatched) portions, we get at least $1-3\varepsilon$ of weight that the opponent needs to match twice: it was matched above x for P_x, and then should be matched again along the limit path (that does not go through x; recall that we consider the vertices outside the limit path). So the opponent needs to spend at least $2-6\varepsilon$, otherwise he loses.

In the second case each process P_x for x on the limit path is awake starting from some moment, and is never stopped, so it reaches its target value $\varepsilon\eta$ and adds $\varepsilon\eta$ to the object i (here (i,ε) is the request in vertex x). So our weights on the objects match $\mathbf{m}^A$ for limit path A up to factor ε, and the opponent loses. (We know also that the total weight on objects does not exceed 1, since it is bounded by the total weight on lengths.)

We therefore have constructed a winning strategy for $2-6\varepsilon$ game, and by choosing a small ε can win c-game for every $c<2$.

Using this strategy on a subtree

Preparing for the recursion, let us look at the described strategy and modify it for using in a subtree rooted at some vertex x. We also scale the game and

assume that we have some budget α that we are allowed to use (instead of total weight 1). To guarantee that the strategy does not interfere with other actions outside x-subtree we agree that it uses lengths only from some infinite set L of length and nobody else touches these lengths. Then we can assign $l_y \in L$ for every y in the subtree and use them as before.

Let us describe the strategy in more details. It is composed of processes P_y for all y above x. When x is not on the actual path, all these processes sleep, and the strategy is sleeping. But when path goes through x, some processes P_y (for y on the path) become active and start increasing the weight of length l_y by small portions δ_y (the sum of all δ_y now is bounded by $\alpha\varepsilon$, since we scaled everything by α). A supervisor controls the total weight used by all P_y, and as soon as it reaches α, terminates all P_y. When the process P_y reaches the weight $\alpha\varepsilon\eta$, it increases the weight of object i by $\alpha\varepsilon\eta$ (here (i,η) is the request at vertex y). So everything is as before, but with factor α and only on x-subtree.

What does this strategy guarantee?

- The total weight on lengths used by it is at most α.
- The total weight on objects does not exceed the total weight on lengths.
- If the limit path A exists and goes through x, then either
  - the strategy halts and the opponent either fails to match all the weights or spends more than $c\alpha$ on x-subtree; or
  - the strategy does not halt, and the semimeasure on objects generated by this strategy, $*$-exceeds $\mathbf{m}^A$, if we omit from $\mathbf{m}^A$ all the requests on the path to x.

The argument is the same: if the limit path exists and the strategy does not halt, then all the requests along the limit path (except for finitely many of them below x) are fulfilled with coefficient $\alpha\varepsilon$. If the strategy halts, the weight used along the limit path does not exceed $\alpha\varepsilon$ (since the sum of requests along each path is bounded by 1), the weight used in the other vertices of x-subtree is at least $\alpha(1-2\varepsilon)$, including at least $\alpha(1-3\varepsilon)$ matched weight that should be doubled along the limit path, and we achieve the desired goal for $c = 2-6\varepsilon$.

Remark. In the statement above we have to change $\mathbf{m}^A$ by deleting the requests on the path to x. We can change the construction by moving requests up the tree when processing vertex x to get rid of this problem. One may also note that omitted requests deal only with finitely many objects, so one can average the resulting semimeasure with some semimeasure that is positive everywhere. So we may ignore this problem in the sequel.

How to win the game for $c < 3$

Now we make the crucial step: show how we can increase c by recursively using our strategies. Recall our strategy for $c < 2$, and change it in the following ways:

- Instead of assigning some length l_x for each vertex x, let us assign an infinite (decidable uniformly in x) set L_x of integers; all elements should be greater than $|x|$ and for different x these sets should be disjoint (it is easy to achieve this).
- We agree that process P_x (to be defined) uses only lengths from L_x.
- Again P_x is active when x is on the current path, and sleeps otherwise.
- Previously P_x increased the weight of l_x in small portions, and after each small increase waited until the opponent matched this increase along the current path. Now, instead of that, P_x calls the x-strategy described in the previous section, with small $\alpha = \delta_x$, waits until this strategy terminates forcing the opponent to spend almost $2\delta_x$, then calls another instance of x-strategy, waits until it terminates, and so on. For that, P_x divides L_x into infinite subsets $L_x^1 + L_x^2 + \ldots$, using L_x^s for sth call of x-strategy, and using δ_x as the budget for each call.

There are several possibilities for the behavior of x-strategy called recursively. It may happen that it does infinitely many steps. This happens when x is a part of the limit path, x-strategy never exceeds its budget δ_x, and the entire strategy does not come close to 1 in its total spending. It this case we win the game (the part of the semimeasure on objects due to x-strategy is enough to $*$-exceed $\mathbf{m}^A$).[4]

If x is not in the limit path, the execution of x-strategy may be interrupted; in this case we know only that it spent not more than its budget, and that the weight used for objects does not exceed the weight used for lengths. (This is similar to the case when an increase at l_x was not matched because the path changed.)

The x-strategy may also terminate. In this case we know that the opponent used almost twice the budget (δ_x) on the extensions of x, and a new call of x-strategy is made for another set of lengths. (This is similar to the case when the increase at l_x was matched; the advantage is that now the opponent used almost twice our weight!)

Finally, the strategy may be interrupted because the total weight used by x-processes for all x came close to 1. Similar thing happened for the case $c < 2$. After that everything stops, and we just wait until the opponent will be unable to match all the existing weights or forced to use total weight close to 3. Indeed, most of our weight (except for $O(\varepsilon)$) was used not on the limit path and already matched with factor close to 2 there — so matching it again on the limit path makes the total weight close to 3.

Induction step

Now it is clear how one can continue this reasoning and construct a winning strategy for arbitrary c. To get a strategy for some c, we follow the described

[4] This case is called "the golden run" in the original exposition of the proof.

scheme, and the process P_x makes sequential recursive calls of c'-strategies for smaller c' (the difference should be less than 1, for example, we can use $c' = c - 0.5$). More formally, we recursively define a process $S(c, x, \alpha, L)$ where c is the desired amplification, x is a vertex, α is a positive rational number (the budget), and L is an infinite set of integers greater than $|x|$.[5] The requirements for $S(c, x, \alpha, L)$:

- It increases only weights of lengths in L.
- The total weight used for lengths does not exceed α.
- At each step the total weight used for objects does not exceed the total weight used for lengths.
- Assuming that the process is not terminated externally (this means that x belongs to the current path, starting from some moment), it may halt or not, and:
  - If the process halts, the opponent uses more that $c\alpha$ on strings that have length in L and are above x.
  - If the process does not halt and the limit path A exists, the semimeasure generated on objects by this process only, $*$-exceeds $\mathbf{m}^A$.

The implementation of $S(c, x, \alpha, L)$ uses recursive calls of $S(c-0.5, y, \beta, L')$; for each y above x a sequence of those calls is made with $\beta = \delta_y$ and L' that are disjoint subsets of L (for different y these L' are also disjoint), similar to what we have described above for the case $c < 3$. □

5 *K*-low and ML-low oracles

There is one more description of K-low (or K-trivial) sequences: they are sequences that (being used as oracles) do not change the notion of Martin-Löf randomness. As almost all notions of the computability theory, the notion of Martin-Löf randomness can be relativized by an oracle a (this means that the algorithm that enumerates covers for an effectively null set, now may use oracle a). In this way we get (in general) more effectively null sets, and therefore less random sequences. However, for some a the class of Martin-Löf random sequences remains unchanged.

Definition. *A sequence a is MLR-low if every Martin-Löf random sequence remains Martin-Löf random with oracle a.*

[5] The pedantic reader may complain that the parameter is an infinite sets. It in enough to consider infinite sets from some class, say, decidable sets (as we noted in the previous section), or just arithmetic progressions, they are enough for our purposes. Indeed, an arithmetic progression can be split into countably many arithmetic progressions. For example, $1, 2, 3, 4, \ldots$ can be split into $1, 3, 5, 7, \ldots$ (odd numbers), $2, 6, 10, 14 \ldots$ (odd numbers times 2), $4, 12, 20, 28, \ldots$ (odd numbers times 4), etc.

The Schnorr–Levin criterion of randomness in terms of prefix complexity shows that if a is K-low, then a is also *MLR*-low. The other implication is also true (but more difficult).

Theorem. *Every MLR-low sequence is K-low.*

We will prove a more general result.

Definition. *Let a and b be two sequences (considered as oracles). We say that $a \leq_{LK} b$ if*

$$K^b(x) \leq K^a(x) + O(1).$$

We say that $a \leq_{LR} b$ if every Martin-Löf random with oracle b sequence is also Martin-Löf random with oracle a.

If an oracle b is stronger in Turing sense (=computes) some oracle a, then b gives bigger effectively null sets, smaller set of random sequences, and smaller complexity function, so we have $a \leq_{\mathrm{LR}} b$ and $a \leq_{\mathrm{LK}} b$. So both orderings are more coarse than the Turing degree ordering.

Using this definitions, we can reformulate the definitions: sequence a is K-low if $a \leq_{\mathrm{LK}} \mathbf{0}$ and is *MLR*-low if $a \leq_{\mathrm{LR}} \mathbf{0}$. So it is enough to prove the following result:

Theorem. *$a \leq_{LK} b \Leftrightarrow a \leq_{LR} b$ for every a and b.*

Proof. Again in one direction (from left to right) it follows directly from the Schnorr–Levin randomness criterion. The other direction is more difficult[6], and will be split in several steps.

Recall that the set of non-random (in Martin-Löf sense; we do not use other notions of randomness here) sequences can be described using universal Martin-Löf test, as the intersection of effectively open sets

$$U_1 \supset U_2 \supset U_3 \supset \dots$$

where U_i has measure at most 2^{-i}. The following observation goes back to Kučera and says that the first layer of this test, the set U_1, is enough to characterize all non-random sequences.

Lemma 1. *Let U be an effectively open set of measure less than 1 that contains all non-random sequences. Then a sequence $a = a_0a_1a_2\dots$ is non-random if and only if all its tails $a_ka_{k+1}a_{k+2}\dots$ belong to U.*

Proof of Lemma 1. If a is non-random, then all the tails are non-random and therefore belong to U. In the other direction: represent U as a union of disjoint intervals $[u_i]$ (by $[v]$ we denote the set of all sequences that have prefix v). Then $\rho = \sum 2^{-|u_i|}$ is less than 1. If all tails of a belong to U, then a starts with

[6]This could be expected. The relation $a \leq_{\mathrm{LK}} b$ is quantitative: two functions coincide with $O(1)$-precision; the relation $a \leq_{\mathrm{LR}} b$ is more qualitative, we speak here about yes/no question. One can also consider the quantitative version, with randomness deficiencies, but this is not needed: the relation $\leq_{\mathrm{LR}}$ is strong enough.

some u_i, the rest is a tail that starts with some u_j, etc., so a can be split into pieces that are among u_i. The set of sequences of the form "some u_i, then something" has measure ρ, the set of sequence of the form "some u_i, some u_j, then something" has measure ρ^2, etc. These sets are effectively open and their measures ρ^n effectively converge to 0. So their intersection is an effectively null set and a is non-random. Lemma is proven. □

The argument gives also the following

Corollary: *a sequence x is non-random if there exists an effectively open set U of measure less than* 1 *such that all tails of x belong to U.*

This argument can be relativized, so randomness with oracle X can be characterized in terms of X-effectively open sets of measure less that 1: *a sequence x is X-nonrandom if there exists an X-effectively open set U of measure less than* 1 *such that all tails of x belong to U.* This gives one implication in the following equivalence (we denote here the oracles by capital letter not to mix them with sequences):

Lemma 2. *Let A and B be two oracles. Then $A \leq_{LR} B$ if and only if every A-effectively open set of measure less than* 1 *can be covered by some B-effectively open set of measure less than* 1*.*

Proof of Lemma 2. In one direction (if, $\Leftarrow$) it follows for the discussion above: if a is not A-random, its tails can be covered by some A-effectively open set of measure less than 1 and therefore by some B-effectively open set of measure less than 1, so a is not B-random.

In the other direction: let U be an A-effectively open set of measure 1 that cannot be covered by any B-effectively open set of measure less than 1. The set U is the union of A-enumerable sequence of disjoint intervals $[u_1], [u_2], [u_3]$, etc. Consider a set V that is B-effectively open, contains all B-non-random sequences and has measure less than 1 (e.g., the first level of universal B-Martin-Löf test). By assumption U is not covered by V, so some interval $[u_i]$ of U is not (entirely) covered by V.

The set V has the following special property: if it does not contain *all* points of some interval, then it cannot contain *almost all* points of this interval: the non-covered part has some positive measure. Indeed, the non-covered part is a B-effectively closed set, and if it has measure zero, it has B-effectively measure zero, so all non-covered sequences are B-non-random, and therefore should be covered by V.

So we found an interval $[u_i]$ in U whose part of positive measure is outside V. Then consider the set $V_1 = V/u_i$, i.e., the set of infinite sequences α such that $u_i\alpha \in V$. This is a B-effectively open set of measure less than 1, so it does not cover U (again by our assumption). So there exists some interval $[u_j]$ not covered by V/u_i. This means that $[u_iu_j]$ is not covered by V. Then we repeat the argument and conclude that non-covered part has positive measure, so V/u_iu_j is a B-effectively open set of measure less than 1, so it does not cover some $[u_k]$, etc. In the limit we get a sequence $u_iu_ju_k\ldots$ whose prefixes define intervals not covered fully by V. Since V is open, this sequence does not

belong to V, so it is B-random. On the other hand, it is not A-random, as the argument from the proof of Lemma 1 shows. □

Let us summarize where we are now. Assuming that $A \leq_{\mathrm{LR}} B$, we have shown that every A-effectively open set of measure less than 1 can be covered by some B-effectively open set of measure less than 1. And we need to show somehow that $A \leq_{\mathrm{LK}} B$, i.e., $K^B \leq K^A$ (up to an additive constant), or $\mathbf{m}^A \leq \mathbf{m}^B$ (up to a constant factor). This can be reformulated as follows: *for every lower A-semicomputable converging series $\sum a_n$ of reals there exists a converging lower B-semicomputable series $\sum b_n$ of reals such that $a_n \leq b_n$ for every n.*

So to connect our assumption and our goal, we need to convert somehow a converging lower semicomputable series into an effectively open set of measure less than 1 and vice versa. We may assume without loss of generality that all a_i are strictly less than 1. Then $\sum a_n < \infty$ is equivalent to

$$(1 - a_0)(1 - a_1)(1 - a_2) \ldots > 0.$$

This product is a measure of an A-effectively closed set

$$[a_0, 1] \times [a_1, 1] \times [a_2, 1] \times \ldots$$

so its complement

$$\{(x_0, x_1, \ldots) \mid (x_0 < a_0) \vee (x_1 < a_1) \vee \ldots\}$$

is an A-effectively open set of measure less than 1. (Here we split the Cantor space into a countable product of Cantor spaces and identify each of them with $[0, 1]$ equipped with standard uniform measure on the unit interval.) We are finally ready to apply our assumption and find some B-effectively open set V that contains this complement.

Now let us define b_0 as supremum of all z such that

$$[0, z] \times [0, 1] \times [0, 1] \times \ldots \subset V$$

This product is compact for every z, and V is B-effectively open, so we can B-enumerate all rational z with this property, and their supremum b_0 is lower B-semicomputable. Note that all $z < a_0$ have this property, so $a_0 \leq b_0$. In a similar way we define all b_i and get a lower B-semicomputable series b_i such that $a_i \leq b_i$. It remains to show that $\sum b_i$ is finite. Indeed, the set

$$\{(x_0, x_1, \ldots) \mid (x_0 < b_0) \vee (x_1 < b_1) \vee \ldots\}$$

is a part of V, and therefore has measure less than 1; its complement

$$[b_0, 1] \times [b_1, 1] \times [b_2, 1] \times \ldots$$

has measure $(1 - b_0)(1 - b_1)(1 - b_2) \ldots$, therefore this product is positive and the series $\sum b_i$ converges. This finishes the proof. □

Acknowledgments

This exposition was finished while one of the authors (A.S.) was invited to National University of Singapore IMS *Algorithmic Randomness* program. The authors thank IMS for the support and for the possibility to discuss several topics (including this exposition) with the other participants (special thanks to Rupert Hölzl and Nikolai Vereshchagin). We thank also Andre Nies for the suggestion to write down this exposition for the Logic Blog and comments, and all the participants of the course taught by L.B. at Poncelet lab (CNRS, Moscow), especially Misha Andreev and Gleb Novikov. Last but not least, we are grateful to Joe Miller who presented the proof of equivalence between $\leq_{\mathrm{LK}}$ and $\leq_{\mathrm{LR}}$ while visiting LIRMM several years ago.

References

[1] Downey R., Hirschfeldt D. *Algorithmic Randomness and Complexity*, Springer, 2010.

[2] Nies, A., *Computability and Randomness*, Oxford University Press, 2009.

[3] Shen A., Algorithmic information theory and Kolmogorov complexity, lecture notes, Uppsala University Technical Report TR2000-034, `http://www.it.uu.se/research/publications/reports/2000-034/2000-034-nc.ps.gz`

[4] Shen A., Uspensky V., Vereshchagin N., Kolmogorov complexity and algorithmic randomness, MCCME, 2012 (Russian). Draft translation: `www.lirmm.fr/~ashen/kolmbook-eng.pdf`.