

Fundamentals of Computing

Leonid A. Levin (<https://www.cs.bu.edu/fac/lnl/>)

Abstract

These are notes for the course CS-172 I first taught in the Fall 1986 at UC Berkeley and subsequently at Boston University. The goal was to introduce students to basic concepts of Theory of Computation and to provoke their interest in further study. Model-dependent effects were systematically ignored. Concrete computational problems were considered only as illustrations of general principles.

The notes are skeletal: they do have (terse) proofs, but exercises, references, intuitive comments, examples are missing or inadequate. The notes can be used for designing a course or by students who want to refresh the known material or are bright and have access to an instructor for questions.

Each subsection takes about a week of the course. Versions of these notes appeared in [Levin 91].

Acknowledgments. I am grateful to the University of California at Berkeley, its MacKey Professorship fund and Manuel Blum who made possible for me to teach this course. The opportunity to attend lectures of M. Blum and Richard Karp and many ideas of my colleagues at BU and MIT were very beneficial for my lectures. I am also grateful to the California Institute of Technology for a semester with light teaching load in a stimulating environment enabling me to rewrite the students' notes. NSF grants #DCR-8304498, DCR-8607492, CCR-9015276 also supported the work. And most of all I am grateful to the students (see sec.7.2) who not only have originally written these notes, but also influenced the lectures a lot by providing very intelligent reactions and criticism.

Contents

I	Basics	2
1	Deterministic Models; Polynomial Time & Church's Thesis	2
1.1	Rigid Models	3
1.2	Pointer Machines	4
1.3	Simulation	5
2	Universal Algorithm; Diagonal Results	6
2.1	Universal Turing Machine	6
2.2	Uncomputability; Goedel Theorem	7
2.3	Intractability; Compression and Speed-up Theorems	8
3	Games; Alternation; Exhaustive Search; Time vs. Space	9
3.1	How to Win	9
3.2	Exponentially Hard Games	10
3.3	Reductions; Non-Deterministic, Alternating TM; Time / Space	11
II	Mysteries	12
4	Nondeterminism; Inverting Functions; Reductions	12
4.1	An Example of a Narrow Computation: Inverting a Function	12
4.2	Complexity of NP Problems	13
4.3	An NP-Complete Problem: Tiling	14
5	Probability in Computing	15
5.1	A Monte-Carlo Primality Tester	15
5.2	Randomized Algorithms and Random Inputs	16
5.3	Arithmetization: One-Player Games with Randomized Transition	17
6	Randomness	18
6.1	Randomness and Complexity	18
6.2	Pseudo-randomness	19
6.3	Cryptography	20
7	End Matter.	20
7.1	Go On!	20
7.2	Writing Contributions.	21
	References	21

Part I

Basics

1 Deterministic Models; Polynomial Time & Church's Thesis

Sections 1, 2 study deterministic computations. Non-deterministic aspects of computations (interaction, randomization, errors, etc.) are crucial and challenging in advanced theory and practice. Defining them as an extension of deterministic computations is simple. The latter, however, while much simpler conceptually, require elaborate models for definition. These models may be sophisticated if we need to measure all required resources precisely. However, if we only need to define what is computable and get a very rough magnitude of the needed resources, all reasonable models turn out equivalent, even to the simplest ones. We will pay significant attention to this surprising and important fact. The simplest models are most useful for proving negative results and the strongest ones for positive results.

We start with terminology common to all models, later adding specifics of models we actually study. We represent *computations* as graphs: the edges reflect various relations between nodes (*events*). Nodes, edges have attributes: labels, states, colors, parameters, etc. (affecting the computation, or only its analysis). *Causal* edges run from each event to all events immediately essential for its occurrence or attributes. They form a directed acyclic graph (though cycles may be added artificially to mark external input parts).

We will study only *synchronous* computations. Their nodes have a *time* parameter. It reflects logical steps, not necessarily a precise value of any physical clock. Causal edges only span short (typically, ≤ 3 moments) time intervals. One event among the causes of a node is called its *parent*. *Pointer* edges connect the parent of each event to all its other possible causes and reflect connections that allow simultaneous events to interact and have a joint effect. Pointers with the same source have different labels (colors). The colored subgraph of events/edges at a given time is an instant memory *configuration* of the model.

Each non-terminal configuration has *active* nodes/edges around which it may change. The models with only a small active area at any step of the computation are *sequential*. Others are called *parallel*.

Complexity. We use the following measures of computing resources of a machine A on input x :

Time: The greatest depth $D_{A(x)}$ of causal chains is the number of computation steps. The volume $V_{A(x)}$ is the combined number of active edges in all steps. Time $T_{A(x)}$, depending on the context, means either depth or volume (close for sequential models). Note that time complexity is robust only up to a constant factor: Machines can be modified for a larger labels alphabet, representing several locations in one. It would produce identical results in a fraction of time and space (provided, the time limits suffice for transforming the input and output into the other alphabet).

Space: $S_{A(x)}$ or $S_A(x)$ of a synchronous computation is the greatest (over time) size of its configurations. Sometimes excluded are nodes/edges unchanged since the input.

Growth Rates (typically expressed as functions of bit length $n = \|x, y\|$ of input/output x/y):

$$O, \Omega: f(n) = O(g(n)) \iff g(n) = \Omega(f(n)) \iff \sup_n \frac{f(n)}{g(n)} < \infty.$$

$$o, \omega: f(n) = o(g(n)) \iff g(n) = \omega(f(n)) \iff \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0.$$

$$\Theta: f(n) = \Theta(g(n)) \iff (f(n) = O(g(n)) \text{ and } g(n) = O(f(n))).$$

(These are customary but somewhat misleading notations. E.g., $n = O(n^3)$ and $n^2 = O(n^3)$ do not imply $n = n^2$. The clear notations would be like $f(n) \in O(g(n))$.)

Here are examples of frequently appearing growth rates: negligible $(\log n)^{O(1)}$; moderate $n^{\Theta(1)}$ (called polynomial or P, like in P-time); infeasible: $2^{n^{\Omega(1)}}$, also $n! = (n/e)^n \sqrt{\pi(2n+1/3)} + \varepsilon/n$, $\varepsilon \in [0, .1]$. (A rougher estimate $\ln n! = t \ln(t/e)|_{t=1.5}^{n+.5} + O(1)$ follows by using that $|\sum_{i=2}^n g(i) - \int_{1.5}^{n+.5} g(t) dt|$ is bounded by the total variation v of $g'/8$. For monotone $g'(t) = \ln'(t) = 1/t$ the $O(1)$ is $< v < 1/12$.)

The reason for ruling out exponential (and neglecting logarithmic) rates is that the known Universe is too small to accommodate exponents. Its radius is about 46.5 giga-light-years $\sim 2^{204}$ Plank units. A system of $\gg R^{1.5}$ atoms packed in R Plank Units radius collapses rapidly, be it Universe-sized or a neutron star. So the number of atoms is $< 2^{306} \ll 4^{4^4} \ll 5!!$. (Actually, $< 10^{80}$.)

1.1 Rigid Models

Rigid computations have another node parameter: *location (cell)*. Combined with time, it designates the event uniquely. Cells have *structure* or *proximity* edges between them. They (or their short chains) indicate the neighbors to which pointers may be directed.

Cellular Automata (CA)

(CA) are a parallel rigid model. Its sequential restriction is the **Turing Machine (TM)**. The configuration of CA is a (possibly multi-dimensional) grid with a finite, independent of the grid size, alphabet of *states* to label the events. The states include, among other values, pointers to the grid neighbors. At each computation step, the state of each cell can change as prescribed by a *transition* function (also called program) applied to the previous states of the cell and its pointed-to neighbors. The initial state of the cells is the input for the CA. All subsequent states are determined by the transition function.

An example of technology CA represents is a VLSI chip (Very Large Scale Integration) represented as a grid of cells connected by wires (chains of cells) of different lengths. The propagation of signals along the wires is simulated by changing the state of the wire cells step by step. The clock interval can be set to the time the signals propagate through the longest wire. This way the delays affect the simulation implicitly.

An example: the Game of Life (GL). GL is a plane grid of cells, each holds a 1-bit state (dead/alive) and pointers to the 8 adjacent cells. A cell remains dead or alive if the number i of its live neighbors is 2. It becomes (or stays) alive if $i=3$. In all other cases it dies (of overcrowding or loneliness) or stays dead.

A *simulation* of a machine M_1 by M_2 is a correspondence between their memory configurations which is preserved during the computation (may be with some time dilation). Such constructions show that the computation of M_1 on any input x can be performed by M_2 as well. GL can simulate any CA in this formal sense: (See a sketch of an ingenious proof in the last section of [Berlekamp, Conway, Guy 82].)

We fix space and time periods a, b . Cells (i, j) of GL are mapped to cell $(\lfloor i/a \rfloor, \lfloor j/a \rfloor)$ of CA M (compressing $a \times a$ blocks). We represent cell states of M by states of $a \times a$ blocks of GL . This correspondence is preserved after any number t steps of M and bt steps of GL regardless of the starting configuration.

Turing Machines (TMs).

TM is a *minimal CA*. Its configuration – *tape* – is an infinite to the right chain of cells. Cells have pointers to adjacent cells. At each time one of the cells, called *head* is in a state with only one pointer. Only it and its pointer’s target are active, i.e. can change state. The head’s state is often set in a separate alphabet. It is sometimes convenient to assume all cells have single pointers (in the direction toward the head). The input is an array of non-blank cells followed by all blanks at the right. It is often treated as set in unchangeable “ink” part of each cell’s state, excluded from the output. The rest of the cell’s state is in changeable “pencil”.

Another type of CA are **Multi-head TMs (MhTm)** with several heads. $O(1)$ heads may fit in a cell. Heads can split or merge only in the first cell (which, thus, controls the number of active cells). When the last head drops off it, computation halts. This model is convenient for its volume V (the combined number of heads at all steps) being **constructible** with $O(1)$ accuracy, i.e. itself computable in volume $O(V)$.

Exercise: Prove the above statement, i.e. find a method to transform any given MhTm A running in volume V_A into another one B whose volume and the value of output are $\Theta(V_A)$.

Hint: Cells of B may have a field to simulate A and maintain (in other fields) two binary counters with their least significant digits at the leftmost cell: H for the number of heads of A , and V for A ’s volume. H adds its highest digit to the same place in V at each step of A . H maintains heads in every second cell. To move the carry, V borrows a head from the H segment. The head moves it right, to drop to the leftmost 0 digit of V or to a next adjacent head freed already of its carry. Heads freed of carry move left toward the H segment which absorbs them. Borrowed or returned heads make sparse or dense heads regions in H (with adjacent heads or adjacent no-heads). Those shift leftward via transitions $|-|h|h| \rightarrow |h|-|h|$, and $|h|-|-| \rightarrow |-|h|-|$, until absorbed at the leftmost cell.

1.2 Pointer Machines

The memory configuration of a *Pointer Machine (PM)*, called *pointer graph*, is a finite directed labeled multigraph. Edges (*pointers*) are labeled with *colors* from a finite alphabet common to all graphs handled by a given program. The pointers coming out of a node must all differ in colors (which bounds the outdegree). A node is said *seeing* its pointers, their targets, and pointers of those targets.

Some colors are designated as *work* colors and not used in inputs or outputs. One of them is called *active*, as also are pointers carrying it and nodes seeing them. The computation is initiated by inserting an active loop-edge into a node called *root*. It must have directed paths to all nodes. Active pointers must have inverses, form a tree to the root, and can be recolored only in its leaves.

At each step each active node A executes the *transition* specified by the program. In its first *pulling* stage, A acquires copies of all pointers of its pointers targets using “composite” colors: e.g., for a two-pointer path (A, B, C) colored x, y , the new pointer (A, C) is colored xy , or an existing z -colored pointer (A, C) is recolored $\{z, xy\}$. In the next *recoloring* stage, A transforms the colors of its set of pointers, drops the pointers left with composite colors, and may spawn a new node with pointers to and from it. Nodes with no pointers left vanish. Nodes with no path from the root are forever invisible and considered dropped, too. When no active pointers remain, the graph, with all working colors dropped, is the output.

Exercise: Design a PM transforming the input graph into the same one with two extra pointers from each node: to its parent in a DFS spanning tree and to the root. Hint: Nodes with no path to the root can never be activated. But the graph can be copied, copies have extra pointers to the parent, the original, and the root. The originals then dropped. Care is needed not to copy any node twice.

PM can be *parallel, PPM* [Barzdin’, Kalnin’s 74] or *sequential, SPM*: In SPM only pointers to the root, their sources, and nodes connected by two-way pointers with these sources can be active.

A *Kolmogorov* or *Kolmogorov-Uspenskii* Machine (KM) [Kolmogorov, Uspenskii 58], is a special case of Pointer Machine [Schoenhage 80] with all pointers having inverses. This implies the graph’s $O(1)$ in/out-degree which we further assume constant.

Fixed Connection Machine (FCM) is a variant of the PKM with the restriction that pointers, once created, *cannot* be removed, only re-colored. So when the memory limits are reached, the pointer structure freezes, and the computation can be continued only by changing the colors of the pointers.

PPM is the most powerful model we consider: it can simulate the others in the same space/time. E.g., cellular automata are a simple special case of a PPM, restricting the Pointer Graph to be a grid.

Exercise. Design a machine of each model (TM, CA, KM, PPM) which determines if an input string x has a form ww , $w \in \{a, b\}^*$. Analyze time (depth) and space. KM/PPM takes input x in the form of colors of edges in a chain of nodes, with root linked to both ends. The PPM nodes also have pointers to the root. Below are hints for TM,SPM,CA. The space is $O(\|x\|)$ in all three cases.

Turing and Pointer Machines. TM first finds the middle of ww by capitalizing the letters at both ends one by one. Then it compares letter by letter the two halves, lowering their case. The complexity is: $T(x) = O(\|x\|^2)$. SPM acts similarly, except that the root keeps and updates the pointers to the borders between the upper and lower case substrings. This allows constant time access to these borders. So, $T(x)=O(\|x\|)$.

Cellular Automata. At the start, the leftmost cell sends right two signals. Reaching the end the first signal turns back. The second signal propagates three times slower, so they meet in the middle of $w'w$ and disappear. While alive, the second signal copies the input field i of each cell into a special field c . The c symbols will try to move right whenever the next cell’s c field is blank.

So the chain of these symbols alternating with blanks will start growing both ways from the middle of $w'w$. Upon reaching the ends they will push the blanks left and pack themselves into a copy of the w' right of the middle. Having no blank at the right to move to, a c symbol compares itself with the i field of the same cell. If they differ, it generates a signal to halt all activity and reject x . If all comparisons succeed, the last c generates an accepting signal. The depth is: $T(x) = O(\|x\|)$.

1.3 Simulation

We looked at several models of computation. We will see now how the simplest of them – Turing Machine – can simulate all others: these powerful machines can compute no more functions than TM.

Church-Turing Thesis is a generalization of this conclusion: TMs can compute every function computable in any thinkable realistic model of computation. This is not a math theorem because the notion of model is not formally specified. Yet, the long history of studying ways to design real and ideal computing devices makes it very convincing. Moreover, this Thesis has a stronger *Polynomial Time* version which bounds the volume of computation required by that TM simulation by a polynomial of the volume used by the other models. Both forms of the Thesis play a significant role in foundations of Computer Science.

PKM Simulation of PPM. Assume all PPM nodes have pointers to root, as generated in the above HW. PKM represents PPM configuration with extra colors l, r, u used in a u -colored binary tree added to each node X . All (unlimited in number) PPM pointers to X are reconnected to its leaves. The inverses, colored l, r , added to all u -pointers. The number of pointers increases at most 4 times.

To simulate PPM, X gets a binary *name* formed by the l, r colors on its path through the root tree, and broadcasts it down its own tree. For pulling stage X extends its tree to double depth and merges (with combined colors) its own 2-pointer paths with the same targets (seeing identical names). Then X re-colors its pointers as PPM program requires and rebalances its tree. This simulation of a PPM step takes polylogarithmic parallel time.

TM Simulation of PKM. Assume the PKM keeps a roughly balanced root tree. TM tape reflects PKM state as the list of all pointers sorted by (source (l, r)-name, color). TM's transition table reflects the PKM program. To simulate PKM's pulling stage TM creates a copy of each pointer and sorts copies by their sinks.

Now each pointer, located at source, has its copy near its sink. So both components of 2-pointer paths are nearby, stored as double-colored special pointers, and moved to their sources by resorting on the source names. The re-coloring stage is local: all relevant pointers with the same source are nearby.

Once the root has no active pointers, the TM stops. If a PPM computes $f(x)$ in $t(x)$ steps, using $s(x)$ nodes, the simulating TM uses space $S = O(s \log s)$, ($O(\log s)$ bits for each of $O(s)$ pointers) and time $T = O(S^2)t$, as TM sorting takes quadratic time.

Squaring matters ! TM cannot outperform Bubble Sort. Is its quadratic overhead a big deal? In a short time all silicon gates on your PC run, say, Avogadro order clock cycles $X=6 \cdot 10^{23} \sim 2^{2^{6.3}}$ combined. Silicon parameters double almost annually. Decades may bring clouds of μm -thin sunlight sails in space in of great computing and physical (light beam) power. Centuries may turn them into a Dyson Sphere enveloping the solar system. Still, the power of such an ultimate computer is limited by the number of photons the Sun emits per second: $Y \sim 2^{2^{7.3}} = X^2$. Giga-years may turn much of the known universe into a computer, but its might is still limited by its total entropy $2^{2^{8.3}} = Y^2$.

Faster PPM Simulations. Parallel Bubble-Sort on CA or Merge-Sort on sequential FCM take nearly linear time. Parallel FCM can do much better [Ofman 65]. It represents and updates pointer graphs as the above TM. All steps are straightforward to do locally in parallel polylog time except sorting of pointers. Sophisticated sorting networks sort arbitrary arrays of n integers in $O(\log n)$ parallel steps.

We need only a simpler polylog method. Merge-Sort splits an array of two or more entries in two halves and sorts each recursively. Batcher-Merge combines two sorted lists in $O(\log n)$ steps as follows.

Batcher-Merge. A *bitonic cycle* is the combination of two sorted arrays (one may be shorter), connected by max-to-max and min-to-min entries. Entries in a contiguous half (*high-half*) of the cycle are \geq than all entries in the other (*low*) half. Each half (with its ends connected) forms a bitonic cycle itself.

Shuffle Exchange graph links nodes in a 2^k -nodes array to their *flips* and *shifts*. The flip flips the highest bit of a node's address; the *shift* cycle-shifts that bit to the end.

We merge-sort two sorted arrays given as a bitonic cycle on such a graph as follows. Comparing each entry with its flip (half-a-cycle away), and switching if wrongly ordered, fits the high and low halves into respectively the first and last halves of the array. (This rotates the array, and then its left and right halves.) We do so for each half recursively (decrementing k via shift edges).

2 Universal Algorithm; Diagonal Results

2.1 Universal Turing Machine

The first computers were hardware-programmable. To change the function computed, one had to reconnect the wires or even build a new computer. John von Neumann suggested using Turing's Universal Algorithm. The function computed is then specified by giving its description (program) as part of the input instead of changing the hardware. This was a radical idea, since universal functions do not exist in classical math (as we will see in Sec. 2.2).

Let C be the class of all TM-computable functions: total (defined for all inputs) and partial (which may diverge). Surprisingly, there is a universal function u in C . It simulates every Turing Machine M with a given tape alphabet (but different head alphabets). Let M compute $f \in C$ in time T and space S . u uses a program m of length c listing the commands and initial head state of M .

Then $u(mx)$ simulates $M(x)$. It operates in cycles, each simulating one step of M in time c^2 and space $S+c$. Let the tape of M after i steps be $t_i = l_i s_i r_i$, where s_i is the state of the pair of active cells. After i cycles u has $t_i = l_i m s_i r_i$ on its tape. It looks up m to find the command corresponding to s_i , and modifies t_i accordingly. When $M(x)$ halts, $u(mx)$ erases the (penciled) m and halts too. Universal Multi-head TM works similarly but can also determine in time $O(t(x))$ whether it halts in t steps (given $x, t(x)$).

Exercise. Design a universal multi-head TM with a constant factor volume overhead c^2 .

Hint: When heads split or merge in the first cell, the room u needs for their programs creates sparse or dense content regions that propagate right (sparse faster).

We now describe in detail a simpler but slower universal [Ikeno 58] TM U . It simulates any other TM M that uses only 0, 1 bits on the tape, lacking even the blank that usually marks the end of input. (Larger alphabets can be encoded as bit strings.)

U has 6 symbols + 11 head states. Its transition table at the right shows the states and tape symbols only when changed, except that primes always show. The head is on the tape: lower case states look left, upper case – right. Input calls, halt, etc. are special states for M ; for U they are shown as “A/B” or “=”. U works in cycles, each simulating one transition of M . The tape consist of 0/1 segments, each preceded with a *. Some symbols are primed.

	1	0	*	1'	0'	*'
A	c	c	e0			
B	C	C	e1			
C,c	a*	b*	C			f
D	d'	–				e'
E	=	–	'	'	'	e'
d	'	'		'	'	D
a	'	'	'		b'	D
b	'	'	'	a'	C	E'
e	'	'	'	B	A	A/B
f	'	'		=	C	E'

The rightmost part of one or two segments is always a copy of M 's (infinite to the right) tape. The other segments describe one transition command each: $(s, \beta) \rightarrow (s', \beta', \delta)$ for M to change head state s to s' , tape bit β to β' and turn left or right. The commands segments are sorted in order of (s, β) and never change, except for priming. Each transition is represented as $*S\delta\beta$, where β is the bit to write, δ the direction $R=0/L=1$ to turn. S points to the next state represented as 1^k , if it is k segments to the left, or 0^k (if to the right). Each cycle starts with U 's head in state C or c , located at the site of M 's head. Primed are the digits of S in the prior command and all digits to their left. An example of the configuration: $*'0'0'0'1'0' *' 0'0'0'0'01 * 011 * \dots * 00$ head $00\dots$. The leftward head in the leftmost cell halts.

U first reads the bit of an M 's cell changing the state from C or c to b/a , puts $*$ there, moves left to the primed state segment S , finds from it the command segment and moves there. It does this by repeatedly priming nearest unprimed $*$ and 1s of S (or unpriming 0s) while alternating the states f/C or d/D .

When S is exhausted, the target segment $\|S\| + \beta$ stars away is reached. Then U reads (changing state from e to A or B) the rightmost symbol β' of the command, copies it at the $*$ in the M area, goes back, reads the next symbol δ , returns to the just overwritten (and first unprimed) cell of M area and turns left or right.

In CA format, M and U have in each cell three standard bits: present δ and previous δ' pointer directions and a “content” bit to store M 's symbol. In addition U needs just one “trit” of its own!

2.2 Uncomputability; Goedel Theorem

Universal and Complete Functions.

Notations: Let us choose a special mark (not of form aba) and after its k -th occurrence, break any string x into $\text{Prefix}_k(x)$ and $\text{Suffix}_k(x)$. Let $f^+(x)$ be $f(\text{Prefix}_k(x) x)$ and $f^-(x)$ be $f(\text{Suffix}_k(x))$. We say u ***simulates*** f iff for some $p = \text{Prefix}_k(p)$ and all s , $u(ps) = f(s)$. The prefix can be intuitively viewed as a program which simulating function u applies to the suffix (input). We also use a symmetric variant of relation “ k -simulate” which makes some proofs easier. Namely, u ***k -intersects*** f iff $u(ps) = f(ps)$ for some $\text{prefix}_k p$ and all s . E.g., length preserving functions can intersect but cannot simulate one another.

We call ***universal*** for a class F , any u which k -simulates all functions in F for a fixed k . When F contains f^-, f^+ for each $f \in F$, universality is equivalent to (or implies, if only $f^+ \in F$) ***completeness***: u k -intersects all $f \in F$. Indeed, u k -simulates f iff it k -intersects f^- ; u $2k$ -intersects f if it k -simulates f^+ .

Exercise: Describe **explicitly** a function, **complete** for the class of all *linear* (e.g., $5x$ or $23x$) functions.

A ***negation*** of a (partial or total) function f is the total predicate $\neg f$ yielding 1 iff $f(x)=0$ and 0 otherwise. Obviously, no closed under \neg class of functions contains a complete one.

So, the class of all (computable or not) predicates contains no universal one. This is the prominent Cantor Theorem that the set of all sets of strings (as well as the sets of all functions, reals etc.) is not countable.

Goedel’s Theorem.

Formal proof systems are computable functions $A(P)$ which check if P is an acceptable proof and output the proven statement. $\vdash s$ means $s = A(P)$ for some P . A is ***rich*** iff it allows computable translations s_x of statements “ $u_2(x) = 0$ ”, provable whenever true, and refutable ($\vdash \neg s_x$), whenever $u_2(x) = 1$.

A is ***consistent*** iff **at most** one of any such pair $s_x, \neg s_x$ is provable, and ***complete*** iff **at least** one of them always (even when $u(x)$ diverges) is. There is no complete function among the **total** computable ones, as this class is closed under negation. So the universal in C function u (and $u_2 = (u \bmod 2)$) has no total computable extensions. Thus, rich consistent and complete formal systems cannot exist, since they would provide an obvious total extension u_A of u_2 (by exhaustive search for P to prove or refute s_x).

This is the famous Goedel’s Theorem – one of the shocking surprises of the 20th century science. (Here A is any extension of the formal Peano Arithmetic; we skip the details of its formalization and proof of richness.)¹

Computable Functions. Another byproduct is that the Halting (of u) Problem would yield a total extension of u and, thus, is not computable. This leads to many other uncomputability results. Another source is an elegant ***Fixed Point*** Theorem by S. Kleene: any total computable A transforming programs (prefixes) p maps some p into equivalent ones (computing the same functions). Indeed, the complete $u(ps)$ intersects computable $u(A(p)s)$. This implies Rice theorem: the only total computable invariant (i.e. equal on equivalent programs) property of programs is constant (**exercise**).

Computable (partial and total) functions are also called recursive (due to an alternative definition). Their ranges (and, equivalently, domains) are called (computably) ***enumerable*** or ***c.e.*** sets. A c.e. set with a c.e. complement is called computable or ***decidable***. A function f is computable iff its graph G is c.e. If f is also total, G is decidable. Each infinite c.e. set is the range of an injective total computable function (“enumerating” it, hence the name c.e.).

We can reduce membership problem of a set A to one of a set B by finding a computable function f s.t. $x \in A \iff f(x) \in B$. Then A is called ***m-*** (or ***many-to-1-***) ***reducible*** to B . More complex ***Turing*** reduction are by algorithms which, starting from input x , interact with B by generating strings s and receiving answers to $s \in ? B$ questions. Eventually this stops and tells if $x \in A$. C.e. sets (like Halting Problem) to which all c.e. sets m-reduce are called c.e.-complete. This (and, thus, undecidability) can be shown by reducing the Halting Problem to them. Ju.Matijasevich so proved completeness of Diophantine Equations Problem: find if a given multivariate integer polynomial of degree 4 has integer roots. Such ideas are broadly used in Theory of Algorithms and should be learned from any standard text, e.g., [Rogers 67].

¹A closer look at this proof reveals another famous Goedel theorem: Consistency C_n of A (expressible in rich A , as divergence of the search for contradictions) is itself an unprovable example. Indeed, u_2 intersects $1 - u_A$ for some prefix a . C_n implies that u_A extends u_2 and, thus, $u_2(a), u_A(a)$ both diverge. So, $C_n \Rightarrow \neg s_a$. Peano Arithmetic can formalize this proof, thus $\vdash C_n \Rightarrow \vdash \neg s_a$. But $\vdash \neg s_a$ implies $u_A(a)$ converges, so $\vdash C_n$ contradicts C_n . So, consistency of A is provable in A if and only if false !

2.3 Intractability; Compression and Speed-up Theorems

The *t-restriction* u_t of u aborts and outputs 1 if $u(x)$ does not halt within $t(x)$ steps, i.e. u_t computes the *t-Bounded Halting Problem (t-BHP)*. It remains complete for the \neg -closed class of functions computable in time $o(t(x))$. ($o(1)$ and padding of p absorb $O(\|p\|^2)$ overhead.) So, u_t is not in the class, i.e. cannot be computed in time $o(t(x))$ [Tseitin 56]. (And neither can be any function agreeing with *t-BHP* on a *dense* (i.e. having strings with each prefix) subset.) E.g. $2^{\|x\|}$ -BHP requires exponential time. However for some trivial input programs the BHP can obviously be answered by a fast algorithm. The following theorem provides another function $P_f(x)$ (which can be made a predicate) which has only a finite number of such trivial inputs.

We state the theorem for the Multi-Head Turing Machine volume of computation. It can be reformulated in terms of Pointer Machine time or regular Turing Machine space (or, with smaller accuracy, time).

Definition: A function f is called *constructible* iff $\Theta(f)$ can be computed in volume $O(f)$.

Example: $f(x)=2^{\|x\|}$ is constructible, as $V_f(x) = O(\|x\|) \ll 2^{\|x\|}$.

Compression Theorem [Rabin 59]. For each constructible function f , a function P_f exists such that for all functions t , the following two statements are equivalent:

1. P_f is computable in volume $\Theta(t)$.
2. t is constructible and $f = O(t)$.

Speed-up Theorem [Blum 67]. There exists a total computable predicate P such that if any algorithm computes $P(x)$ in volume $t(x)$ then $t(x) > \|x\| - O(1)$ and some other algorithm does it in volume $O(\log t(x))$.

Though stated here for exponential speed-up, this theorem holds for any computable unbounded monotone function in place of log. So, not even remotely optimal algorithm computing P exists. This case is in a sharp contrast with Compression Theorem cases.

The general case. So, the complexity of some predicates P cannot be characterized by a single constructible function f . However, Compression Theorem remains true for any *semi-constructible* f , i.e. computable in volume b as $f^{(b)}(x)$ given any its upper bound $b > f(x)$. In this form it is fully general (every computable function P satisfies the Compression Theorem with an appropriate semi-constructible f). There is no contradiction with Blum's Speed-up, since the bound f (not constructible itself) cannot be reached.

Proof. Let *Timed Kolmogorov Complexity* $\text{KT}(i|x)$ of i given x be the least $\lceil 3\|p\| + \log v \rceil$ for programs p generating i from x in volume $v = V_{u(p,x)}$. Let $P_f(x)$ be the least i , with $\text{KT}(i|x) > \log f$. f and P_f can be computed in volume $O(b)$ if $b > f$: by generating all i of low KT , sorting them, and taking the first missing. It satisfies the Theorem, as computing $i = P_f(x)$ faster than f violates the complexity bound defining it. \square

(Some extra efforts can make P Boolean valued.)

Moreover, any **computable** bound f has an equivalent semi-constructible bound g with the same constructible bounds t above either of them: $t = \Omega(f) \iff t = \Omega(g)$. (For (1) or (2) of the Theorem (with computable P, f), the set M of constructible t satisfying it is *directed*, i.e. for each t_1, t_2 in M includes all $t > \min(t_1, t_2)/2$. Their programs form a Σ_2^0 class (i.e. of form $\{p : \exists c \forall d A(p, c, d)\}$). Such M are shown to be the $\Omega(f)$ for some semi-constructible f .) See a review in [Seiferas, Meyer 95].

3 Games; Alternation; Exhaustive Search; Time vs. Space

3.1 How to Win

In this section we consider a more interesting *provably* intractable problem: playing two players zero sum games with full information. We will see that even some simple games yield to no much faster algorithms than exhaustive search of all possible configurations. The rules of an *n-player game* G are set by families f, v of **information** and **value** functions and a **transition rule** g . Players $i \in I$ at each step participate in transforming a configuration (game position) $x \in C$ into the new configuration $g(x, m)$, $m : I \rightarrow M$ by choosing moves $m_i = m(i)$ based only on their knowledge $f_i(x)$ of x . The game ends upon reaching a **terminal** configurations $t \in T \subset C$. Then $v_i(t)$ is the loss or gain of the i -th player.

Our games will have **zero sum** $\sum v_i(t) = 0$ and **full information**: $f_i(x) = x$, $g(x, m) = g'(x, m_{a(x)})$: $a(x)$ indicates the single **active** player. We consider binary, two-players, no-draw games, taking $0 \notin C \subset \mathbb{Z}$, $M \subset \mathbb{Z}$, $T = I = \{\pm 1\}$, $a(x) = x/|x|$ (its sign), $v_i(t) = a(t)i$, $|g(x, m)| < |x|$. (In our examples this “ $<|x|$ ” is by implicitly prepending configurations with a counter of remaining steps.)

An example of such games is chess. Examples of games without full information are card games, where only a part $f_i(x)$ (player’s own hand) of the position x is known. Each player may have a **strategy** providing a move for each position. A strategy S is **winning** at x if starting at a position x it guarantees victory whatever the opponent does, even if he knows S . We can extend v_1 on T to V on all positions with a winning strategy for one side so that $a(x)V(x) = \sup_m \{a(x)V(g(x, m))\}$. (sup $\{\}$ taken as -1 .)

Evaluating or **solving** a game, means computing V . This ability is close to the ability to find a good move in a modified game. Indeed, any game can be modified into an equivalent game with two moves: $M \subset \{0, 1\}$ by breaking a string move into several single bit-moves. Evaluating such games clearly suffices for choosing the right move. And vice versa: modify a game G into G' by adding a preliminary stage to it. At this stage the player A offers a starting position for G and her opponent B chooses which side to play. Then A may either start playing G or decrement the counter of allowed offers and give another one. The ability to find a good move in the preliminary stage means the ability to evaluate positions of G .

Theorem. *Each position of any full information game G has a winning strategy for one side.*

[Neumann, Morgenstern 44]

(This theorem fails for **partial** information games: either player may lose if the adversary knows his strategy. E.g.: 1. Blackjack (21); 2. Each player picks a bit; their equality determines the winner.)

Playing all strategies against each other can solve the game.

There are 2^n positions of length n , $(2^n)^{2^n} = 2^{n \times 2^n}$ strategies and $2^{n \times 2^{n+1}}$ pairs of them. For a 5-bit game that is 2^{320} . But the proof of this Theorem gives a much faster (yet still exponential time!) strategy.

Proof. Make the graph of all $\leq x$ positions and moves; Set $V \leftarrow 0$; reset $V \leftarrow v$ on T . Repeat until idle: If $V(x) = 0$, set $V(x) \leftarrow a(x) \sup_m \{a(x)V(g(x, m))\}$. Eventually $N \stackrel{\text{def}}{=} V^{-1}(0)$ gets empty since all nodes staying in N have a move to N , forming a cycle, impossible, as $|g(x, m)| < |x|$ in G keep decreasing. \square

Games may be categorized by the difficulty to compute g . We will consider only g computable in linear space $O(\|x\|)$. Then, the $2^{2\|x\|}$ possible moves can be computed in exponential time, say $2^{3\|x\|}$. The algorithm tries each move in each step. Thus, its total running time is $2^{3\|x\|+1}$: *extremely* slow (2^{313} for a 13-byte game) but still *much* faster than the previous (double exponential) algorithm.

Can the remaining exponent be eliminated, too? See the sec.3.2.

Exercise: the Nim Game. Consider 3 boxes, each with 3 matches: $\boxed{!!!} \boxed{!!!} \boxed{!!!}$. The players alternate turns taking any *positive* number of matches from a *single* box. The taker of the last match from table loses. Use the above algorithm to evaluate all positions and list the evaluations after each its cycle.

Exercise: Modify the chess game by giving one side the option to make an extra move out of turn during the first 10 moves. Prove that this side has a non-losing strategy.

3.2 Exponentially Hard Games

A simple example of a full information game is **Linear Chess**, played on a finite linear board. Each piece has a 1-byte type, including *loyalty* to one of two sides: W (weak) or S (shy), *gender* M/F and a 6-bit *rank*. All cells of the board are filled and all W's are always on the left of all S's. Changes occur only at the **active** border where W and S meet (and fight). The winner of a fight is determined by the following Gender Rules: Shy gets confused and loses if Weak is of different sex. Otherwise, Weak loses. The party of a winning piece A replaces the loser's piece B by its own piece C. The choice of C is restricted by the table of rules listing all allowed triples (ABC). We will see that this game *cannot* be solved in a *subexponential* time.

We first prove that (see [Chandra, Kozen, Stockmeyer 81]) for an artificial game.

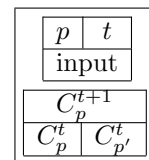
Then we reduce this **Halting Game** to Linear Chess showing that any fast algorithm to solve Linear Chess, could be used to solve Halting Game, thus requiring exponential time. For Exp-Time Completeness of regular (but $n \times n$) Chess, Go, Checkers see: [Fraenkel, Lichtenstein 81, Robson 83, 84].

ExpTime Complete Halting Game.

We use a universal Turing Machine u (defined as 1-pointer cellular automata) restricted to either diverge, or halt within $2^{\|x\|}$ steps by its head rolling off of the tape's left end, leaving a blank. This cannot be determined in $o(2^{\|x\|})$ steps. We will now convert this problem into the Halting Game:

The players are: W claiming $u(x)$ halts (and has winning strategy iff this is true);

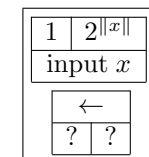
His opponent is S . The **board** has four parts: the C diagram, the input x to u , positive integers p (position) and t (time in the execution of $u(x)$) The diagram shows the states C_p^{t+1}, C_p^t of cell p at times $t+1, t$, and $C_{p'}^t$ pointed to by C_p^t . C include present and previous pointers directions; C^t may be replaced by "?".



Some board configurations are illegal: if (1) both $C_p, C_{p'}$ point away from each other, or (2) C^{t+1} differs from the result prescribed by the transition rules for C^t , or (3) $t=1$, while $C_p^1 \neq x_p$. (At $t=1$, u has the input x on its tape at the left with the head in the initial state, followed by blanks at the right.)

The **Game Rules**: The starting configuration of the game is at the right.

W , in its moves, replaces the ?s with symbols claiming to reflect the state of cells p', p at step t of $u(x)$. S optionally replaces p with p' , then moves C_p to top C box, fills lower C boxes with ?s, and decrements t . The game can only check that compliance with "local" rules, not refer to past moves or to actual computation of $u(x)$.



Note that W may lie (i.e distort the actual computation of $u(x)$ in filling "?"), as long as he is compliant.

Strategy for W: If $u(x)$ does indeed halt, then the initial configuration is true to the computation of $u(x)$. Then W has an obvious (though hard to compute) winning strategy: just tell truly (and thus always consistently) what happens in the computation. W will win when $t=1$, so S cannot decrement it further.

Strategy for S: If the initial configuration is a lie, S can force W to lie all the way down to $t=1$ (when the lie is exposed). How is this done? If the upper box C_p^{t+1} of a legal configuration is false then the lower boxes $C_{p'}, C_p^t$ cannot both be true, since the rules of u determine C_p^{t+1} uniquely from them. If S correctly points the false C and brings it to the top on his move, then W is forced to keep on lying. At time $t=1$ the lie is illegal, as the configuration doesn't match the actual input string x . So, solving this game is to decide if the initial configuration is correct, i.e. $u(x)$ halts in $2^{\|x\|}$ steps: impossible in time $o(2^{\|x\|})$.

This Halting Game is artificial, still has a Halting Problem flavor, though it does not refer to exponents. We will now reduce it to a sleeker game – Linear Chess – to prove it exponentially hard, too.

3.3 Reductions; Non-Deterministic, Alternating TM; Time / Space

To reduce (see definition in sec. 2.2) Halting Game to Linear Chess, we introduce a few concepts.

A *non-deterministic* Turing Machine (NTM) is a TM that sometimes offers a (restricted) transition choice, made by a *driver*, a function (of the TM configuration) of unrestricted complexity. A deterministic (ordinary) TM M accepts a string x if $M(x)=\text{yes}$; an NTM M does if there exists a driver d s.t. $M_d(x)=\text{yes}$.

NTM represent single player games – puzzles – with a simple transition rule, e.g., Rubik’s Cube.

One can compute the winning strategy in exponential time by exhaustive search of all d .

Home Work: Prove or refute that all such games have P-time winning strategies.

Will get you “A” for the course, \$1,000,000 award and a senior faculty rank at a school of your choice.

Alternating TM (ATM) are a variation of the NTM driven by two alternating drivers l, r .

A string is accepted iff $M_{l,r}(x)=\text{yes}$ for some l and all r . Our games could be viewed as ATM returning the result of the game in linear space but possibly exponential time. M prompts players l and r alternately to choose their moves (in several steps for multi-bit moves) and computes the resulting positions, until a winner emerges. Accepted strings describe winning (i.e. having a winning strategy) positions.

Linear Chess (LC) Simulation of TM-Games. We first represent the Halting Game as an ATM computation simulated by the Ikeno TM (Sec.2.1) (using the “A/B” command for players’ input).

It is viewed as an array of 1-pointer cellular automata: Weak cells as rightward, Shy leftward.

Upon termination, the TM head is set to move to the end of the tape, eliminating all loser pieces.

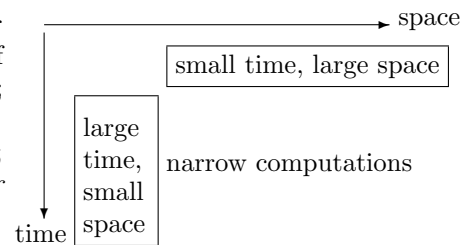
This is viewed as a game of *1d-Chess (1dC)*, a variant of LC, where a table, not the “Gender Rule”, determines the victorious piece, and not only the losing piece is replaced, but also the winning piece may change rank. The types are states of Ikeno TM showing Loyalty (pointer direction $d \in \{W, S\}$), gender δ (previous d), and 6/6/6/5 ranks (trit $t \in \{0, 1, *\}$ with ‘ bit p).

Exercise: Describe LC simulation of 1dC. **Hint:** Each transition of 1dC is simulated in several LC stages. Let L,R be the pieces active in 1dC. In odd stages L (in even stages R) turns pointer twice and changes gender. The last stage turns pointer once and possibly changes gender. In the first stage L appends its rank with R’s p bit. All other stages replace old 1dC rank with the new one. R appends its old t bit (only if $t \neq *$) to its new rank. Subsequent stages drop both old bits, marking L instead if it is the new 1dC head.

Up to 4 more stages are used to fix any mismatch with 1dC new d, δ bits.

Space-Time Trade-off. *Deterministic* linear space computations are games where each position has a single (and easy to compute) move. A big open problem: what time is required to compute their outcome.

We know no general subexponential upper or superlinear lower bounds. Recall that on a parallel machine *time* is the number of steps until the last processor halts, *space* is the memory size used; *volume* is the combined number of steps of all processors. “*Small*” will refer to values bounded by a polynomial of the input length; “*large*” to exponential. Call computations *narrow* if *either* time *or* space are small, and *compact* if both (thus, volume too) are.



An open question: Do all exponential volume algorithms (e.g., one solving Linear Chess) allow an equivalent *narrow* computation? **Alternatively, can every *narrow* computation be converted into a compact one?** This is equivalent to the existence of a P-time algorithm for solving any *fast* game, i.e. a game with a P-time transition rule and a move counter with a **small bound**.

The sec. 3.1 algorithm can run in parallel P-time for such games. Converse, too, is similar to Halting Game.

[Stockmeyer, Meyer 73] solve compact games in P-space: With $M \subset \{0, 1\}$ run depth-first search on the tree of all games (sequences of moves). On exiting each node it is marked as the active player’s win if some move leads to a child so marked; else as his opponent’s. Children’s marks are then erased.

Conversely, compact games can simulate any k -space algorithms (even non-deterministic). Player A declares a state achievable in 2^k steps. If he lies, player B asks him to declare the state in the middle of that time interval, etc. So by a k -step binary search A is caught on a mismatch of states at two adjacent times.

Thus, while narrow alternating computations (general games) correspond to large deterministic ones, the compact alternating computations (fast games) correspond to either type of narrow deterministic ones. This has some flavor of trade-offs such as saving time at the expense of space in dynamic programming.

Part II

Mysteries

4 Nondeterminism; Inverting Functions; Reductions

4.1 An Example of a Narrow Computation: Inverting a Function

We now enter Terra Incognita with deterministic computations extended by tools like random choices, non-deterministic guesses, etc., the power of which is completely unknown.

Yet fascinating discoveries were made there, a glimpse of which we will get here.

Consider a P-time function F . Assume $\|F(x)\| = \|x\|$, for convenience (often $\|F(x)\| = \|x\|^{\Theta(1)}$ suffices). Inverting F means given y , finding some $x \in F^{-1}(y)$, i.e. such that $F(x) = y$.

To invert F we may check $F(x) = y$ for all possible x . This takes small *space* but absolutely infeasible $\Omega(2^{\|x\|})$ *time*. No method is currently proven much better in the worst case. Nor could we prove some inversion problems to require more time than F does. This is the sad present state of Computer Science !

An Example: Factoring. Let $F(x_1, x_2) = x_1 x_2$ be the product of integers. For simplicity, assume x_1, x_2 are primes. A fast algorithm in sec. 5.1 determines if an integer is prime. If not, no factor is given, only its existence. To invert F means to factor $F(x)$. The density of n -bit primes is $\Theta(1)/n$. So, factoring by exhaustive search takes *exponential* time! In fact, even the best known algorithms for this ancient problem run in time about $2^{\sqrt{\|x\|}}$, despite centuries of efforts by most brilliant people. The task is commonly believed infeasible and the security of many popular cryptographic schemes depends on this *unproven* faith.

One-Way Functions: $F : x \rightarrow y$ are those easy to compute but hard to invert ($y \mapsto x$) for most x . Even their existence is a sort of religious belief in Computer Theory. It is unproven, though many functions *seem* to be one-way. Some functions, however, are proven to be one-way **iff** any one-way functions **exist**. Many theories and applications are based on this hypothetical existence.

Search and NP Problems.

Let us compare inversion problems with another type – search problems – solving P-time computable relations F : given x , (a) find w (called *witness*) s.t. $F(x, w), \|w\| = \|x\|$ (constructive problem) or (b) just decide if such w exist (decision problem). Any inversion problem is a search problem and any search problem can be restated as an inversion problem. E.g., finding Hamiltonian cycles C in a graph G , is to invert f where $f(G, C)$ is $(G, 0 \dots 0)$ if C is in fact a Hamiltonian cycle of G , or $(0 \dots 0)$ if C is not. Similarly any search problem can be reduced to another one equivalent to its decision version. For instance, factoring x reduces to bounded factoring: given x, b find $p|x$ such that $1 < p < b$. (Decisions yield construction by binary search.)

Exercise: Generalize the above to reduce any search problem to an inversion and to a decision problems.

(Search problems are games with P-time transition rules and one move duration. A great hierarchy of problems results from allowing more moves and/or other complexity bounds for transition rules.)

The problem's *language* is the set of instances it accepts: the range of f for inversion problems, $\{x : \exists w F(x, w)\}$ for search. An **NP language** is the set of inputs acceptable by a P-time **non-deterministic** TM (sec. 3.3). All three language classes – search, inversion and NP – coincide. (NP \iff search is trivial.)

Interestingly, *P-space* bounded deterministic and non-deterministic TMs have equal power [Savitch 70], as we saw in Sec.3.3.

4.2 Complexity of NP Problems

Nobody knows if *all* search (or inversion, NP, etc.) problems are solvable in P-time (said being in P). They all yield to exponential time but no better bounds are known. This question (P=?NP) is probably the most famous one in Computer Science. Many problems seem hopeless to solve, while similar or slightly modified problems yielded to (often extraordinary) efforts. Examples:

1. Linear Programming: Given an integer $n \times m$ matrix A and vector b , find a rational vector x with $Ax < b$. Note, if n and entries in A have $\leq k$ -bits and x exists then an $O(nk)$ -bit x exists, too. Solution: The Dantzig's **Simplex** algorithm finds x quickly for many A . Some A , however, take exponential time. After frustratingly long efforts, a worst case P-time Ellipsoid Algorithm was finally found in [Yudin and A.S. Nemirovsky 76].

2. Primality test: Determine if a given integer p has a factor? Solution: Trying all $2^{\|p\|}$ possible integer factors of p is infeasible, but more sophisticated algorithms, run fast (see section 5.1).

3. Graph Isomorphism: Are two given graphs G_1, G_2 , isomorphic? I.e., can the vertices of G_1 be re-numbered to make it equal to G_2 ? Solution: Checking all $n!$ enumerations of vertices is infeasible (exceeds the number of atoms in the known Universe for $n=100$). [Luks 80] found an $O(n^d)$ steps algorithm where d is the degree. This is a P-time for $d = O(1)$.

4. Independent Edges (Matching): Find a given number of independent (i.e., not sharing nodes) edges in a given graph. Solution: Max flow algorithm solves a bipartite graph case. A more sophisticated algorithm by J.Edmonds solves the general case.

Many other problems have been battled for decades or centuries and no P-time solution has been found. Even modifications of the previous four examples have no known answers:

1. Linear Programming: All known solutions produce rational x .
No reasonable algorithm is known to find integer x .
2. Factoring: Given an integer, find a factor. Centuries of quest for fast algorithm were unsuccessful.
Takes $n^{\sqrt{n}}$ time so far.
3. Sub-graph isomorphism: find if a graph is isomorphic to a part of another.
No P-time solutions known, even for $O(1)$ -degrees.
4. Independent nodes: Find k independent (i.e., not sharing edges) nodes in a given graph.
No P-time solution is known.

Exercise: Restate the above problems as inverting easily computable functions.

We proved the exponential complexity of Linear Chess and some other games. None of the above or any other search/inversion/NP problem, however, have been proven to require super-polynomial time. When, therefore, do we stop looking for an efficient solution?

NP-Completeness theory is an attempt to answer this question.

See results by S.Cook, R.Karp, L.Levin, and others surveyed in [Garey, Johnson 79, Trakhtenbrot 84].

A P-time function r reduces one NP-predicate $p_1(x)$ to $p_2(x)$ iff $p_1(x) = p_2(r(x))$, for all x . p_2 is NP-complete if *all* NP problems can be so reduced to it. Thus, NP-complete problems are the hardest in the worst-case in the class NP. Any P-time solution for one of them would solve all other NP (or inversion, or search) problems. This may suggest giving up on fast algorithms for them.

Faced with problem's NP-completeness we may try to restate it, find a similar one which is easier (possibly with additional tools) but still gives the information we really want. Relaxing factoring to testing primality (see Sec. 5.1) is such a case. Now we proceed with an example of NP-completeness.

4.3 An NP-Complete Problem: Tiling

A tile is one of 26^4 unit squares with a Latin letter at each corner. Two tiles may be placed next to each other if the letters on the shared side match. (See an example at the right.)

a	x	x	c
m	r	r	z
m	r	r	z
n	s	s	z

Tiling Problem. Find a tiled square, given its first row and the list of tiles allowed to use.

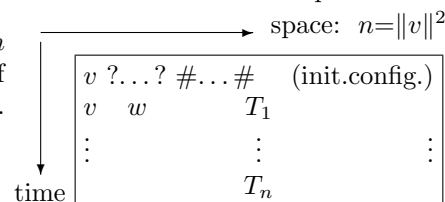
We will now reduce all search problems to that one, starting with some “standard” NP problem. An obvious candidate is existence of w with $U(v, w)$, where the universal Turing Machine U simulates $F(x, w)$ for $v = px$. But U has no P-time limit, so we must impose one on it. How to make such fixed degree polynomial sufficient to simulate any polynomial (even of higher degree) time?

Padding Argument. Let $u(v, w)$ for $v=00\dots 01px$ simulate $\|v\|^2$ steps of $U(px, w)=F(x, w)$. If the 0^* *padding* in v is long enough, u has time to simulate F , even though u runs in quadratic time, while F 's time limit may be, say, cube (of a shorter string xw).

So an NP problem $F(x, w)$ is reduced to $u(v, w)$ by mapping instances x to $v=0^*1px$, with $\|0^*\|$ determined by F 's time limit. So, if some NP problem *cannot* be solved in P-time then neither can be $\exists?w : u(v, w)$. And if the latter *HAS* a P-time solution then so have *all* search problems. (Which of these alternatives is true we do not know.) It remains to reduce the search problem u to Tiling.

The Reduction. $u(v, w)$ is run by a TM represented as an array of 1-pointer cellular automata that runs for $\|v\|^2$ steps and stops if w does NOT solve the relation F . Otherwise it enters an infinite loop.

An instance x has a solution iff $u(v, w)$ runs forever for some w . Set n to u 's time/space $\|v\|^2$. In the space-time diagram of computation of $u(v, w)$, each row represents configuration of u at the respective time. the second step fills in the solution w below a special symbol “?”. A faulty table may reflect no actual computation.



Then it must have 4 cells sharing a corner impossible to appear in the computation of u on any input.

Proof. As the input v and the guessed solution w are the same in both the right and wrong tables, the first 2 rows agree. The third row starts the actual computation. In the first mismatching row a transition of some cell from the previous row is wrong. This is clear from the state in both rows of this cell and the cell it points to, making an impossible combination of four cells sharing a corner.

For a given x , the existence of w satisfying $F(x, w)$ is equivalent to the existence of a table with the prescribed first row, no halting state, and permissible patterns of each 4 adjacent squares (cells).

We now convert the table into the **Tiling Problem**: Cut each cell into 4 parts by a vertical and a horizontal lines through its center and copy cell's content in each part. Combine into a tile the 4 parts sharing a corner of 4 cells. The cells at the right are separated by “—”; the tiles by dashes; If these cells are permissible in the table, then so is the respective tile. Thus any matching tiled square solves the respective NP problem. This reduces inversion of u , and thus every NP problem, to Tiling.

u	u	v	v
u	u	v	v
v	v	x	x
v	v	x	x

Exercise: Find a P-time algorithm for $n \times \log n$ Tiling Problem.

5 Probability in Computing

5.1 A Monte-Carlo Primality Tester

The factoring problem seems very hard. But to test if a number has factors turns out to be much easier than to find them. This is crucial for many applications, such as, e.g., security tools. It also helps to supply the computer with a coin-flipping device. We now consider a Monte Carlo algorithm, i.e. one that with high probability rejects any composite number, but never a prime. See: [Miller 76, Solovay, Strassen 77, Rabin 80].

Residue Arithmetic. $p|x$ means p divides x . $x \equiv y \pmod{p}$ means $p|(x-y)$. $y = (x \bmod p)$ denotes the residue of x when divided by p , i.e. $x \equiv y \in [0, p-1]$. Residues can be added, multiplied and subtracted with the result put back in the range $[0, p-1]$ via shifting by an appropriate multiple of p . E.g., $-x$ means $p-x$ for residues mod p . $\pm x$ means either x or $-x$.

The Euclidean Algorithm finds $\gcd(x, y)$ – the greatest (multiple of any other) common divisor of x, y : $\gcd(x, 0)=x$; $\gcd(x, y) = \gcd(y, (x \bmod y))$ for $y>0$. By induction, $d = \gcd(x, y) = A*x - B*y$, where integers $A=(d/x \bmod y)$ and $B=-(d/y \bmod x)$ are produced as a byproduct of that algorithm.

It allows division $(\bmod p)$ by any r **coprime** with p (i.e. $\gcd(r, p) = 1$), and operations $+, -, *, /$ obey all usual arithmetical laws. We also need to compute $(x^q \bmod p)$ in polynomial time.

We cannot do $q>2^{\|q\|}$ multiplications. Instead we compute all $x_i=(x_{i-1}^2 \bmod p)=(x^{2^i} \bmod p)$, $i<\|q\|$. Then we take q in binary, i.e. as a sum of powers of 2 and multiply $\bmod p$ the needed x_i 's.

Fermat Test. The Little Fermat Theorem for each prime $p \nmid x$ says: $x^{(p-1)} \equiv 1 \pmod{p}$. Indeed, $i \rightarrow (ix \bmod p)$ is a permutation of $[1, p-1]$. So, $1 \equiv (\prod_{i<p} (ix)) / (\prod_{i<p} i) \equiv x^{p-1} \pmod{p}$. This test rejects typical composite p . The other composite (**Carmichael**) numbers are actually factored by the following tests.

Square Root Test. For each y and prime p , $x^2 \equiv y \pmod{p}$ has at most one pair of solutions $\pm x$.

Proof. Let x, x' be two solutions: $y \equiv x^2 \equiv x'^2 \pmod{p}$. Then $p | x^2 - x'^2 = (x-x')(x+x')$. Thus $p|(x-x')$ or $p|(x+x')$, making $x \equiv \pm x'$, if p is prime. Otherwise p is composite, and $\gcd(p, x+x')$ *even gives its factor*.

Random Choice. We say d **kills** \mathbb{Z}_p^* if $x^d \equiv 1 \pmod{p}$ for all x (in \mathbb{Z}_p^* , i.e. coprime with p).

If $x^d \not\equiv 1$, then for all y either $y^d \not\equiv 1$ or $(xy)^d \not\equiv 1$. Same with $x^d \not\equiv \pm 1 \pmod{p}$.

So existence of a single such x implies the same for **most** of randomly chosen y .

Miller-Rabin Test T_x factors a composite p given d that kills \mathbb{Z}_p^* . If $d = p-1$ does not, then Fermat Test confirms p is composite. If it does, let $p-1=2^k q$, with odd q . T_x sets $x_0=(x^q \bmod p)$, $x_i = (x_{i-1}^2 \bmod p) = (x^{2^i q} \bmod p)$, $i \leq k$. $x_k=1$. If $x_0=1$, or one of x_i is -1 , T_x gives up for this x .

Otherwise $x_i \not\equiv \pm 1$ for some $i < k$, while $x_i^2 \equiv x_{i+1} \equiv 1$, and the Square Root Test factors p . Now, T succeeds with some (thus most!) $x \in \mathbb{Z}_p^*$. If $p=a^{i+1}$, $i>0$, then $(1+a^i)^{p-1} = 1+a^i(p-1)+a^{2i}c \equiv 1-a^i \not\equiv 1 \pmod{p}$. Else, $p=ab$, $\gcd(a, b)>1$: Take the **greatest** i such that $2^i q$ does not kill \mathbb{Z}_p^* . It exists (as $(-1)^q \equiv -1$ for odd q) and has $x_i \not\equiv 1 \equiv (x_i)^2 \pmod{p}$.

Then $x'=1+b(1/b \bmod a)(x-1) \equiv 1 \equiv x'_i \pmod{b}$, while $x'_i \equiv x_i \not\equiv 1 \pmod{a}$. So, $x'_i \not\equiv \pm 1 \pmod{p}$.

5.2 Randomized Algorithms and Random Inputs

Las-Vegas algorithms, unlike Monte-Carlo, never answer wrongly. Unlucky coin-flips just make them run longer than expected. Quick-Sort is a simple example. It is about as fast as deterministic sorters, but popular due to its simplicity. It sorts an array by choosing a random *pivot* in it, splitting the rest of it in two by comparing with the pivot, and calling itself recursively on each half.

For easy reference, rename the entries with their positions $1, \dots, n$ in the *sorted output* (no effect on the algorithm). Denote $t(i)$ the (random) time i is chosen as a pivot. Then i will ever be compared with j iff either $t(i)$ or $t(j)$ is the smallest among $t(i), \dots, t(j)$. This has 2 out of $|j-i|+1$ chances. So, the expected number of comparisons is $\sum_{i,j>i} 2/(1+j-i) = -4n + (n+1) \sum_{k=1}^n 2/k = 2n(\ln n - O(1))$. Note, that the expectation of the sum of variables is the sum of their expectations (not true, say, for product).

The above Monte-Carlo and Las-Vegas algorithms flip random coins themselves. (Or use **pseudo-random generators** instead, in hope, rarely supported by proofs, that their outputs have all the statistical properties of truly random coin flips used in the analysis.) These scenarios should not be confused with analysis of algorithms when inputs are chosen randomly with uniform distribution.

Random Inputs. Consider an example: Someone is interested in knowing whether or not certain graphs contain Hamiltonian Cycles. He offers graphs G and pays \$100 if we show either that G has or has not one. The problem is NP-Complete, so it likely is very hard for *some*, but not necessarily for *most* graphs. In fact, if our patron chooses G uniformly, a fast algorithm can earn us the \$100 *most of the time* ! Let the graphs have n nodes, $d < n(\ln n)/2$ edges, and be equally likely. Then use the following (deterministic) algorithm:

Output “**No** Hamiltonian Cycles” and collect the \$100, if G has isolated nodes. Otherwise, pass on that graph and the money. Now, how often do we get our \$100 ?

The probability that a given node A of the graph is isolated is $(1 - 1/n)^d > (1 - O(1/n))/\sqrt{n}$. The probability that *none* of them are (and we lose our \$100) is $O((1 - 1/\sqrt{n})^n) = O(1)/e^{\sqrt{n}}$, vanishing fast. Similar calculations can be made whenever $r = \lim d/(n \ln n) < 1$.

If $r > 1$, other fast algorithms can find a Hamiltonian Cycle. See: [Johnson 84, Karp 76, Gurevich 85].

(See also [Levin Venkatesan 18] for a proof that another graph problem is NP-complete even on average.)

How do this HC algorithm and the above primality test differ? The primality test works for *all* instances. It tosses its own coin and can repeat it for a more reliable answer. The HC algorithms only work for *most* instances (with isolated nodes or generic HC). In the HC case, we trust the customer’s honest random choices. If he cheats, producing rare graphs often, the analysis breaks down.

Randomness comes into computing in great many very different ways, we can only have a glimpse at few examples. Here, one more: **Symmetry Breaking** (avoiding conflicts for shared resources).

Dining Philosophers. They sit at a circular table. Between each pair is either a knife or a fork, alternating. Neighbors must share the utensils, cannot eat at the same time. How can they complete the dinner while all following the same procedure with no central organizer? If two diners grab at the same utensil, neither succeeds. They could each flip a coin, and sit still if it comes up heads, otherwise try to grab the utensils. If they repeat this procedure enough times, most likely each philosopher will soon get a turn for both knife and fork without interference.

5.3 Arithmetization: One-Player Games with Randomized Transition

In section 3, to win games, the player (call him Merlin), must beat another almighty wizard (say, Lady of the Lake). To make the challenge realistic we replace the Lady with a simple player, Arthur, who chooses at random the moves for both sides. Merlin, instead, must evaluate (by a formula) all moves. Any wrong formula of his will be clearly inconsistent with correct formulas for the results of most choices of the next move.

So, wrongness will persist. This trick, called *arithmetization*, was proposed in Noam Nisan's Fall-89 article widely distributed over email, and quickly used in a flood of follow-ups for matching various high complexity classes. We follow [Shamir 90, Fortnow, Lund 93]. The trick is based on that degree d polynomials agree on either $<d$ points or the whole field. We express boolean functions as low degree polynomials, and apply them to \mathbb{Z}_p -tokens (let us call them *bytes*) instead of bits. This reduces generic games to those in which any Merlin's strategy in any losing configuration has exponentially small chance to win.

The reduction holds for games with any (exponential, polynomial, etc.) limit on the counter c of the remaining moves. This c will be included implicitly in games configurations below.

Take the (ATM-complete) game of 1d-Chess (Sec.3.3), $g(m, x)$ with $x=x_1 \dots x_s$, $m, x_i \in \{0, 1\}$ being its transition rule. Configurations include x and a remaining moves counter $c \leq 2^s$. They are terminal if $c=0$, player x_1 winning. Intermediate configurations (m, x, y) have y claimed as a prefix of $g(m, x)$.

Let $t(m, x, y)$ be 1 if $y=g(m, x)$, else $t=0$. 1d-Chess is simple, so t can be expressed as a product of s multilinear $O(1)$ -sized terms, any variable shared by ≤ 2 terms. Thus t is a polynomial, quadratic in each m, x_i, y_i . Let $V_c(x)$ indicate if the active player has a strategy to win in c moves, i.e. $V_0(x) \stackrel{\text{df}}{=} x_1$, $V_{c+1}(x) = 1 - V_c(g(0, x))V_c(g(1, x))$, $V_c(g(m, x)) = V'_c(m, x, \{\})$, where $V'_c(m, x, y) \stackrel{\text{df}}{=} V_c(y)t(m, x, y)$ for $y = y_1 \dots y_s$ and $V'_c(m, x, y) \stackrel{\text{df}}{=} V'_c(m, x, y \circ 0) + V'_c(m, x, y \circ 1)$ for shorter y . (\circ denotes concatenation.) In our game G Merlin must prove his value v for V' . At the start he chooses a $2s$ -bit prime p . Configurations $X=(m, x, y, c, v)$ of G replace x_i, m, y_i, v bits with \mathbb{Z}_p -bytes. The polynomial V' retains the above inductive definition, thus is quadratic in each x_i, m , as $t(m, x, y)$ is. Then y_i have degree ≤ 4 in $V_c(y)$ and ≤ 6 in V' .

At his steps Merlin chooses a degree 6 polynomial $P(r)$. v must be $t(m, x, y)(1 - P(1)P(0))$ for s -byte y , or $P(0) + P(1)$ for shorter y . Arthur then takes a random $r \in \mathbb{Z}_p$, sets X to $(r, y, \{\}, c-1, P(r))$, or $(m, x, y \circ r, c, P(r))$, respectively. For X with a correct v Merlin's obvious winning strategy is to always provide the correct P . If v is wrong then $P(1), P(0)$ cannot both be right.

So P will differ from V , they can agree only on few (up to degree) points (i.e. on exponentially small fraction of random r). The wrong v will then propagate throughout the game, becoming obvious at $c=0$. Thus any Merlin strategy has exponentially small winning chance.

This reduction of Section 3 games yields a hierarchy of powers of Arthur-Merlin games and of computations mutually reducible with $V_c(x)$ of such games. The one-player games with randomized transition rule g running in space $O(\|x\|)$ are equivalent to exponential time deterministic computations. If instead the running time T of g combined for all steps is limited by a polynomial, then the games (called *interactive proofs, IP*) are equivalent to P-space deterministic computations.

An interesting twist comes in 1-move games with polylog T , too tiny to examine the initial configuration x and the move m . Yet, a little care not only removes this obstacle but achieves equivalence to NP. Namely, x, m are set in an error correcting code, and g is given $O(\log \|x\|)$ coin-flips and random access to the digits of x, m . Then the membership proof m is reliably verified by the polylog randomized g . See [Holographic proof] for details and references.

6 Randomness

6.1 Randomness and Complexity

Now let us look into the concept of Randomness itself. Intuitively, random sequences are those with the same properties as coin flips. But what *are* these properties? Kolmogorov resolved the issue with a robust definition of random sequences: those with no description noticeably shorter than their length. See survey and history in [Kolmogorov, V.A.Uspenskii 87, Li, Vitanyi 19].

Kolmogorov Complexity $K_A(x|y)$ of the string x given y is the length of the shortest program p which lets algorithm A transform y into x : $\min\{(\|p\|) : A(p, y) = x\}$. A Universal Algorithm U exists s.t, $K_U(x) \leq K_A(x) + O(1)$, for every algorithm A . This $O(1)$ is bounded by the length of the program U needs to simulate A . And as $|K_U(x|y) - K_{U'}(x|y)| = O(1)$ for any such U, U' , we select some U and abbreviate $K_U(x|y)$ as $K(x|y)$, or $K(x)$ for empty y . E.g.: for $A : x \rightarrow x$, $K_A(x) = \|x\|$, so $K(x) \leq K_A(x) + O(1) = \|x\| + O(1)$.

We cannot compute $K(x)$ by running all programs p of length $\leq \|x\| + O(1)$ to find the shortest one generating x , as some p diverge, and the halting problem is unsolvable. In fact, no algorithm can compute K or even any its lower bounds except $O(1)$. Consider the Berry Paradox expressed in the phrase: “*The smallest integer that cannot be uniquely and clearly defined by an English phrase of less than two hundred characters.*”

There are $< 128^{200}$ English phrases of < 200 characters. So there must be integers not expressible by such phrases and the smallest one among them. But isn't it described by the above phrase?

A similar argument proves that K is not computable. Assume an algorithm $L(x) \neq O(1)$ gives a lower bound for $K(x)$. We can use it to compute $f(n)$ that finds x with $n < L(x) \leq K(x)$. But $K(x) \leq K_f(x) + O(1)$ and $K_f(f(n)) \leq \|n\|$. So $n < K(f(n)) \leq \|n\| + O(1) = \log O(n) \ll n$: a contradiction. Thus, K and its non-constant lower bounds are not computable. An important application of Kolmogorov Complexity measures the Mutual Information: $I(x : y) = K(x) + K(y) - K(x, y)$. It has many uses which we cannot consider here.

Rarity (Deficiency of Randomness).

Some upper bounds of $K(x)$ are close in some important cases. One such case is of x generated at random. Define its **rarity** for uniform on $\{0, 1\}^n$ distribution as $d(x) = n - K(x|n) \geq -O(1)$.

What is the probability of $d(x) > i$ (i.e. of $K(x|n) < n - i$), for uniformly random n -bit x ? $n - i$ -bit programs generate $< 2^{n-i}$ strings. So, probability of such x is $< 2^{n-i} / 2^n = 2^{-i}$ for **any** n . Even for a billion-bit n , the probability of $d(x) > 300$ is really negligible (assuming x was indeed generated by fair coin flips).

Small rarity implies all other enumerable properties of random strings. Indeed, let such property “ $x \notin P$ ” have a negligible chance: a small, say $< 2^{-s_n}$, fraction of n -bits strings violate P . To generate x , we need only the algorithm enumerating S_n and the $n - s_n$ -bit position of x in that enumeration.

Then the rarity $d(x) > s_n - O(1)$ is large. Each x violating P will thus also violate the “small rarity” requirement. In particular, the small rarity implies unpredictability of bits of random strings: Any short algorithm with high prediction rate assures large $d(x)$. However, the randomness can only be refuted, cannot be confirmed: we saw, K and its lower bounds are not computable.

Rectification of Distributions. Sources of randomness with precisely known distribution are rare. (Note: distributions of 10^6 coin flips with .49 and .51 head chances are nearly orthogonal.) But there are very efficient ways to convert “roughly” random sources into perfect ones. We follow [Goldreich, Levin 89], which also uses some earlier U. and V. Vazirani ideas. Assume a sequence with weird unknown distribution. We only know that its m bits long segments have some, say $> k + i$, min-entropy i.e. probability $< 1/2^{k+i}$, given all previous bits. (Without such m arbitrary long segments could be fully predictable, thus useless.)

No relation is required between n, m, i, k , but useful are small m, i, k and huge n (within $o(2^k/i)$). Fold X into an $n \times m$ matrix. We also need a **small** $m \times i$ matrix Z , **really** independent of X and uniformly random (or random Toeplitz, i.e. s.t. $Z_{a+1, b+1} = Z_{a, b}$). The same Z can be reused repeatedly with many X . Then the $n \times i$ product XZ has uniform with accuracy $O(\sqrt{ni}/2^k)$ distribution.

6.2 Pseudo-randomness

The above definition of randomness is very robust, if not practical. True random generators are rarely used in computing. The problem is *not* in physics of random sources: we just saw efficient ways to perfect the distributions of messy random noise. The reason lies in many extra benefits provided by pseudorandom generators. E.g., designing, debugging, or using programs often requires to repeat the exact same sequence. With a physical random generator, one would need to record all its bits: slow and costly.

[Blum, Micali 84, Yao 82] justified an alternative: generating **pseudo-random** strings from a short seed. First, take any one-way permutation $f_n(x)$ (see sec. 6.3) with a **hard-core** bit $b_t(x)$ (in $\{0, 1\}$ or $\{\pm 1\}$): easy to compute from (x, t) , but infeasible to guess from $(f_n(x), n, t)$ with any noticeable correlation (see below). Then take random k -bit **seeds** x_0, t, n , and run: $x_{i+1} \leftarrow f_n(x_i)$, $S_i \leftarrow b_t(x_i) = b_t((f_n)^i(x_0))$.

We will see how distinguishing outputs S of this generator from strings of coin flips would imply the ability to invert f : infeasible if f is one-way. But if $P=NP$ (a famous open problem), no one-way f , and no pseudorandom generators could exist. By Kolmogorov standards, pseudo-random strings $S=G(s)$, are not random: Take $\|S\| \gg \|s\|$. Then $K(S) \leq O(1) + \|s\| \ll \|S\|$, so fail Kolmogorov's definition.

We can distinguish between truly random and pseudo-random strings by simply trying all short seeds. This, however, takes exponential time $2^{\|s\|}$. Realistically, pseudo-random strings will be as good as truly random ones if they can't be distinguished in any feasible time. Such generators we call **perfect**.

Theorem: [Yao 82] Let $G(s) = S \in \{0, 1\}^n$ run in time T_G . Let an algorithm A_w in expected (over internal coin flips w) time T_A accept $G(s)$ and truly random strings with probabilities different by d . Then, for random i , one can use A to guess S_i from $S_{i+} \stackrel{\text{def}}{=} S_{i+1}, \dots, S_n$ in time $T_A + T_G$ with correlation d/n .

Proof. Let $p_{i,\sigma}$ (or p_i for $\sigma=\{\}$) be the probabilities of A accepting $r\sigma S_{i+}$, for randomly chosen r and $\|r\sigma\| = i$. So A accepts $G(s)$ and random r with respective probabilities p_0 , $p_0 = p_n + d$. So, $p_{i-1} - p_i = d/n$, for randomly chosen i . Then $(p_{i,1} + p_{i,0})/2$ averages to p_i , while $p_{i,S_i} = p_{i,0} + (p_{i,1} - p_{i,0})S_i$ averages to p_{i-1} and $(p_{i,1} - p_{i,0})(S_i - 1/2)$ to $p_{i-1} - p_i = d/n$. So, (samplable) $p_{i,1} - p_{i,0}$ has the stated correlation with S_i . \square

If the above generator $S_i = b_t(f_n(x_i))$ was not perfect, one could guess S_i from S_{i+} with correlation d/n . But, S_{i+} can be computed from $(f_n(x_i), n, t)$. So, this employs A to guess $b_t(x_i)$ from $(f_n(x_i), n, t, w)$ with correlation d/n contrary to b being hard-core.

Hard Core. The key to pseudorandom generation is a one-way f (e.g., Rabin's, sec. 6.3) with a hard core. And $(x \cdot t) = \sum_i x_i p_i \bmod 2$ is hard-core for any one-way $F(x, n, t) = (f_n(x), n, t)$. Indeed, [Goldreich, Levin 89] converts any method a of guessing $(x \cdot t)$ from $(f_n(x), n, t)$ with correlation ε into one for inverting f (slower ε^2 times than a). [Knuth 97] has more details and references.

Proof. (Simplified with some ideas of Charles Rackoff.) Let $k = \|x\| = \|f_n(x)\|$, $j = \log(2k/\varepsilon^2)$, $v_i = 0^{i-1}10^{k-i}$, $b_t(x) = (-1)^{(x \cdot t)}$. Throughout the proof $y = (f_n(x), n, w)$ is fixed. Assume $a_t(y) \in \{\pm 1\}$ guesses $b_t(x)$ with correlation $\sum_t 2^{-\|t\|} b_t(x) a_t(y) > \varepsilon$. $b_t(x) a_t$ averaged over $> 2k/\varepsilon^2$ random pairwise independent t is ε -close to its mean (over all t) (and so > 0) with probability $> 1 - 1/2k$. Same for $b_{t+v_i}(x) a_{t+v_i} = (-1)^{(x \cdot t)} a_{t+v_i} (-1)^{(x \cdot v_i)}$.

Take a random $k \times j$ binary matrix P . The vectors Pr , $0 \neq r \in \{0, 1\}^j$, are pairwise independent. So, for a fraction $\geq 1 - 1/2k$ of P , sign $(\sum_r (-1)^{xPr} a_{Pr+v_i}) = (-1)^{(x \cdot v_i)}$. We could thus find all bits $(x \cdot v_i)$ of x with probability $> .5$ if we knew $z = xP$. But z is short: we can try all its 2^j possible values and check $f_n(x)$ for each! So the inverter computes $A_i(r) = a_{Pr+v_i}$, for a random P and all i, r . It uses Fast Fourier on A_i to compute $h_i(z) = \sum_r b_r(z) A_i(r)$. The sign of $h_i(z)$ is the i -th bit of z -th member of output list. \square

6.3 Cryptography

A very useful type of one-way permutations, called “trap-doors”, has many applications, especially in Cryptography. Here is a popular example:

Rabin’s One-way Function. Pick random prime numbers p, q , $\|p\|=\|q\|$ with two last bits =1, i.e. with odd $(p-1)(q-1)/4$. Then $n = pq$ is called a Blum number. Its length should make factoring infeasible. All residues below are mod n , i.e. in Z_n . Let Q_n be the group of **quadratic residues**, i.e. squares in Z_n^* .

Lemma. Let $n = pq$ be a Blum number, $F : x \mapsto x^2 \pmod{n}$. Then (1) F is a permutation on Q_n and (2) The ability to invert F on random x is equivalent to that of factoring n .

Proof. (1) $v \stackrel{\text{def}}{=} (p-1)(q-1)/4$ is odd, so $u=(v+1)/2$ is an integer. Let $x=F(z)$. Both $p-1$ and $q-1$ divide $2v$. So, by Fermat’s little theorem, both p, q (and, thus n) divide $x^v - 1 \equiv z^{2v} - 1$. Then $F(x)^u \equiv x^{2u} = x^{v+1} \equiv x$.

(2) The above y^u inverts F . Conversely, let $F(A(y)) = y$ for a fraction ε of $y \in Q_n$.

Each $y \in Q_n$ has $x, x' \neq \pm x$ with $F(x)=F(x')=y$, both with equal chance to be chosen at random.

If $F(x)$ generates y while $A(y)=x'$ the Square Root Test (Sec.5.1) has both x, x' for factoring n . \square

Picking random primes is easy: their density is $1/O(\|p\|)$. Indeed, all primes in $[n, 2n]$ divide $\binom{2n}{n}$ but none in $[\frac{2}{3}n, n]$ and no prime power $p^i > 2n$ do. So, $\log_n \binom{2n}{n} = 2n/\log n - O(1)$ is an upper bound on the number of primes in $[n, 2n]$ and a lower bound on that in $[1, 2n]$ (and in $[3n, 6n]$ (i.e. in $[1, 6n] \setminus [1, 2n]$). And fast VLSI exist to multiply long numbers and check primality.

Public Key Encryption. A perfect way to encrypt a message m is to add it mod 2 bit by bit to a random string S of same length k . The resulting encryption $m \oplus S$ has the same uniform probability distribution, no matter what m is. So it is useless to the adversary for learning something about m , if S is hidden.

A disadvantage is that the parties must secretly share S as large as all messages to be exchanged, combined.

Public Key Cryptosystems use two keys. One, needed to encrypt the messages, may be completely disclosed to the public. The **secret decryption** key need not be sent to the encrypting party.

The same keys may be used repeatedly for many messages.

Such cryptosystem can be obtained [Blum, Goldwasser 82] by replacing the above random S by pseudo-random $S_i = (x_i \cdot t)$; $x_{i+1} = (x_i^2 \pmod{n})$. Here a Blum number $n = pq$ is chosen by the Decryptor and is public, but p, q are kept secret. The Encryptor chooses $t \in \mathbb{Z}_2^{\|n\|}$, $x_0 \in \mathbb{Z}_n$ at random and sends $t, x_k, m \oplus S$.

Assuming factoring is intractable for the adversary, S should be indistinguishable from random strings (even with known t, x_k). Then this scheme is as secure as if S were random. The Decryptor knows p, q and can compute u, v (see above) and $w=(u^{k-1} \pmod{v})$. So, he can find $x_1=(x_k^w \pmod{n})$, then S and m .

Another use of the intractability of factoring is digital signatures [Rivest, Shamir, Adleman 78, Rabin 79]. Strings x can be released as authorizations of $y = (x^2 \pmod{n})$. Verifying x , is easy but the ability of forging it for generic y is equivalent to that of factoring n .

7 End Matter.

7.1 Go On!

You saw, most of our burning questions are still open. Take them on! Start with reading recent results (FOCS/STOC is a good source). See where you can improve them. Start writing, first notes just for your friends, then real papers. Here is a little writing advice: A well written paper has clear components: skeleton, muscles, etc. The skeleton is an acyclic digraph of basic definitions and statements, with cross-references. The meat consists of proofs (muscles) each *separately* verifiable by competent graduate students having to read no other parts but statements and definitions cited. Intuitive comments, examples and other comfort items are fat and skin: a lack or excess will not make the paper pretty. Proper scholarly references constitute clothing, no paper should ever appear in public without! Trains of thought which led to the discovery are blood and guts: keep them hidden. Metaphors for other vital parts, like open problems, I skip out of modesty.

7.2 Writing Contributions.

Sec. 1 was originally prepared by Elena Temin, Yong Gao and Imre Kifor (BU), others by Berkeley students: 2.3 by Mark Sullivan, 3.1 by Eric Herrmann and Elena Eliashberg, part of 3.2 by Wayne Fenton and Peter Van Roy, 3.3 by Carl Ludewig, Sean Flynn, and Francois Dumas, 4.1 by Jeff Makaiwi, Brian Jones and Carl Ludewig, 4.2 by David Leech and Peter Van Roy, 4.3 by Johnny and Siu-Ling Chan, 5.2 by Deborah Kordon, 6.1 by Carl Ludewig, 6.2 by Sean Flynn, Francois Dumas, Eric Herrmann, 6.3 by Brian Jones.

References

- [Levin 91] L. Levin. Fundamentals of Computing: a Cheat-List. *SIGACT News; Education Forum*. Special 100-th issue, 27(3):89-110, 1996. Errata: *ibid.* 28(2):80. Earlier version: *ibid.* 22(1), 1991.
- [Kleinberg, Tardos 06] Jon Kleinberg, Eva Tardos. *Algorithm design*. 2006. Pearson.
- [Knuth 97] Donald E. Knuth. *The Art of Computer Programming*. Vol. 1-3. Addison-Wesley, 3d ed., 1997. New to 3d ed. Sec.3.5.F of v.2 is also on pp. 10, 29-36 of <https://www-cs-faculty.stanford.edu/~knuth/err2-2e.ps.gz>
- [Feller 68] William Feller. *An Introduction to Probability Theory and Its Applications*. Wiley & Sons, 1968.
- [Lang 93] S.Lang. *Algebra*. 3rd ed. 1993, Addison-Wesley.
- [Rogers 67] H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967.
- [] References for section 1:
- [Barzdin', Kalnin's 74] Ja.M. Barzdin', Ja.Ja. Kalnin's. A Universal Automaton with Variable Structure. *Automatic Control and Computing Sciences*. 8(2):6-12, 1974.
- [Berlekamp, Conway, Guy 82] E.R.Berlekamp, J.H.Conway, R.K.Guy. *Winning Ways*. Sec.25. 1982.
- [Kolmogorov, Uspenskii 58] A.N. Kolmogorov, V.A. Uspenskii. On the Definition of an Algorithm. *Uspekhi Mat. Nauk* 13:3-28, 1958; AMS Transl. 2nd ser. 29:217-245, 1963.
- [Schoenrage 80] A. Schoenrage. Storage Modification Machines. *SIAM J. on Computing* 9(3):490-508, 1980.
- [Ofman 65] Yu. Ofman. A Universal Automaton. *Trans. of the Moscow Math. Soc.*, pp.200-215, 1965.
- [] Section 2:
- [Blum 67] M. Blum. A machine-independent theory of the complexity of recursive functions. *JACM* 14, 1967.
- [Davis 65] M. Davis, ed. *The Undecidable*. Hewlett, N.Y. Raven Press, 1965. (The reprints of the original papers of K.Goedel, A.Turing, A.Church, E.Post and others).
- [Ikeno 58] Shinichi Ikeno. A 6-symbol 10-state Universal Turing Machine. *Proceedings, Institute of Electrical Communications, Tokyo*, 1958.
- [Seiferas, Meyer 95] Joel I. Seiferas, Albert R. Meyer. Characterization of Realizable Space Complexities. *Annals of Pure and Applied Logic* 73:171-190, 1995.
- [Rabin 59] M.O. Rabin. Speed of computation of functions and classification of recursive sets. *Third Convention of Sci.Soc.* Israel, 1959, 1-2. Abst.: Bull. of the Research Council of Israel, 8F:69-70, 1959.
- [Tseitin 56] G.S. Tseitin. Talk: seminar on math. logic, Moscow university, 11/14, 11/21, 1956. Also pp. 44-45 in: S.A. Yanovskaya, Math. Logic and Foundations of Math., *Math. in the USSR for 40 Years*, 1:13-120, 1959, Moscow, Fizmatgiz, (in Russian).
- [] Section 3:
- [Neumann, Morgenstern 44] J. v.Neumann, O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton Univ. Press, 1944.
- [Stockmeyer, Meyer 73] L.Stockmeyer, A.Meyer. Word problems requiring exponential time. *STOC-1973*
- [Chandra, Kozen, Stockmeyer 81] Ashok K. Chandra, Dexter C. Kozen, Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114-133, 1981.

- [Robson 83, 84] J.M. Robson. N by N checkers is EXPTIME-complete. *SIAM J. Comput* 13(2), 1984. Also: The complexity of Go. *Proc. 1983 IFIP World Computer Congress*, p. 413-417.
- [Fraenkel, Lichtenstein 81] A.S. Fraenkel, D. Lichtenstein. Computing a perfect strategy for $n \times n$ chess requires time exponential in n . *J. Combin. Theory (Ser. A)* 31:199-214. ICALP-1981.
- [] Section 4:
- [Savitch 70] W.J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.* 4:177-190, 1970.
- [Yudin and A.S. Nemirovsky 76] D.B. Yudin and A.S. Nemirovsky. Informational Complexity and Effective Methods for Solving Convex Extremum Problems. *Economica i Mat. Metody* 12(2):128-142; transl. *MatEcon* 13:3-25, 1976.
- [Luks 80] E.M. Luks: Isomorphism of Graphs of Bounded Valence Can Be Tested in Polynomial Time. *FOCS-1980*.
- [Garey, Johnson 79] M.R.Garey, D.S.Johnson. *Computers and Intractability*. W.H.Freeman & Co. 1979.
- [Trakhtenbrot 84] B.A.Trakhtenbrot. A survey of Russian approaches to *Perebor* (brute-force search) algorithms. *Annals of the History of Computing*, 6(4):384-400, 1984.
- [] Section 5:
- [Rabin 80] M.O.Rabin. Probabilistic Algorithms for Testing Primality. *J. Number Theory*, 12: 128-138, 1980.
- [Miller 76] G.L.Miller. Riemann's Hypothesis and tests for Primality. *J. Comp. Sys. Sci.* 13(3):300-317, 1976.
- [Solovay, Strassen 77] R. Solovay, V. Strassen. A fast Monte-Carlo test for primality. *SICOMP* 6:84-85, 1977.
- [Karp 86] R. Karp. Combinatorics, Complexity and Randomness. (Turing Award Lecture) *Communication of the ACM*, 29(2):98-109, 1986.
- [Johnson 84] David S. Johnson. The NP-Completeness Column. *J. of Algorithms* 5:284-299, 1984.
- [Karp 76] R. Karp. The probabilistic analysis of some combinatorial search algorithms. *Algorithms and Complexity*. (J.F.Traub, ed.) pp. 1-19. Academic Press, NY 1976.
- [Gurevich 85] Y. Gurevich, Average Case Complexity. *Internat. Symp. on Information Theory, IEEE*, 1985.
- [Levin Venkatesan 18] Leonid A Levin, Ramarathnam Venkatesan. An average case NP-complete graph coloring problem. *Combinatorics, Probability, and Computing*, 27(5), 2018. <https://arxiv.org/abs/cs/0112001>
- [Shamir 90] A. Shamir. IP = PSPACE. *JACM* 39/4:869-877, 1992.
- [Fortnow, Lund 93] Lance Fortnow, Carsten Lund. Interactive proof systems and alternating time—space complexity. *Theor.Comp.Sci.* 113(1):55-73, 1993. [https://doi.org/10.1016/0304-3975\(93\)90210-K](https://doi.org/10.1016/0304-3975(93)90210-K)
- [Holographic proof] Holographic proof. *The Encyclopedia of Mathematics, Supplement II*, Hazewinkel, M. (Ed.), Kluwer, 2000. https://encyclopediaofmath.org/wiki/Holographic_proof
- [] Section 6:
- [Kolmogorov, V.A.Uspenskii 87] A.N.Kolmogorov, V.A.Uspenskii. Algorithms and Randomness. *Theoria Veroyatnostey i ee Primeneniya = Theory of Probability and its Applications*, 3(32):389-412, 1987.
- [Li, Vitanyi 19] M. Li, P.M.B. Vitanyi. *Introduction to Kolmogorov Complexity and its Applications*. Springer Verlag, New York, 2019.
- [Blum, Micali 84] M.Blum, S.Micali. How to generate Cryptographically Strong Sequences. *SICOMP*, 13, 1984.
- [Yao 82] A. C. Yao. Theory and Applications of Trapdoor Functions. *FOCS-1982*.
- [Goldreich, Levin 89] O.Goldreich, L.Levin. A Hard-Core Predicate for all One-Way Functions. *STOC-1989*.
- [Rivest, Shamir, Adleman 78] R.Rivest, A.Shamir, L.Adleman. A Method for Obtaining Digital Signature and Public-Key Cryptosystems. *Comm. ACM*, 21:120-126, 1978.
- [Blum, Goldwasser 82] M. Blum, S. Goldwasser. An Efficient Probabilistic Encryption Scheme Hiding All Partial Information. *Crypto-1982*.
- [Rabin 79] M. Rabin. *Digitalized Signatures as Intractable as Factorization*. MIT/LCS/TR-212, 1979.