Fundamentals of Computing

Leonid A. Levin

These are the notes for the course CS-172 I taught in the Fall of 1986 at UC Berkeley. The goal was to introduce the undergraduates to basic concepts of Theory of Computation and to provoke their interest in further study. The model-dependent effects were systematically ignored. Concrete computational problems were considered only as illustrations of general principles. The notes (prepared by the students and revised by me) are skeletal: they do have (terse) proofs, but exercises, references, intuitive comments and examples are missing or inadequate. The better is English in a paragraph the smaller was my contribution and the greater caution is needed.

The notes can be used by an instructor designing a course or by students who either know the material and want to refresh the memory or are exceptionally bright and have access to an instructor for questions. Each subsection takes about a week of the course. I distribute these notes to generate some discussion on the content of the introductory theory course. I am most interested in your comments, both general and on particular account details. My home address is: 460 Commonwealth Avenue, Newton, MA 02459-1333; tel.: (617) 332-9492; e-mail: Lnd (@bu.edu). I keep updating these notes: take a fresh copy from http://www.cs.bu.edu/fac/lnd/toc/ (z.ps or z.dvi).

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 who not only have originally written these notes, but also influenced the lectures a lot by providing very intelligent reactions and criticism.

Contents

 Last revised: June 21, 1999.

1 Models of Computations; Polynomial Time & Church's Thesis.

1.1 Deterministic Computation.

Sections 1,2 study deterministic computations. Non-deterministic aspects of computations (inputs, interaction, errors, randomization, etc.) are crucial and challenging in advanced theory and practice. Defining them as an extension of deterministic computations is simple. The latter, however, while simpler conceptually, require elaborate models for definition. These models may be sophisticated if we need a precise measurement of all required resources. 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, gradually making it more specific to the models we actually study. *Computations* consist of *events* and can be represented as graphs, where edges between events reflect various relations. Nodes and edges will have attributes called labels, states, values, colors, parameters, etc. We require different labels for any two edges with the same source. Edges of one type, called *causal,* run from each event x to all events essential for the occurrence or attributes of x. They form a directed acyclic graph (though cycles are sometimes added artificially to mark the input parts of the computation).

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 run between events with close (typically, consecutive) values of time. 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. Pointers reflect the connection between simultaneous events that allows them to interact and have a joint effect. The subgraph of events at a particular value of time (with pointers and attributes) 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 single active area at any step of the computation are *sequential.* Others are called *parallel.*

Complexity.

The following measures of computing resources of a machine A on input x will be used throughout the course:

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 during all steps. Time $T_{A(x)}$ is used (depending on the context) as either depth or volume, which coincide for sequential models.

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 unchanged since the input.

Note that time complexity is robust only up to a constant factor: a machine can be modified into a new one with a larger alphabet of labels, representing several locations in one. It would produce identical results in a fraction of time and space (provided that the time limits are sufficient for the transformation of the input into and output from the new alphabet).

Growth Rate Notations: $f(x) = O(g(x))$[1] $\iff$ $g(x) = \Omega(f(x))$ $\iff$ $\sup_x \frac{f(x)}{g(x)} < \infty$.

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

$\theta : f(x) = \theta(g(x)) \iff (f(x) = O(g(x))$ and $g(x) = O(f(x)))$.

Here are a few examples of frequently appearing growth rates: negligible $(\log n)^{O(1)}$; moderate $n^{\theta(1)}$ (called polynomial or P, like in P-time); infeasible $2^{n^{\Omega(1)}}$; another infeasible: $n! = (n/e)^n\sqrt{t + 2\pi n}$, $t \in [1, 2]$.

The reason for ruling out exponential (and neglecting logarithmic) rates is that the known Universe is too small to accommodate exponents. Being about 15 billions years old, it is at most 15 billion light years $\sim 10^{61}$ Plank Units wide. A system of $\gg R^{1.5}$ particles packed in R Plank Units radius collapses rapidly, be it the Universe or a neutron star. So the number of particles is $< 10^{91.5} \sim 2^{304} \ll 4^{4^4} \ll 5!!$.

[1] This is a customary but somewhat misleading notation. The clear notations would be like $f(x) \in O(g(x))$

1.2 Rigid Models.

Rigid computations have another node parameter: *location* or *cell*. Combined with time, it designates the event uniquely. Locations have *structure* or *proximity* edges between them. They (or their short chains) indicate all neighbors of a node 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 fixed (independent of the grid size) number of *states* to label the events. The states include, among other values, pointers to the grid neighbors. At each step of the computation, the state of each cell can change as prescribed by a *transition* function of 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 (also called program).

An example of a possible application of CA is a VLSI (very large scale integration) chip 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 delays implicitly affect the simulation.

An example: the Game of Life (GL).

Consider a plane grid of cells, each having a 1-bit state (dead/alive) and pointers to the 8 natural neighbors. The 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 overpopulation or solitude).

A *simulation* of a machine M_1 by M_2 is a correspondence between memory configurations of M_1 and M_2 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 (see a sketch of an ingenious proof in the last section of [Berlekamp, Conway Guy 82]) in this formal sense:

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.

Consider an infinite (say, to the right) chain or *tape* of cells with two adjacent neighbors each. Each state of a cell has a pointer to one neighbor. The input to this CA is an array of rightward cells followed at the right by leftward blanks. A cell changes state, if and only if it and its neighbor "face", i.e. point to each other. The transition function prevents the cells from ever turning "back-to-back." We can use these 1-pointer CA as a definition of the TM. The pair of active cells can be viewed as the TM's moving head (the cell which just changed the pointer direction) and the tape symbol it works on.

Another type of CA represents a TM with several non-communicating heads which can emerge from and disappear into the first cell (which, thus, controls the number of active cells). The computation halts when all heads vanish. This model has convenient theoretical features. E.g. with linear (in T) number ($|p|^2 T$) of state changes (volume) one can solve the *Bounded Halting Problem* $H(p, x, T)$: find out whether the machine with a program p stops on an input x within volume T of computation.

Problem: Find a method to transform any given multi-head TM A into another one B such that the value of the output of $B(x)$ (as a binary integer) and the volumes of computation of $A(x)$ and of $B(x)$ are all equal within a constant factor (for all inputs x).

Hint: B may keep a field to simulate A and maintain (in other fields) two binary counters h for the number of heads of A and v for A's volume. The least significant digits of h, v would be at the leftmost cell. The most significant digit of h would be added at each step to the same digit of v.

1.3 Pointer Machines.

The memory configuration of a *Pointer Machine (PM)*, called *pointer graph*, is a finite directed labeled graph. One node is marked as *root* and has directed paths to all other nodes. The *nodes* act as automata changing the graph locally. Edges (*pointers*) are labeled with *colors* from a finite alphabet fixed for all inputs. The pointers coming out of a node must have different colors (which bounds the outdegree).

Nodes *see* the configuration of their out-neighborhood of constant (2 suffices) depth. Some colors (as well as their carrying edges and the nodes seeing them) are called *active* and are not used in input/output. Active pointers must have inverses; their source nodes must have active loop-edges. An active node can create/delete pointers to neighbors and create new nodes with pointers to and from it. It does that, based on its neighborhood, according to the *program* which is the same for all nodes. Nodes with no pointers vanish. Nodes with no path from the root are permanently invisible and effectively removed. The computation is initiated by inserting an active loop-edge into the root. When no active pointers remain, the graph is the output.

The subgraph of active pointers is kept connected and acyclic: A node cannot set or remove active pointers to other nodes with active loops. Active loops must be removed if no incident active edges exist.

It is convenient to assume PM nodes to act in two stages: At *pulling* stage one, for each 2-pointer path colored x, y, creates a new pointer with a special color xy. Then the node removes/recolors pointers and creates new nodes based on what colors its pointers have and which of their sinks are identical.

Problem: Design a PM transforming the input graph into the same one with an extra pointer from each node to the root. Hint: Nodes with no path *to* the root can never be activated. They should be copied with pointers, copies connected to the root, then the original input removed.

Pointer Machines can be either *Parallel, PPM* [Barzdin' Kalnin's 74] or *Sequential*. The latter differ by the restriction that only nodes pointed to by the root can be active.

A *Kolmogorov* or *Kolmogorov-Uspenskii* Machine (KM) [Kolmogorov Uspenskii 58], is a special case of Pointer Machine [Shonhage 80] with the restriction that all pointers have inverses. This implies the bounded in/out-degree of the graph which we further assume to be 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 structure of the machine cannot be altered 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 make a simple special case of a PPM which restricts the Pointer Graph to be a grid.

Example Problem. Design a machine of each model (TM, CA, KM, PPM) which determines if an input string x is a double (i.e. has a form ww, $w \in \{a,b\}^*$). Analyze time 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,PM,CA. The space is $O(|x|)$ in all three cases.

Turing and Pointer Machines. TM uses extra symbols A, B. First find the middle of ww by capitalizing the letters at both ends one by one. Then compare letter by letter the two halves, lowering the case of the compared letters. The complexity is: $T(x) = O(|x|^2)$. PM algorithm is similar to the TM's, except that the root keeps and updates the pointers to the borders between the upper and lower case substrings. This allows commuting between these substrings in constant time. So, the complexity is: $T(x) = O(|x|)$.

Cellular Automata. The computation begins from the leftmost cell sending right two signals. Reaching the end the first signal turns back. The second signal propagates three times slower than the first. They meet in the middle of ww and disappear. While alive, the second signal copies the input field i of each cell into a special field a. The a symbols will try to move right whenever the next cell's a field is blank. So the chain of these symbols alternating with blanks will start moving right from the middle of ww. When they reach the end they will push the blanks out and pack themselves back into a copy of the left half of ww shifted right. When an a symbol does not have a blank at the right to move to, it compares itself with the i field of the same cell. They should be identical, if the ww form is correct. Otherwise a signal is generated which halts all activity and rejects x. If all comparisons are successful, the last symbol generates the accepting signal. The complexity is: $T(x) = O(|x|)$.

1.4 Simulation.

We considered several types of machines (models of computation). We will see now that all these machines can be simulated by the simplest of them: the Turing Machine. In other words, these powerful machines can compute only those functions computable by a TM.

Church-Turing Thesis is a generalization of this conclusion: TMs can compute every function computable in any thinkable physical model of computation. This is not a mathematical result because the notion of model is not formally specified. But the long history of investigations of ways to design real and ideal computing devices makes it very convincing. Moreover, this Thesis has a stronger *Polynomial Time* version which states that if any model computes a function in polynomial volume, TM can do the same. Both forms of the Thesis play a significant role in foundations of Computer Science.

PKM Simulation of PPM. For convenience, we assume each PPM node has an edge into root. Now, for each node, we reconnect its incoming (in unlimited number) PPM edges, 2 per leaf, to a bidirectional binary tree with new PKM colors l, r, u. The number of edges increases at most 4 times. The nodes simulate PPM's pulling stage by extending their trees to double depth. To simulate the re-coloring stage, each node gets a binary *name* formed by the l, r colors on its path through the root tree. Then it broadcasts its name down its own tree. When each node thus receives identities and pointers of its PPM neighbors, it stores them by creating a little auxiliary chain (acting as TM). Then it computes and implements the actions of the original PPM and rebalances its tree. This simulation of a PPM step takes polylogarithmic time.

TM Simulation of PPM. To simulate the above PKM by a TM, we first represent its memory configuration on the TM tape as the list of all pointers, sorted by the source names (described above) and then by color. The PKM program is reflected in the TM's transition table. Now the TM can simulate a PKM's pulling stage as follows: It 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: the special double-colored pointers can be created and moved to their sources by sorting. The re-coloring stage is straightforward, as all relevant pointers have the same source and are located together. When no active edges remain in the root, the Turing machine stops and its tape represents the PKM output. If a PPM computes a function $f(x)$ 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.

FCM Simulation of PPM [Ofman 65]. FCM represents the pointer graph in the same way as the above TM and uses a similar procedure. All steps are straightforward to do locally in parallel polylogarithmic time except sorting pointers. We need to create a fixed connection sorting network which takes an arbitrary array of integers as input and outputs it sorted. Sophisticated networks take logarithmic time. But we need only a simpler polylogarithmic method, Merge-Sort with parallel Batcher-Merge: Arrays with two or more entries are separated in two halves and each sorted recursively. The Batcher-Merge combines two sorted lists in *parallel* logarithmic time.

Batcher Merge: Call array entries *i-th partners* when their addresses differ only in i-th bit. Operations on partners can be implemented on a *Shuffle Exchange* graph of 2^k nodes. Each node has pointers to its k-th partner and to and from its *shift* node obtained by moving its first address bit to the end.

A *bitonic cycle* is the combination of two sorted arrays (one may be shorter), connected by max-to-max and min-to-min entries. We merge sorted arrays by appending the reversed second one to the first, considering the last and first entries as neighbors, and sorting the resulting bitonic cycle.

The sorting of a 2^k long bitonic cycle proceeds by comparing each entry with its k-th partner (i.e. diametric opposite on the cycle) and switching if wrongly ordered. Each half becomes then a bitonic cycle and any two entries from different halves are in proper order. The process then repeats for each half recursively (decrementing k through the graph's 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 can be then specified by just giving its description (program) as part of the input rather than by changing the hardware. This was a radical idea, since in the classical mathematics universal functions do not exist (as we will see).

Let R be the class of all TM-computable total (defined for all inputs) and partial (which may diverge) functions. Surprisingly, there is a universal function u in R. It simulates any other $f \in R$ in time c^2T and space $S + c$, where $S, T > |x|$ are space and time of computing $f(x)$ and c is its program length. This u expects the prefix m of its input mx to list the commands of a Turing Machine M and its initial head state. Then $u(mx)$ operates in cycles. Each cycle simulates one step of $M(x)$. Let after i steps of $M(x)$, l_i be the left (from the head) part of its tape; r_i be the rest of the tape and s_i be the head's state. The tape configuration of $u(mx)$ after i cycles is $t_i = l_i m s_i r_i$. Then u looks up m to find the command corresponding to the state s_i and the first character of r_i and modifies t_i accordingly. When $M(x)$ halts, $u(mx)$ erases ms_i from the tape 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 $x, t(x)$ and an appropriate program). We now describe in detail a simpler but slightly slower universal TM.

The transition table at the right defines a small (11 states + 6 symbols) TM U by Ikeno which can simulate any other TM M over $\{0,1\}$ tape alphabet with the following stipulations (which still allow M to simulate any other TMs): The direction of head shift is a function of the new post-transition state (lower case - left, upper case - right). And so is, for M only, the digit typed. The tape is infinite to the right only: the left states in the leftmost cell remain there. For M only, the new state is the tape bit read plus a function of the old state. In the U table the states and tape digits are shown only when changed; except that the prime is always shown. The halt and external input/output commands are special states for M; for U they are shown as =.

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

U's tape consist of segments: each is a 0/1 string preceded with a *. Some symbols are primed. Each finite segment describes a transition performed by one state of M and never changes (except for primes). The rightmost (infinite) segment is always a copy of M's tape, initially with U's head at the same location in the state C. Each transition is represented as STW, where W is the symbol to write, T the direction L/R to turn, represented as 0/1, S the new state (when 0 is read). S is represented as 1^k, if the next state is k segments to the left, or 0^k (if to the right). Initially, primed are the digits of S in the segment corresponding to the initial state of M and all digits to their left. An example of the configuration: $*'0'0'0'1'0' *' 0'0'0'0'01 * 011 * ... * 00$ $\boxed{\text{head}}$ 00.

U first reads the digit of an M's cell changing the state from C or f to a/b, puts a * there, moves left to the primed state segment S, finds from it the new state segment and moves there. With only 10 head states, U can't find the new segment at once. So, it (alternating the states c/C or d/D) keeps priming nearest unprimed * and 1s of S (or unpriming 0s). When S is exhausted the target segment, $|S|$ stars away, is reached. Then U reads (changing state from e to A/B) the rightmost symbol W of the new segment, copies it at the * in the M area, goes back, reads the next symbol T, returns to the just overwritten (and first unprimed) cell of M area and turns left or right. As CA, M and U have in each cell three standard bits: present and previous pointer directions and a "content" bit to store M's symbol. In addition U needs just 3 states of its own!

2.2 Uncomputability; Gödel Theorem.

Universal and Complete Functions.

Notations: Let us choose a special mark 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 *k-simulates* f iff for some $p = \text{Prefix}_k(q), q \neq p$ and all x, $u(px) = f(x)$. The prefix can be intuitively viewed as a program which simulating function u applies to the suffix (input). We also consider a symmetric variant of relation "k-simulate" which makes some proofs easier. Namely, u *k-intersects* f iff $u(px) = f(px)$ for some prefix_k p and all x. E.g., length preserving functions can intersect but not 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^+.

A *negation* of a (partial or total) function f is the total predicate $\neg f$ which yields 1 iff $f(x) = 0$ and yields 0 otherwise. Obviously, no closed under negation class of functions contains a complete one. So, there is no universal function in the class of all (computable or not) predicates. This is the well known Cantor Theorem that the set of all sets of strings (as well as the sets of all partial functions, reals etc.) is not countable.

Gödel Theorem.

There is no complete function among the *total* computable (recursive) ones, as this class is closed under negation. So the universal in R function u (and $u_2 = (u \bmod 2)$) has no total computable extensions.

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. A rich consistent and complete formal system cannot exist, since it 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 Gödel's Theorem which was one of the shocking surprises of the science of our century. (Here A is an extension of the formal Peano Arithmetic; we skip the details of its formalization and proof of richness.)[2]

Recursive Functions. Another byproduct is that the Halting (of $u(x)$) Problem would yield a total extension of u and, thus, is not computable. This is the source of many other uncomputability results. Another source is an elegant *Fixed Point* Theorem by S. Kleene: any total computable transformation A of programs (prefixes) maps some program into an equivalent one. Indeed, the complete/universal $u(px)$ intersects computable $u(A(p)x)$. This implies (exercise), e.g., that the only computable invariant (i.e. the same on programs computing the same functions) property of programs is constant (Rice-Uspenskii).

Computable (partial and total) functions are also called *recursive* (due to an alternative definition). Their ranges (and, equivalently, domains) are called (recursively) *enumerable* or *r.e.* sets. An r.e. set with an r.e. complement is called recursive (as is its yes/no characteristic function) or *decidable*. A function is recursive iff its graph is r.e. An r.e. graph of a total function is recursive. Each infinite r.e. set is the range of a 1-to-1 total recursive function ("enumerating" it, hence the name r.e.).

We can reduce membership problem of a set A to the one of a set B by finding a recursive function f s.t. $x \in A \iff f(x) \in B$. Then A is called *m-* (or *many-to-1-*) *reducible* to B. A more complex *Turing* reduction is given by an algorithm which, starting from input x, interacts with B by generating strings s and receiving answers to $s \in ?B$ questions. Eventually it stops and tells if $x \in A$. R.e. sets (like Halting Problem) to which all r.e. sets can be m-reduced are called r.e.-complete. One can show a set r.e.-complete (and, thus, undecidable) by reducing the Halting Problem to it. So Ju.Matijasevich proved r.e.-completeness of Diophantine Equations Problem: given a multivariate polynomial of degree 4 with integer coefficients, find if it has integer roots. The above (and related) concepts and facts are broadly used in Theory of Algorithms and should be learned from any standard text, e.g., [Rogers 67].

---

[2] A closer look at this proof reveals the second famous Gödel theorem: the consistency itself is an example of unprovable $\neg s_x$. Consistency C of A is expressible in A as divergence of the search for contradiction. u_2 intersects $1 - u_A$ for some prefix a. C implies that u_A extends u_2, and, thus, $u_2(a), u_A(a)$ both diverge. So, $C \Rightarrow \neg s_a$. This proof can be formalized in A which yields $\vdash C \Rightarrow \vdash \neg s_a$. But $\vdash \neg s_a$ implies $u_A(a) = 1$, so $C, \vdash C$ are incompatible.

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 class of functions computable within $o(t(x))$ steps which is closed under negation. So, u_t does not belong to the class, i.e. requires time $\Omega(t(x))$ [Tseitin 55]. E.g. $2^{|x|}$-BHP requires exponential time. For similar reasons any function which agrees with t-BHP on a *dense* (i.e. having strings with each prefix) subset cannot be computed in $o(t(x))$ steps either.

On the other hand, we know that for some trivial input programs the BHT can be answered by a fast algorithm. The following Rabin's Compression Theorem provides another predicate $P_f(x)$ for which there is only a finite number of such trivial inputs. The theorem is stated for the volume of computation for Multi-Head Turing Machine. It can be reformulated in terms of time of Pointer Machine and space (or, with smaller accuracy, time) of regular Turing Machine.

Definition: A function $f(x)$ is *constructible* if some algorithm F computes it, in binary, within volume $O(f(x))$, i.e. $V_{F(x)} = O(f(x))$.

Here are two examples: $2^{|x|}$ is constructible, as $V_n = O(n \log n) \ll 2^n$. $2^{|x|} + h(x)$, where $h(x)$ is 0 or 1, depending on whether $U(x)$ halts within $3^{|x|}$ steps is not constructible.

Theorem: For any constructible function f, there exists a function P_f such that for all functions T, the following two statements are equivalent:

1. There exists an algorithm A such that for all inputs x, $A(x)$ computes $P_f(x)$ in volume $T(x)$.
2. t is constructible *and* $f(x) = O(T(x))$.

Let *t-bounded Kolmogorov Complexity* $K_t(i/x)$ of i given x be the length of the shortest program p for the Universal Multi-Head Turing Machine transforming x into i with $< t$ volume of computation. Let $P_f(x)$ be the smallest i, with $2K_t(i/x) > \log(f(x)/t)$ for all t. P_f can be computed in volume f by generating all i of low complexity, sorting them and taking the first missing. It satisfies the Theorem, since computing $i = P_f(x)$ faster causes a violation of complexity bound defining it. P_f can be made a predicate: see [Rabin 59].

Speed-up Theorem.

This theorem will be formulated for exponential speed-up, but it remains true if log is replaced by any computable unbounded monotone function [Blum 67].

Theorem: There exists a total computable predicate P such that for any algorithm computing $P(x)$ with running time $T(x)$, there exists another algorithm computing $P(x)$ with running time $O(\log T(x))$.

This procedure may continue any constant number of steps. In other words, there is no even nearly optimal algorithm for the predicate P.

So, the complexity of some predicates P cannot be characterized by a single constructible function f, as in Compression Theorem. However, the Compression Theorem can be generalized by removing the requirement that f is constructible (it still must be computable or enumerable from below). In this form it is general enough so that every computable predicate (or function) P satisfies the statement of the theorem with an appropriate computable function f. There is no contradiction with Blum's Speed-up Theorem, since the complexity f cannot be reached (f is not constructible itself). See a review in [Seiferas, Meyer].

Rabin's predicate has an optimal algorithm. Blum's does not. In general, one can't tell whether a predicate has an optimal algorithm or not (see the Rice-Uspenskii Theorem in Sec. 2.2).

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

3.1 How to Win.

In this section we consider a more interesting *provably* intractable problem: playing games with *full information*, two players and *zero sum*. We will see that even for some simple games there cannot be a much more efficient algorithm, than exhaustive search through all possible configurations.

The rules of an *n-player game* G are set by a family f of *information functions* and a *transition rule* r. Each player $i \in I$ at each step participates in transforming a configuration (game position) $x \in S$ into the new configuration $r(x, m)$, $m : I \to M$ by choosing a move $m_i = m(i)$ based only on his knowledge $f_i(x)$ of x. The game proceeds until a *terminal* configurations $t \in T \subset S$ is reached. Then $f_i(t)$ is the *loss (or gain)* of the i-th player. We consider games with two-players. We identify S and M with the set of integers and take $f(T) = T = I = \{\pm 1\}$. Our games will have *zero sum* $\sum f_i(t) = 0$ and *full information:* $f_i(x) = ix, r(x, m) = r'(x, m_{a(x)})$, where $a(x)$ points to the *active* player. We take $a(x) = \mathbf{sign}(x)$ and $r'(x, m) \in \{m, -a(x)\}$. The predicate $R(x, y)$ checks if $r(x, y) = y$ i.e. whether the rule accepts the transition from the configuration x to y (a *legal* move) or terminates the game.

An example of such games is chess. Examples of games without full information are card games, where only part $f_i(x)$ (player's own hand) of the position x is known. Each player may have a strategy S providing a next position $y = S(x)$ for each position x. Once players choose strategies, the game (its moves) is completely determined. A strategy is *winning* (*non-losing*) if it guarantees victory (resp. survival) whatever the opponent does, even if he knows S. A position is *winning*, labeled W_i, if the active player has an i-step winning strategy, *losing* (L_i) if his opponent does, and *neutral* (N) if both have non-losing strategies. The latter can be excluded by adding a time counter decremented each step.

Solving a game, means determining the winning side for each position. The ability is close to the ability to find a good move in a modified game. Indeed, modify a game R into R' by adding a preliminary stage to it. At this stage the player A offers a starting position for R and her opponent B chooses which side to play. Then A may either start playing R or offer a new (lower in some order) starting position. Obviously, B wins if he can determine the winning side of every position. If he cannot while A can, she wins.

Conversely, any game R can be modified into a *choosing* game R' in which the list of all legal moves is easy to compute for any position. Solving such games is obviously sufficient for choosing the right move. A position of R' consists of a position x of R and a segment y of another position extended with "?"s to the board size. The active side replaces one ? with the next bit of y. When ?s are gone, he checks $R(x, y)$ and if the transition is legal, replaces x with y, empties y and switching the active side (i.e. the sign).

Games may be categorized by the difficulty to compute R. We will consider only R with $< 2^{|x|}$ time (mostly much faster). Our boards also never change size: $R(x, y) \Rightarrow |x| = |y|$.

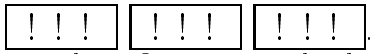
Theorem. *Each position of any full information game is winning, losing or neutral.*

(This theorem [Neumann, Morgenstern 44] fails for games with *partial* information: either player may lose if his strategy is known to the adversary. E.g.: 1. Blackjack (21); 2. Each player picks a bit; their equality determines the winner.) The best move can be found by playing all strategies against each other. 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}. The proof of this Theorem gives a much faster (but still exponential time!) strategy.

Proof: Make a graph of all $|x|$-bit positions and legal moves; set $i = 0$. Repeat: label L_i all nodes without moves (if none, exit); label W_i all nodes with a move to L_i; remove all labeled nodes and add 1 to i.

For all positions, this procedure guarantees existence of the legal moves: $W_i \to L_i$; $N \to N$ (unlabeled); $L_i \to W_{i-1}$. *Any strategy choosing these moves is obviously optimal.* The moves guaranteed not to exist are $N \to L$; $L \to N$; $L \to L$; $L_i \to W_{\geq i}$; $W_i \to L_{<i}$. Such labeling is called *consistent* and is unique.

Indices are bounded by the number $2^{|x|}$ of game configurations, since the number of Ns shrinks each step. There are $< 2^{2|x|}$ moves. The time to determine the legal moves is $< 2^{3|x|}$. The algorithm tries each legal move in each relabeling 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.

Problem: the Match Game. Consider 3 boxes with 3 matches each:  . The players alternate turns taking any *positive* number of matches from any *one* box. One cannot take the last match on the whole table. Use the above algorithm to list all winning and all losing positions.

3.2 Exponentially Hard Games.

A simple example of a full information game is *Linear Chess*, played on a finite linear board. Each piece is loyal to one of two sides: W (weak) or S (shy). It is assigned a gender M or F and a rank from a set of ranks Σ; this set doesn't depend on the board size. All the W's are always on the left side and all the S's on the right. All cells of the board are filled. 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:

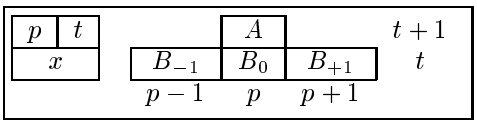
1. If S and W are of the same sex, W (being weaker) loses.
2. If S and W are of different sexes, S gets confused and loses.

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 (i.e. consistent labeling computed) in a *subexponential* time. We first prove that (see [Chandra, Kozen, Stockmeyer 1981]) for an artificial game. Then we reduce this *Halting Game* to Linear Chess.

For Exp-Time Completeness of regular (but $n \times n$) Chess, Go and Checkers see: [Fraenkel, Lichtenstein 1981], [Robson 1983, 1984].

Exptime Complete Halting Game.

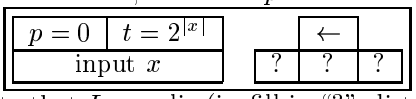
We use a universal Turing Machine u (defined as 1-pointer cellular automata) which halts only by its head rolling off of the tape's left end. Bounded Halting Problem BHP(x) determines if $u(x)$ stops (i.e. the leftmost tape cell points left) in $2^{|x|}$ steps. This requires $\Omega(2^{|x|})$ steps. We now convert u into the Halting Game. The players are: L claiming $u(x)$ halts in time (and should have winning strategy iff this is true); His opponent S. The *board* has four parts: the diagram, the input x to u, positive integers p (position) and t (time in the execution of $u(x)$):



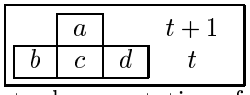
The diagram shows the states A of cell p at time $t+1$ and B_s, $s \in \{0, \pm 1\}$ of cells $p+s$, at time t. A, B include the pointer direction; B may be replaced by "?". Some board configurations are illegal: if (1) two of B_s point away from each other, or (2) A differs from the result prescribed by the transition rules for B_s, or (3) $t = 1$, while $(B_s) \neq x_{p+s}$. (At $t = 1$, $u(x)$ is just starting, so its tape has the input x at the left, the head in the initial state at the end with blanks leading off to the right.) Here are the **Game Rules:**

The game starts in the configuration shown below. L moves first replacing the ?'s with symbols that claim to reflect the state of cells $p+s$ at step t of $u(x)$. When S moves, he: chooses s, copies B_s into A and fills all B with ?'s; adds s to p and 1 to t.

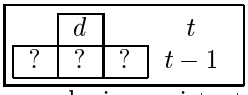
Start:



L puts:



S puts:



Note that L may lie (i.e fill in "?" distorting the actual computation of $u(x)$), as long as he is consistent with the above "local" rules. All S can do is to check the two consecutive board configurations. He cannot refer to past moves or to actual computation of $u(x)$ as an evidence of L's violation.

Strategy: If $u(x)$ does indeed halt within $2^{|x|}$ steps, then the initial configuration is true to the computation of $u(x)$. Then L has an obvious (though hard to compute) winning strategy: he just tells the truth, what actually happens during the computation. He will be always consistent; S will lose when $t = 1$ and cannot decrease any more. If the initial configuration is false then S can win exploiting that L must lie. If L lies once, S can force L to lie all the way down to $t = 1$. How?

If the upper box a of a legal configuration is false then the lower boxes b, c, d cannot all be true, since the rules of u determine a uniquely from them. If S guesses correctly which of b, c, or d is false and brings it to the top on his move, then L is forced to keep on lying. At time $t = 1$ all chips are down: the lying of L is exposed since the configuration doesn't match the actual input string x, i.e. is illegal. In other words, L can't consistently fool S all the time: eventually he is caught.

Solving this game (i.e. telling winning configurations from losing) amounts to answering whether the initial configuration is correct, i.e. whether $u(x)$ halts in $2^{|x|}$ steps, which requires $\Omega(|x|)$ steps. This Halting Game is artificial, still with a flavor of BHP, although it does not mention the exponent in its definition. We now reduce it to a nicer game (linear chess) to prove it exponential too.

3.3 Reductions; Non-Deterministic and Alternating TM; Time and 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 deterministic (ordinary) TM M accepts a string x if $M(x) =$ yes; a non-deterministic TM M does if there exists a driver d s.t. $M_d(x) =$ yes.

NTM represent single player games, puzzles, e.g. Rubik's Cube with a simple transition rule. We can compute the winning strategy in exponential time (exhausting all positions).

Home Work: Is there a P-time winning strategy for every such game? Nobody knows. Alternatively, show it requires exponential time. Grade A for the course and, probably, a senior faculty position at the university of your choice will be awarded for a solution.

The *alternating* TM (ATM) is a variation of the NTM driven by two alternating players l and r. A string is accepted if there is l such that for any $r : M_{l,r}(x) =$ yes. Our games could be viewed as ATM using a small space but up to an exponential time and returning the result of the game. It prompts l and r alternatingly to choose their moves (in several steps if the move is specified by several bits) and computes the resulting position, until a winner emerges. Accepted strings describe winning positions.

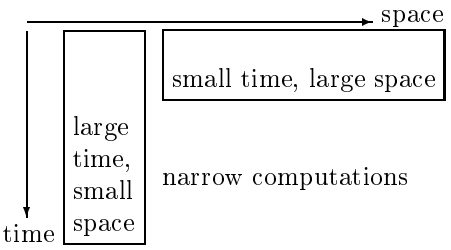
Linear Chess Simulation of TM-Game. We first simulate our Halting Game by *L-Chess*, a variant of Linear Chess. It has the same board: Weak ||||| ||||| Shy and, like regular chess, 6 ranks. Unlike Linear Chess, where only the vanquished piece is replaced, in L-chess the winning piece may also be replaced by ("promoted to") a piece of the same side; the gender bit is set to the side bit of the previous step, and an arbitrary table rather than the simple "Gender Rule" determines the winning piece.

The simulation is achieved simply by representing the Halting Game as an ATM computation simulated by the universal TM (using "=" commands for players' input). The UTM is viewed as an array of 1-pointer cellular automata: Weak cells as rightward, Shy leftward. The TM head is set to move upon termination to the end of the tape, so that no loser pieces are left. To transform L-Chess to Linear Chess it is left (as an exercise) to modify genders, extend ranks, and replace each transition by several, so that the winning piece is determined by the Gender Rule and is not replaced. So, any fast algorithm to solve Linear Chess, could be used to solve any game. Since Halting Game requires exponential time, so does the Linear Chess.

A Question Still Open: Space-Time Trade-off.

Deterministic linear space computations are games where any position has at most one (and easily computable) legal move. We know no general non-linear lower bound or subexponential upper bound for time to determine their outcome. This is the space-time trade-off problem. You met with such trade-offs using techniques like dynamic programming: saving time at the expense of space.

Recall that on a parallel machine: *time* is the number of steps until the last processor halts; *space* is the amount of memory 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. Let us call computations *narrow* if *either* time *or* space are polynomial, and *compact* if both (and, thus, volume too) are. Sec. 3.4 reduces the computations with *large* time and *small* space to (parallel) ones with *large* space and *small* time, and vice versa.



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 counter decremented at each move, limiting *the number of moves* to a polynomial. The sec. 3.1 algorithm can be implemented in parallel P-time for such games. Conversely, any narrowly computable predicate may be expressed as one determining the winning side of a fast game (similar to the Halting Game). Thus, fast games (i.e. compact alternating computations) correspond to narrow deterministic computations; general games (i.e. narrow alternating computations) correspond to large deterministic ones.

A Related Question: Do all exponential volume algorithms (e.g., one solving Linear Chess) allow an equivalent *narrow* computation? The two conjectures are mutually exclusive: otherwise we could solve the exponential-time Bounded Halting Problem in polynomial volume.

3.4 Fast and Lean Computations.

Based on [Chandra, Stockmeyer 1976], we now reduce the computations with *large* time and *small* space to parallel ones (PPM implemented by sorting networks) with *large* space and *small* time, and vice versa.

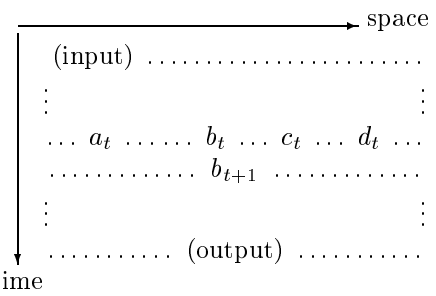
Parallelization. Suppose Professor has in his office a program (for a Small Machine with linear space and exponential time) solving the next exam problems. You break into his office shortly before the test and get the tape. Your time is very limited, insufficient to run the program - but wait! - also in the office you find the password that gains you access to a *really* Big Machine, one with essentially unlimited space resources (exponential number of parallel processors, memory, etc.). How can you, the devious student, exploit in small time this vast resource to solve the exam?

Generate simultaneously all possible configurations of the Small Machine with linear memory space, each as a separate process. There will be $M = n^{O(n)}$ of these configurations/processes, as there are $n^{O(n)}$ possible graphs with n nodes and $O(n)$ edges. Each configuration/process x computes a pointer $x \to x'$, where x and x' are successive configurations of Professor's Small Machine. Now, each process gets a copy of its successor's successor pointer: $x \to x' \to x''$ leads to $x \to x''$. Next, the single step pointers are erased and the procedure repeats for the 4-step pointers, 8-step pointers, etc.

If the Professor's Small Machine halts, then it cannot repeat a configuration and must stabilize in time $< M$. So, our pointer-compressing procedure will take at most $\log M = O(n \log n)$ steps, which is only P-time. Once it is complete, we need only take the input configuration for the test, and it will have a pointer to the answer configuration. The *volume* (time $*$ space) of computation is still vast. There is no way known to reduce volume to a polynomial, but you see how we can trade space for time.

Computing in Tight Space: a Pebble Game.

Consider now a *large* array M of interacting automata running in parallel for a P-time. We want to compute in polynomial space its output, however long it takes. Assume M to be a fixed connection network (which, we know, can simulate any PPM). The directed, acyclic graph at the right serves as a space-time diagram of the operation of M: rows represent time steps. Each node stores an *(event)*, i.e. the state of a particular automaton at a particular time. Each event is a function of its parents i.e. the events of the previous step in the automaton and its ($O(1)$, say, 3) neighbors. Here a_t, b_t, c_t and d_t represent the states at time t of b and its neighbors which determine b_{t+1}.



We will compute M's output (in the *central* automaton O) assuming the computation time n (i.e. the depth of the graph) is *small*. Each node can be described by a triplet (i, t, s), where i is the position of the automaton; s its state; t the time. The length of this triplet is *small*: $|s| = O(1)$; $|t| = O(\log(n))$. How long i's may we need? An automaton can only be relevant if it has time to propagate its information to O, i.e. is $\leq n$ links away from it. There are only 3^n of those. So: $|i| = O(\log(3^n)) = O(n)$.

We therefore can store each event in a *small* space. But it doesn't help if we need to store a *large* number of events. To know how many events we need to store we consider the *Pebble Game* with the following rules: The goal is to pebble (put a pebble in) a marked node O of a digraph; one can only pebble a node if all its parents are pebbled; there are k pebbles, they can be removed and reused.

Note that input nodes have no parents and can always be pebbled. You can win with $k = O(dn)$ (d: degree of the graph; n: its depth). The proof is by induction. Suppose you can pebble any node at level $t \geq 0$ with $1 + (d-1)t$ pebbles. Then you can pebble any node at level $t+1$ with $(d-1)t + d = 1 + (d-1)(t+1)$ pebbles. You just pebble each of the node's parents, leave a pebble there and reuse the rest of the pebbles for the next parent. Finally you put a pebble in the node itself. However, the time needed to pebble this graph may be *large* (you may have to traverse all descending paths).

Pebbling a node corresponds to computing an event in our diagram. Each event can be computed once its parents' are. k is actually the number of events we may have to store simultaneously. Since we can pebble the graph with $3n$ pebbles we can solve the problem in space $3n * |(i, t, s)|$. We transformed *large* space, *small* time into *small* space, *large* time. But the volume (space$*$time) is still *large*.

4 Nondeterminism; Inverting Functions; Reductions.

4.1 Example of a Narrow Computation: Inverting a Function.

Consider a P-time function F. For convenience, assume $|F(x)| = |x|$, (often it is enough if $|x|$ and $|F(x)|$ are bounded by polynomials of each other). Inverting F means given y, find $x \in F^{-1}(y)$, i.e. such that $F(x) = y$. There may be multiple solutions if F is many-to-one; we need to find only one. How?

We may try all possible x for $F(x) = y$. Assume F runs in linear time on a Pointer Machine. What is the cost of inverting F? The *space* used is $|x| + |y| + \text{space}_F(x) = O(|x|)$. But time is $O(|x|2^{|x|})$: absolutely infeasible. No method is currently proven much better in the worst case. And neither can we prove some inverting problems to require *super-linear* time. This is the sad present state of Computer Science!

An Example: Factoring. Let $F(x_1, x_2) = x_1 x_2$ be the the product of integers x_1, x_2 $|x_1| = |x_2|$. $|F(x)|$ is almost $|x|$. 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 $F(x)$. How many primes we might have to check? The density of n-bit prime numbers is approximately $1/(n \ln 2)$. So, factoring by exhaustive search takes *exponential* time! In fact, even the best known algorithms for this ancient problem run in time about $2^{\sqrt{|y|}}$, despite centuries of efforts by most brilliant people. The task is now commonly believed infeasible and the security of many famous cryptographic schemes depends on this *unproven* faith.

One-Way Functions: $x \xrightarrow{F} y$ are those easy to compute ($x \mapsto y$) and hard to invert ($y \mapsto x$) for most x. Even their existence is sort of a 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 one-way functions EXIST. Many theories and applications are based on this hypothetical existence.

Search and NP Problems.

Let us compare the inverting problems with another type: the search problems. They are, given x, to find w satisfying a given predicate $P(x, w)$ computable in time $|x|^{O(1)}$. Any inverting problem is a search problem and any search problem can be restated as an inverting problem. E.g., finding a Hamiltonian cycle C in a graph G, can be stated as inverting a $f(G, C)$, which outputs $G, 0 \ldots 0$ if C is in fact a Hamiltonian cycle of G. Otherwise, $f(G, C) = 0 \ldots 0$. There are two parts to a search problem, (a) decision problem: does w (called *witness*) exists, and (b) a constructive problem: actually find w.

A time bound for solving one of these types of problems gives a similar bound for the other.

Suppose a P-time $A(x)$ finds w satisfying $P(x, w)$ (if w exists). If A does not produce w within the time limit then it does not exist. So we can use the "witness" algorithm to solve the decision problem.

On the other hand, an algorithm deciding if the witness exists can be used to find it. Assume, given x and a predicate P, A quickly determines if there is w satisfying $P(x, w)$. Consider a predicate $P'(y, w)$ stating that $y = x, z \,\&\, P(x, zw)$. "Decision" answers for P' could then be used to find w for $P(x, w)$. First, take $z = 1$ and check the existence of w s.t. $P'(y, w)$ for $y = x, 1$. If it exists then a solution w for $P(x, w)$ begins with a 1. If not, w could only start with 0. Then do the same extending the found z with 1 and so on. You will find w in $|w|$ iterations. Unfortunately, for many problems such A is not known to exist.

The *language* of a problem is the set of all acceptable inputs. For the inverting problem it is the range of f. For the search problem it is the set of all x s.t. $P(x, w)$ holds for some w. An *NP language* is the set of all inputs acceptable by a P-time *non-deterministic* Turing Machine (sec. 3.4). All three classes of languages – search, inverse and NP – coincide. What NP machine accepts x if the search problem with input x and predicate P is solvable? This is just the machine which prompts the driver for digits of w and checks $P(x, w)$. Conversely, which P corresponds to a non-deterministic TM M? $P(x, w)$ just checks if M accepts x, when the driver chooses the states reflecting the digits of w.

Interestingly, polynomial *space* bounded deterministic and non-deterministic TMs have equivalent power. It is easy to modify TM to have a unique accepting configuration. Any acceptable string will be accepted in time $s2^s$, where s is the space bound. Then we need to check $A(x, w, s, k)$: whether the TM can be driven from the configuration x to w in time $< 2^k$ and space s. For this we need for every z, to check $A(x, z, s, k-1)$ and $A(z, w, s, k-1)$, which takes space $t_k \le t_{k-1} + |z|$. So, $t_k = O(sk) = O(s^2)$ [Savitch 1970].

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.

4.2 Complexity of NP Problems.

We determined that inversion, search, and NP types of problems are equivalent. Nobody knows whether *all* such problems are solvable in P-time (i.e. belong to P). This question (called P=?NP) is probably the most famous one in Theoretical Computer Science. All such problems are solvable in exponential time but it is unknown whether any better algorithm generally exists. For many problems the task of finding an efficient algorithm may seem hopeless, while similar or slightly relaxed problems can be solved. Examples:

1. Linear Programming: Given an integer $n \times m$ matrix A, find a rational vector x with $Ax > 0$. Note that if A contains k-bit coefficients and x exists then an x with $O(nk)$ bit numbers also exists.

   Solution: The Dantzig's *Simplex* algorithm finds x quickly for most A. Some A, however, take exponential time. After long frustrating efforts, a worst case P-time Ellipsoid Algorithm was finally found in [Yudin Nemirovsky 1976].

2. Primality test: Determine whether a given integer p has a factor?

   Solution: A bad (exponential time) way is to try all $2^{|p|}$ possible integer factors of p. More sophisticated algorithms, however, run fast (see section 5.1).

3. Graph Isomorphism: Problem: Given two graphs G_1 and G_2, is G_1 isomorphic to G_2? i.e. Can the vertices of G_1 be re-numbered so that it becomes equal G_2?

   Solution: Checking all $n!$ enumerations of vertices is not practical (for $n = 100$, this exceeds the number of particles in the known Universe). [Luks 1980] found an $O(n^d)$ steps algorithm where d is the degree. This is a P-time for $d = O(1)$.

Many other problems have been battled for decades or centuries and no P-time solution has been found. Even modifications of the previous three 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. Can be done in about exponential time $n^{\sqrt{n}}$. Seems very hard: Centuries of quest for fast algorithm were unsuccessful.

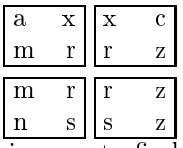
3. Sub-graph isomorphism: In a more general case where one graph may be isomorphic to a part of another graph, no P-time solution has been found.

We learned the proofs that Linear Chess and some other games have exponential complexity. None of the above or any other search/inversion/NP problem, however, have been proven to require super-P-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] [Trakhtenbrot]. A P-time function f reduces one NP-predicate $p_1(x)$ to $p_2(x)$ iff $p_1(x) = p_2(f(x))$, for all x. p_2 is NP-complete if *all* NP problems can be reduced to it. Thus, each NP-complete problem is as least as worst case hard as all other NP problems. This may be a good reason to give up on fast algorithms for it. Any P-time algorithm for one NP-complete problem would yield one for all other NP (or inversion, or search) problems. No such solution has been discovered yet and this is left as a homework (10 years deadline). What do we do when faced with an NP-complete problem? Sometimes one can restate the problem, find a similar one which is easier but still gives the information we really want, or allow more powerful means. Both of these we will do in Sec. 5.1 for factoring. Now we proceed with an example of NP-completeness.

4.3 An NP-Complete Problem: Tiling.

Example: NP-complete Tiling Problem. Invert the function which, given a tiled square, outputs its first row and the list of tiles used. A tile is one of the 26^4 possible squares containing a Latin letter at each corner. Two tiles may be placed next to each other if the letters on the mutual side are the same. E.g.:

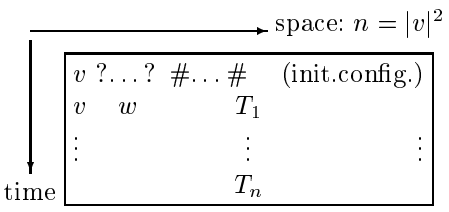


We now reduce any NP/search problem P to Tiling. Recall: A search problem is, given x, to find w which satisfies a P-time computable property $P(x, w)$. Existence of w is an NP problem since w can be "guessed" non-deterministically and verified in P-time.

Padding Argument. First, we need to reduce it to some "standard" NP problem. An obvious candidate is the problem [Is there? $w : U(v, w)$], where U is the universal Turing Machine, simulating $P(x, w)$ for $v = px$. A difficulty is that U does not run in P-time. We must restrict U to u which stops within some P-time limit. How to make this fixed degree limit sufficient for simulating any polynomial (even of higher degree) time P? Let the TM $u(v, w)$ for $v = 00\ldots01px$ simulate about $|v|^2$ steps of $U(px, w)$ (and, thus, of $P(x, w)$). If the *padding* of 0's in v is sufficiently long, u will have enough time to simulate P, even though u runs in quadratic time, while P's time limit may be, say, cube (of a shorter "un-padded" string). So the NP problem $P(x, w)$ is reduced to $u(v, w)$ by mapping instances x into $f(x) = 0\ldots01px = v$, with $|v|$ determined by the time limit for P. Notice that program p for $P(x, w)$ is fixed.

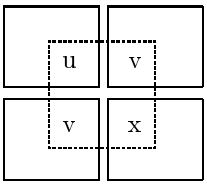
So, if some NP problem *cannot* be solved in P-time then neither can be the u-problem. Equivalently, if the problem [is there? $w : u(v, w)$] *IS* solvable in P-time then so is *any* search problem. We do not know which of these alternatives is true. It remains to reduce the search problem u to Tiling.

The Reduction. We compute $u(v, w)$ (where $v = 00\ldots01px$) 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 predicate P. Otherwise it enters an infinite loop. An instance x has a solution iff $u(v, w)$ runs forever for some w and $v = 0\ldots01px$. Here is the space-time diagram of computation of $u(v, w)$. We set n to u's time (and space) $|v|^2$. Each row in this table represents the configuration of u in a different moment of time. The solution w is filled in at the second step below a special symbol "?". Suppose somebody fills in a wrong table that doesn't reflect the actual computation. We claim that any wrong table has four adjacent squares that couldn't possibly 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 the wrong tables, the first 2 lines agree. The actual computation starts on the third line. Obviously, in the first mismatching line a transition of some cell from the previous line is wrong. This is visible from the state in both lines of this cell and the cell it points to, resulting in an impossible combination of four adjacent squares.

For a given x, the existence of w satisfying $P(x, w)$ is equivalent to the existence of a table with the prescribed first row, no halting state, and permissible patterns of each four adjacent squares. Conversion of our table to the *Tiling Problem*:
The squares in the table are separated by "—" ; the tiles by "..."; Break every square in the table into 4 parts, each part represents a corner of 4 separate tiles. If the 4 adjacent squares in the table are permissible, then the square is also tiled permissibly.



So, any P-time algorithm extending a given first line to the whole table of matching tiles from a given set would solve all NP problems by converting them to Tiling as shown.

5 Randomness in Computing.

5.1 A Monte-Carlo Primality Tester.

The factoring problem seems very hard. But to test a number for having factors turns out to be much easier than to find them. It also helps if we 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: [Rabin 1980], [Miller 1976], [Solovay, Strassen 1977].

Residue Arithmetic. $p|x$ means p is a divisor of x. $y = (x \mod p)$ denotes the residue of x when divided by p, i.e. $y \in [0, p-1]$, $p|(x-y)$. $x \equiv y \pmod p$ means $p|(x-y)$. Residues can be added, multiplied and subtracted with the result put back in the range $[0, p-1]$ (by adding an appropriate multiple of p). E.g., $-x$ means $p-x$ for residues mod p. We use $\pm x$ to mean either x or $-x$. If r and p have no common divisors > 1 (are mutually prime), division $(x/r \mod p)$ is possible, since $x \to (r * x \mod p)$ is one-to-one on $[0, p-1]$. Operations $+, -, *, /$ obey all usual arithmetical laws. $\gcd(x, y)$ is the greatest (and divisible by any other) common divisor of x and y. It can be found by Euclid's Algorithm: $\gcd(x, 0) = x$; $\gcd(x, y) = \gcd(y, (x \mod y))$, for $y > 0$. By induction, $g = \gcd(x, y) = A * x - B * y$, where integers $A = (g/x \mod y)$ and $B = (g/y \mod x)$ are produced as a byproduct of Euclid's Algorithm.

We will need to compute $(x^q \mod p)$ in polynomial time. We cannot multiply x q times, since it takes $q > 2^{|q|}$ steps. Instead we compute all numbers $x_i = (x_{i-1}^2 \mod p) = (x^{2^i} \mod p), i < |q|$. Then we represent q in binary, i.e. as a sum of powers of 2 and multiply mod p the needed x_i's.

Fermat Test. The Little Fermat Theorem for every $x \in [1, p-1]$ and prime p says: $x^{(p-1)} \equiv 1 \pmod p$. Indeed, the sequence $(xi \mod p)$ is a permutation of $i = 1, \ldots, p-1$. So, $1 \equiv (\prod_{i<p}(xi))/(p-1)! \equiv x^{p-1} (\mathrm{mod} p)$.

This test rejects typical composite p. Other composite p can be actually factored by the following test:

Square Root Test.

Lemma: For each y and prime p, the equation $(x^2 \mod p) = y$ 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 $x^2 - x'^2 = (x - x') * (x + x') \equiv 0 \pmod p$. We have a product of two integers which is congruent to 0, i.e. divisible by p. Therefore p must divide at least one of the factors. (Otherwise p is composite, and $\gcd(p, x + x')$ *actually gives* us its factor). So, either $x - x' \equiv 0 \pmod p$ or $x + x' \equiv 0 \pmod p$ i.e. $x \equiv \pm x' \pmod p$. **End of proof.**

In particular $x^2 \equiv 1 \pmod p$ implies $x \equiv \pm 1$. Note that this does NOT hold if p is composite, since its factors can be spread between $(x - x')$ and $(x + x')$. Then y could have more than one $\pm$ pair of roots.

Rabin-Miller Test. Let us combine these tests into $T(x, p)$ which uses given x to prove p is composite. Let $p-1 = q * 2^k$, with odd q. T sets $x_0 = (x^q \mod p)$, $x_i = (x_{i-1}^2 \mod p) = (x^{q*2^i} \mod p)$, $i \le k$. If $x_0 = 1$, or one of x_i is -1, T gives up. If $x_k \neq 1$, Fermat Test rejects p. Otherwise there is $z = x_i \neq \pm 1$, such that $(z^2 \mod p) = x_{i+1} = 1$. Then the Square Root Test factors p.

First, for each odd composite p, we show that T succeeds with *some* x, mutually prime with p. If $p = a^j, j > 1$, then $x = (1 + p/a)$ is good for T : $(1 + p/a)^{p-1} = 1 + (p/a)(p-1) + (p/a)^2 * (p-1)(p-2)/2 + \ldots \equiv 1 - p/a \neq 1 \pmod p$, since $(p/a)^2 \equiv 0 \pmod p$. So the Fermat Test works. Otherwise $p = a * b$, $\gcd(a, b) = 1$. Take the **greatest** $i \le k$ such that $x_i \neq 1$, for some x mutually prime with p. Such i exists, since $(-1)^q \equiv -1$ for odd q. If $i < k$ then $(x_i)^2 \equiv 1 \pmod p$. Take $x' = 1 + b * (1/b \mod a) * (x-1)$. Check: $x' \equiv 1 \equiv x'_i \pmod b$, while $x'_i \equiv x_i \pmod a$. Thus, either x_i or x'_i is not ± 1.

Now, $T(y, p)$ succeeds on step i with *most* y: function $y \mapsto x * y$ is 1-1 and T cannot fail with both y and $x * y$. This test can be repeated for many randomly chosen x. Each time T fails, we are twice more sure that p is prime. The probability that T fails 300 times on a composite p is $< 2^{-300} < 1/N$, where N is the number of particles in the known Universe.

5.2 Randomized Algorithms and Random Inputs.

Las-Vegas algorithms, unlike Monte-Carlo, never give wrong answers. 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 is popular due to its simplicity. It sorts an array $a[1..n]$ of > 2 numbers by choosing in it a random *pivot*, splitting the remaining array in two by comparing with the pivot, and calling itself recursively on each half.

For easy reference, replace the array entries with their positions $1, ..., n$ 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 $t(i), ..., t(j)$. This has 2 out of $|j - i| + 1$ chances. So, the expected number of comparisons is $\sum_{i,j>i} 2/|1 + j - i| = 3 - n + (n + 1) \sum_{k=3}^{n} 2/k = 2n \ln n - O(n)$. Note, that the expectation of the sum of variables is the sum of their expectations (not true, say, for product).

There is also a Las-Vegas primality tester, but it goes far beyond the scope of these notes.

The above Monte-Carlo and Las-Vegas algorithms require choosing strings *at random* with uniform distribution. We mentally picture that as flipping a coin. (Computers use *pseudo-random generators* rather than coins in hope, rarely supported by proofs, that their outputs have all the statistical properties of truly random coin flips needed for the analysis of the algorithm.)

Random Inputs to Deterministic Algorithms are analyzed similarly to algorithms which flip coins themselves and the two should not be confused. Consider an example: Someone is interested in knowing whether or not certain graphs contain Hamiltonian Cycles. He offers graphs and pays \$100 if we show either that the graph *has* or that it *has not* Hamiltonian Cycles. Hamiltonian Cycle problem is NP-Complete, so it should be very hard for *some*, but not necessarily for *most* graphs. In fact, if our patron chooses the graphs uniformly, a fast algorithm can earn us the \$100 *most of the time*! Let all graphs have n nodes and, say, $k(n) < n \ln n/4$ edges and be equally likely. Then we can use the following (deterministic) algorithm: output "**No** Hamiltonian Cycles" and collect the \$100, if the graph has an isolated node. 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 $(1 - 2/n)^k > (1 - O(1/n))/\sqrt{n}$. Thus, the probability that *none* of n nodes is isolated (and we lose our \$100) is $O((1 - 1/\sqrt{n})^n) = O(e^{-\sqrt{n}})$ and vanishes fast. Similar calculations can be made whenever $r = \lim 2k(n)/(n \ln n) < 1$. If $r > 1$, other fast algorithms can actually find a Hamiltonian Cycle. See: [Johnson 1984], [Karp 1976], [Gurevich 1985]. See also [Venkatesan, Levin] for a proof that another problem is NP-complete even on average. How do this HC algorithm and the above primality test differ?

- The primality algorithm works for *all* instances. It tosses the coin itself and can repeat it for a more reliable answer. The HC algorithm only works for *most* instances (with isolated nodes).
- In the HC algorithm, we must trust the opponent to follow the presumed random procedure. If he cheats and produces connected graphs often, our analysis breaks down.

Symmetry Breaking. Randomness comes into Computer Science in many other ways besides those we considered. Here is a simple example: avoiding conflicts for shared resources.

Several philosophers dine at a circular table. Before each of them is a plate, and left of each plate is either a knife or a fork arranged so that each diner has a knife on his right and a fork on his left or vice versa. The problem is that neighboring diners must share the utensils: neighbors cannot eat at the same time. How can the philosophers complete the dinner given that all of them must act in the same way without any central organizer? Trying to grab the knives and forks at once may turn them into fighting philosophers. Instead they could each flip a coin, and sit still if it comes up heads, otherwise try to grab the utensils. If two diners try to grab the same utensil, neither succeeds. If they repeat this procedure enough times, most likely each philosopher will eventually get both a knife and fork without interference.

We have no time to actually analyze this and many other scenaria, where randomness is crucial. Instead we will take a look into the concept of Randomness itself.

5.3 Randomness and Complexity.

The obvious definition of a random sequence is one that has the same properties as a sequence of coin flips. But this definition leaves the question, what *are* these properties? Kolmogorov resolved these problems with a new definition of random sequences: those with no description shorter than their full length. See survey and history in [Kolmogorov, Uspensky 1987], [Li, Vitanyi 1993].

Kolmogorov Complexity $K_A(x/y)$ of the string x given y is the length of the shortest possible program p which lets algorithm A transform y into x: $\min\{(|p|) : A(p,y) = x\}$. There exists a Universal Algorithm U such that, $K_U(x) < K_A(x) + O(1)$, for every algorithm A. This constant $O(1)$ is bounded by the length of the program U needs to simulate A. We abbreviate $K_U(x/y)$ as $K(x/y)$ or $K(x)$, for empty y.

An example: For $A : x \mapsto x$, $K_A(x) = |x|$, so $K(x) < K_A(x) + O(1) < |x| + O(1)$.

Can we compute $K(x)$? One could try all programs p, $|p| < |x|+O(1)$ and find the shortest one generating x. This does not work because some programs diverge, and the halting problem is unsolvable. In fact, no algorithm can compute K or even any its lower bounds except $O(1)$.

Consider an old paradox expressed in the following phrase: "The smallest integer which 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 that cannot be expressed by such phrases. Then there is the smallest such integer, but isn't it described by the above phrase?

A similar argument proves that K is not computable. Suppose an algorithm $L(x) \neq O(1)$ computes a lower bound for $K(x)$. We can use it to compute $f(n)$ that finds x with $n < L(x) < K(x)$, but $K(x) < K_f(x) + O(1)$ and $K_f(f(n)) \leq |n|$, so $n < K(f(n)) < |n| + O(1) = \log O(n) \ll n$: a contradiction. So, K and its non-constant lower bounds are not computable.

A nice 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.

Deficiency of Randomness.

There are ways to estimate complexity that are correct in some important cases, but not always. One such case is when a string x is generated at random. Let us define $d(x) = |x| - K(x/|x|)$; $(d(x) > -O(1))$.

What is the probability of $d(x) > i$, for $|x| = n$? There are 2^n strings x of length n. If $d(x) > i$, then $K(x/|x|) < n - i$. There are $< 2^{n-i}$ programs of such length, generating $< 2^{n-i}$ strings. So, the probability of such strings is $< 2^{n-i}/2^n = 2^{-i}$ (regardless of n)! So even for $n = 1,000,000$, the probability of $d(x) > 300$ is absolutely negligible (provided x was indeed generated by fair coin flips). We call $d(x)$ the ***deficiency of randomness*** of a string x with respect to the uniform probability distributions on all strings of that length.

If $d(x)$ is small then x satisfies all other reasonable properties of random strings. Indeed, consider a property "$x \notin P$" with enumerable $S = \neg P$ of negligible probability. Let S_n be the number of strings of length n, violating P and $\log(S_n) = s_n$. What can be said about the complexity of all strings in S? S is enumerable and sparse (has only S_n strings). To generate x, using the algorithm enumerating S, one needs only the position i of x in the enumeration of S. We know that $i \leq S_n \ll 2^n$ and, thus, $|i| \leq s_n \ll n$. Then the deficiency of randomness $d(x) > n - s_n$ is large. Every x which violates P will, thus, also violate the "small deficiency of randomness" requirement. In particular, the small deficiency of randomness implies unpredictability of random strings: A compact algorithm with frequent prediction would assure large $d(x)$. Sadly, the randomness can not be detected: we saw, K and its lower bounds are not computable.

Rectification of Distributions. We rarely have a source of randomness with precisely known distribution. But there are very efficient ways to convert "imperfect" random sources into perfect ones. Assume, we have a "junk-random" sequence with weird unknown distribution. We only know that its long enough (m bits) segments have min-entropy $> k + i$, i.e. probability $< 1/2^{k+i}$, given all previous bits. (Without such m we would not know a segment needed to extract even one not fully predictable bit.) We can fold X into an $n \times m$ matrix. No relation is required between n, m, i, k, but useful are small m, i, k and huge $n = o(2^k/i)$. We also need a small $m \times i$ matrix Z, independent of X and really uniformly random (or random Toeplitz, i.e. with restriction $Z_{a+1,b+1} = Z_{a,b}$). Then the $n \times i$ product XZ has uniform distribution with accuracy $O(\sqrt{ni/2^k})$. This follows from [Goldreich, Levin], which uses earlier ideas of U. and V. Vazirani.

5.4 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* that making a true random generator is impossible: we just saw efficient ways to perfect the distributions of biased random sources. The reason lies in many extra benefits provided by pseudorandom generators. E.g., when experimenting with, debugging, or using a program one often needs to repeat the exact same sequence. With a truly random generator, one actually has to record all its outcomes: long and costly. The alternative is to generate *pseudo-random* strings from a short seed. Such methods were justified in [Blum Micali], [Yao]:

First, take any one-way permutation $F_n(x)$ (see sec. 5.5) with a *hard-core* bit (see below) $B_p(x)$ which is easy to compute from x, p, but infeasible to guess from $p, n, F_n(x)$ with any noticeable correlation.
Then take a random *seed* $x_0, p, n \in \{0,1\}^k$ and Repeat: $(S_i \leftarrow B_p(x_i);\ x_{i+1} \leftarrow F_n(x_i);\ i \leftarrow i+1)$.

We will see how distinguishing outputs S of this generator from strings of coin flips would imply a fast inverting of F (believed impossible).

But if P=NP (a famous open problem), no one-way F, and no pseudorandom generators could exist.

By Kolmogorov's standards, pseudo-random strings are not random: let G be the generator; s be the seed, $G(s) = S$, and $|S| \gg k = |s|$. Then $K(S) \leq O(1) + k \ll |S|$, so this violates Kolmogorov's definition. We can distinguish between truly and pseudo- random strings by simply trying all short seeds. However this takes time exponential in the seed length. Realistically, a pseudo-random string will be as good as a truly random one if they can't be distinguished in feasible time. Such generators we call *perfect*.

Theorem: [Yao] Let $G(s) = S \in \{0,1\}^n$ run in time t_G. Let a probabilistic algorithm A in expected (over internal coin flips) time t_A accept $G(s)$ and truly random strings with different by d probabilities. Then, for random i, one can use A to guess S_i from $S_{i+1}, S_{i+2}, \ldots$ in time $t_A + t_G$ with correlation $d/O(n)$.

Proof: Let p_i be the probability that A accepts $S = G(s)$ modified by replacing its first i digits with truly random bits. Then p_0 is the probability of accepting $G(s)$ and must differ by d from the probability p_n of accepting random string. Then $p_{i-1} - p_i = d/n$, for randomly chosen i. Let $P_0(x)$ and $P_1(x)$ be the probabilities of acceptance of $r0x$ and $r1x$ for random r of length $i-1$. Then $(P_1(x)+P_0(x))/2$ averages to p_i for $x = S_{i+1}, S_{i+2}, \ldots$, while $P_{S_i}(x) = P_0(x)+(P_1(x)-P_0(x))S_i$ averages to p_{i-1} and $(P_1(x)-P_0(x))(S_i-1/2)$ to $p_{i-1} - p_i = d/n$. So, $P_1(x) - P_0(x)$ has the stated correlation with S_i. Q.E.D.

If the above generator was not perfect, one could guess S_i from the sequence $S_{i+1}, S_{i+2}, \ldots$ with a polynomial (in $1/|s|$) correlation. But, $S_{i+1}, S_{i+2} \ldots$ can be produced from p, n, x_{i+1}. So, one could guess $B_p(x_i)$ from $p, n, F(x_i)$ with correlation d/n, which cannot be done for hard-core B.

Hard Core. The key to constructing a pseudorandom generator is finding a hard core for a one-way F. The following B is hard-core for any one-way F, e.g., for Rabin's OWF in sec. 5.5. [Knuth 1997, v.2, 3d ed., Chap.3.5.F Pseudorandom numbers, pp.36, 170-179] has more details and references.

Let $B_p(x) = (x \cdot p) = (\sum_i x_i p_i \bmod 2)$. [Goldreich Levin] converts any method g of guessing $B_p(x)$ from $p, n, F(x)$ with correlation ε into an algorithm of finding x, i.e. inverting F (slower ε^2 times than g).

Proof. Take $k = |x| = |y|$, $j = \log(2k/\varepsilon^2)$, $v_i = 0^i 10^{k-i}$. Let $B_p(x) = (x \cdot p)$ and $b(x,p) = (-1)^{B_p(x)}$.

Assume, for $y = F_n(x)$, $g(y,p,w) \in \{\pm 1\}$ guesses $B_p(x)$ with correlation $\sum_p 2^{-|p|} b(x,p) g_p > \varepsilon$, where g_p abbreviates $g(y,p,w)$, since w, y are fixed throughout the proof.

Averaging $(-1)^{(x\cdot p)} g_p$ over $> 2k/\varepsilon^2$ random pairwise independent p deviates from its average by $< \varepsilon$ (and so is > 0) with probability $> 1 - 1/2k$. The same for $(-1)^{(x\cdot[p+v_i])} g_{p+v_i} = (-1)^{(x\cdot p)} g_{p+v_i} (-1)^{x_i}$.

Take a random matrix $P \in \{0,1\}^{k\times j}$. Then the vectors Pr, $r \in \{0,1\}^j \setminus \{0^j\}$ are pairwise independent. So, for a fraction $\geq 1 - 1/2k$ of P, $\mathrm{sign}\sum_r (-1)^{xPr} g_{Pr+v_i} = (-1)^{x_i}$. We could thus find x_i for all i with probability $> 1/2$ if we knew $z = xP$. But z is short: we can try all 2^j possible values!

So the inverter, for a random P and all i, r, computes $G_i(r) = g_{Pr+v_i}$. It uses Fast Fourier on G_i to compute $h_i(z) = \sum_r b(z,r) G_i(r)$. The sign of $h_i(z)$ is the i-th bit for the z-th member of output list. Q.E.D.

5.5 Cryptography.

Rabin's One-way Function. Pick random prime numbers $p, q, |p| = |q|$ with 2 last bits $= 1$, i.e. with odd $(p-1)(q-1)/4$. Then $n = p * q$ is called a Blum number. Its length should make factoring infeasible.

Let $Q(n)$ be the set of *quadratic residues*, i.e. numbers of the form $(x^2 \bmod n)$.

Lemma. If $n = pq$ is a Blum number then $F : x \mapsto (x^2 \bmod n)$ is a permutation of $Q(n)$.

Proof: Let $x = F(z) \in Q(n)$ and $y = F(x)$. Let $t = (p-1)(q-1)/4$. It is odd, so $u = u(n) = (t+1)/2$ is an integer. Since $2t$ is a multiple of both $p-1$ and $q-1$, according to the Little Fermat Theorem $x^t - 1 \equiv z^{2t} - 1$ divides both p and q (and, thus n). Then $y^u \equiv x^{2u} = xx^t \equiv x \pmod n$. Q.E.D.

Lemma. Inverting F on random x is equivalent to factoring n.

Proof. Let $F(A(y)) = y$ for a fraction ε of $y \in Q(n)$. Then $F(A(y)) = y = F(x)$, while $A(y) \neq \pm x$ for a fraction $\varepsilon/2$ of x. The Square Root Test would factor n given any such $x, A(F(x))$ which could be found in about $2/\varepsilon$ random trials. Conversely, knowing a secret (factors of n) makes inverting F easy; such one-way permutations, called "trap-door," have many applications, such as cryptography (see below).

Picking a prime number is easy since primes have density $1/O(|p|)$. Indeed, one can see that $\binom{2n}{n}$ is divisible by every prime $p \in [n, 2n]$ but by no prime power $p^i > 2n$. So, $(\log \binom{2n}{n}) / \log n = 2n/\log n - o(1)$ is an upper bound on the number of primes in $[n, 2n]$ and a lower bound on that in $[1, 2n]$.

Note that fast VLSI circuits exist to multiply large 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 the same length k. The resulting encryption $m \oplus S$ has the same uniform probability distribution, no matter what m is. So it is useless for the adversary who wants to learn something about m, without knowing S. A disadvantage is that the communicating parties must share a secret S as large as all messages to be exchanged combined. *Public Key* Cryptosystems use two keys. One key is needed to encrypt the messages and may be completely disclosed to the public. The *decryption* key must still be kept secret, but 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 1982] by replacing the above random S by pseudorandom $S_i = (s_i \cdot x)$; $s_{i+1} = (s_i^2 \bmod n)$. Here a Blum number $n = pq$ is chosen by the Decryptor and is public, but p, q are kept secret. The Encryptor chooses x, s_0 at random and sends $x, s_k, m \oplus S$. Assuming factoring is intractable for the adversary, S should be indistinguishable from random strings (even when s_k is known). Then this scheme is as secure as if S were random. The Decryptor knows p, q and can compute u, t (see above) and $v = (u^{k-1} \bmod t)$. So, he can find $s_1 = (s_k^v \bmod n)$, and then S and m.

Another use of the intractability of factoring is digital signatures [Rivest, Shamir, Adleman 1978], [Rabin, 1979]. Strings x can be released as authorizations of $y = (x^2 \bmod n)$. Anyone can verify x, but nobody can forge it since only the legitimate user knows factors of n and can take square roots.

Go On!

You noticed that 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 the 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! The trains of thought which led to the discovery are blood and guts: keep them hidden. Other vital parts, like open problems, I skip out of modesty.

Writing Contributions. Section 1 was prepared by Elena Temin, Yong Gao and Imre Kifor (BU), others by Berkeley students: 2.3 by Mark Sullivan, 3.0 by Eric Herrmann, 3.1 by Elena Eliashberg, 3.2 by Wayne Fenton and Peter Van Roy, 3.3 by Carl Ludewig, 3.4 by Sean Flynn, 3.4 by Francois Dumas, 4.1 by Jeff Makaiwi, 4.1.1 by 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, 5.3 by Carl Ludewig, 5.4 by Sean Flynn, Francois Dumas, Eric Herrmann, 5.5 by Brian Jones.

References

[1] On-line bibliographies. http://theory.lcs.mit.edu/~dmjones/hbp.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest. Introduction to Algorithms. 1990, McGraw-Hill.

[3] Donald E. Knuth. *The Art of Computer Programming.* Vol. 1-3. Addison-Wesley, 3d ed., 1997. Additions to v.2 can be found in http://www-cs-faculty.stanford.edu/~knuth/err2-2e.ps.gz .

[4] William Feller. *An Introduction to Probability Theory and Its Applications.* John Wiley & Sons, 1968.

[5] S. Lang. *Algebra.* 3rd ed. 1993, Addison-Wesley.

[6] H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill Book Co., 1967.

[] References for section 1:

[7] Ja.M. Barzdin', Ja.Ja. Kalnin's. A Universal Automaton with Variable Structure. *Automatic Control and Computing Sciences.* 8(2):6-12, 1974.

[8] E.R. Berlekamp, J.H. Conway, R.K. Guy. *Winning Ways.* Section 25. Academic Press, 1982.

[9] 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.

[10] A. Schönhage. Storage Modification Machines. *SIAM J. on Computing* 9(3):490-508, 1980.

[11] Yu. Ofman. A Universal Automaton. *Transactions of the Moscow Math. Society,* pp.200-215, 1965.

[] Section 2:

[12] M. Blum. A machine-independent theory of the complexity of recursive functions. *J. ACM* 14, 1967.

[13] M. Davis, ed. *The Undecidable.* Hewlett, N.Y. Raven Press, 1965.
(The reprints of the original papers of K.Gödel, A.Turing, A.Church, E.Post and others).

[14] Shinichi Ikeno. A 6-symbol 10-state Universal Turing Machine. *Proc. Inst. of Elec. Comm.* Tokyo, 1958.

[15] Joel I. Seiferas, Albert R. Meyer. Characterization of Realizable Space Complexities.
Annals of Pure and Applied Logic 73:171-190, 1995.

[16] 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.

[17] 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:

[18] J. v.Neumann, O. Morgenstern. *Theory of Games and Economic Behavior.* Princeton Univ. Press, 1944.

[19] A.K.Chandra and L.J.Stockmeyer, Alternation. *FOCS*-1976.

[20] Ashok K. Chandra, Dexter C. Kozen, Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114-133, 1981.

[21] J.M. Robson. N by N checkers is EXPTIME-complete. *SIAM J. Comput* 13(2), 1984.

[22] J.M. Robson. The complexity of Go. *Proc. 1983 IFIP World Computer Congress*, p. 413-417.

[23] A.S. Fraenkel and 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:

[24] W.J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.* 4:177-190, 1970.

[25] 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.

[26] E.M. Luks: Isomorphism of Graphs of Bounded Valence Can Be Tested in Polynomial Time. *FOCS*-1980.

[27] M.R.Garey, D.S.Johnson. *Computers and Intractability.* W.H.Freeman & Co. 1979.

[28] 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:

[29] M.O.Rabin. Probabilistic Algorithms for Testing Primality. *J. Number Theory*, 12: 128-138, 1980.

[30] G.L.Miller. Riemann's Hypothesis and tests for Primality. *J. Comp. Sys. Sci.* 13(3):300-317, 1976.

[31] R. Solovay, V. Strassen. A fast Monte-Carlo test for primality. *SIComp* 6:84-85, 1977.

[32] R. Karp. Combinatorics, Complexity and Randomness. (Turing Award Lecture) *Communication of the ACM*, 29(2):98-109, 1986.

[33] David S. Johnson. The NP-Completeness Column. *J. of Algorithms* 5, (1984) 284-299.

[34] R. Karp. The probabilistic analysis of some combinatorial search algorithms. *Algorithms and Complexity.* (J.F.Traub, ed.) Academic Press, NY 1976, pp. 1-19.

[35] Y. Gurevich, Average Case Complexity, Internat. Symp. on Information Theory, IEEE, Proc. 1985.

[36] 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.

[37] M. Li, P.M.B. Vitányi. Introduction to Kolmogorov Complexity and its Applications. Springer Verlag, New York, 1993.

[38] M. Blum, S. Micali. How to generate Cryptographically Strong Sequences. *SIAM J. Comp.*, 13, 1984.

[39] A. C. Yao. Theory and Applications of Trapdoor Functions. *FOCS*-1982.

[40] O.Goldreich, L.Levin. A Hard-Core Predicate for all One-Way Functions. *STOC*-1989, pp. 25-32.

[41] R.Rivest, A.Shamir, L.Adleman. A Method for Obtaining Digital Signature and Public-Key Cryptosystems. *Comm. ACM*, 21:120-126, 1978.

[42] M. Blum, S. Goldwasser. An Efficient Probabilistic Encryption Scheme Hiding All Partial Information. *Crypto*-1982.

[43] M. Rabin. *Digitalized Signatures as Intractable as Factorization.* MIT/LCS/TR-212, 1979.

[44] R.Venkatesan, L.Levin. Random Instances of a Graph Coloring Problem are Hard. *STOC*-1988.