

Computational Model

Slides based on *S.Aurora, B.Barak. Complexity Theory: A Modern Approach.*

Ahto Buldas

Ahto.Buldas@ut.ee

Some Citations

“The idea behind digital computers may be explained by saying that these machines are intended to carry out any operations which could be done by a human computer. The human computer is supposed to be following fixed rules; he has no authority to deviate from them in any detail. We may suppose that these rules are supplied in a book, which is altered whenever he is put on to a new job. He has also an unlimited supply of paper on which he does his calculations.” Alan Turing, 1950

“[Turing] has for the first time succeeded in giving an absolute definition of an interesting epistemological notion, i.e., one not depending on the formalism chosen.” Kurt Gödel, 1946

Importance of Turing Machine

Surprising discovery I (in 1930s): all known computational models are able to simulate each other, i.e. *the set of computable problems does not depend upon the computational model.*

How about computational efficiency? ... Even a kid knows that whether or not a new video game program is “efficiently computable” depends upon his computers hardware.

Surprising discovery II (in 1960s): Turing Machine (as a single abstraction) seems able to simulate all physically realizable computational models with very little loss of efficiency, i.e. *the set of “efficiently computable” problems is at least as large for the Turing Machine as for any other model.*

Quantum computer is a possible exception, but it is not known if it is physically realizable.

Agenda of Lecture 1

Turing machine—an embodiment of the intuition that computation consists of applying mechanical rules to manipulate numbers, by using a scratch pad on which to write intermediate results.

The Turing Machine can be also viewed as the equivalent of any modern programming language—this intuition will suffice for most of the course.

During Lecture 1 we:

- Give a formal definition of the Turing Machine and its running time
- Present the important notion of the universal Turing machine.
- Introduce a class of “efficiently computable” problems called P
- Define many other models (nondeterministic, probabilistic and quantum Turing machines, Boolean circuits, parallel computers, decision trees, and communication games).

Representing Objects as Strings

In general we study the complexity of computing functions whose input and output are finite strings of bits.

The set of all strings of length n is denoted by $\{0, 1\}^n$, while the set of all strings is denoted by $\{0, 1\}^* = \cup_{n \geq 0} \{0, 1\}^n$.

Integers can be represented by binary expansion (e.g., 34 is represented as 100010), a graph as its adjacency matrix (i.e., an n vertex graph G is represented by an $n \times n$ 0/1-valued matrix \vec{A} such that $\vec{A}_{i,j} = 1$ iff the edge (i, j) is present in G).

We will avoid low level issues of representation, and will use $\lfloor x \rfloor$ to denote some canonical binary representation of x . Often we simply use x to denote both the object and its representation.

Pairs and Tuples

We denote by $\langle x, y \rangle$ the ordered pair consisting of x and y .

A canonical representation for $\langle x, y \rangle$ can be easily obtained from the representations of x and y .

For example, we can first encode $\langle x, y \rangle$ as the string $\lfloor x \rfloor \circ \# \circ \lfloor y \rfloor$ over the alphabet $\{0, 1, \#\}$ (where \circ denotes concatenation) and then use the mapping $0 \mapsto 00$, $1 \mapsto 11$, $\# \mapsto 01$ to convert this into a string of bits.

To reduce notational clutter, instead of $\lfloor \langle x, y \rangle \rfloor$ we use $\langle x, y \rangle$ to denote not only the pair consisting of x and y but also the representation of this pair as a binary string.

Similarly, we use $\langle x, y, z \rangle$ to denote both an ordered triple and its representation, and use similar notation for 4-tuples, 5-tuples etc..

Decision Problems and Languages

An important special case of functions mapping strings to strings is the case of Boolean functions, whose output is a single bit. We identify such a function f with the set $L_f = \{x: f(x) = 1\}$ and call such sets *languages* or *decision problems*.

We identify the computational problem of computing f (i.e., given x compute $f(x)$) with the problem of *deciding the language* L_f (i.e., given x , decide whether $x \in L_f$).

Example: By representing the possible invitees to a dinner party with the vertices of a graph having an edge between any two people that can't stand one another, the dinner party computational problem from the introduction becomes the problem of finding a maximum sized independent set:

$$\text{INDSET} = \{ \langle G, k \rangle : \exists S \subset V(G) [|S| \geq k, \forall u, v \in S (\overline{uv} \notin E(G))] \} .$$

Big-Oh Notation

Typically, the efficiency of an algorithm is measured as the maximum number $T(n)$ of basic operations it performs in case of input of length n .

The exact definition of $T(n)$ depends on the notion of “basic operation”. For example, addition will take three times more steps if the bit-operations are defined as “basic” ... we do not want to be that precise ...

For any two functions $f: \mathbb{N} \rightarrow \mathbb{N}$ and $g: \mathbb{N} \rightarrow \mathbb{N}$:

- $f = O(g)$ iff $\exists c \in \mathbb{R}$ such that $f(n) \leq c \cdot g(n)$ for sufficiently large n ;
- $f = \Omega(g)$ iff $g = O(f)$;
- $f = \Theta(g)$ iff both $f = O(g)$ and $f = \Omega(g)$;
- $f = o(g)$ iff $\forall \epsilon > 0$ we have $f(n) \leq \epsilon \cdot g(n)$ for sufficiently large n ;
- $f = \omega(g)$ iff $g = o(f)$.

Modeling Computation and Efficiency

Let f be a function that takes a string of bits (i.e., a member of the set $\{0, 1\}^*$) and outputs, say, either 0 or 1.

Informally, an *algorithm for computing f* is a set of mechanical rules, that allow to compute $f(x)$ given any input $x \in \{0, 1\}^*$. The set of rules is fixed, though each rule may be applied arbitrarily many times.

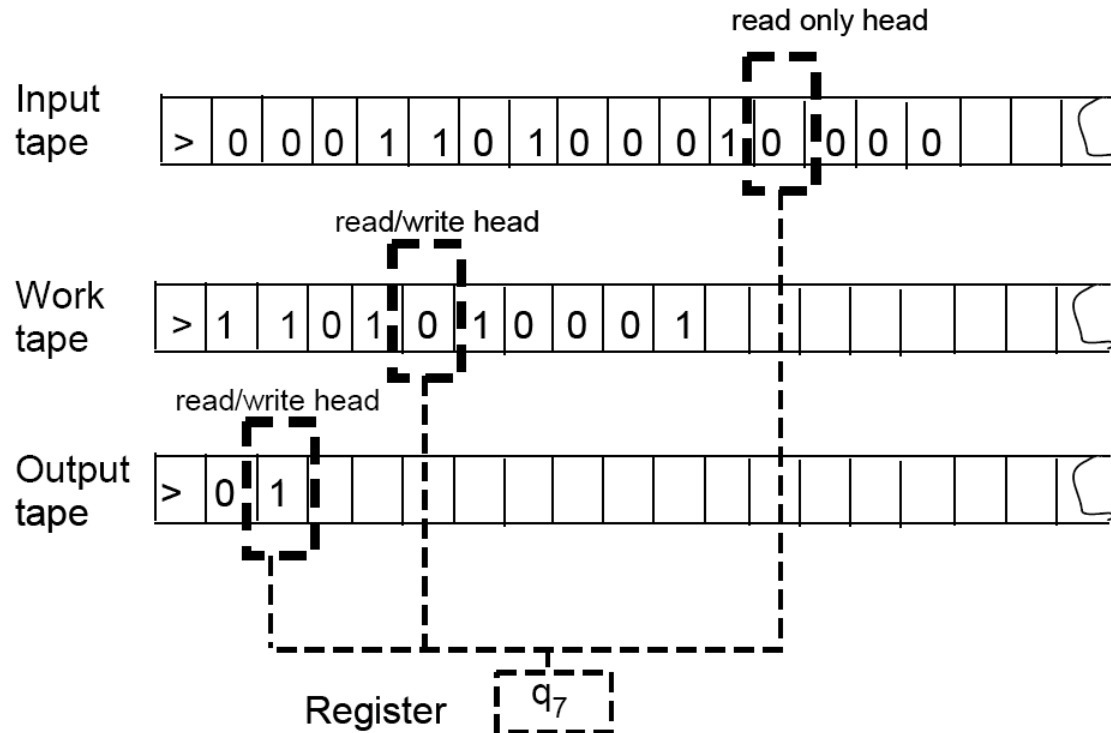
Each rule involves one or more of the following “elementary” operations:

- Read a bit of the input.
- Read a bit (or a symbol from a larger alphabet, say $\{0, \dots, 9\}$) from the “scratch pad”.
- Write a bit/symbol to the scratch pad.
- Either stop and output 0 or 1, or choose the rule that will be applied next.

The running time is the number of these basic operations performed.

The Turing Machine

The *k-tape Turing Machine* is a concrete realization of this informal notion:



Formal Definition of a Turing Machine

Formally, a TM M is described by a tuple (Γ, Q, δ) containing:

- A set Γ of the symbols that M 's tapes can contain. We assume that Γ contains a designated “blank” symbol, denoted by \square , a designated “start” symbol, denoted by \triangleright and the numbers 0 and 1. We call Γ the *alphabet* of M .
- A set Q of possible states M 's register can be in. We assume that Q contains a designated start state, denoted by q_{start} and a designated halting state, denoted by q_{halt} .
- A function $\delta: Q \times \Gamma^k \rightarrow Q \times \Gamma^{k-1} \times \{L, S, R\}^k$ describing the rule M uses in performing each step. This function is called the *transition function* of M .

Transition Function as a Table

IF			THEN			
input symbol read	work/output tape symbol read	current state	move input head	new work/output tape symbol	move work/output tape	new state
⋮	⋮	⋮	⋮	⋮	⋮	⋮
a	b	q	→	b'	←	q'
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Transition function $\delta: Q \times \Gamma^2 \rightarrow Q \times \Gamma^1 \times \{L, S, R\}^2$ of a two-tape Turing machine.

How Turing Machine "Works" ?

If the machine is in state $q \in Q$ and reads $(\sigma_1, \dots, \sigma_k)$ from the k tapes, and if

$$\delta(q, (\sigma_1, \dots, \sigma_k)) = (q', (\sigma'_2, \dots, \sigma'_k), z) \quad (z \in \{L, S, R\}^k) ,$$

then:

- $(\sigma_2, \dots, \sigma_k)$ will be replaced with $(\sigma'_2, \dots, \sigma'_k)$ in the last $k - 1$ tapes;
- the machine will be in state q' ; and
- for each $i = 1, \dots, k$, the i -th head will be moved as indicated by z_k .

The machine is started/stopped as follows:

- All tapes except the input tape are initialized with $[\triangleright, \square, \square, \dots]$.
- Input tape consists of a non-blank input string and a sequence of \square .
- The machine begins from state q_{start} .
- After reaching q_{halt} it is not allowed to change the output tape (the machine stops!)

Running Time

Def. Let $f: \{0, 1\}^* \rightarrow \{0, 1\}^*$ and $T: \mathbb{N} \rightarrow \mathbb{N}$ be some functions, and let M be a Turing machine. We say that M computes f in $T(n)$ -time if for every $x \in \{0, 1\}^*$, if M is initialized to the start configuration on input x , then after at most $T(|x|)$ steps it halts with $f(x)$ written on its output tape. We say that M computes f if it computes f in $T(n)$ -time for some function $T: \mathbb{N} \rightarrow \mathbb{N}$.

Def. We say that $T: \mathbb{N} \rightarrow \mathbb{N}$ is *time constructible* if $T(n) \geq n$ and there is a TM M that computes the function $x \mapsto \lfloor T(|x|) \rfloor$ in time $T(n)$.

Example: n , $n \log n$, n^2 , and 2^n are time-constructible.

The restriction $T(n) \geq n$ is to allow the algorithm time to read its input.

Example: Palindromes

Let PAL be the Boolean function defined for every $x \in \{0, 1\}^*$ as follows:

$$\text{PAL}(x) = \begin{cases} 1 & \text{if } x \text{ is a palindrome,} \\ 0 & \text{if } x \text{ is not a palindrome.} \end{cases}$$

A *palindrome* x reads the same from left to right as from right to left (i.e., $x_1x_2 \dots x_n = x_nx_{n-1} \dots x_1$).

We show that PAL can be computed within less than $3n$ steps:

- Copy the input to the read/write work tape.
- Move the input head to the beginning of the input.
- Move the input-tape head to the right while moving the work-tape head to the left. If at any moment the machine observes two different values, it halts and outputs 0.
- Halt and output 1.

Robustness of Definition

Most of the specific details of our definition of Turing machines are quite arbitrary. We will show that:

- restricting the alphabet Γ to be $\{0, 1, \square, \triangleright\}$,
- restricting the machine to have a single work tape, or
- allowing the tapes to be infinite in both directions

will not have a significant effect on the time to compute functions.

Small Alphabet

Claim 1.8. For every $f: \{0, 1\}^* \rightarrow \{0, 1\}^*$ and time-constructible $T: \mathbb{N} \rightarrow \mathbb{N}$, if f is computable in time $T(n)$ by a TM M using alphabet Γ then it is computable in time $3 \cdot \log |\Gamma| \cdot T(n)$ by a TM \tilde{M} with alphabet $\{0, 1, \square, \triangleright\}$.

Proof Idea: Use $\log \Gamma$ bits to encode every symbol of Γ in all tapes of \tilde{M} .

M's tape:

>	m	a	c	h	i	n	e												
---	---	---	---	---	---	---	---	--	--	--	--	--	--	--	--	--	--	--	--

\tilde{M} 's tape:

>	0	1	1	0	1	0	0	0	0	1	0	0	0	1	1				
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	--	--	--	--

For simulating a computational step of M , the machine \tilde{M} will: (1) read the $\log_2 |\Gamma|$ bits from all tapes, determine the corresponding symbol of Γ , (2) using the state function δ of M , write down the code of the new symbol on each tape (by moving the heads backwards), (3) move all heads to their next places.

Single Work Tape

Simulation of one step of M by \tilde{M} :

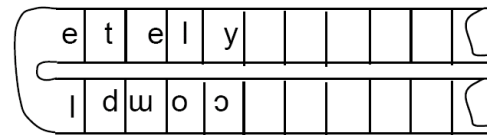
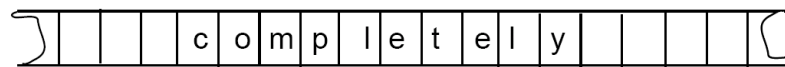
- Move $kT(n)$ steps to the right, read the tape and the symbols under all cursors.
- Move $kT(n)$ steps to the left, and correct the left-moving cursors.
- Move $kT(n)$ steps to the right, and correct the right-moving cursors.
- Move $kT(n)$ steps to the left to reach the initial position.

Unidirectional Infinite Tape

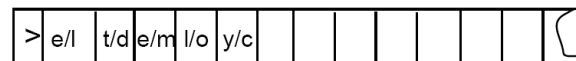
Claim 1.11. For every $f: \{0, 1\}^* \rightarrow \{0, 1\}^*$ and time constructible $T: \mathbb{N} \rightarrow \mathbb{N}$, if f is computable in time $T(n)$ by a bidirectional TM M then it is computable in time $4T(n)$ by a standard (unidirectional) $TM\tilde{M}$.

Proof Idea: Use alphabet Γ^2 . The machine \tilde{M} will ignore one of the components of a symbol. If a head goes "over the edge" then the other component will be ignored.

M 's tape is infinite in both directions:



\tilde{M} uses a larger alphabet to represent it on a standard tape:



Machines as Strings

Turing machines can be represented as strings: since we can write the description of any TM M on paper, we can definitely encode this description.

For convenience we assume that in our representation:

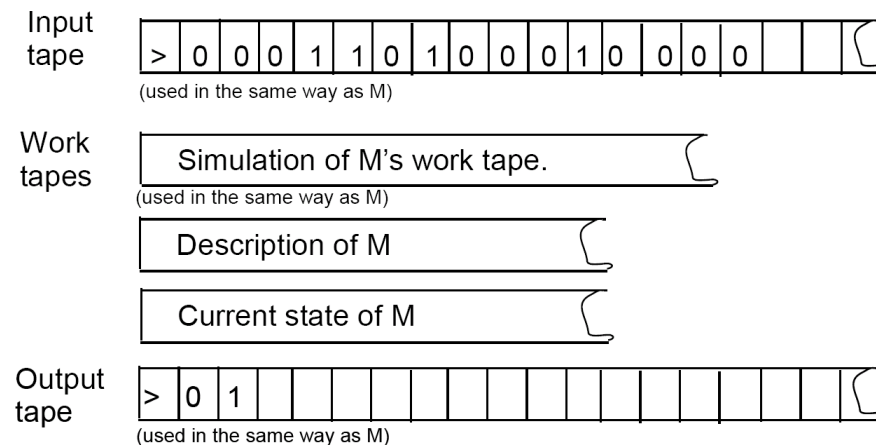
- *Every string represents some Turing machine.* This is easy to ensure by mapping strings that are not valid encodings into some trivial TM.
- *Every TM is represented by infinitely many strings.* This can be ensured by specifying that the representation can end with an arbitrary number of 1s, that are ignored. (e.g., the `/* . . . */` construct in C, C++ and Java) that allows to add superfluous symbols to any program.

If M is a Turing machine, then we use $\lfloor M \rfloor$ to denote M 's representation as a binary string. If α is a string then we denote the TM that α represents by M_α . As is our convention, we will also often use M to denote both the TM and its representation as a string.

Universal Turing Machine

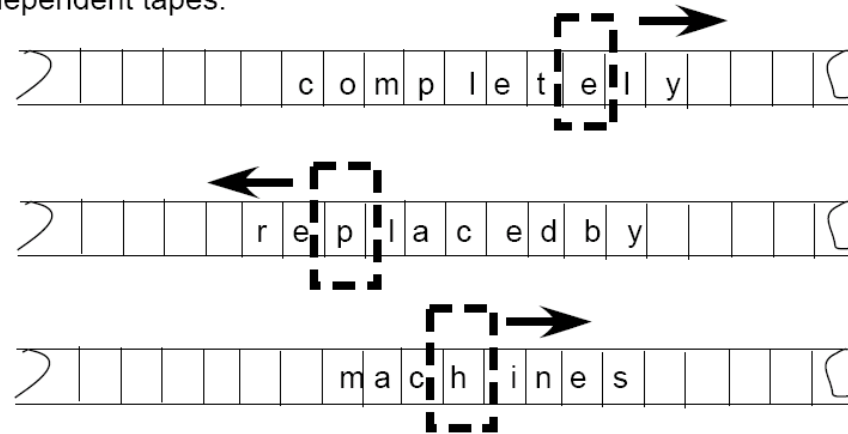
Theorem 1.13 There exists a TM \mathcal{U} such that for every $x, \alpha \in \{0, 1\}^*$, $\mathcal{U}(x, \alpha) = M_\alpha(x)$, where M_α denotes the TM represented by α . Furthermore, if M_α halts on input x within T steps then $\mathcal{U}(x, \alpha)$ halts within $CT \log T$ steps, where C is a number independent of $|x|$ and depending only on M_α 's alphabet size, number of tapes, and number of states.

Proof Idea:

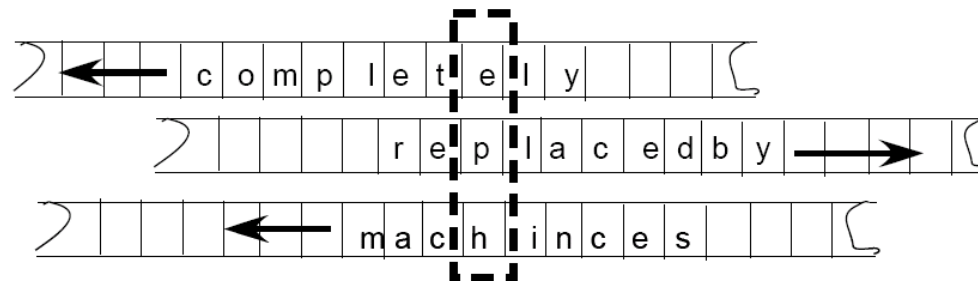


Universal Simulation in $O(T \log T)$ Time

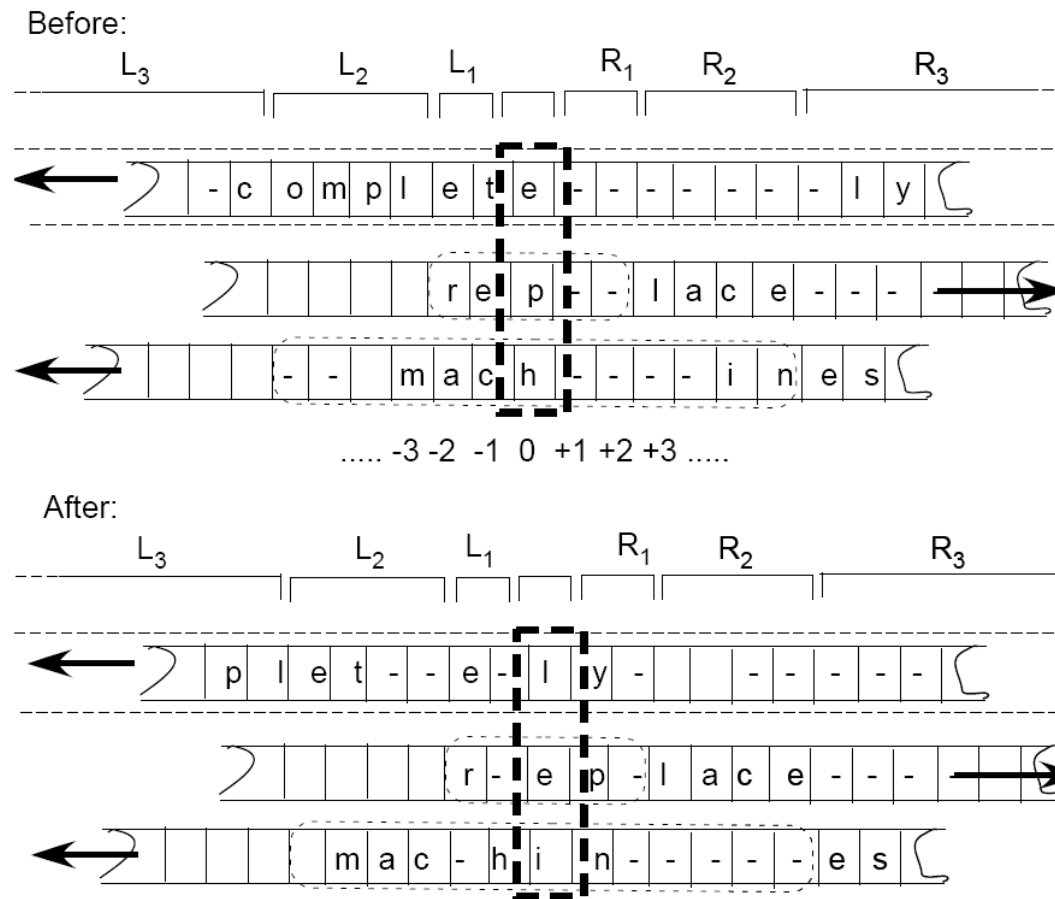
M's 3 independent tapes:



U's 3 parallel tapes (i.e., one tape encoding 3 tapes)



Universal Simulation in $O(T \log T)$ Time



Uncomputable Functions: Why such Functions Must Exist?

There are only *countably many* finite bit-strings $\alpha \in \{0, 1\}^*$.

Hence, there are only countably many computable functions f , because for every such function, there is a finite bit-string α , such that M_α computes f .

The set of all functions $f: \{0, 1\}^* \rightarrow \{0, 1\}$ is *not countable*. A famous theorem by Cantor says that the power-set (the set of all subsets) of any set is strictly larger than the set itself.

Therefore, there exist functions that are not computable.

But, are we able to define any?

Uncomputable Functions: An Example

Theorem: There exists a function $UC: \{0, 1\} \rightarrow \{0, 1\}$ that is not computable by any TM.

Proof: The function UC is defined so that for every $\alpha \in \{0, 1\}$:

$$UC(\alpha) = \begin{cases} 0 & \text{If } M_\alpha(\alpha) \text{ halts within a finite number of steps and outputs 1,} \\ 1 & \text{otherwise.} \end{cases}$$

Suppose that there exists a Turing Machine M such that computes UC, i.e. $M(x) = UC(x)$ for every $x \in \{0, 1\}$. Then, for $\alpha = \perp M \perp$, we have $M(\alpha) = UC(\alpha)$. But then ...

- if $UC(\alpha) = 0 (\neq 1)$ then by definition $M_\alpha(\alpha) = 1$, i.e. $UC(\alpha) = 1$, because $UC \equiv M_\alpha$. A contradiction!
- if $UC(\alpha) = 1$ then (by $UC \equiv M_\alpha$) we have $M_\alpha(\alpha) = 1$, which by the definition of UC implies $UC(\alpha) = 0$. Again a contradiction!

This proof technique is called *diagonalization*.

Diagonalization

	0	1	00	01	10	11	...	α
0	0	1	*	0	1	0		$M_0(\alpha)$	
1	1	1	0	1	*	1		...	
00	*	0	0	1	0	*			
01	1	*	0	0	1	*	0		
...									
α	$M_\alpha(0)$...						$M_\alpha(\alpha)$	$1 - M_\alpha(\alpha)$
...									

Uncomputable Functions: Halting Problem

The following "natural" function definition is also uncomputable:

$$\text{HALT}(\alpha, x) = \begin{cases} 1 & \text{If } M_\alpha(x) \text{ halts,} \\ 0 & \text{otherwise.} \end{cases}$$

Indeed, if HALT is computable, then we would have an algorithm for the UC function:

- On input α , compute $\text{HALT}(\alpha, \alpha)$ first, and then ...
- If $\text{HALT}(\alpha, \alpha) = 0$ then output 1
- Otherwise, use the universal Turing Machine to compute $M_\alpha(\alpha)$,
- If $M_\alpha(\alpha) = 1$ then output 0, otherwise output 1.

Obviously enough, this program computes UC, which is impossible.

Complexity Classes: Class P

A *complexity class* is a set of functions that can be computed within a given resource.

We will pay special attention to computing Boolean functions $\{0, 1\}^* \rightarrow \{0, 1\}$, also known as *decision problems* or *languages*.

We identify a Boolean function f with the language $L_f = \{x : f(x) = 1\}$.

Class DTIME: Let $T: \mathbb{N} \rightarrow \mathbb{N}$ be some function. We let $\text{DTIME}(T(n))$ be the set of all Boolean (one bit output) functions that are computable in $c \cdot T(n)$ -time for some constant $c > 0$.

Definition 1.20 (The class P):

$$\mathbf{P} = \bigcup_{c \geq 0} \text{DTIME}(n \cdot c) .$$

What have we learned?

- There are many equivalent ways to mathematically model computational processes; we use the standard Turing machine formalization.
- Turing machines can be represented as strings. There is a universal TM that can emulate (with small overhead) any TM given its representation.
- There exist functions, such as the Halting problem, that cannot be computed by any TM regardless of its running time.
- The class \mathbf{P} consists of all decision problems that are solvable by Turing machines in poly-time. We say that problems in \mathbf{P} are efficiently solvable.
- All low-level choices (number of tapes, alphabet size, etc..) in the definition of Turing machines are immaterial, as they will not change \mathbf{P} .