MTAT.07.004 — Complexity Theory September 1st, 2011

Lecture 1

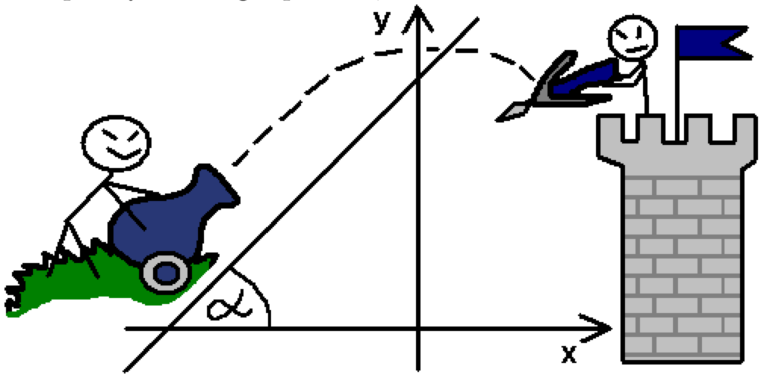
Lecturer: Peeter Laud *Scribe(s): Alisa Pankova*

1 Introduction

The first lecture gives an overview of the models of computation and the most important computational resources. Different measures of complexity have been proposed for different models. The Turing machines are described in more details, including the reduction of the alphabet size, and the Universal Turing Machine. The lecture finishes with the formal definition of the complexity class P.

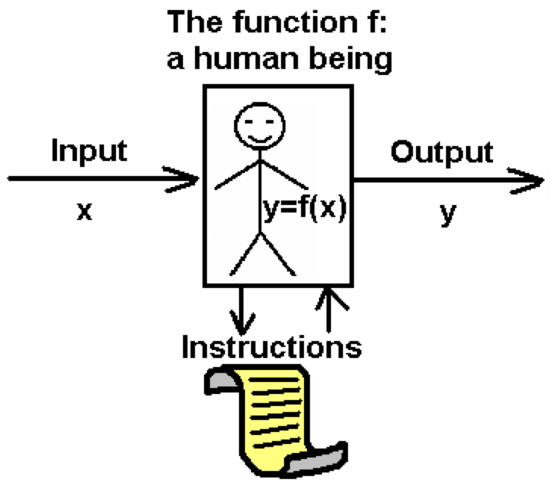
2 About Computation

Complex computations had been used long before the computers emerged, and the algorithms had practical applications. For example, if an artilleryman wanted to attack a fortress, he had to calculate the angle from which he had to shoot and the amount of gunpowder he had to use in order to hit the fortress precisely. These parameters depended on several other things, like the distance to the fortress, the height of its walls, the height difference of terrain between the fortress and the artillery piece, the wind speed and direction, the quality of the gunpowder, etc.



Of course, it would take a lot of time to do such computations each time on the battlefield, and therefore the artilleryman had to consult a table, in which the cannon position depended on different parameters, such as the distance from the target or the elevation. The table had to be pre-computed by some function that took these parameters as an input and produced an output. Similar tables were needed for many other purposes, such as bookkeeping and accounting.

In order to create a table, someone had to compute the outputs on different inputs. This function could be fulfilled by a human being, and in order to produce the correct output, it had to consult some instructions. It seems that a human being in principle is not much different from a computer program, but it is much slower and may make more mistakes.



The official notion of *computaion* and *computability* has been formalized only in the middle of 20^{th} century. Computation complexity in general means how much effort should be applied to compute a function, on how many resources should be spent on it, depending on the size of input.

3 Models of Computation

There exist different universal models of computation. Any computation (with the possible and likely exception of quantum computers) performed in one of them can be modeled in another, with a similar amount of computational effort. Similar means that computation time is not exactly the same, but the computing functions would have the same or similar order of magnitude.

Suppose that we are given two models, $Model_1$ and $Model_2$. We want to perform a task f in both models. This task can be computing some function that takes an input x and outputs y. Let M_1 be the machine that performs that task in $Model_1$. It computes $M_1(x) \to y$ with computational effort bounded by $T(|x|)$ (where T is a time function, and $|x|$ is the length of the input). If $Model_2$ can emulate $Model_1$ with a similar amount of computational effort (in the case of our universal models it is polynomial time), it means that it is possible to construct a machine M_2 which performs the same task in $Model_2$ (computes $M_2(x) \to y$), such that there exists a polynomial p, such that the computational effort of M_2 is bounded by $p(T(|x|))$.

The following models are all universal.

3.1 λ-calculus

This is a practical model that is used in functional programming. A λ-expression is either a variable x, or an *application* MN, where M and N are λ-expressions, or an *abstraction* (or function definition) $\lambda x.M$, where x is a variable and M is a λ-expression. Since M may include variables other than x, we need to show which of them is the argument that the function actually takes. For example, if M contains variables x and y, then $\lambda x.M$ applied to 2 may output a new function where all instances of x are replaced by 2, but the instances of y are still not evaluated.

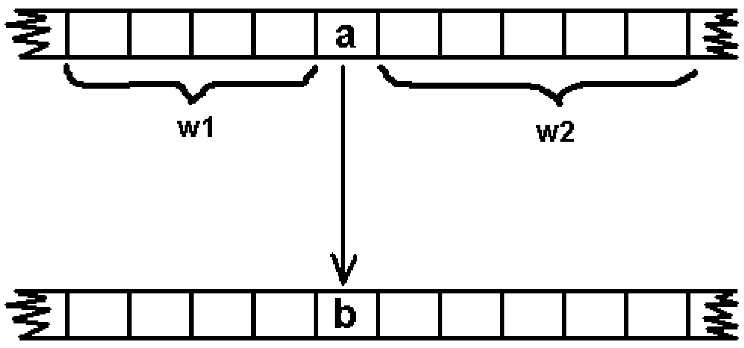
A computation step in λ-calculus applies to an expression of the form $(\lambda x.M)N$, where M and N are λ-expressions. In one computation step, this expression may evolve to $M[x \leftarrow N]$. This expression denotes M, where each occurrence of x (that was bound by λx before) is replaced with the expression N.

The computational effort for λ-calculus is determined by the number of steps.

3.2 Cellular Automata

This model is only theoretical. Its work is shown on example of one-dimensional cellular automata. In this case, the main component of the automaton is a doubly infinite tape that is segmented into cells, where each cell contains a symbol from a finite alphabet Σ. Theoretically, the tape is infinite in both directions, but we may place the input into the middle and fill the rest of the tape with blank symbols.

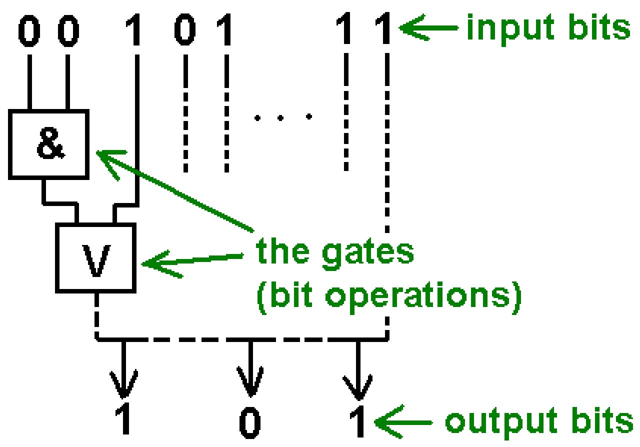
In one computation step, all cells of the tape evolve and their contents may change into a different symbol from the alphabet Σ. The evolution rules have the form $(w_1, a, w_2) \to b$, where $w_1, w_2 \in \Sigma^*$ and $a, b \in \Sigma$. A cell containing a before the computation step will contain b afterwards if the cells to the left of that cell contain w_1 before the computation step, and the cells to the right contain w_2. The program of the automaton is a set of such rules. For each symbol and context, there must be exactly one applicable rule.



The computational effort is determined by the number of steps.

3.3 Boolean circuits

Boolean circuits are very practical, and they are used in computer hardware. A circuit takes a bit sequence of fixed length as an input. Each bit enters its wire. The circuit contains gates that perform logical operations on the bits. A gate outputs just one bit, but it may send it to several places. The output of the whole circuit may be not a single bit, but a bit sequence.



The circuit does computation on all the input bits preferrably in parallel, but the computational effort is still defined by the total number of gates.

3.4 Quantum Circuits

These circuits are very similar to Boolean circuits, but instead of common bits they use quantum bits. The computational effort is also determined by the number of gates in the circuit. There is not such hardware by the moment, but the scientists are trying to bring this model from theory to practice.

3.5 Random Access Machines

This model is very similar to modern computers. The main idea is the interaction between the set of instructions (the program) and the memory (the register bank). The computational effort is defined by the number of steps.

3.6 Turing Machines

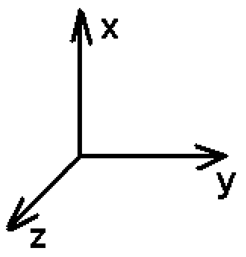
The most famous computational model is a Turing machine. Although Turing machines are only theoretical, and nobody has found it necessary to construct a real one, they are often being used in algorithm theory.

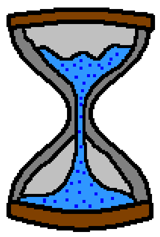
The efficiency and computational effort is determined by the number of steps that a Turing machine makes before producing the correct output. This model is described in more details further.

4 Resources

Each computation requires resources. They can be the following:

- **Time:** how long it takes for a function to produce the correct output.
- **Space:** how much space for temporary values the function requires during the computation.





These two resources are the most important for computation. Although time can be considered as an additional dimension to space, there is a significant difference between these two resources. The space can be re-used, and the time can not (at least people haven't learned to do it in this reality). Therefore, the time complexity can be considered more important.

- **Program size:** someone has to write the program that computes the funcion. If the program text turns out to be infinite, it is impossible to run it.

  For Boolean circuits, the program size is equivalent to time resource.

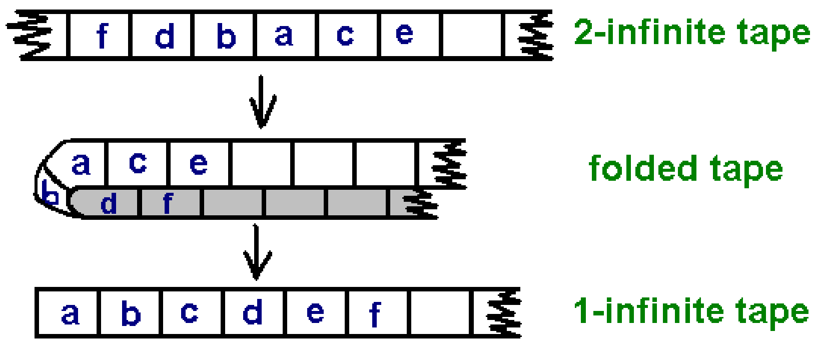
- **Randomness:** a non-deterministic algorithm that uses randomization may be much faster, but with some probability it may output a wrong answer or not stop at all.

- **Coherence:** This resource is special for quantum bits. They need to avoide noise to preserve their entangled states.

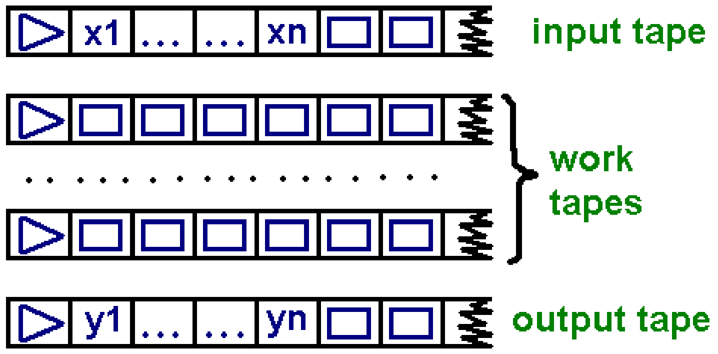
5 Turing Machines

The main component of a Turing Machine (TM) is a set of tapes. In the model we're considering in this course, a Turing machine has at least two tapes, one for input (this tape is read-only) and one for output. The rest of them are the work tapes, where temporary results may be stored. The output tape can also be used as a work tape. Each tape has its own reading/writing head.

Another detail that we're going to introduce is, that the tapes of a Turing machine are infinite only in one direction. A two-way infinite tape can be modeled by a one-way infinite tape if necessary.



Let $\Gamma = \{0, 1, \Box, \triangleright\}$ be the alphabet of the symbols that can be present on the tapes. We may define a TM with more symbols, but these four are sufficient. Each tape begins with a start symbol $\triangleright$. On the input tape, the input is represented as a sequence of $\{0, 1\}$ symbols, followed by an infinite amount of blank symbols $\Box$. The work tapes initially consist only of the start symbol and the blank symbols. In the end of the computation, the output emerges on the output tape in the beginning right after the $\triangleright$ (also as a sequence of $\{0, 1\}$ symbols).

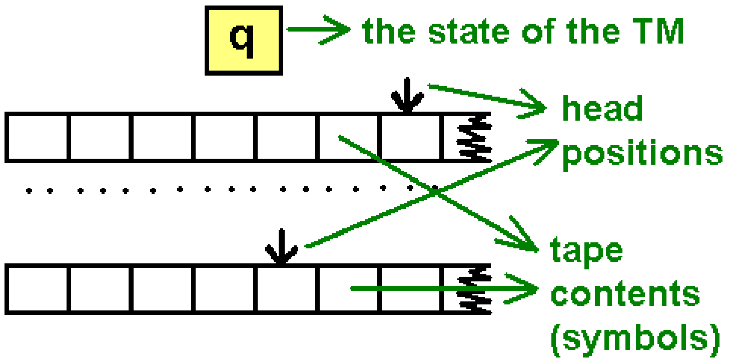


Additionally to the tapes, a Turing machine has a state. The set of all the possible states of a TM is denoted as Q. The states can be represented by different natural numbers. In some states, that are defined as final states ($Q_F \subseteq Q$), the Turing machine stops its work. There is also the initial state ($q_0 \in Q$), in which the TM starts its work. There can be only one initial state.

The contents of all k tapes, the head positions, and the current state of a TM define its configuration. We can write it down as $\langle q; w_1, \dots, w_k; p_1, \dots, p_k \rangle$, where:

- $q \in Q$ is the current state of the TM
- $w_i \in \Gamma^* \cdot \{\Box^\omega\}$ is the contents of the i-th tape.
- $p_i \in \mathbb{N}$ is the position of the i-th head. Let leftmost position be 1.

The set of all the possible configurations for a TM with k tapes, an alphabet Γ, and the states Q is denoted as $CONF^k_{\Gamma,Q}$.

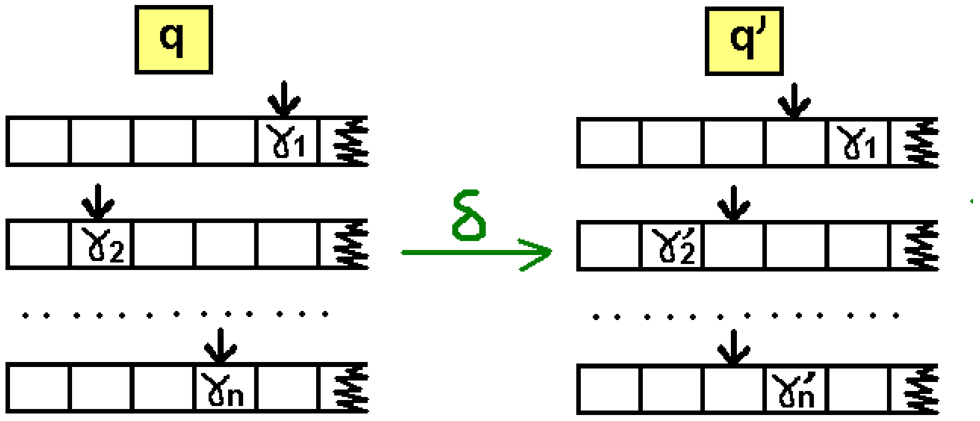


One very important thing is the transition function. A Turing Machine changes its current configuration according to the predifined function. It can be denoted as $\delta : Q \times \Gamma^k \rightarrow$

$Q \times \Gamma^{k-1} \times Move^k$, where $Move = \{-1, 0, 1\}$, and $Move^k$ shows in which direction the head of each of the k tapes has to move (-1 is "left", 1 is "right", 0 is "do not move"). In the right part, we write Γ^{k-1} instead of Γ^k because the contents of the input tape do not change.

A Turing machine can be in general represented as a tuple $(\Gamma, Q, \delta, q_0, Q_F)$. It defines a relation (a partial function) $\overset{M}{\to}$ on the set of all the possible configurations $CONF^k_{\Gamma,Q}$:
$\langle q; w_1, \ldots, w_k; p_1, \ldots, p_k\rangle \overset{M}{\to} \langle q'; w_1, w'_2, \ldots, w'_k; p'_1, \ldots, p'_k\rangle$ iff

- $\underline{q \notin Q_F}$: if a Turing machine is it its final state, it cannot perform any transition.
- $\underline{\gamma_i = w_i[p_i]}$: let γ_i denote the symbol on the i^{ith} tape on which its head pointed before the transition.
- $\underline{(q'; \gamma'_2, \ldots, \gamma'_k; s_1, \ldots, s_k) = \delta(q; \gamma_1, \ldots, \gamma_k)}$: the transition fuctions changes the contents of the symbols that have been under the heads. Each γ_i except γ_1 (which is on the input tape) becomes γ'_i. The transition also moves the heads of each tape either left, right, or does not move (the movements are denoted by s_i).
- $\underline{w'_i = w_i[p_i \mapsto \gamma'_i]}$: the symbols that have been under the heads change.
- $\underline{p'_i = \max(1, p_i + s_i)}$: the max function is here because the head cannot move out of the tape beyond the first symbol.



Now we may define the work process of a Turing machine applied to bit strings. Let $C_0 = \langle q_0; \triangleright \cdot x \cdot \square^\omega, \triangleright \cdot \square^\omega, \ldots, \triangleright \cdot \square^\omega; 1, \ldots, 1\rangle$ be the initial configuration. The relation defined by the TM produces a series of configurations $C_1, C_2, \ldots$, such that $C_{i-1} \overset{M}{\to} C_i$.

- If there exists $C_n = \langle q_n; w_1, \ldots, w_k; p_1, \ldots, p_k\rangle$ with $q_n \in Q_F$ then
  - we say that M stops on x in n steps in state q_n.
  - If also $w_k = \triangleright \cdot y \cdot \square^\omega$ where $y \in \{0,1\}^*$ then we say that M outputs y on input x.
- If there is no such C_n, then M does not stop on x.

5.1 The Tasks of Turing Machines

There are two main things that a Turing machine may perform:

- **Accept a language.** Let L be a language (a subset of bitstrings). Let the TM have two final states, $Q_F = \{q_{acc}, q_{rej}\}$. Let x be an arbitrary bit string. If the TM always stops in q_{acc} iff $x \in L$, and it stops on all inputs, it means that the TM accepts the language L.
- **Compute a function.** Let f be a function defined on bit strings ($f : \{0,1\}^* \to \{0,1\}^*$). If on each input x the TM stops, outputs something, and this something equals to $f(x)$, it means that the TM computes the function f.

5.2 Time Function

For a Turing machine, we may define an upper bound of the steps that it takes to compute the corresponding function, depending on the size of the input. First, we define a function $T : \mathbb{N} \to \mathbb{N}$ (the natural numbers will be encoded as bit srings in the case of Turing machine). Let M be a Turing machine. We may say that M computes some function f in time T, if for any input it makes at most $T(|x|)$ steps, and also returns the correct output for f (here $|x|$ is the length of the input bit string).

The function T is supposed to return the number of steps that M makes. But if we simply run M, we get the output of f, not the number of steps. Suppose that we want to compute $T(|x|)$. This will already be another Turing machine, and it in turn takes some time to compute the value of $T(|x|)$, which also depends on $|x|$. If for any input x the time that this new Turing machine takes to compute $T(|x|)$ is not more complex than $T(|x|)$ itself (is upper bounded by $c \cdot T(|x|)$ for some constant c), then we say that the function T is **time constructible**.

The inputs and outputs of time functions are also represened by bit strings.

5.3 Function Comparison

If we want to compare time functions of Turing machines, we first need to define a way of comparing them. One Turing machine may be faster that another one when the input size is relatively small, but it may become slower when the input size increases. We need to introduce several notations.

Let $f, g : \mathbb{N} \to \mathbb{N}$. Let $\mathbb{R}^+$ denote the set of all positive real numbers. Define sets of functions from $\mathbb{N}$ to $\mathbb{N}$: $O(f), o(f), \Theta(f), \Omega(f)$, and $\omega(f)$. Their meaning is the following:

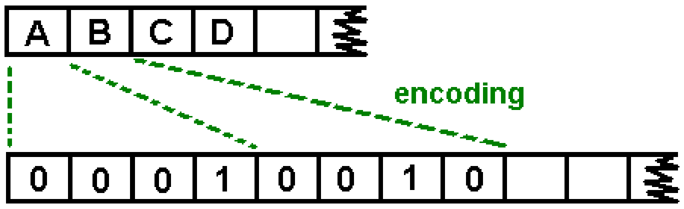
- $g \in O(f)$ (or g is $O(f)$) if
  $\exists c \in \mathbb{R}^+ \ \exists n_0 \in \mathbb{N} \ \forall n \in \mathbb{N} : n \geq n_0 \Rightarrow g(n) \leq c \cdot f(n)$
- $g \in o(f)$ if
  $\forall c \in \mathbb{R}^+ \ \exists n_0 \in \mathbb{N} \ \forall n \in \mathbb{R}^+ : n \geq n_0 \Rightarrow g(n) \leq c \cdot f(n)$
- $g \in \Omega(f)$ if $f \in O(g)$.
- $g \in \omega(f)$ if $f \in o(g)$.
- $\Theta(f)$ is the intersection of $O(f)$ and $\Omega(f)$.

5.4 Reducing the Size of the Tape Alphabet

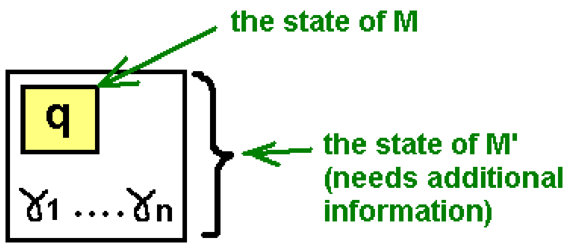
We have freedom to construct Turing machines with different sizes of the alphabet. But what is the minimal number of symbols that are needed to compute a particular function? Is it possible to reduce the alphabet and retain the same computational time? There is a theorem that states that it is sufficient to have $\Gamma = \{\triangleright, \Box, 0, 1\}$ symbols to compute the same function without adding extra tapes (the exception is if the initial TM had 2 tapes), and without increasing the time complexity.

Theorem 1 *Let $M = (\Gamma, Q, \delta, q_0, Q_F)$ with k tapes accept a language / compute a function in time T. There exists a TM M' with max(k, 3) tapes and tape alphabet $\{\triangleright, \Box, 0, 1\}$ that accepts the same language / computes the same function in time O(T).*

Proof: First, we need to encode all the symbols that are present in the alphabet of M with the symbols available for M'. Let the start $\triangleright$ and the blank $\Box$ symbols remain the same. We need to encode the rest of the symbols by 0 and 1: define a function $\Gamma \to \{0,1\}^{\ulcorner log|\Gamma|\urcorner}$, which encodes each symbol by a bit string of fixed length defined by the size of Γ.



It is easy to encode the contents of the tapes, but we have to consider the new length of values in the transition function. Reading/writing a cell in M means reading/writing several cells in M', and moving once to the left or to the right in M means passing by several cells in M'. This increase in movement is a constant that depends on $|\Gamma|$, not on the input. The state of M' that is equivalent to q in M can be defined as q', and it has to contain at least as much information as q, but it is not enough. The state q' needs to remember not only the current symbol, but the whole sequence that equals to one symbol in M. If the M performs a step $(q, \gamma_1, \ldots, \gamma_n) \to \ldots$, the state q' of M' needs to contain information not only about the q, but also about $\gamma_1 \ldots \gamma_n$.



All the additional operations of M' do not increase the complexity. The time has definitely increased by some constant, but the constants do not matter since they mostly depend on the Turing machine (in the case of computers, on the hardware). The better hardware becomes, the less will be the constant.

The other thing that we have to prove is that the theorem holds only for $k' \geq 3$ (k' is the number of tapes in M'). Supose that $k' = 2$. Since we have only one work tape, we have to perform all the operations on the same tape, and it will already require square steps (it is no longer linear). $\square$

5.5 Universal Turing Machine

Each Turing machine can be represented as a tuple $(\Gamma, Q, \delta, q_0, Q_F)$. If we encode each of the elements in that tuple to bit strings, we get an encoding from the set of Turing Machines to bit strings. Let for any bit string α there exist a TM M_α, that is encoded by α. Now we may refer to any Turing machine by its index α.

We may now define a universal Turing machine $\mathcal{U}$, which computes every possible function that any other Turing machine may compute. There exists a five-tape TM $\mathcal{U}$ with tape alphabet $\{0, 1, \triangleright, \square\}$ and a function $C : \{0,1\}^* \to \mathbb{N}$, such that, for all $x, \alpha \in \{0,1\}^*$, if M_α on input x stops in t steps then:

- $\mathcal{U}$ on input (α, x) stops in at most $C(\alpha) \cdot t \log t$ steps;
- the output of $\mathcal{U}$ on (α, x) equals the output of M_α on x.

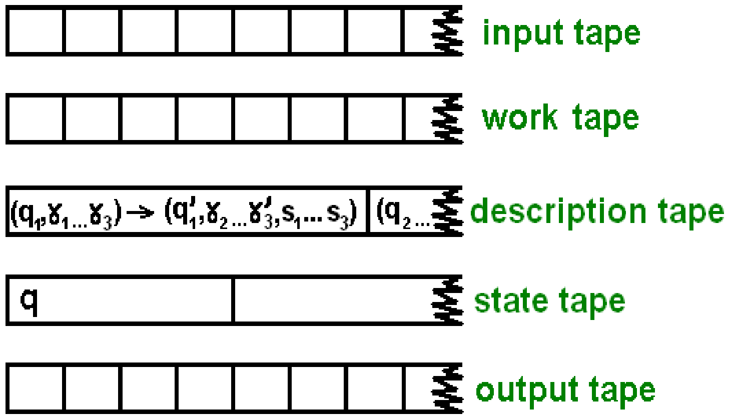
First, we check which Turing machine the $\mathcal{U}$ has to simulate. We can see this from the argument α. We want to compute the same function that M_α computes. We may construct a TM which computes the same function with only two tapes without exceeding polynomial time (this theorem has not been proven on the lecture). According to Theorem 1, we may construct another TM that still computes the same function, but needs an alphabet that consists only of four symbols ($\Gamma = \{0, 1, \triangleright, \square\}$). We need to add an additional tape if we want to use this theorem, since the number of the tapes must be at least three. Let this new TM be denoted as as M'.

This 3-tape Turing machine M' could do its work if we defined its transition function and the states. But the $\mathcal{U}$ has its own transition function and the states, and we cannot just substitute them. It means that this information has to be stored somewhere else. Let us encode all the possible states and transitions of M' as bit strings. We may now take all the encoded transitions of M' and write them onto an additional tape (this would be the description of M'). The current state of M' should also be stored on a separate tape. We get five tapes in total.

We also need to define the rules, how this new 5-tape TM would compute the same function as M' by consulting these auxiliary tapes. The new TM should do something like this:

1. Look at the current state of M' (see it from the state tape).
2. Select an appropriate transition rule from the description tape.
3. Move the heads on the three other tapes and change the symbols according to the transition function.
4. Change the value on the state tape according to the transition.

Now denote this Turing machine as $\mathcal{U}$. When we run $\mathcal{U}(\alpha, x)$, it fills its description tape with the transitions that relate to M_α, writes the initial state of M_α onto the state tape, leaves the x on the input tape, and computes the value of M_α with help of the auxiliary tapes.



The number of steps involved in computation of $\mathcal{U}(\alpha, x)$ is definitely larger than the number of steps done by $M_\alpha(x)$, but it still in polynomial time. If M_α on input x stops in t steps, then U on input (α, x) stops in at most $C(\alpha) \cdot t \ log \ t$ steps, where $C(\alpha)$ is a constant that depends on M_α, not on the input. There are in total $log \ t$ times more steps since U has only one working tape. If M_α also has only one working tape, then U stops in $C(\alpha) \cdot t$ steps.

6 The Class P

Let f be any function defined on natural numbers ($f : \mathbb{N} \to \mathbb{N}$). We may define a class $\mathsf{DTIME}(f) \subseteq 2^{\{0,1\}^*}$ which is the set of all languages that can be accepted in time $O(f)$ (this class is a set of sets of bit strings). The class P is defined as $\mathsf{P} = \bigcup_{c\in\mathbb{N}} \mathsf{DTIME}(\lambda n.n^c) = \mathsf{DTIME}(n) \cup \mathsf{DTIME}(n^2) \cup \mathsf{DTIME}(n^3) \cup \ldots$, which actually means that the time is polynomial in relation to the input length n.

The letter D in the word DTIME means ”deterministic”.