Computational Complexity

Manuel Mignogna

May 10, 2025

Contents

1 Turing Machine

1.1 Definition

Formally, a Turing Machine is a quadruple $M = (K, \Sigma, \delta, s)$ where: K is a finite set of states, $s \in K$ is the initial state, Σ is a finite set of symbols (we say that Σ is the alphabet of M), $\sqcup \in \Sigma$ is a special symbol called the blank symbol, $\triangleright \in \Sigma$ is a special symbol called the first symbol, δ is a transition function

$$\delta : K \times \Sigma \to (K \cup \{h, \text{"yes"}, \text{"no"}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}$$

We assume that h (the halting state), "yes" (the accepting state), and "no" (the rejecting state), and the cursor directions $\leftarrow$ for "left", $\rightarrow$ for "right", and $-$ for "stay", are not in $K \cup \Sigma$. The function δ is the "program" of the machine. It specifies, for each combination of current state $q \in K$ and current symbol $\sigma \in \Sigma$, a triple

$$\delta(q, \sigma) = (p, \rho, D)$$

where: p is the next state, ρ is the symbol to be written on σ and $D \in \{\leftarrow, \rightarrow, -\}$ is the direction in which the cursor will move. For $\triangleright$ we require that, if for states q and p, $\delta(q, \triangleright) = (p, \rho, D)$, then $\rho = \triangleright$ and $D = \rightarrow$. That is, $\triangleright$ always directs the cursor to the right, and is never erased.

1.2 Program execution

How is the program to start? Initially, the state is s. The string is initialized to a $\triangleright$, followed by a finitely long string $x \in (\Sigma - \{\sqcup\})^*$. We say that x is the input of the Turing machine. The cursor is pointing to the first symbol, always a $\triangleright$.

From this initial configuration, the machine takes a step according to δ, changing its state, printing a symbol, and moving the cursor; then it takes another step, and another. Note that, by our requirement on $\delta(p, \triangleright)$, the string will always start with a $\triangleright$, and thus the cursor will never "fall off" the left end of the string.

Although the cursor will never fall off the left end, it will often wander off the right end of the string. In this case, we think that the cursor scans a $\sqcup$, which of course may be overwritten immediately. This is how the string becomes longer—a necessary feature if we wish our machines to perform general computation. The string never becomes shorter.

Since δ is a completely specified function, and the cursor never falls off the left end, there is only one reason why the machine cannot continue: One of the three halting states h, "yes", and "no" has been reached. If this happens, we say that the machine has halted. Furthermore, if state "yes" has been reached, we say the machine accepts its input; if "no" has been reached, then it rejects its input. If a machine halts on input x, we can define the output of the machine M on x, denoted $M(x)$. If M accepts or rejects x, then $M(x)$ = "yes" or "no", respectively. Otherwise, if h is reached, then the output is the string of M at the time of halting. Since the computation has gone on for finitely many steps, the string consists of a $\triangleright$, followed by a finite string y, whose last symbol is not a $\sqcup$, possibly followed by a string of $\sqcup$s (y could be empty).

We consider string y to be the output of the computation, and write $M(x) = y$. Naturally, it is possible that M will never halt on input x. If this is the case, we write $M(x) = \nearrow$.

1.3 Configuration

We can define the operation of a Turing machine formally using the notion of a *configuration*. Intuitively, a configuration contains a complete description of the current state of the computation. Formally, a configuration of M is a triple (q, w, u), where $q \in K$ is a state, and w, u are strings in Σ^*.

w is the string to the left of the cursor, including the symbol scanned by the cursor, and u is the string to the right of the cursor, possibly empty. q is the current state.

1.3.1 Configuration Yields

We say that configuration (q, w, u) *yields* configuration (q', w', u') in one step, denoted

$$(q, w, u) \xrightarrow{M} (q', w', u'),$$

intuitively if a step of the machine from configuration (q, w, u) results in configuration (q', w', u'). Formally, it means that the following holds.

First, let σ be the last symbol of w, and suppose that $\delta(q, \sigma) = (p, \rho, D)$. Then we must have that $q' = p$. We have three cases: If $D = \rightarrow$, then w' is w with its last symbol (which was a σ) replaced by ρ, and the first symbol of u appended to it ($\sqcup$ if u is the empty string); u' is u with the first symbol removed (or, if u was the empty string, u' remains empty). If $D = \leftarrow$, then w' is w with σ omitted from its end, and u' is u with ρ attached in the beginning. Finally, if $D = -$, then w' is w with the ending σ replaced by ρ, and $u' = u$.

Once we defined the relationship of *yields in one step* among configurations, we can define *yields* to be its transitive closure. That is, we say that configuration (q, w, u) *yields* configuration (q', w', u') in k steps, denoted

$$(q, w, u) \xrightarrow{M^k} (w', q', u'),$$

where $k \geq 0$ is an integer, if there are configurations (q_i, w_i, u_i), $i = 1, \ldots, k+1$, such that $(q_i, w_i, u_i) \xrightarrow{M} (q_{i+1}, w_{i+1}, u_{i+1})$ for $i = 1, \ldots, k$, $(q_1, w_1, u_1) = (q, w, u)$, and $(q_{k+1}, w_{k+1}, u_{k+1}) = (q', w', u')$.

Finally, we say that configuration (q, w, u) *yields* configuration (q', w', u'), denoted

$$(q, w, u) \xrightarrow{M^*} (q', w', u'),$$

if there is a $k \geq 0$ such that $(q, w, u) \xrightarrow{M^k} (q', w', u')$.

1.4 Turing Machine for palindromes

$p \in K,$	$\sigma \in \Sigma$	$\delta(p, \sigma)$
s	0	$(q_0, \triangleright, \rightarrow)$
s	1	$(q_1, \triangleright, \rightarrow)$
s	$\triangleright$	$(s, \triangleright, \rightarrow)$
s	$\sqcup$	("yes", $\sqcup, -)$
q_0	0	$(q_0, 0, \rightarrow)$
q_0	1	$(q_0, 1, \rightarrow)$
q_0	$\sqcup$	$(q_0', \sqcup, \leftarrow)$
q_1	0	$(q_1, 0, \rightarrow)$
q_1	1	$(q_1, 1, \rightarrow)$
q_1	$\sqcup$	$(q_1', \sqcup, \leftarrow)$

$p \in K,$	$\sigma \in \Sigma$	$\delta(p, \sigma)$
q_0'	0	$(q, \sqcup, \leftarrow)$
q_0'	1	("no", $1, -)$
q_0'	$\triangleright$	("yes", $\sqcup, \rightarrow)$
q_1'	0	("no", $1, -)$
q_1'	1	$(q, \sqcup, \leftarrow)$
q_1'	$\triangleright$	("yes", $\triangleright, \rightarrow)$
q	0	$(q, 0, \leftarrow)$
q	1	$(q, 1, \leftarrow)$
q	$\triangleright$	$(s, \triangleright, \rightarrow)$

The machine works as follows: In state s, it searches its string for the first symbol of the input. When it finds it, it makes it into a $\triangleright$ (thus effectively moving the left end of the string inward) and remembers it in its state. By this, we mean that M enters state q_0 if the first symbol is a 0, and state q_1 if it is a 1 (this important capability of Turing machines to remember finite information in their state will be used over and over). M then moves to the right until the first $\sqcup$ is met, and then once to the left to scan the last symbol of the input (now M is in state q_0' or q_1', still remembering the first symbol). If this last symbol agrees with the one remembered, it is replaced with a $\sqcup$ (so that the string implodes on the right as well). Then the rightmost $\triangleright$ is found using a new state q, and then the process is repeated. Notice that, as the two boundaries of the string (a $\triangleright$ on the left, a $\sqcup$ on the right) have "marched inwards," the string left is precisely the string that remains to be shown a palindrome. If at some point the last symbol is different from the first symbol as remembered by the machine, then the string is not a palindrome, and we reach state "no". If we end up with the empty string (or we fail to find the last symbol, which means that the string was a single symbol) we express our approval by "yes".

On input 0010, the following configurations, among others, will be yielded:

$$(s, \triangleright, 0010) \xrightarrow{M^5} (q_0, \triangleright 010\sqcup, \epsilon) \xrightarrow{M} (q_0', \triangleright 010, \sqcup) \xrightarrow{M} (q, \triangleright 01, 01\sqcup) \xrightarrow{M} (s, \triangleright 0, 1\sqcup) \xrightarrow{M}$$

$$(q_0, \triangleright\triangleright, 1\sqcup) \xrightarrow{M^2} (q_0, \triangleright\triangleright 1, \sqcup) \xrightarrow{M} (q_0', \triangleright\triangleright, \sqcup) \xrightarrow{M} (\text{"no"}, \triangleright\triangleright\sqcup, \sqcup).$$

On input 101, the computation is as follows:

$$(s, \triangleright, 101) \xrightarrow{M^3} (q_1, \triangleright 01, \sqcup) \xrightarrow{M} (q, \triangleright 0, 01\sqcup) \xrightarrow{M} (s, \triangleright 0, \sqcup) \xrightarrow{M}$$

$$(q_0, \triangleright\triangleright, \sqcup) \xrightarrow{M} (q_0', \triangleright\triangleright, \sqcup) \xrightarrow{M} (\text{"yes"}, \triangleright\triangleright, \sqcup).$$

On input ϵ (the shortest palindrome in the world), here is the computation:

$$(s, \triangleright, \epsilon) \xrightarrow{M} (s, \triangleright\sqcup, \epsilon) \xrightarrow{M} (\text{"yes"}, \triangleright\sqcup, \epsilon).$$

1.5 Turing Machines as Algorithms

Recursive Language

Let $L \subset (\Sigma - \{\sqcup\})^*$ be a language, that is, a set of strings of symbols. Let M be a Turing machine such that, for any string $x \in (\Sigma - \{\sqcup\})^*$: If $x \in L$, then $M(x) =$ "yes" (that is, M on input x halts at the "yes" state), and if $x \notin L$, then $M(x) =$ "no".
Then we say that M *decides* L. If L is decided by some Turing machine M, then L is called a *recursive language*. For example, palindromes over $\{0,1\}^*$ constitute a recursive language decided by machine M defined previously.
We say that M simply *accepts* L whenever, for any string $x \in (\Sigma - \{\sqcup\})^*$: If $x \in L$, then $M(x) =$ "yes"; however, if $x \notin L$, then $M(x) = \nearrow$.
If L is accepted by some Turing machine M, then L is called *recursively enumerable.*

Proposition. *If L is recursive, then it is recursively enumerable.*

Proof. Suppose that there is a Turing machine M that decides L. We shall construct from M a Turing machine M' that *accepts* L, as follows: M' behaves exactly like M. Except that, whenever M is about to halt and enter state "no", M' moves to the right forever, and never halts. $\square$

Recursive Function

We shall not only deal with the decision and acceptance of languages, but also occasionally with the *computation of string functions*. Suppose that f is a function from $(\Sigma - \{\sqcup\})^*$ to Σ^*, and let M be a Turing machine with alphabet Σ. We say that M *computes* f if, for any string $x \in (\Sigma - \{\sqcup\})^*$, $M(x) = f(x)$. If such an M exists, f is called a *recursive function.*

1.6 Problems encoding

Thus, Turing machines can be thought of as algorithms for solving string-related problems. But how about our original project, to develop a notation for algorithms capable of attacking problems like those identified in the previous chapter, whose instances are mathematical objects such as graphs, networks, and numbers?

To solve such a problem by a Turing machine, we must decide how to *represent by a string* an instance of the problem. Once we have fixed this representation, an algorithm for a decision problem is simply a Turing machine that decides the corresponding language.

That is, it accepts if the input represents a ”yes” instance of the problem, and rejects otherwise. Similarly, problems that require more complex output, such as Max Flow, are solved by the Turing machine that computes the appropriate function from strings to strings (where the output is similarly represented as a string).

It should be clear that this proposal is quite general. Any ”finite” mathematical object of interest can be represented by a finite string over an appropriate alphabet. For example: - Elements of finite sets, such as the nodes of a graph, can be represented as integers in binary. - Pairs and k-tuples of simpler mathematical objects are represented by using parentheses and commas. - Finite sets of simpler objects are represented by using set brackets, and so on.

Or, perhaps, a graph can be represented by its *adjacency matrix*, which in turn can be arranged as a string, with rows separated by some special symbol such as ‘;‘.

There is a wide range of acceptable representations of integers, finite sets, graphs, and other such elementary objects. They may differ a lot in form and succinctness. *However, all acceptable encodings are related polynomially.* That is, if A and B are both ”reasonable” representations of the same set of instances, and representation A of an instance is a string with n symbols, then representation B of the same instance has length at most $p(n)$, for some polynomial p. For example, representing a graph with no isolated points by its adjacency matrix is at most quadratically more wasteful than representing it by an adjacency list.

Representing numbers in unary, instead of binary or decimal, is about the only possible slip in this regard. Obviously, the unary representation (in which, for example, number 14 is ”IIIIIIIIIIIIII”) requires exponentially more symbols than the binary representation. As a result, the complexity of an algorithm, measured as a function of the length of the input of the Turing machine, may seem deceptively favorable.

2 Multi-tape Turing Machine

Definition of Multi-tape Turing Machine

A *k-string Turing machine*, where $k \geq 1$ is an integer, is a quadruple $M = (K, \Sigma, \delta, s)$, where K, Σ, and s are exactly as in ordinary Turing machines. δ is a program that must reflect the complexities of multiple strings. Intuitively, δ decides the next state as before, but also decides *for each string* the symbol overwritten, and the direction of cursor motion by looking at the current state and the current symbol *at each string.*

Formally, δ is a function from $K \times \Sigma^k$ to

$$(K \cup \{h, \text{"yes"}, \text{"no"}\}) \times (\Sigma \times \{\leftarrow, \rightarrow, -\})^k.$$

Intuitively, $\delta(q, \sigma_1, \ldots, \sigma_k) = (p, \rho_1, D_1, \ldots, \rho_k, D_k)$ means that, if M is in state q, the cursor of the first string is scanning a σ_1, that of the second a σ_2, and so on, then the next state will be p, the first cursor will write ρ_1 and move in the direction indicated by D_1, and so on for the other cursors.

$\triangleright$ still cannot be overwritten or passed on to the left: If $\sigma_i = \triangleright$, then $\rho_i = \triangleright$ and $D_i = \rightarrow$. Initially, all strings start with a $\triangleright$; the first string also contains the input. The outcome of the computation of a k-string machine M on input x is as with ordinary machines, with one difference: In the case of machines that compute functions, the output can be read from the *last (kth) string* when the machine halts.

2.1 2-tape Turing Machine for palindromes

$p \in K,$	$\sigma_1 \in \Sigma$	$\sigma_2 \in \Sigma$	$\delta(p, \sigma_1, \sigma_2)$
$s,$	0	$\sqcup$	$(s, 0, \rightarrow, 0, \rightarrow)$
$s,$	1	$\sqcup$	$(s, 1, \rightarrow, 1, \rightarrow)$
$s,$	$\triangleright$	$\triangleright$	$(s, \triangleright, \rightarrow, \triangleright, \rightarrow)$
$s,$	$\sqcup$	$\sqcup$	$(q, \sqcup, \leftarrow, \sqcup, -)$
$q,$	0	$\sqcup$	$(q, 0, \leftarrow, \sqcup, -)$
$q,$	1	$\sqcup$	$(q, 1, \leftarrow, \sqcup, -)$
$q,$	$\triangleright$	$\sqcup$	$(p, \triangleright, \rightarrow, \sqcup, \leftarrow)$
$p,$	0	0	$(p, 0, \rightarrow, \sqcup, \leftarrow)$
$p,$	1	1	$(p, 1, \rightarrow, \sqcup, \leftarrow)$
$p,$	0	1	$(\text{"no"}, 0, -, 1, -)$
$p,$	1	0	$(\text{"no"}, 1, -, 0, -)$
$p,$	$\sqcup$	$\triangleright$	$(\text{"yes"}, \sqcup, -, \triangleright, \rightarrow)$

We can decide palindromes more efficiently with a 2-tape Turing machine. This machine starts by copying its input in the second string. Next,it positions the cursor of the first string at the first symbol of the input, and the cursor of the second string at the last symbol of the copy. Then, it moves the two cursors in opposite directions, checking that the two symbols under them are identical at all steps, at the same time erasing the copy.

2.2 Configuration of a Multi-tape Turing Machine

A configuration of a k-string Turing machine is defined analogously with ordinary Turing machines. It is a $(2k+1)$-tuple $(q, w_1, u_1, \ldots, w_k, u_k)$, where q is the current state, the ith string reads $w_i u_i$, and the last symbol of w_i is holding the ith cursor. We say that $(q, w_1, u_1, \ldots, w_k, u_k)$ yields in one step $(q', w'_1, u'_1, \ldots, w'_k, u'_k)$ denoted

$$(q, w_1, u_1, \ldots, w_k, u_k) \xrightarrow{M} (q', w'_1, u'_1, \ldots, w'_k, u'_k)$$

if the following is true.

First, suppose that σ_i is the last symbol of w_i, for $i = 1, \ldots, k$, and suppose that

$$\delta(q, \sigma_1, \ldots, \sigma_k) = (p, \rho_1, D_1, \ldots, \rho_k, D_k).$$

Then, for $i = 1, \ldots, k$ we have the following: - If $D_i = \rightarrow$, then w'_i is w_i with its last symbol (which was a σ_i) replaced by ρ_i, and the first symbol of u_i appended to it ($\sqcup$ if u_i is the empty string); u'_i is u_i with the first symbol removed (or, if u_i was the empty string, u'_i remains empty). - If $D_i = \leftarrow$, then w'_i is w_i with σ_i omitted from its end, and u'_i is u_i with ρ_i attached in the beginning. - Finally, if $D_i = -$, then w'_i is w_i with the ending σ_i replaced by ρ_i, and $u'_i = u_i$.

In other words, the conditions for yielding in single-string Turing machines must hold at each string. The relations "yields in n steps" and plain "yields" are defined analogously with ordinary Turing machines.

A k-string Turing machine starts its computation on input x with the configuration

$$(s, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright, \epsilon);$$

that is, the input is the first string, and all strings start with an $\triangleright$. If

$$(s, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright, \epsilon) \xrightarrow{M^*} (\text{"yes"}, w_1, u_1, \ldots, w_k, u_k),$$

for some strings $w_1, u_1, \ldots, u_k$, then we say that $M(x) =$ "yes"; if

$$(s, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright, \epsilon) \xrightarrow{M^*} (\text{"no"}, w_1, u_1, \ldots, w_k, u_k),$$

then we say that $M(x) =$ "no". Finally, if the machine halts at configuration $(h, w_1, u_1, \ldots, w_k, u_k)$, then $M(x) = y$, where y is $w_k u_k$ with the leading $\triangleright$ and all trailing $\sqcup$s removed. That is, if M halts at state h (the state that signals the output is ready), the output of the computation is contained in the last string.

Notice that, by these conventions, an ordinary Turing machine is indeed a k-string Turing machine with $k = 1$. Also, once the meaning of $M(x)$ has been defined, we can simply extend to multistream Turing machines the definitions of function computation, and language decision and acceptance of the previous section.

2.3 TIME class

TIME class

We shall use the multistreaming model of the Turing machine as the basis of our notion of the time expended by Turing machine computations (for space we shall need a minor modification introduced later in this chapter).
If for a k-string Turing machine M and input x we have

$$(s, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright, \epsilon) \xrightarrow{M^t} (H, w_1, u_1, \ldots, w_k, u_k)$$

for some $H \in \{h, \text{"yes"}, \text{"no"}\}$, then the time required by M on input x is t. That is, the time required is simply the number of steps to halting. If $M(x) = \nearrow$, then the time required by M on x is thought to be ∞ (this will rarely be the case in this book).
Defining the time requirements of a single computation is only the start. What we really need is a notion that reflects our interest in solving any instance of a problem, instead of isolated instances. Recall that the performance of algorithms in the previous chapter was characterized by the amount of time and space required on instances of “size” n, when these amounts were expressed as a function of n. For graphs, we used the number of nodes as a measure of “size.” For strings, the natural measure of size is the length of the string. Accordingly, let f be a function from the nonnegative integers to the nonnegative integers. We say that machine M operates within time $f(n)$ if, for any input string x, the time required by M on x is at most $f(|x|)$ (by $|x|$ we denote the *length* of string x). Function $f(n)$ is a time bound for M.
Suppose now that a language $L \subset (\Sigma - \{\sqcup\})^*$ is decided by a multistream Turing machine operating in time $f(n)$. We say that $L \in \text{TIME}(f(n))$. That is, $\text{TIME}(f(n))$ is a set of languages. It contains exactly those languages that can be decided by Turing machines with multiple strings operating within the time bound $f(n)$. **TIME**$(f(n))$ is what we call a *complexity class*. It is a set of languages (hopefully, including many that represent important decision problems). The property shared by these languages is that they can all be decided within some specified bound on some aspect of their performance (time, soon space, and later others). Complexity classes, and their relationship with the problems they contain, are the main objects of study in this book.

2.4 Speed up Multi-tape

Theorem (Speed up Multi-tape)

Given any k-string Turing Machine M operating within time $f(n)$, we can construct single string Turing Machine M' operating within time $O(f(n)^2)$ and such that, for any input x, $M(x) = M'(x)$

Proof. Suppose that $M = (K, \Sigma, \delta, s)$; We shall describe $M' = (K', \Sigma', \delta', s)$. M' single string must "simulate" the k strings of M.

One way to do this would be to maintain in M''s string the concatenation of the strings of M (without, of course the $\triangleright_s$ that would come in the way of going back and forth in the string). We must also "remember" the position of each cursor, as well as the current right end of each string. To accomplish all this, we let

$$\Sigma' = \Sigma' \cup \underline{\Sigma} \cup \{\triangleright', \triangleleft\}$$

Here $\underline{\Sigma} = \{\underline{\sigma} : \sigma \in \Sigma\}$ is a set of cursor versions of the symbol in Σ. $\triangleright'$ is a new version of $\triangleright$ which can be passed over to the left, and $\triangleleft$ marks the right end of a string.

Thus, any configuration $(q, w_1, u_1, \ldots, w_k, u_k)$ can be simulated by the configurations of M' $(q, \triangleright, w_1'u_1 \triangleleft w_2'u_2 \triangleleft \ldots w_k'u_k \triangleleft \triangleleft)$. Here w_i' is w_i with the leading $\triangleright$ replaced by $\triangleright'$, and the last symbol σ_i by $\underline{\sigma_i}$. The last two $\triangleleft$s signal the end of M''s string.

For the simulation to begin, M' has simply to shift its input one position to the right, precede it with a $\triangleright'$, and write the string $\triangleleft(\triangleright'\triangleleft)^{k-1}\triangleleft$ after its input. This can be easily accomplished by adding to the states of M' $2k+2$ new states, whose sole purpose is to perform this writing.

To simulate a move of M, M' scans twice its string from left to right and back. In the first scan, M' gathers information concerning the k currently scanned symbols in M: They are the k underlined symbols encountered. To do this "remembering," M' must contain new states, each of which corresponds to a particular combination of a state of M and of a k-tuple of symbols of M.

Based on its state at the end of the first scan, M' knows what changes need to be performed on the string to reflect changes in the strings of M at the move being simulated. Then M' scans its string again from left to right, stopping at each underlined symbol to rewrite one or two symbols nearby, in a manner that reflects the symbols overwritten and the cursor motions of M at this string during this move. These updates can be easily performed based on the information available to M'.

There is however one complication: If a cursor scanning the right end of a string needs to move right, we must create space for a new symbol (a $\sqcup$). This is done by first marking the currently scanned $\triangleleft$ by a special mark, making it $\triangleleft'$, moving M''s cursor all the way to the $\triangleleft$ on the right end, and moving all symbols one position to the right, as was done by the Turing machine in Example 2.1. When the $\triangleleft'$ is encountered, it is moved to the right as a simple $\triangleleft$, and a $\sqcup$ is overwritten in its old position. Then we go on to implement the changes in the next string of M.

The simulation proceeds until M halts. At this point, M' erases all strings of M except the last (so that its output is the same as that of M) and halts.

How long does the operation of M' on an input x take? Since M halts within time $f(|x|)$, during its operation none of its strings ever becomes longer than $f(|x|)$ (this is a very basic fact about any reasonable model of computation: It cannot waste more space than time!). Thus the total length of the string of M' is never more than $k(f(|x|)+1)+1$ (to account for the $\triangleleft$s). Simulating a move thus takes at most two traversals of this string from left to right and back ($4k(f(|x|) + 1) + 4$ steps), plus at most $3k(f(|x|) + 1) + 3$ steps per each string of M simulated. The total is $\mathcal{O}(k^2 f(|x|)^2)$, or, since k is fixed and independent of x, $\mathcal{O}(f(|x|)^2)$. □

2.5 Linear Speedup Theorem

As a result of this theorem, we have that e.g.

$$\text{if } L \in \textbf{TIME}(150n^2) \text{ then } L \in \textbf{TIME}(n^2 + n + 2)$$

From this we can conclude that, $O()$ is the only significative notation for time complexity.

Linear Speedup Theorem

Let $L \in \textbf{TIME}(f(n))$. Then, for any $\epsilon > 0$, $L \in \textbf{TIME}(f'(n))$, where

$$f'(n) = \epsilon f(n) + n + 2$$

2.6 Space Bounds

At first it seems straightforward to define the space used by the computation

$$(s, \triangleright, \epsilon, \ldots, \triangleright, \epsilon) \xrightarrow{M^*} (H, w_1, u_1, \ldots, w_k, u_k)$$

of a Turing machine with k strings. Since the strings cannot become shorter in our model, the lengths of the final strings $w_i u_i$, $i = 1, \ldots, k$, capture the amount of storage required by each during the computation. We can either add up these lengths, or take their maximum. There are arguments in favor of both approaches, but we shall cut the discussion short by noting that the two estimates differ by at most k, a constant. Let us say for the time being that the space used by the machine M on x is

$$\sum_{i=1}^{k} |w_i u_i|.$$

It turns out that this estimate represents a serious overcharge. Consider the following example:

Example

Can we recognize palindromes in space substantially less than n?
In view of our definition of the space used by a computation, the sum of the lengths of the strings at halting, this is impossible: One of these lengths, namely that of the first string carrying the input, will always be at least $|x|+1$.
But consider the following 3-string Turing machine that recognizes palindromes: The first string contains the input, and is never overwritten. The machine works in stages. At the ith stage the second string contains integer i in binary. During the ith stage we try to identify and remember the ith symbol of x. We do this by initializing the third string to $j=1$, and then comparing i and j. Comparisons of the contents of the two different strings are easy to do by moving the two cursors. If $j<i$, then we increment j in binary (recall the machine in Example 2.2), and advance the first cursor to the right, to inspect the next input symbol. If $i=j$ we remember in our state the currently scanned symbol of the input, we reinitialize j to 1, and try to identify the ith from the last symbol of x. This is done much in the same way, with the first cursor now marching from right to left instead from left to right.
If the two symbols are unequal, we halt with "no". If they are equal, we increment i by one, and begin the next stage. If, finally, during a stage, the alleged ith symbol of x turns out to be a $\sqcup$, this means that $i>n$, and the input is a palindrome. We halt with a "yes".

What space is needed for the operation of this machine? It seems fair to assess that the space used up by this machine is $\mathcal{O}(\log n)$. The machine surely looks at the input (computation would be impossible otherwise) *but in a read-only fashion.* There is no writing on the first string. The other two strings are kept at most $\log n+1$ long. □

This example leads us to the following definition: Let $k>2$ be an integer. A *k-string Turing machine with input and output* is an ordinary k-string Turing machine, with one important restriction on the program δ: Whenever

$$\delta(q,\sigma_1,\ldots,\sigma_k)=(p,\rho_1,D_1,\ldots,\rho_k,D_k),$$

then (a) $\rho_1=\sigma_1$, and (b) $D_k\neq\leftarrow$. Furthermore, (c) if $\sigma_1=\sqcup$, then $D_1=\leftarrow$.

Requirement (a) says that at each move, the symbol "written" on the first string is always the same as the old symbol, and hence the machine effectively has a read-only input string. Requirement (b) states that in the last (output) string the cursor never moves to the left, and hence the output string is effectively write-only. Finally, (c) guarantees that the cursor of the input string does not wander off into the $\sqcup$s after the end of the input. It is a useful technical requirement, but not necessary.

These restrictions lead to an accurate definition of space requirements, without "overcharging" the computation for reading the input and for providing output, that is, for functions involving no "remembering" of earlier results of the computation. With these restrictions we shall be able to study space bounds smaller than n. However, let us first observe that these restrictions do not change the capabilities of a Turing machine.

Proposition

For any k-string Turing machine M operating within time bound $f(n)$ there is a $(k+2)$-string Turing machine M' with input and output, which operates within time bound $\mathcal{O}(f(n))$.

Proof. Machine M' starts by copying its input on the second string, and then simulating the k strings of M on its strings numbered 2 through $k+1$. When M halts, M' copies its output to the $k+2$nd string, and halts. $\square$

Space Complexity Class

Suppose that, for a k-string Turing machine M and an input x

$$(s, \triangleright, x, \ldots, \triangleright, \epsilon) \xrightarrow{M^*} (H, w_1, u_1, \ldots, w_k, u_k),$$

where $H \in \{h, \text{"yes"}, \text{"no"}\}$ is a halting state. Then the *space required by M on input x* is

$$\sum_{i=1}^{k} |w_i u_i|.$$

If, however, M is a machine with input and output, then the space required by M on input x is

$$\sum_{i=2}^{k-1} |w_i u_i|.$$

Suppose now that f is a function from $\mathbb{N}$ to $\mathbb{N}$. We say that *Turing machine M operates within space bound $f(n)$* if, for any input x, M requires space at most $f(|x|)$.

Finally, let L be a language. We say that L is in the *space complexity class* $\mathbf{SPACE}(f(n))$ if there is a Turing machine with input and output that decides L and operates within space bound $f(n)$. For example, palindromes were shown to be in the space complexity class $\mathbf{SPACE}(\log n)$. This important space complexity class is usually referred to as $\mathbf{L}$.

Theorem

Let L be a language in $\mathbf{SPACE}(f(n))$. Then, for any $\epsilon > 0$,

$$L \in \mathbf{SPACE}(2 + \epsilon f(n)).$$

3 Nondeterministic Turing Machines

We shall now break our chain of "reasonable" models of computation that can simulate each other with only a polynomial loss of efficiency. We shall introduce an *unrealistic* model of computation, the nondeterministic Turing machine. And we shall show that it can be simulated by our other models with an *exponential* loss of efficiency.

Nondeterministic Turing Machine

A *nondeterministic Turing machine* is a quadruple $N = (K, \Sigma, \Delta, s)$, much like the ordinary Turing machine. K, Σ, and s are as before. Reflecting the fact that a nondeterministic Turing machine does not have a single, uniquely defined next action, but a choice between several next actions, Δ is no longer a function from $K \times \Sigma$ to $(K \cup \{h, \text{"yes"}, \text{"no"}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}$, but instead a *relation* $\Delta \subseteq (K \times \Sigma) \times [(K \cup \{h, \text{"yes"}, \text{"no"}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}]$. That is, for each state-symbol combination, there may be *more than one* appropriate next step—or none at all.

Nondeterministic Turing machine configurations are exactly like configurations of deterministic machines, but "yields" is no longer a function; it is instead a *relation*. We say that configuration (q, w, u) of the nondeterministic Turing machine N *yields* configuration (q', w', u') in one step, denoted

$$(q, w, u) \xrightarrow{N} (q', w', u'),$$

intuitively if there is a rule in Δ that makes this a legal transition. Formally, it means that there is a move $((q, \sigma), (q', \rho, D)) \in \Delta$ such that, either (a) $D =\rightarrow$, and w' is w with its last symbol (which was a σ) replaced by ρ, and the first symbol of u appended to it ($\sqcup$ if u is the empty string), and u' is u with the first symbol removed (or, if u was the empty string, u' remains empty); or (b) $D =\leftarrow$, w' is w with σ omitted from its end, and u' is u with ρ attached in the beginning; or finally, (c) $D = -$, w' is w with the ending σ replaced by ρ, and $u' = u$. $N^k \xrightarrow{k}$ and $N^* \xrightarrow{*}$ can be defined as usual, but $N^k \xrightarrow{k}$ is, once more, no longer a function.

What makes nondeterministic machines so different and powerful is the very weak "input-output behavior" we demand of them, that is, our very liberal notion of what it means for such a machine to "solve a problem." Let L be a language and N a nondeterministic Turing machine. We say that N *decides* L if for any $x \in \Sigma^*$, the following is true:

$$x \in L \iff (s, \triangleright, x) \xrightarrow{N^*} (\text{"yes"}, w, u)$$

for some strings w and u.

This is the crucial definition that sets nondeterministic machines apart from other models. An input is accepted if there is *some* sequence of nondeterministic choices that results in "yes." Other choices may result in rejection; just one accepting computation is enough. The string is rejected only if no sequence of choices can lead to acceptance.

We say that nondeterministic Turing machine N decides language L *in time* $f(n)$, where f is a function from the nonnegative integers to the nonnegative integers, if N decides L, and, moreover for any $x \in \Sigma^*$, if

$$(s, \triangleright, x) \stackrel{N^k}{\rightarrow} (q, u, w),$$

then $k \leq f(|x|)$. That is, we require that N, besides deciding L, does not have computation paths longer than $f(n)$, where n is the length of the input. Thus, we charge to a nondeterministic computation an amount of time that may be considered unrealistically small. The amount of time charged is the "depth" of the computational activity. Obviously, the "total computational activity" generated can be exponentially bigger.

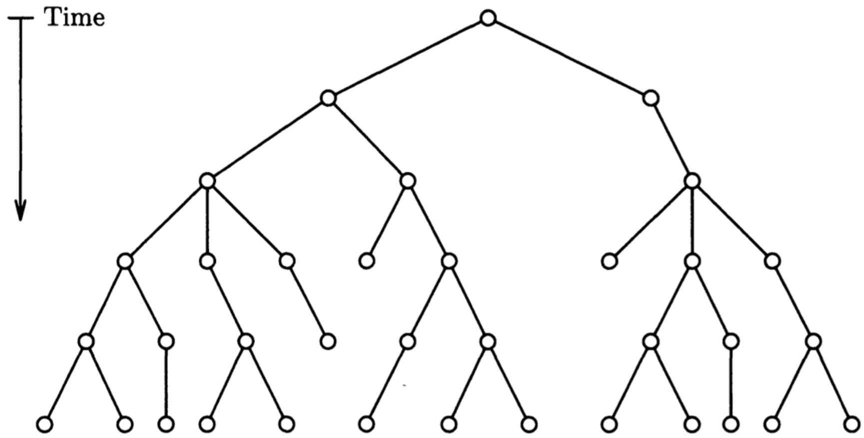

Figure 1: Nondeterministic computation

The set of language decided by nondeterministic Turing machines within time f is a new, important kind of complexity class, denoted **NTIME**$(f(n))$. A most important nondeterministic complexity class is **NP**, the union of all **NTIME**(n^k). Notice immediately that $\mathbf{P} \subseteq \mathbf{NP}$; the reason is that deterministic machines from a subclass of the nondeterministic ones: They are precisely those for which the relation Δ happens to be a function.

Example

Recall that we do not know whether the decision version of the traveling salesman problem, TSP (D) (recall Section 1.3), is in **P**. However, it is easy to see that TSP (D) is in **NP**, because it can be decided by a nondeterministic Turing machine in time $\mathcal{O}(n^2)$. This machine, on an input containing a representation of the TSP instance, goes on to write an arbitrary sequence of symbols, no longer than its input. When this writing stops, the machine goes back and checks to see whether the string written is the representation of a permutation of the cities, and, if so, whether the permutation is a tour with cost B or less. Both tasks can easily be carried out in $\mathcal{O}(n^2)$ time by using a second string (multistring nondeterministic Turing machines are straightforward to define; they can be simulated by single-string ones again with a quadratic slowdown). If the string indeed encodes a tour cheaper than B, the machine accepts; otherwise, it rejects (perhaps hoping that another choice of symbols, written at some other branch of the tree in Figure 2.9, will end up accepting).

This machine indeed decides TSP (D), because it accepts its input if and only if it encodes a "yes" instance of TSP (D). If the input is a "yes" instance, this means that there is a tour of the cities with cost B or less. Therefore, there will be a computation of this machine that "guesses" precisely this permutation, checks that its cost is indeed below B, and thus ends up accepting. It does not matter that other computations of the machine may have guessed very costly tours, or plain nonsense strings, and will therefore reject; a single accepting computation is all that is needed for the machine to accept. Conversely, if there are no tours of cost less than B, there is no danger that a computation will accept: All computations will discover that their guess is wrong, and will reject. Thus, according to our convention, the input will be rejected. □

It would be remarkable if we could somehow turn this nondeterministic idea into a viable, polynomial-time algorithm for TSP (D). Unfortunately, at present we know of only exponential general methods for turning a nondeterministic algorithm into a deterministic one. We describe the most obvious such method next.

Theorem (from nondeterministic to deterministic)

Suppose that language L is decided by a nondeterministic Turing machine N in time $f(n)$. Then it is decided by a 3-string deterministic Turing machine M in time $\mathcal{O}(c^{f(n)})$, where $c > 1$ is some constant depending on N.

Notice that, using complexity class notation, we can restate this result very succinctly:

$$\mathbf{NTIME}(f(n)) \subseteq \bigcup_{c>1} \mathbf{TIME}(c^{f(n)})$$

Proof. Let $N = (K, \Sigma, \Delta, s)$. For each $(q, \sigma) \in K \times \Sigma$, consider the set of choices $C_{q,\sigma} = \{(q', \sigma', D) : ((q, \sigma), (q', \sigma', D)) \in \Delta\}$. $C_{q,\sigma}$ is a finite set. Let $d = \max_{q,\sigma} |C_{q,\sigma}|$ (this d could be called the "degree of nondeterminism" of N), and assume that $d > 1$ (otherwise, the machine is deterministic, and there is nothing to do).

The basic idea for simulating N is this: Any computation of N is a sequence of nondeterministic choices. Now, any sequence of t nondeterministic choices made by N is essentially a sequence of t integers in the range $0, 1, \ldots, d-1$. The simulating deterministic machine M considers all such sequences of choices, in order of increasing length, and simulates N on each (notice that we cannot simply consider the sequences of length $f(|x|)$, where x is the input, because M must operate with no knowledge of the bound $f(n)$). While considering a sequence $(c_1, c_2, \ldots, c_t)$, M maintains the sequence on its second string. Then M simulates the actions that N would have taken had N taken choice c_i at step i for its first t steps. If these choices would have led N to halting with a "yes" (possibili earlier than tth step), then M halts with a "yes" as well. Otherwise, M proceeds to the next sequence of choices. Generating the next sequence is akin to calculating the next integer in d-ary, and can be done easily. How can M detect that N rejects, with no knowledge of the bound $f(n)$? The solution is simple: If M simulates all sequences of choice of a particular length t, and finds among them *no continuing computation* (that is, all computations of length t end with halting, presumably with "no" or h), then M can conclude that N rejects its input.

The time required by M to complete the simulation is bounded from above by

$$\sum_{t=1}^{f(n)} d^t = O(d^{f(n)+1})$$

(the number of sequences it needs to go through) times the time required to generate and consider each sequence, and this latter cost is easily seen to be $O(2^{f(n)})$. □

We shall also be very interested in the *space* used by nondeterministic Turing machines. To do this correctly for space bounds smaller than linear, we shall need to define a *nondeterministic Turing machine with input and output.* It is a straightforward task (which we omit) to combine the two extensions of Turing machines into one.

Given an k-string nondeterministic Turing machine with input and output $N = (K, \Sigma, \Delta, s)$, we say that N decides language L within space $f(n)$ if N decides L and, moreover, for any $x \in (\Sigma - \{\sqcup\})^*$, if

$$(s, \triangleright, x, \ldots, \triangleright, \epsilon) \xrightarrow{N^*} (q, w_1, u_1, \ldots, w_s, u_s),$$

then

$$\sum_{j=2}^{s-1} |w_j u_j| \leq f(|x|).$$

That is, we require that N under no circumstances uses space in its "scratch" strings greater than function f in the input length. Notice that our definition does not even require N to halt on all computations.

4 Proper Complexity Functions

Definition (Proper Complexity Function)

Let f be a function from the nonnegative integers to the nonnegative integers. We say that f is a *proper complexity function* if f is nondecreasing (that is, $f(n+1) \geq f(n)$ for all n), and furthermore the following is true: There is a k-string Turing machine $M_f = (K, \Sigma, \delta, s)$ *with input and output* such that, for any integer n, and any input x of length n,

$$(s, \triangleright, x, \triangleright, \epsilon, \ldots, \triangleright, \epsilon) \xrightarrow{M_f^t} (h, x, \triangleright, \sqcup^{j_2}, \triangleright, \sqcup^{j_3}, \ldots, \triangleright, \sqcup^{j_{k-1}}, \triangleright, \sqcup^{f(|x|)}),$$

such that $t = O(n + f(n))$, and the $j_i = O(f(|x|))$ for $i = 2, \ldots, k-1$, with t and the j_i's depending only on n. In other words, on input x, M_f computes the string $\sqcup^{f(|x|)}$, where $\sqcup$ is a "quasi-blank" symbol. And, on any input x, M_f halts after $O(|x| + f(|x|))$ steps and uses $O(f(|x|))$ space besides its input. □

Using only proper complexity functions we can standardize the behavior of Turing machines.

Precise Turing Machines

A Turing Machine (of any type) is *precise* if and only if there exist two proper functions f and g such that, for any input x, every computation of M on x halts after exactly $f(|x|)$ steps, and every tape of M beside the input and the output tapes are long exactly $g(|x|)$.

Theroem (Precise Solutions)

Let M be any Turing machine. If M decides a language L in time or space $f(n)$, and f is a proper complexity function, then there exist a precise Turing machine M' that decides L in time or space (respectively) $O(f(n))$.

Proof. In all four cases (deterministic time, deterministic space, nondeterministic time, or nondeterministic space) the machine M' on input x starts off by simulating the machine M_f associated with the proper function f on x, using a new set of strings. After M_f's computation has halted, M uses the output string of M_f as a "yardstick" of length $f(|x|)$, to guide its further computation.

If $f(n)$ is a time bound, then M' simulates M on a different set of strings, *using the yardstick as an "alarm clock"*. That is, it advances its cursor on the yardstick after the simulation of each step of M, and halts if and only if a true blank is encountered, after precisely $f(|x|)$ steps of M. If $f(n)$ is a space bound, then M' simulates M on the quasiblanks of M_f's output string. In either case, the machine is precise.

If M is nondeterministic, then precisely the same amount of time or space is used over all possible computations of M' on x, as this amount depends only on the deterministic phase of M' (the simulation of M_f). In both cases, the time or space is the time or space consumed by M_f, plus that consumed by M, and is therefore $\mathcal{O}(f(n))$.

In the case of space-bounded machines, the "space yardstick" can be used by M' to also "count" steps in an appropriate base representing the size of the alphabet of M, and thus M' will never indulge in a meaninglessly long computation. Thus, we can assume that space-bounded machines always halt. □

Theorem (Termination of TM space bounded)

If M decides L in space $f(n)$, with f being a proper function then exists a Turing machine M' that decides L in space $O(f(n))$ and such that every computation halts.

Proof. If M works in space $f(n)$, then the set of different configurations in which it can go through is finite. Upper bounder by

$$c = |K| \times |\Sigma|^{f(n)} \times (f(n) + 1)$$

If M makes more than c steps, then it must have repeated a configuration and so it must be in a loop. If M doesn't halt in at most c steps we can force termination in "no". Proof for 1 tape: concatenates M_f and a simulation of M. The last tape of M_f is used as a yardstick. This tape will contain a cell in which we have the symbol representing the current state, it can contain $|K|$ different symbols. Then the next $f(n)$ cells will contain the symbols of Σ (represented by an integer in base $|\Sigma|$). The last $(f(n) + 1)$ cells will be used as an unary representation of the cursor position. At every transition of M we increment the counter and if in at most c steps M has not halted, we can force it to halt in "no". □

From this theorem we ca conclude that if the space is finite then the machine will halt. Another important result is that time is exponentially bigger than space.

5 Complexity Classes

$$
\begin{aligned}
&\mathrm{TIME}(n^k) &&= \bigcup_{j>0} \mathrm{TIME}(n^j) &&= \mathbf{P}, \\
&\mathrm{NTIME}(n^k) &&= \bigcup_{j>0} \mathrm{NTIME}(n^j) &&= \mathbf{NP}, \\
&\mathrm{TIME}(2^{n^k}) &&= \bigcup_{j>0} \mathrm{TIME}(2^{n^j}) &&= \mathbf{EXP}, \\
&\mathrm{SPACE}(n^k) &&= \bigcup_{j>0} \mathrm{SPACE}(n^j) &&= \mathbf{PSPACE}, \\
&\mathrm{NSPACE}(n^k) &&= \bigcup_{j>0} \mathrm{NSPACE}(n^j) &&= \mathbf{NPSPACE}, \\
&\mathrm{SPACE}(\log n) && &&= \mathbf{L}, \\
&\mathrm{NSPACE}(\log n) && &&= \mathbf{NL}.
\end{aligned}
$$

5.1 Exercises

- $TIME(n^3) \subseteq EXP$? Yes
- $TIME(n^3) \subseteq P$? Yes
- $EXP \subseteq P$? No (there is a theorem that proves it)
- $P \subseteq NTIME(n^k)$? Yes, because $P \subseteq NP$.
- $NTIME(n^k) \subseteq P$? We don't know.
- $NTIME(n^k) \subseteq NP$? Yes, because $NTIME(n^k) = NP$.
- $TIME(2^{n^2}) \subseteq EXP$? Yes, because $EXP = \bigcup_{k>0} TIME(2^{n^k})$.
- $NTIME(n^2) \subseteq EXP$? Yes, because

  $$NTIME(n^2) \subseteq TIME(k^{n^2}) = TIME(2^{(log_2 k)n^2}) \subseteq TIME(2^{n^3}) \subseteq EXP$$

  to switch from nondeterministic to deterministic we used the theorem that says that nondeterministic Turing machines can be simulated by deterministic Turing machines with an exponential loss of efficiency.
- $PSPACE \subseteq SPACE(n^k)$? Yes, because $PSPACE = \bigcup_{k>0} SPACE(n^k)$.
- $PSPACE \subseteq NPSPACE$? Yes, because deterministic Turing machines are a subset of nondeterministic Turing machines.
- $SPACE(2^n) \subseteq SPACE(2^{n^2})$? Yes
- $SPACE(n) \subseteq NSPACE(n)$? Yes

- $NPSPACE \subseteq PSPACE$? Yes, (proven by Savitch corollary)
- $TIME(n^k) \subseteq PSPACE$? Yes, because $TIME(n^k) \subseteq SPACE(n^k) = PSPACE$, obviously if we have a Turing machine that works in time n^k then it will use at most n^k space.
- $PSPACE \subseteq EXP$? Yes, because we can upper bound time number of computations by $c = |k| \times |\Sigma|^{f(n)} \times f(n)$ which is exponential in $f(n)$.
- $NP \subseteq PSPACE$? Yes, because we can prove that any NP problem can be solved in polynomial space.

6 Complementary Complexity Classes

When we first defined nondeterministic computation, we were struck by an asymmetry in the way "yes" and "no" inputs are treated. For a string to be established as a string in the language (a "yes" input), one successful computational path is enough. In contrast, for a string not in the language, all computational paths must be unsuccessful.

Complement of a Language

Let $L \subseteq \Sigma^*$ be a language. The *complement* of L, denoted $\overline{L}$, is the language $\Sigma^* - L$, that is, the set of all strings in the appropriate alphabet that are not in L. We now extend this definition to decision problems. The complement of a decision problem A, usually called *A complement*, will be defined as the decision problem whose answer is "yes" whenever the input is a "no" input of A, and vice versa.

For example, *SAT complement* is this problem: Given a Boolean expression ϕ in conjunctive normal form, is it *unsatisfiable*? *Hamilton path complement* is the following: Given a graph G, is it true that G *does not* have a Hamilton path? And so on. Notice that, strictly speaking, the languages corresponding to the problems *Hamilton path* and *Hamilton path complement*, for example, are not complements of one another, as their union is not Σ^* but rather the set of all strings that encode graphs; this is of course another convenient convention with no deep consequences.

Complementary Complexity Class

For any complexity class C, **co**C denotes the class $\{\overline{L} : L \in C\}$. It is immediately obvious that if C is a deterministic time or space complexity class, then $C = \mathbf{co}C$; that is, *all deterministic time and space complexity classes are closed under complement.* The reason is that any deterministic Turing machine deciding L within a time or space bound can be converted to a deterministic Turing machine that decides $\overline{L}$ within the same time or space bound: The same machine with the roles of "yes" and "no" reversed. But it is an important open problem whether nondeterministic time complexity classes are close under complement. For nondeterministic space complexity classes we can prove closure under complement, but we will see that later.

7 The Hierarchy Theorem

We shall assume that, although a machine may have an arbitrary alphabet, the languages of interest, and therefore the inputs x, contain the symbols used for encoding Turing machines (0, 1, "(", ";", etc.). This is no loss of generality, since languages over different alphabets obviously have identical complexity properties, as long as these alphabets contain at least two symbols. We represent the states and the alphabet symbols of turing machines as integers:

- $\Sigma \rightsquigarrow \{1, \ldots, |\Sigma|\}$
- $K \rightsquigarrow \{|\Sigma| + 1, \ldots, |\Sigma| + |K|\}$
- $\{\leftarrow, \rightarrow, -, h, \text{"yes"}, \text{"no"}\} \rightsquigarrow \{|\Sigma| + |K| + 1, \ldots, |\Sigma| + |K| + 6\}$

Every integer is represented in binary, with strings of length $log(|\Sigma|+|K|+6)$. We denote with $\underline{i}$ the binary encoding of the integer i, with $\underline{q}$ the encoding of the state $q \in K$ and with $\underline{\sigma}$ the encoding of the symbol $\sigma \in \Sigma$. Codifica di $(q, \sigma, q', \sigma', D) \in \Delta$:$(\underline{q}, \underline{\sigma}, \underline{q}', \underline{\sigma}', \underline{D})$. Codifica di Δ, denotata con $\underline{\Delta}$:$(\underline{q}_1, \underline{\sigma}_1, \underline{q}'_1, \underline{\sigma}'_1, \underline{D}_1), \ldots, (\underline{q}_m, \underline{\sigma}_m, \underline{q}'_m, \underline{\sigma}'_m, \underline{D}_m) \quad (m = |\Delta|)$. Codifica di M, denotata con $\underline{M}$:$|\underline{\Sigma}|, |\underline{K}|, \underline{\Delta}$.

Universla Turgin Machine

There exist a **universal** turing machine U that given the string $\underline{M}; x$ can simulate M on input x. We describe U as a two tape turing machine since we know that it can be reduced to a one tape turing machine with just a quadratic slowdown. U keeps the configuration of M on the second tape with such a format: $(\underline{w, q, u})$ where w is the left side of the string with respect to the cursor, q is the state of the machine and u is the right side of the string. U simulates M with the following steps: searches for the encoding of the current state q in the second tape, then it searches in the first tape for a tuple $(q, \sigma, \dots)$ starting with the q state. Then it keeps moving left on the second string until it reaches the left and of w to verifies that the cursor is under the symbol σ. If this is true then it writes the new symbol σ' in the first tape and moves the cursor according to D otherwise searches for the next tuple starting with q and keeps repeating these steps. If it finds an encoding error for $\underline{M}$ it keeps moving the cursor to the right forever (diverging).

In this section, we prove a *quantitative hierarchy result*: With sufficiently greater time allocation, Turing machines are able to perform more complex computational tasks. Predictably, our proof will employ a quantitative sort of diagonalization.

Let $f(n) \geq n$ be a proper complexity function, and define H_f to be the following time-bounded version of the *halting* language H:

$$H_f = \{M; x \mid M \text{ accepts input } x \text{ after at most } f(|x|) \text{ steps}\},$$

where M in the definition of H_f ranges over all descriptions of deterministic multi-string Turing machines.

Lemma (complexity upper bound of H_f)

$$H_f \in TIME(f(n)^3)$$

Proof. We shall describe a Turing machine U_f with four strings, which decides H_f in time $(f(n))^3$. U_f is based on several machines that we have seen previously: The universal Turing machine U; the single-string simulator of multi-string machines; the linear speedup theorem machine that shaves constants off the time bound; and the ,machine M_f that computes a "yardstick" of length precisely $f(n)$, which exists because we are assuming that f is a proper complexity function. First, U_f uses M_f on the second part of its input, x, to initialize on its fourth string an appropriate "alarm clock" $\triangleright\sqcup^{f(|x|)}$, to be used during the simulation of M. (Here we assume that M_f has at most four strings; if not, the number of strings of U_f has to be increased accordingly.) M_f operates within time $\mathcal{O}(f(|x|))$ (where the constant depends on f alone, and not on x or M). U_f also copies the description of the machine M to be simulated on its third string, and converts x on its first string to the encoding of $\triangleright x$ (that is, it removes $\underline{M}$ from the first string and adds a $\triangleright$

at the beginning of x). The second string is initialized to the encoding of the initial state s (that is, it replaces $\underline{(w, q, u)}$ with only $\underline{q}$). We can also at this point check that indeed $\underline{M}$ is the description of a legitimate Turing machine, and reject if it is not (this requires linear time using two strings). The total time required so far is $\mathcal{O}(f(|x|) + n) = \mathcal{O}(f(n))$ where n is the size of the input $\underline{M}; x$.

The main operation of U_f starts after this initial stage. Like the U we defined before (universal turing machine), U_f simulates one-by-one the steps of M on input x. The simulation is confined in the first string, where the encodings of the contents of all strings of M are kept. Each step of M is simulated by two successive scans of the first string of U_f. During the first scan, U_f collects all relevant information concerning the currently scanned symbols of M, and writes this information on the second string. The second string also contains the encoding of the current state. U_f then matches the contents of the second string with those of the third (containing the description of M) to find the appropriate transition of M. U_f goes on to perform the appropriate changes to its first string, in a second pass. It also advances its "alarm clock" by one.

U_f simulates each step of M in time $\mathcal{O}(\ell_M k_M^2 f(|x|))$, where k_M is the number of strings of M, and ℓ_M is the length of the description of each state and symbol of M. Since, for legitimate Turing machines, these quantities are bounded above by the logarithm of the length of the description of the machine, the time to simulate each step of M is $\mathcal{O}(f^2(n))$, where, once more, the constant does not depend on M.

If U_f finds that M indeed accepts x within $f(|x|)$ steps it accepts its input $M; x$. If not (that is, if M rejects x, or if the alarm clock expires) then U_f rejects its input. The total time is $\mathcal{O}(f(n)^3)$. It can be easily made at most $f(n)^3$, by modifying U_f to treat several symbols as one, as in the proof of the linear speedup theorem. □

Lemma (complexity lower bound of H_f)

$$H_f \notin \mathrm{TIME}\left(f\left(\left\lfloor \frac{n}{2} \right\rfloor\right)\right)$$

Proof. Suppose for the sake of contradiction, that there is a Turing machine M_{H_f} that decides H_f in time $f\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$. This leads to the construction of a "diagonalizing" machine D_f, with the following behavior:

$$D_f(\underline{M}) : \text{if } M_{H_f}(\underline{M}; \underline{M}) = \text{"yes" then "no" else "yes"}.$$

D_f on input $\underline{M}$ runs in the same time as M_{H_f} on input $\underline{M}; \underline{M}$, that is, in time

$$f\left(\left\lfloor \frac{2n+1}{2} \right\rfloor\right) = f(n).$$

Does then D_f accept its own description? Suppose that $D_f(\underline{D_f}) =$ "yes". This means that $M_{H_f}(\underline{D_f}; \underline{D_f}) =$ "no", or, equivalently, $\underline{D_f}; \underline{D_f} \notin H_f$. By the definition of H_f, this means that D_f fails to accept its own description in $f(n)$ steps, and, since we know that D_f always accepts or rejects its input within $f(n)$ steps, $D_f(\underline{D}_f) =$ "no".

Similarly, $D_f(\underline{D}_f) =$ "no" implies $D_f(\underline{D}_f) =$ "yes", and our assumption that $H_f \in \mathrm{TIME}\left(f\left(\lfloor \frac{n}{2} \rfloor\right)\right)$ has led us to a contradiction. $\square$

The Time Hierarchy Theorem

If $f(n) \geq n$ is a proper complexity function, then the class $\mathrm{TIME}(f(n))$ is strictly contained within $\mathrm{TIME}((f(2n+1))^3)$.

Proof. Since $f(n)$ is a proper complexity function and $f(n) \geq n$, we have that

$$f(n) \leq f(2n+1) \leq f(2n+1)^3$$

and so $TIME(f(n)) \subseteq TIME(f(2n+1)^3)$.

To prove the strict inclusion, we shall dimostrate that for an appropriate g,

$$H_g \in TIME(f(2n+1)^3) \text{ but } H_g \notin TIME(f(n))$$

Let $g(n) = f(2n+1)$. Notice that

$$2\left\lfloor \frac{n}{2} \right\rfloor + 1 = \begin{cases} n+1 & \text{se } n \text{ pari} \\ n & \text{se } n \text{ dispari} \end{cases} \geq n$$

since f is a proper complexity function, it does not decrease.

$$g\left(\left\lfloor \frac{n}{2} \right\rfloor\right) = f\left(2\left\lfloor \frac{n}{2} \right\rfloor + 1\right) \geq f(n)$$

and so we have that

$$\mathrm{TIME}\left(g\left(\left\lfloor \frac{n}{2} \right\rfloor\right)\right) \supseteq \mathrm{TIME}(f(n)).$$

For lemma (2), $H_g \notin \mathrm{TIME}(g(\lfloor n/2 \rfloor))$, and since $g\left(\lfloor \frac{n}{2} \rfloor\right) \geq f(n) \implies H_g \notin \mathrm{TIME}(f(n))$. Furthermore for lemma (1), $H_g \in \mathrm{TIME}(g(n)^3) = \mathrm{TIME}(f((2n+1)^3))$. $\square$

It has been proven that

$$TIME(f(n)) \subset TIME(f(n)\log^2 f(n))$$

Separation of P and EXP

P is a proper subset of **EXP**

Proof. Any polynomial will ultimately become smaller than 2^n, and so

$$\mathbf{P} \subseteq \mathbf{TIME}(2^n) \subseteq \mathbf{EXP}$$

To prove proper inclusion, notice that, by the Time Hierarchy Theorem,

$$\mathbf{TIME}(2^n) \subset \mathbf{TIME}((2^{2n+1})^3) = \mathbf{TIME}((2^{6n+3})) \subseteq \mathbf{TIME}(2^{n^2}) \subseteq \mathbf{EXP}$$

$\square$

The Space Hierarchy Theorem

If f is a proper complexity function, then

$$\mathbf{SPACE}(f(n)) \subset \mathbf{SPACE}(f(n)\log f(n))$$

8 Nondeterministic hierarchies

- $\mathbf{NTIME}(n^c) \subset \mathbf{NTIME}(n^d)$ for all the constants $1 \leq c < d$.
- for all increasing functions h (doesn't matter how slow):
  - $\mathbf{NSPACE}(\log n) \subset \mathbf{NSPACE}(\log(n) \cdot h(n))$
  - $\mathbf{NSPACE}(n^k) \subset \mathbf{NSPACE}(n^k \cdot h(n))$
  - $\mathbf{NSPACE}(2^n) \subset \mathbf{NSPACE}(2^n \cdot h(n))$
  - $\mathbf{NSPACE}(2^{2^n}) \subset \mathbf{NSPACE}(2^{2^{n+1}})$
    Notice how hierarchies are much more dense then the deterministic ones.

9 Time limit given space

$$\mathbf{TIME}(f(n)) \subseteq \mathbf{SPACE}(\frac{f(n)}{\log f(n)})$$

If to sort an array we need $O(n \log n)$, then we can do it in $O(n)$ space. If we use more space, it means that we are wasting space. If a problem cannot be solved in polynomial space then it cannot be solved in polynomial time.

10 Space limit given time

$$\mathbf{SPACE}(f(n)) \subseteq \mathbf{TIME}(k^{f(n)})$$

In particular, when $f(n)$ is a polynomial, this theorem implies

$$\mathbf{PSPACE} \subseteq \mathbf{EXP}$$

We don't know if the inclusion is strict, but many believe so. This is because **EXP** can use exponential space, tanks to the relation $\mathbf{TIME}(f(n)) \subseteq \mathbf{SPACE}(f(n))$

11 Relation between deterministic and nondeterministic complexity classes

$$\textbf{NTIME}(f(n)) \subseteq \textbf{SPACE}(f(n))$$

Proof. Every possibile run of any nondeterministic Turing machine can be represented with a sequence of integers in the range $1, \dots, d$ (where d is the degree of nondeterminism). If M runs in time $f(n)$, then every possibile run is at most $f(n)$ long and so is the space used. In the deterministic simulation we need to try every possibile run to see if at least one of them is accepted. We can use the same space for every possible run erasing the previous computation. The maximum space used is then the space used by the longest run plus the space used to enumerate the runs. The space used by the longest run is $O(f(n))$ and the space used to enumerate the runs is $O(f(n)\log d) = O(f(n))$. So the total space used is $O(f(n))$. □

$$\textbf{NSPACE}(f(n)) \subseteq \textbf{TIME}(k^{\log(n)+f(n)})$$

Proof. omitted □

Corollary

$$\textbf{L} \subseteq \textbf{NL} \subseteq \textbf{P} \subseteq \textbf{NP} \subseteq \textbf{PSPACE}$$

11.1 Savitch's Theorem

$$\textit{REACHABILITY} \in \textbf{SPACE}(log^2 n)$$

Proof. Let G be a graph with n nodes, let x and y be nodes of G, and let $i \geq 0$. We say that predicate PATH(x, y, i) holds if the following is true: There is a path from x to y in G, of length at most 2^i. Notice immediately that, since a path in G need be at most n long, we can solve reachability in G if we can compute whether PATH$(x, y, \lceil \log n \rceil)$ for any two given nodes of G.

We shall design a Turing machine, with two working strings besides its input string, which decides whether PATH(x, y, i). The adjacency matrix of G is given at the input string. We assume that the first work string contains nodes x and y and integer i, all in binary. The first work string will typically contain several triples, of which the (x, y, i) one will be the leftmost. The other work string will be used as scratch space—$\mathcal{O}(\log n)$ space will suffice there.

We now describe how our machine decides whether PATH(x, y, i). If $i = 0$, we can tell whether x and y are connected via a path of length at most $2^0 = 1$ by simply checking whether $x = y$, or whether x and y are adjacent by looking at the input. This takes care of the $i = 0$ case. If $i \geq 1$, then we compute PATH(x, y, i) by the following recursive algorithm:

for all nodes z test whether PATH$(x, z, i-1)$ and PATH$(z, y, i-1)$

This program implements a very simple idea: *Any path of length 2^i from x to y has a midpoint z, and both x and y are at most 2^{i-1} away from this midpoint.*

To implement this elegant idea in a space-efficient manner, we generate all nodes z, one after the other, reusing space. Once a new z is generated, we add a triple $(x, z, i-1)$ to the main work string and start working on this problem, recursively. If a negative answer to PATH$(x, z, i-1)$ is obtained, we erase this triple and move to the next z. If a positive answer is returned, we erase the triple $(x, z, i-1)$, write the triple $(z, y, i-1)$ on the work string (we consult (x, y, i), the next triple to the left, to obtain y), and work on deciding whether PATH$(z, y, i-1)$. If this is negative, we erase the triple and try the next z; if it is positive, we detect by comparing with triple (x, y, i) to the left that this is the second recursive call, and return a positive answer to PATH(x, y, i). Notice that the first working string of the machine acts like a *stack of activation records* to implement the recursion indicated above.

It is clear that this algorithm implements the recursive one displayed above, and thus correctly solves PATH(x, y, i). The first work string contains at any moment $\lceil \log n \rceil$ or fewer triples, each of length at most $3 \log n$. And in order to solve *REACHABILITY*, we start this algorithm with $(x, y, \lceil \log n \rceil)$ written on the main work string. The last string used as a counter to enumerate the z nodes uses at most $\log n$.
Total space used:

$$\lceil \log n \rceil \cdot 3 \log n + \log n = O(\log^2 n)$$

□

Corollary

For every proper complexity function $f(n) \geq \log n$

$$\mathbf{NSPACE}(f(n)) \subseteq \mathbf{SPACE}(f^2(n))$$

Proof. We can solve reachability on the graph configurations using the Savitch's algorithm and then we can generate the configurations graph to save space. The space required is:

$$\log^2(c^{f(n)}) = (\log(c^{f(n)}))^2 = O(f(n)^2) = O(f^2(n))$$

□

Corollary

Since we know that $\mathbf{SPACE}(f(n)) \subseteq \mathbf{NSPACE}(f(n))$ and $\mathbf{NSPACE}(f(n)) \subseteq \mathbf{SPACE}(f^2(n))$ when f is polynomial, we can conclude that:

$$\mathbf{PSPACE} = \mathbf{NSPACE}$$

That is, nondeterminism does not extend the class of problems that can be solved in polynomial space.

12 Reductions

Like all complexity classes, **NP** contains an infinity of languages. Of the problems and languages we have seen so far, **NP** contains TSP (D) and the SAT problem for Boolean expressions. In addition, **NP** certainly contains REACHABILITY and CIRCUIT VALUE, both of which are in **P** and thus in **NP**.

However, it is intuitively clear that the former two problems (TSP (D) and SAT) are somehow more representative of the complexity of **NP** than the latter two. They seem to capture more faithfully the power and complexity of **NP**, and are not known (or believed) to be in **P** like the other two.

To formalize the notion of one problem being "at least as hard as" another, we introduce the concept of *reduction*. We say that problem A is at least as hard as problem B if B reduces to A. That is, there is a transformation R which, for every input x of B, produces an equivalent input $R(x)$ of A, such that the answer to $R(x)$ for A is the same as the answer to x for B. In other words, to solve B on input x, we just have to compute $R(x)$ and solve A on it.

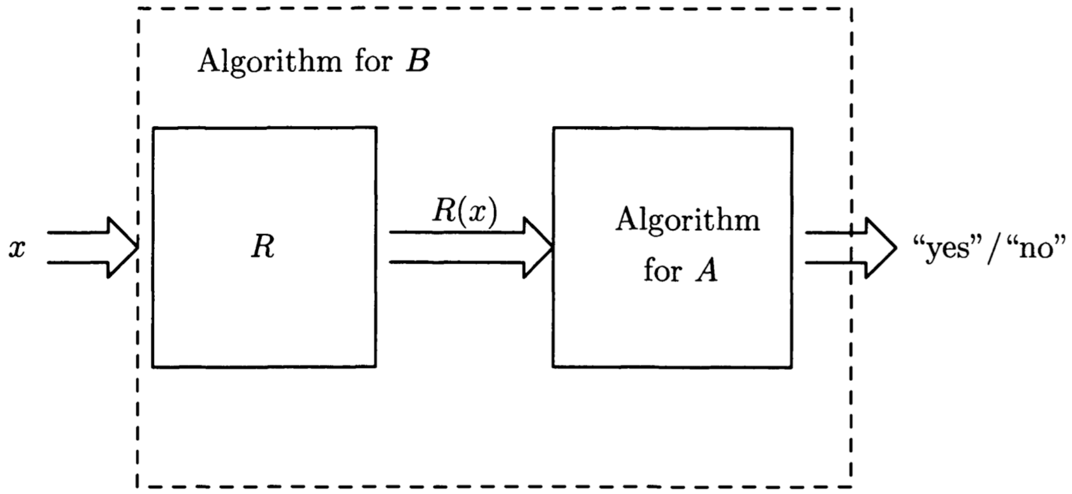

Figure 2: Reduction from B to A

If the scenario above is possible, it seems reasonable to say that A is at least as hard as

B. With one proviso: *That R should not be fantastically hard to compute.* If we do not limit the complexity of computing R, we could arrive at absurdities such as TSP(D) reduced to REACHABILITY, and thus REACHABILITY being harder than TSP(D)! Indeed, given any instance x of TSP(D) (that is, a distance matrix and a budget), we can apply the following reduction: Examine all tours; if one of them is cheaper than the budget, then $R(x)$ is the two-node graph consisting of a single edge from 1 to 2. Otherwise, it is the two-node graph with no edges. Notice that, indeed, $R(x)$ is a "yes" instance of REACHABILITY if and only if x was a "yes" instance of TSP(D). The flaw is, of course, that R is an exponential-time algorithm.

Reduction

As we pointed out above, for our concept of reduction to be meaningful, it should involve the weakest computation possible. *We shall adopt $\log n$ space-bounded reduction as our notion of "efficient reduction."* That is, we say that language L_1 *is reducible to* L_2 if there is a function R from strings to strings computable by a deterministic Turing machine in space $\mathcal{O}(\log n)$ such that for all inputs x the following is true: $x \in L_1$ if and only if $R(x) \in L_2$. R is called a *reduction* from L_1 to L_2. Since our focal problems in complexity involve the comparisons of time classes, it is important to note that reductions are *polynomial-time algorithms.* This is because $\mathbf{SPACE}(\log n) \subseteq \mathbf{TIME}(k^{\log n}) \subseteq \mathbf{TIME}(n^k)$.

Reduction from Hamilton path to SAT

Suppose that we are given a graph G. We shall construct a Boolean expression $R(G)$ such that $R(G)$ is satisfiable if and only if G has a Hamilton path.

Suppose that G has n nodes, $1, 2, \ldots, n$. Then $R(G)$ will have n^2 Boolean variables, $x_{ij} : 1 \leq i, j \leq n$. Informally, variable x_{ij} will represent the fact "node j is the ith node in the Hamilton path," which of course may be either true or false. $R(G)$ will be in conjunctive normal form, so we shall describe its clauses.

The clauses will spell out all requirements on the x_{ij}'s that are sufficient to guarantee that they encode a true Hamilton path. To start, node j must appear in the path; this is captured by the clause $(x_{1j} \vee x_{2j} \vee \ldots \vee x_{nj})$; we have such a clause for each j. But node j cannot appear both at ith and kth; this is expressed by clause $(\neg x_{ij} \vee \neg x_{kj})$, repeated for all values of j, and $i \neq k$. Conversely, some node must be ith, thus we add the clause $(x_{i1} \vee x_{i2} \vee \ldots \vee x_{in})$ for each i; and no two nodes should be ith, or $(\neg x_{ij} \vee \neg x_{ik})$ for all i, and all $j \neq k$.

Finally, for each pair (i, j) which is not an edge of G, it must not be the case that j comes right after i in the Hamilton path; therefore the following clauses are added for each pair (i, j) not in G and for $k = 1, \ldots, n-1$: $(\neg x_{ki} \vee \neg x_{k+1,j})$. This completes the construction. Expression $R(G)$ is the conjunction of all these clauses.

We claim that R is a reduction from Hamilton Path to SAT. To prove our claim, we have to establish two things: That for any graph G, expression $R(G)$ has a satisfying truth assignment if and only if G has a Hamilton path; and that R can be computed in space $\log n$.

Suppose that $R(G)$ has a satisfying truth assignment T. Since T satisfies all clauses of $R(G)$, it must be the case that, for each j there exists a unique i such that $T(x_{ij}) = \mathbf{true}$, otherwise the clauses of the form $(x_{1j} \vee x_{2j} \vee \ldots \vee x_{nj})$ and $(\neg x_{ij} \vee \neg x_{kj})$ cannot all be satisfied. Similarly, clauses $(x_{i1} \vee x_{i2} \vee \ldots \vee x_{in})$ and $(\neg x_{ij} \vee \neg x_{ik})$ guarantee that for each i there exists a unique j such that $T(x_{ij}) = \mathbf{true}$. Hence, T really represents a permutation $\pi(1), \ldots, \pi(n)$ of the nodes of G, where $\pi(i) = j$ if and only if $T(x_{ij}) = \mathbf{true}$. However, clauses $(\neg x_{k,i} \vee \neg x_{k+1,j})$ where (i, j) is not an edge of G and $k = 1, \ldots, n-1$ guarantee that, for all k, $(\pi(k), \pi(k+1))$ is an edge of G. This means that $(\pi(1), \pi(2), \ldots, \pi(n))$ is a Hamilton path of G. Conversely, suppose that G has a Hamilton path $(\pi(1), \pi(2), \ldots, \pi(n))$, where π is a permutation. Then it is clear that the truth assignment $T(x_{ij}) = \mathbf{true}$ if $\pi(i) = j$, and $T(x_{ij}) = \mathbf{false}$ if $\pi(i) \neq j$, satisfies all clauses of $R(G)$.

We still have to show that R can be computed in space $\log n$. Given G as an input, a Turing machine M outputs $R(G)$ as follows: First it writes n, the number of nodes of G, in binary, and, based on n it generates in its output tape, one by one, the clauses that do not depend on the graph (the first four groups in the description of $R(G)$). To this end, M just needs three counters, i, j, and k, to help construct the indices of the variables in the clauses. For the last group, the one that depends on G, M again generates one by one in its work string all clauses of the form $(\neg x_{ki} \vee \neg x_{k+1,j})$ for $k = 1, \ldots, n-1$; after such a clause is generated, M looks at its input to see whether (i, j) is an edge of G, and if it is not, then it outputs the clause. This completes our proof that Hamilton Path can be reduced to SAT.

Reduction Composition

If R is a reduction from language L_1 to L_2 and R' is a reduction from language L_2 to L_3, then the composition $R' \circ R$ is a reduction from L_1 to L_3.

Proof. That $x \in L_1$ if and only if $R'(R(x)) \in L_3$ is immediate from the fact that R and R' are reductions. The nontrivial part is to show that $R' \circ R$ can be computed in space $\log n$.

One first idea is to compose the two machines with input and output, M_R and $M_{R'}$, that compute R and R' respectively, so that $R(x)$ is first produced, and from it the final output $R'(R(x))$. Alas, the composite machine M must have $R(x)$ written on a work string; and $R(x)$ may be much longer than $\log |x|$.

The solution to this problem is clever and simple: We do not explicitly store the intermediate result in a string of M. Instead, we simulate $M_{R'}$ on input $R(x)$ by remembering at all times the cursor position i of the input string of $M_{R'}$ (which is the output string of M_R). i is stored in binary in a new string of M. Initially $i = 1$, and we have a separate set of strings on which we are about to begin the simulation of M_R on input x.

Since we know that the input cursor in the beginning scans a $\triangleright$, it is easy to simulate the first move of $M_{R'}$. Whenever the cursor of $M_{R'}$'s input string moves to the right, we increment i by one, and continue the computation of machine M_R on input x (on the separate set of strings) long enough for it to produce the next output symbol; this is the symbol currently scanned by the input cursor of $M_{R'}$, and so the simulation can go on. If the cursor stays at the same position, we just remember the input symbol scanned. If, however, the input cursor of $M_{R'}$ moves to the left, there is no obvious way to continue the simulation, since the previous symbol output by M_R has been long forgotten. We must do something more radical: We decrement i by one, and then run M_R on x *from the beginning*, counting on a separate string the symbols output, and stopping when the ith symbol is output. Once we know this symbol, the simulation of $M_{R'}$ can be resumed.

It is clear that this machine indeed computes $R' \circ R$ in space $\log n$ (recall that $|R(x)|$ is at most polynomial in $n = |x|$, and so i has $\mathcal{O}(\log n)$ bits). $\square$

Completeness

Let C be a complexity class, and let L be a language in C. We say that L is C-*complete* if any language $L' \in C$ can be reduced to L.

It is not certain that every class C contains a C-*complete* language, but well will see that it's true for the main complexity classes.

Hardness

To be precise, given a complexity class C, we say that a language L is C-*hard* if every language $L' \in C$ can be reduced to L. We say that L is C-*complete* if it is both C-hard and in C.

Closure under reductions

We say that a class C is *closed under reductions* if it contains every language L' such that $L \in C$ and L' can be reduced to L. That is, if $L \in C$ then C contains every other problem more simple or as hard as L.

$\mathbf{P}, \mathbf{NP}, \mathbf{coNP}, \mathbf{L}, \mathbf{NL}, \mathbf{PSPACE}$ and $\mathbf{EXP}$ are all close under reductions.

Proposition

If two classes C_1 and C_2 are both closed under reductions, and there is a language L which is complete for both classes, then $C_1 = C_2$.

Proof. Since L is complete for C_1, all languages in C_1 can be reduced to $L \in C_2$. Since C_2 is closed under reductions, it follows that $C_1 \subseteq C_2$. The other inclusion follows by symmetry. $\square$

Colrollary

If $C_1 \subseteq C_2$, and C_1 is closed under reductions and contains a C_2-*complete* language L, then $C_1 = C_2$.

Proof. For the sake of contradiction, suppose that $L' \in C_2 \setminus C_1$. For the hypothesis of C_2-*completeness*, L' can be reduces to $L \in C_1$. Then for the hypothesis of closure of C_1, $L' \in C_1$ which is a contradiction. □

12.1 Cook's Theorem

Cook's Theorem

SAT is **NP**-complete.

Proof. We know that given a satisfiable boolean expression exists a nondeterministic turing machine that is able to find and verify a satisfying assignment in polynomial time. So we have that SAT is in **NP**. The computation can be represented as a square table with $p(n)$ rows and columns called computation table. This matrix cab be represented as propositional formula in CNF. We will have a symbol for each cell of the table. Let $\{q_1, \ldots, q_r\}$ the states of M and $\{\sigma_1, \ldots, \sigma_l\}$ the symbols of the alphabet. For every step t (row) end every cell s (column), that is (t, s) is a coordinate of a cell in the table, we will have:

- $P^i_{s,t}$ (true if cell (s,t) contains symbol σ_i) $1 \leq i \leq l$
- Q^i_t (true if the state q at time t is q_i) $1 \leq i \leq r$
- $S_{s,t}$ (true if the cursor at time t is on cell s) $1 \leq s \leq T$ where $T = p(n)$.

We need to define the following CNF clauses: At every step the cursor is on exactly one cell:

$$\forall t \in \{1, \ldots, T\}$$

$$\left(\bigvee_{s \in \{1,\ldots,T\}} S_{s,t} \right) \wedge \left(\bigwedge_{1 \leq s < i \leq T} \neg S_{s,t} \vee \neg S_{i,t} \right)$$

At every step in every cell there is exactly one symbol:

$$\forall s, t \in \{1, \ldots, T\}$$

$$\left(\bigvee_{i \in \{1,\ldots,l\}} P^i_{s,t} \right) \wedge \left(\bigwedge_{1 \leq i < j \leq l} \neg P^i_{s,t} \vee \neg P^j_{s,t} \right)$$

At every step there is exactly one current state:

$$\forall t \in \{1, \ldots, T\}$$

$$\left(\bigvee_{i\in\{1,\ldots,r\}} Q_t^i\right) \wedge \left(\bigwedge_{1\le i<j\le r} \neg Q_t^i \vee \neg Q_t^j\right)$$

The initial condition are satisfied if the initial state is q_1, the blank σ_2 ed the initial string $\sigma_{i_1},\ldots,\sigma_{i_n}$ (followed by blanks):

$$Q_1^1 \wedge S_{1,1} \wedge P_{1,1}^{i_1} \wedge P_{2,1}^{i_2} \wedge \cdots \wedge P_{n,1}^{i_n} \wedge P_{n+1,1}^2 \wedge \cdots \wedge P_{T,1}^2$$

Q_1^1 (the initial state is q_1), $S_{1,1}$ (the cursor is on cell 1), $P_{1,1}^{i_1}$ (the first symbol of the input string is σ_{i_1}), $P_{2,1}^{i_2}$ (the second symbol of the input string is σ_{i_2}), $\ldots$, $P_{n,1}^{i_n}$ (the last symbol of the input string is σ_{i_n}), $P_{n+1,1}^2$ (all other cells are blank), $\ldots$, $P_{T,1}^2$ (all other cells are blank). We want that only the symbols under the cursor can change so:

$$\forall s,t \in \{1,\ldots,T\}\ \forall i \in \{1,\ldots,I\}$$

$$\neg S_{s,t} \wedge P_{s,t}^i \implies P_{s,t+1}^i$$

written in CNF:

$$S_{s,t} \vee \neg P_{s,t}^i \vee P_{s,t+1}^i$$

The computation accepts the input if the final state $q_k =$ "yes"

$$Q_T^k$$

The configurations change as specified by the transition function δ of M.
For all

$$(q_i, \sigma_j, q_k, \sigma_m, \leftarrow) \notin \Delta$$

we have that $\forall t = 1,\ldots,T-1$ and $\forall s = 2,\ldots,T$

$$Q_t^i \wedge S_{s,t} \wedge P_{s,t}^j \implies \neg(Q_{t+1}^k \wedge P_{s,t+1}^m \wedge S_{s-1,t+1})$$

equivalent to the CNF:

$$\neg Q_t^i \wedge \neg S_{s,t} \wedge \neg P_{s,t}^j \wedge \neg Q_{t+1}^k \wedge \neg P_{s,t+1}^m \wedge \neg S_{s-1,t+1}$$

For exercise write the remaining clauses corresponding to the moves $\rightarrow$ e $-$.
The reduction $R(x)$ is the *and* of all the clauses above. The problem is in **NP**: Given a satisfiable Boolean expression, a nondeterministic machine can "guess" the satisfying truth assignment and verify it in polynomial time. Since we know that CIRCUIT SAT reduces to SAT, we need to show that all languages in **NP** can be reduced to CIRCUIT SAT.

Let $L \in$ **NP**. We shall describe a reduction R which for each string x constructs a circuit $R(x)$ (with inputs that can be either variables or constants) such that $x \in L$ if and only if $R(x)$ is satisfiable. Since $L \in$ **NP**, there is a nondeterministic Turing machine $M = (K, \Sigma, \Delta, s)$ that decides L in time n^k. That is, given a string x, there is an accepting computation (sequence of nondeterministic choices) of M on input x if and only if $x \in L$. We assume that M has a single string; furthermore, we can assume that it has at each step *two nondeterministic choices*. If for some state-symbol combinations there are $m > 2$ choices in Δ, we modify M by adding $m-2$ new states so that the same effect is achieved (see Figure below). If for some combination there is only one choice, we consider that the two choices coincide; and finally if for some state-symbol combination there is no choice in Δ, we add to Δ the choice that changes no state, symbol, or position. So, machine M has exactly two choices for each symbol-state combination. One of these choices is called choice 0 and the other choice 1, so that a sequence of nondeterministic choices is simply a bitstring $(c_1, c_2, \ldots, c_{|x|^k-1}) \in \{0,1\}^{|x|^k-1}$.

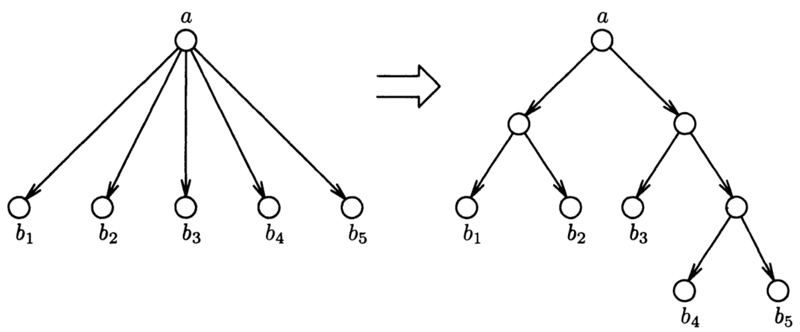

Figure 3: Reducing degree of nondeterminism

Since the computation of nondeterministic Turing machines proceeds in parallel paths, there is no simple notion of computation table that captures all of the behavior of such a machine on an input. *If, however, we fix a sequence of choices* $\mathbf{c} = (c_0, c_2, \ldots, c_{|x|^k-1})$, then the computation is effectively deterministic (at the ith step we take choice c_i), and thus we can define the computation table $T(M, x, \mathbf{c})$ corresponding to the machine, input, and sequence of choices. Again the top row and the extreme columns of the table will be predetermined. All other entries T_{ij} will depend only on the entries $T_{i-1,j-1}$, $T_{i-1,j}$, and $T_{i-1,j+1}$ and the choice c_{i-1} at the previous step (see Figure).

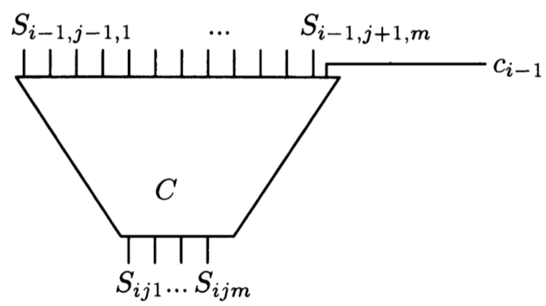

Figure 4: The construction for Cook's theorem

That is, this time the fixed circuit C has $3m+1$ entries instead of $3m$, the extra entry corresponding to the nondeterministic choice.

Thus we can again construct in $\mathcal{O}(\log |x|)$ space a circuit $R(x)$, this time with variable gates $c_0, c_1, \ldots, c_{|x|^k-1}$ corresponding to the nondeterministic choices of the machine. It follows immediately that $R(x)$ is satisfiable (that is, there is a sequence of choices $c_0, c_1, \ldots, c_{|x|^k-1}$ such that the computation table is accepting) if and only if $x \in L$.

$\square$