Introduction to Complexity Theory*
Lecture 3: More on NP and some on DTIME

Notes taken by Michael Elkin and Ekaterina Sedletsky

January 8, 1999

Summary: In this lecture we discuss some properties of the complexity classes P, NP and NPC (theorems of Ladner and Levin). We also define new complexity classes (DTime_i), and consider some relations between them.

1 Non-complete languages in NP

In this lecture we consider several items, that describe more closely the picture of the complexity world. We already know that $P \subseteq NP$ and we conjecture that it's a strict inequality, although we can't prove it. Another important class that we have considered is $NP-complete$ (NPC) problems, and as we have already seen, if there is a gap between P and NP then the class of NPC problems is contained in this gap ($NPC \subseteq NP \backslash P$).

The following theorem of Ladner tells us additional information about this gap $NP \backslash P$, namely that NPC is strictly contained in $NP \backslash P$.

Formally,

Theorem 1 *If $P \neq NP$ then there exists a language $L \in (NP \backslash P) \backslash NPC$.*

That is, NP (or say SAT) is not (Karp) reducible to L. Actually, one can show that SAT is not even Cook-reducible to L.

Oded's Note: *Following is a proof sketch.*

We start with any $B \in NP \backslash P$, and modify it to $B' = B \cap S$, where $S \in P$, so that B' is neither NP-complete nor in P. The fact that S is in P implies that B' is in NP. The "seiving" set S will be constructed to to foil both all possible polynomial-time algorithms for B' and all possible reductions. At the extreme, setting $S = \{0,1\}^*$, foil all algorithms since in this case $B' = B \notin P$. On the other hand, setting $S = \emptyset$, foils all possible reductions since in this case $B' = \emptyset$ and so (under $P \neq NP$) cannot be NP-complete (as reducing to it gives nothing). Note that the above argument extends to the case S (resp., $\overline{S}$) is a finite set.

The "seiving" set S is constructed iteratively so that in odd iterations is fails machines from a standard enumeration of polynomial-time machines (so that in iteration $2i-1$ we fail the i^{th} machine). (Here we don't need to emulate these machines in polynomial-time

* Lecture Notes for a course given by Oded Goldreich at the Weizmann Institute of Science, Israel.

in length of their inputs.) In even iterations we fail oracle-machines from a standard enumeration of such machines (which correspond to Cook-reductions of B to B') so that in iteration $2i$ we fail the $i^{\rm th}$ oracle-machine.

The iteration number is determined by a deciding algorithm for S which is operates as follows. For simplicity, the algorithm puts z in S iff it puts $1^{|z|}$ in S. The decision whether to put 1^n in S is taken as follows. Starting with the first iteration, and using a time-out mechanism with a fixed polynomial bound $b(n)$ (e.g., $b(n) = n^2$ or $b(n) = \log n$ will do), we try to find a input $z \in \{0,1\}^*$ so that the first polynomial-time algorithm, A_1, fails on z (i.e., $A_1(z) \neq \chi_B(z)$). In order to decide $\chi_B(z)$ we run the obvious exponential-time algorithm, but z is expected to be much shorter than n (or else we halt by time-out before). We scan all possible z's in lexicographic order until reaching time-out. Once we reach time-out while not finding such bad z, we let $1^n \in S$. Eventually, for a sufficiently large n we will find a bad z within the time-out. In such a case we let $1^n \notin S$ and continue to the second iteration, where we consider the first polynomial-time oracle-machine, M_1. Now we try to find an input z for B on which M_1 with oracle S. Note that we know the value $n' < n$ so that $1^{n'}$ was the first string not put in S. So currently, S is thought of as containing only strings of length smaller than n'. We emulate M_1 while answering its queries to $B' = B \cap S$ accordingly, using the exponential-time algorithm for deciding B (and our knowledge of the portion of S determined so far). We also use the exponential-time algorithm for B to check if the recuction of z to B' is correct for each z. Once we reach time-out while not finding such bad z, we let $1^n \notin S$. Again, eventually, for a sufficiently large n we will find a bad z within the time-out. In such a case we let $1^n \in S$ and continue to the third iteration, where we consider the second polynomial-time algorithm, and so on.

Some implementation details are provided below. Specifically, the algorithm T below computes the number of iterations completed wrt input $x \in \{0,1\}^n$.

Proof: Let $B \in NP \backslash P$. Let $A_0, A_1, ...$ be the enumeration of all polynomial time bounded Turing machines, that solve decision problems and $M_0, M_1, ...$ be the enumeration of polynomial time bounded oracle Turing machines. Let $L(A_i)$ denote the set recognized by A_i and for every set S let $L(M_i^S)$ to be the set recognized by machine M_i with oracle S.

We construct a polynomial time bounded Turing machine T with range $\{1\}^*$ in such a way that, for $B' = \{x \in B : |T(x)| \text{ is even}\}$, both $B' \notin P$ and $B \not\propto_C B'$ (i.e., B is not Cook reducible to B'). It follows that $B' \notin P \cup NPC$.

We show that for any such T, $B' \propto_K B$ (B' is Karp reducible to B) and $B' \in NP$ follows, since diciding B' can be done by deciding B and $B \in NP$.The Karp-reduction (of B' to B), denoted f, is defined as follows. Let $x_0 \notin B$. (We assume that $B \neq \Sigma^*$, because otherwise $B \in P$). The function f will compute $|T(x)|$ and if $|T(x)|$ is even it will return x and otherwise it will return x_0. Now if $x \in B'$, then $x \in B$ and $|T(x)|$ is even, hence $f(x) = x \in B$. Otherwise $x \notin B'$ and then there are two possibilities.

1. If $x \notin B$, then for even $|T(x)|$ holds $f(x) = x$, and for odd $|T(x)|$ holds $f(x) = x_0$, and so for any x holds $f(x) \notin B$.
2. If $x \in B$ and $|T(x)|$ is odd, then $f(x) = x_0 \notin B$.

(Recall that $x \notin B'$ rules out "$x \in B$ and even $|T(x)|$")

To complete the construction we have to build a Turing machine T such that

(1) $B' \stackrel{\text{def}}{=} \{x \in B : |T(x)| \text{ is even}\} \neq L(A_i)$, for any $i = 0, 1, 2, ...$ (and so $B' \notin P$)

(2) $L(M_i^{B'}) \neq B$, for any $i = 0, 1, 2, ...$ (and so $B \not\propto_C B'$).

A machine T that satisfies these conditions may be constructed in the following way:

On input λ (empty string), T prints λ. On an input x of length n where $x \neq 0^n$ (unary), T prints $T(0^n)$. It remains to say what T does on inputs of the form 0^n where $n \geq 1$.

On input 0^n where $n \geq 1$, T does the following:

1. For n moves, try to reconstruct the sequence $T(\lambda), T(0), T(0^2), ...$ Let $T(0^m)$ be the last number of this sequence that is computed.

2. We consider two cases depending on the parity of $|T(0^m)|$. We associate B with an exponential-time algorithm decoding B (by scanning all possible NP-witnesses).

   Case(i): $|T(0^m)|$ is even. Let $i = |T(0^m)|/2$. For n moves, try to find a z such that $B'(z) \neq A_i(z)$. This is done by successively simulating B and T (to see what B' is) and A_i, on inputs $\lambda, 0, 1, 00, 01, ...$ If no such z is found, print 1^{2*i}; otherwise, print 1^{2*i+1}.

   Case(ii): $|T(0^m)|$ is odd. Let $i = (|T(0^m)| - 1)/2$. For n moves, try to find a z such that $B(z) \neq M_i^{B'}(z)$. This is done by simulating an algorithm B and the procedure M_i successively on inputs $\lambda, 0, 1, 00, 01, ...$ In simulating M_i on some input, whenever M_i makes a query of length at most m, we answer it according to B' determined by B and the values of T computed in Step (1). In case the query has lengths exceeding m, we behave as if we have already completed n steps. (The moves in this side calculation are counted among the n steps allowed.) If no such z is found, print 1^{2*i+1}; otherwise, print 1^{2*i+2}.

Such a machine can be programmed in a Turing machine that runs in polynomial time. For this specific machine T we obtain that $B' = \{x \in B : |T(x)| \text{ is even}\}$, satisfies: $B' \in NP \backslash P$ and $B \not\propto_C B' \Longrightarrow B' \notin NPC \Longrightarrow B' \in (NP \backslash P) \backslash NPC$. ■

The set B' constructed in the above proof is certainly not a natural one (even in case B is). We mention that there are some natural problems conjectured to be in $(NP \backslash P) \backslash NPC$: For example, Graph Isomorphism (the problem of whether two graphs are isomorphic).

2 Optimal algorithms for NP

The following theorem, due to Levin, states an interesting property of the class NP. Informally, Levin's Theorem tells us that exists optimal algorithm for any NP search problem.

Theorem 2 *For any NP-relation R there exist a polynomial $P_R(\cdot)$ and an algorithm $A_R(\cdot)$ which finds solutions whenever they exist, so that for every algorithm A which finds solutions and for any input x*

$$time_{A_R}(x) \leq O(time_A(x) + P_R(|x|)), \tag{1}$$

where $time_A(x)$ is the running time of the algorithm A on input x.

This means that for every algorithm A exists a constant c such that for any sufficiently long x

$$time_{A_R}(x) \leq c * (time_A(x) + P_R(|x|)).$$

This c is not a universal constant, but an algorithm-specific one (i.e., depends on A). The algorithm A_R is optimal in the following sense: for any other algorithm A there exists a constant c such that for any sufficiently long x

$$time_A(x) \geq \frac{1}{c} * time_{A_R}(x) - Q_R(|x|),$$

where $Q_R(|x|) = \frac{1}{c} * P_R(|x|)$.

The algorithms we are talking about are not TM's. For proving the theorem, we should define exactly the computational model we are working with. Either it will be a one-tape machine or two-tape one and etc. Depending on the exact model the constant c may be replaced by some low-n function like $logn$. A constant may be achieved only in more powerful/flexible models of computation that are not discussed here.

We observe also that although the proof of Levin's Theorem is a constructive one, it's completely impractical, since as we'll see it incorporates a huge constant in its running time. On the other hand, it illustrates an important asymptotic property of the NP class.

Proof: The basic idea of the proof is to have an enumeration of all possible algorithms. This set is countable, since the set of all possible deterministic TM's is countable. Using this enumeration we would like to run all the algorithms in parallel. It's, of course, impossible since we can't run a countable set of TM's in parallel, but the solution to the problem is to run them in different rates.

There are several possible implementations of this idea. One of the possibilities is as following. Let us divide the execution of the simulating machine into rounds. The simulating machine runs machine i at round r if and only if $r \equiv 0 \pmod{i^2}$. That is, we let i'th machine to make t steps during $i^2 * t$ rounds of the simulating machine. Also the number of steps made in these $r = i^2 * t$ rounds is $\sum_{j \geq 1} |\frac{r}{j^2}| < r$.

Such a construction would fail if some of these machines would provide wrong answers. To solve this difficulty we "force" the machines to verify the validity of their outputs. So, without loss of generality, we assume that each machine is augmented by a verifying component that checks the validity of machine's output. Since the problem is in NP, verifying the output takes polynomial amount of time. When we estimate the running time of the algorithm A_R, we take into account this polynomial amount of time and it is the reason that P_R arises in Eq. (1). So, without loss of generality, the outputs of the algorithms are correct.

Another difficulty, is that some of these machines could not have sufficient amount of time to halt. On the other hand, since each of the machines solves the problem, it is sufficient for us that one of them will halt.

Levin's optimal algorithm A_R is this construction running interchangeably all these machines.

We'll claim the following property of the construction.

Claim 2.1 *Consider A that solves the problem. Let i be the position of A in the enumeration of all the machines. Let $time_A(\cdot)$ be the running time of A. It is run in A_R in $\frac{1}{f(i)}$ rate. Then A_R runs at most $f(i) * time_A(\cdot)$.*

We took $f(i) = (i+1)^2$, but that is not really important: we observe that i is a constant (and thus, so is $f(i)$). Of course, it's a huge constant, because each machine needs millions of bits to be encoded (even the simplest deterministic TM that does nothing needs several hundreds of bits to be encoded) and the index of machine M in the enumeration (i.e. i) is $i \approx 2^{|M|}$, where $|M|$ is the number of bits needed to encode machine M and then $f(i)$ will be $(i+1)^2 = (2^{|M|}+1)^2$. (This constant makes the algorithm completely impractical.) ■

3 General Time complexity classes

The class P was introduced to capture our notion of efficient computation. This class has some "robustness" properties which make it convenient for investigation.

1. P is not model-dependent: Remain the same if we consider one tape TM or two tape TM. This remains valid for any "reasonable" and "general" enough model of computation.
2. P is robust under "Reasonable" changes to an algorithm: Closed classes under "reasonable" changes of the algorithm, like flipping the output and things like that. This holds for P, but probably not for NP and NPC. The same applies also for (3) and (4).
3. Closed under serial composition: Concatenation of two (efficient) algorithms from a class will produce another (efficient) algorithm from the same class.
4. Closed under subroutine operation: Using one algorithm from a class as a subroutine of another algorithm from the same class provides an algorithm from the class (the class is set of problems and when we are talking about an algorithm from the class we mean an algorithm for solving a problem from the class and the existence of this algorithm is evidence that the problem indeed belongs to this class).

None of these nice properties holds for classes that we will now define.

3.1 The DTime classes

> Oded's Note: *DTime denotes the class of languages which are decideable within a specific time-bound. Since this class specifies one time-bound rather than a family of such bounds (like polinomial-time), we need to be soecific with respect to the model of computation.*

Definition 3 *$DTime_i(t(\cdot))$ is the class of languages decidable by a deterministic i-tape Turing Machine within $t(\cdot)$ steps. That is, $L \in DTime_i(t(\cdot))$ if there exists a deterministic i-tape Turing Machine M which accepts L so that for any $x \in \{0,1\}^*$, on input x machine M makes at most $t(|x|)$ steps.*

Usually, we consider $i = 1$ or $i = 2$ talking about one- and two-tape TM's respectively. When we'll consider space complexity we'll find it very natural to deal with 3-tape TM. If there is no index in DTime, then the default is $i = 1$.

Using this new notation we present the following theorem:

Theorem 4 *For every function $t(\cdot)$ that is at least linear*

$$DTime_2(t(\cdot)) \subseteq DTime_1(t(\cdot)^2)$$

The theorem is important in the sense that it enables us sometimes to skip the index, since with respect to polynomial-time computations both models (one-tape and two-tape) coincide. The proof is by simulating the two-tape TM on a one-tape one.

Proof: Consider a language $L \in DTime_2(t(\cdot))$. Therefore, there exists a two-tape TM M_1 which accepts L in $O(t(\cdot))$. We can imagine that the tape of a one-tape TM as divided into 4 tracks. We can construct M_2, a one-tape TM with 4 tracks, 2 tracks for each of M_1's tapes. One track records the contents of the corresponding tape of M_1 and the other is blank, except for a marker in the cell that holds the symbol scanned by the corresponding head of M_1. The finite control of M_2 stores the state of M_1, along with a count of the number of head markers to the right of M_2's tape head.

Each move of M_1 is simulated by a sweep from left to right and then from right to left by the tape head of M_2, which takes $O(t(\cdot))$ time. Initially, M_2's head is at the leftmost cell containing a head marker. To simulate a move of M_1, M_2 sweeps right, visiting each of the cells with head markers and recording the symbol scanned by both heads of M_1. When M_2 crosses a head marker, it must update the count of head markers to the right. When no more head markers are to the right, M_2 has seen the symbols scanned by both of M_1's heads, so M_2 has enough information to determine the move of M_1. Now M_2 makes a pass left, until it reaches the leftmost head marker. The count of markers to the right enables M_2 to tell when it has gone far enough. As M_2 passes each head marker on the leftward pass, it updates the tape symbol of M_1 "scanned" by that head marker, and it moves the head marker one symbol left or right to simulate the move of M_1. Finally, M_2 changes the state of M_1 recorded in M_2's control to complete the simulation of one move of M_1. If the new state of M_1 is accepting, then M_2 accepts.

Finding the mark costs $O(t(\cdot))$, and as there are not more than $t(\cdot)$ moves, totally the execution costs at most $O(t^2(\cdot))$. ■

The next theorem is less important and brought from elegancy considerations, and says that in general one cannot do a better simulation than one in Theorem 3.
We recall the definition of "O", "Ω" and "o" notations:

- $f(n) = O(g(n))$ means that exists c such that for any sufficiently large n $f(n) \leq c * g(n)$.
- $f(n) = \Omega(g(n))$ means that exists $c > 0$ such that for any sufficiently large n $f(n) \geq c * g(n)$.
- $f(n) = o(g(n))$ means that for any c exists N such that for all $n > N$ it holds $f(n) < c * g(n)$.

Theorem 5 *$DTime_2(O(n))$ is not contained in $DTime_1(o(n^2))$.*

We note that it's much harder to prove that some things are "impossible" or "can not be done", than the opposite, because for the latter, constructive proofs can be used.

There are several possible ways to prove the theorem. The following one uses the notion of communication complexity.

Proof: Define language $L = \{xx : x \in \{0,1\}^*\}$. This language L is clearly in $DTime_2(O(n))$. We will show that $L \notin DTime_1(o(n^2))$ by "reduction" to a communication complexity problem.

Introduce, for the sake of proof, computational complexity model: two parties A and B have two strings, A has $\alpha \in \{0,1\}^n$ and B has $\beta \in \{0,1\}^n$, respectively. Their goal is to calculate $f(\alpha, \beta)$, where f is a function from $\{0,1\}^n * \{0,1\}^n$ to $\{0,1\}$. At the end of computation both parties should know $f(\alpha, \beta)$.

Let us also introduce a notation $R_0(f)$ to be the minimum expected cost of a randomized protocol, that computes f with zero error.

In our case it is sufficient to consider Equality (EQ) function defined by

$$EQ(\alpha, \beta) \stackrel{\text{def}}{=} 1 \text{ if } \alpha = \beta \text{ and } 0 \text{ otherwise.}$$

We state without proof a lower bound on the randomized zero-error communication complexity of EQ.

$$R_0(EQ) = \Omega(n), \tag{2}$$

The lower bound can be proven by invoking the lower bound on the "nondeterministic communication complexity" of EQ, and from the observation that nondeterminism is generally stronger than (zero-error!) randomization: Intuitively, if a randomized algorithm reaches the solution with some non-zero probability, then there is a sequence of values of flipped coins that causes the randomized algorithm to reach the solution. The nondeterministic version of the same algorithm could just guess correctly all these coins' values and to reach the solution as well.

We now get back to the proof. Suppose for contradiction, that there exists a one-tape Turing machine M which decides L in $o(n^2)$ time. Then we will build a zero-error randomized protocol Π that solves EQ in expected complexity $o(n)$, contradicting Eq. (2).

This protocol Π, on input α and β each of length n simulates the Turing machine on input $\alpha 0^n \beta 0^n$ in the following way. They output 1 or 0 depending on whether the machine accepts or rejects this input. They first choose together, uniformly at random, a location at the first 0-region of the tape. The party A simulates the machine whenever the head is to the left of this location, and the party B whenever the head is to the right of this location. Each time the head crosses this location only the state of the finite control ($O(1)$ bits) need to be sent. If the total running time of the machine is $o(n^2)$, then the expected number of times it crosses this location (which has been chosen at random among n different possibilities) is at most $o(n)$, contradicting Eq. (2).

Therefore, we have proved that a one-tape Turing machine which decides L runs $\Omega(n^2)$ time. ∎

An alternative way of proving the theorem, a direct proof, based on the notion of a crossing sequence, is given in the Appendix.

3.2 Time-constructibility and two theorems

Definition 6 *A function $f : N \longrightarrow N$ is time-constructible if there exists an algorithm A st. on input 1^n, A runs at most $f(n)$ steps and outputs $f(n)$ (in binary).*

One motivation to the definition of time-constructible function is the following: if the machine's running time is this specific function of input length, then we can calculate the running time within the time required to perform the whole calculation. This notion is important for simulating results, when we want to "efficiently" run all machines which have time bound $t(\cdot)$. We cannot enumerate these machines. Instead we enumerate all machines and run each with time bound $t(\cdot)$. Thus, we need to be able to compute $t(\cdot)$ within time $t(\cdot)$. Otherwise, just computing $t(\cdot)$ will take too much time.

For example, n^2, 2^n, n^n are all time-constructible functions.

Time Hierarchy: A special case of the Time Hierarchy Theorem asserts that

for every constant $c \in N$, for any $i \in \{1, 2\}$

$$DTime_i(n^c) \subset DTime_i(n^{c+1})$$

(where $A \subset B$ denotes that A is strictly contained in B)

That is, in this case there is no "complexity gaps" and the set of problems that can be solves grows whewn allowing more time: There are computational tasks that can be done in $O(n^{c+1})$, but can not be done in $O(n^c)$. The function n^{c+1} (above) can be replaced even by $n^{c+\frac{1}{2}}$ etc. The general case of the Time Hierarchy Theorem is

Theorem 7 (Time Hierarchy): *For every time-constructible functions* $t_1, t_2 : N \longrightarrow N$ *such that*

$$\lim_{n\to\infty} \frac{t_1(n) * \log t_2(n)}{t_2(n)} = 0$$

then

$$DTime_i(t_1(n)) \subset DTime_i(t_2(n)).$$

The proof for an analogous space hierarchy is easier, and therefore we'll present it first, but in following lectures.

Linear Speed-up: The following *Linear Speed-up Theorem* allows us to discard constant factors in running-time. Intuitively, there is no point in holding such an accurate account when one does not specify other components of the algorithm (like the size of its finite control and work-tape alphabet).

Theorem 8 (Linear Speed-up): *For every function* $t : N \longrightarrow N$ *and for every* i

$$DTime_i(t(n)) \subseteq DTime_i(\frac{t(n)}{2} + O(n)).$$

The proof idea is as following: let Γ be the original work alphabet. We reduce the time complexity by a constant factor by working with larger alphabet $\Gamma_k = \underbrace{\Gamma * \Gamma * ... * \Gamma}_{k \text{ times}}$, which enables us to process adjacent symbols simultaneously. Then we construct a new machine with alphabet Γ_k. Using this alphabet, any k adjacent cells of the original tape are replaced by one cell of the new tape.

So the new input will be processed almost k times faster, but dealing with the input will produce $O(n)$ overhead.

Let M_1 be an i-tape $t(n)$ time-bounded Turing machine. Let L be a language accepted by M_1. Then L is accepted by a i-tape $(\frac{t(n)}{2} + O(n))$ time-bounded TM M_2.

Proof: A Turing machine M_2 can be constructed to simulate M_1 in the following manner. First M_2 copies the input onto a storage tape, encoding 16 symbols into one. From this point on, M_2 uses this storage tape as the input tape and uses the old input tape as a storage tape. M_2 will encode the contents of M_1's storage tape by combining 16 symbols into one. During the course of the simulation, M_2 simulates a large number of moves of M_1 in one basic step consisting of eight moves of M_2. Call the cells currently scanned by each of M_2's heads the home cells. The finite control of M_2 records, for each tape, which of the 16 symbols of M_1 represented by each home cell is scanned by the corresponding head of M_2.

To begin a basic step, M_2 moves each head to the left once, to the right twice, and to the left once, recording the symbols to the left and right of the home cells in its finite control. Four moves of M_2 are required, after which M_2 has returned to its home cells.

Next, M_2 determines the contents of all of M_1's tape cells represented by the home cells and their left and right neighbors at the time when some tape head of M_1 first leaves the region represented by the home cell and its left and right neighbors. (Note that this calculation by M_2 takes no time. It is built into the transition rules of M_2.) If M_1 accepts before some tape head leaves the represented region, M_2 accepts. If M_1 halts, M_2 halts. Otherwise M_2 then visits, on each, the two neighbors of the home cell, changing these symbols and that of the home cell if necessary. M_2 positions each of it heads at the cell that represents the symbol that M_1's corresponding head is scanning at the end of the moves simulated. At most four moves of M_2 are needed.

It takes at least 16 moves for M_1 to move a head out of the region represented by a home cell and its neighbors. Thus in eight moves, M_2 has simulated at least 16 moves of M_1. ■

4 Bibliographic Notes

1. J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

2. E. Kushilevitz and N. Nisan, *Communication Complexity*, Cambridge University Press, 1996.

3. R.E. Ladner, *On the Structure of the Polynomial Time Reducibility*, Journal of the ACM 22(1):155-171, January 1975.

5 Appendix: Proof of Theorem 4, via crossing sequences

Consider one-tape TM M with transition function δ, input w of length m, such that M accepts w and an integer i, $0 < i < m$.

Denote by w_j, $0 \leq j \leq m$, the j'th symbol of the word w.

Consider the computation of the machine M on input w. This computation is uniquely determined by the machine description and the input, since the machine is deterministic.

The computation is a sequence of ID's (instantaneous descriptions), starting with $q_0 w_1 ... w_n$ and ending with $w_1 ... w_n p$ for some $p \in F$ (accepting or rejecting state).

Denote the elements of the computation sequence by $(ID_j)_{j=1}^r$ for some finite r.

Consider a sequence $(L_j)_{j=1}^{r-1}$ of pairs $L_j = (ID_j, ID_{j+1})$. For some $0 \leq i \leq m$ let $(L_{j_l})_{l=1}^t$ $t \leq r-1$ be the subsequence of $(L_j)_{j=1}^{r-1}$ of elements (ID_{j_l}, ID_{j_l+1}) of the form:

either

$ID_{j_l} = w_1 ... w_{i-1} p w_i ... w_n$ and $ID_{j_l+1} = w_1 ... w_i q w_{i+1} ... w_n$, for some $p, q \in Q$ and this specific i,

or

$ID_{j_l} = w_1 ... w_i p w_{i+1} ... w_n$ and $ID_{j_l+1} = w_1 ... w_{i-1} q w_i ... w_n$, for some $p, q \in Q$ and this specific i.

By definition of Turing machine computation $ID_{j_l} \vdash^M ID_{j_l+1}$ and therefore in the first case $\delta(p, w_i) = (q, w_{i+1}, R)$ and in the second case $\delta(p, w_{i+1}) = (q, w_i, L)$.

For every $1 \leq l \leq t$, let q_l be the state recorded in $ID_{j_{l+1}}$, where $ID_{j_{l+1}}$ are as above. Then the sequence $(q_l)_{l=1}^t$ is defined to be the **crossing sequence** of the triple (M, w, i) (machine M on input w in the boundary i).

Consider

$$L = \{wcw : w \in \{0,1\}^*\},$$

where c is a special symbol.

Clearly, a 2-tape TM will decide the language in $O(n)$ just by copying all the symbols before c to another tape and comparing symbol by symbol.

Let's prove that 1-tape TM will need $\Omega(n^2)$ steps to decide this language.

For $w \in \{0,1\}^m$, and $1 \leq i \leq m-1$, let $l_{w,i}$ be the length of the crossing sequence of (M, wcw, i). Denote by s the number of states of M.

Denote the average of $l_{w,i}$ over all m-long words w by $p(i)$.

Then from counting considerations, at least for half of w's it holds $l_{w,i} \leq 2 * p(i)$.

Let $N = 2^m$.So there are at least 2^{m-1} words w for which holds $l_{w,i} \leq 2 * p(i)$.

The number of possible crossing sequences of length $\leq 2 * p(i)$ is

$$\sum_{j=0}^{2*p(i)} s^j < s^{2*p(i)+1},$$

where s is the number of states of M.

So there is at least $\frac{2^{m-1}}{s^{2*p(i)+1}}$ words w of length m with the same crossing sequence for boundary $(i, i+1)$ (by pigeonhole principle). We are interested in such words w with the same suffix $(i+1, ..., m$ symbols). The number of such different suffixes is 2^{m-i}. Therefore if for some i holds

$$\frac{2^{m-1}}{s^{2*p(i)+1}} / 2^{m-i} > 1, \tag{3}$$

then by pigeonhole principle there are two different w's with the same suffix and the same crossing sequence between i and $i+1$ positions. We'll show that this leads to contradiction.

Denote the differing i-prefixes by α_1 and α_2 and the common $(m-i)$-bit suffix by β. Consider the input word $\alpha_1\beta c\alpha_1\beta$ and the input word $\alpha_2\beta c\alpha_1\beta$. Since the crossing sequence between α_1 and β is the same as the one between α_2 and β, the machine will not be able to distinguish between the two cases, and will accept the second word too, contradiction.

So Eq. (3) can not hold for any i and therefore for every i it holds

$$\frac{2^{m-1}}{2^{2*p(i)+1}2^{m-i}} \leq 1$$

and so

$$2^{i-1} \leq 2^{2*p(i)+1},$$

implying

$$i-1 \leq 2*p(i)+1,$$

and

$$p(i) \geq \frac{i-2}{2} \tag{4}$$

follows.

Denote by $T_m(w)$ the time needed to machine M to accept the word wcw. Let us compute the average time $Av_M(w)$ needed for M to accept wcw for an m-long word w.

$$Av_M(w) = \frac{1}{N}\sum_w T_M(w) \geq \frac{1}{N}\sum_w \sum_{i=1}^{m} l_{w,i},$$

because the running time of a TM on input w is the sum of lengths of the crossing sequences of (M, w, i) for $0 < i < m$. We have inequality here since there are crossing sequences for $i > m$ in the word wcw. Now, we have

$$\frac{1}{N}\sum_w \sum_{i=1}^{m} l_{w,i} = \sum_{i=1}^{m}\sum_w \frac{l_{w,i}}{N} = \sum_{i=1}^{m} p(i)$$

and so, by Eq. (4),

$$Av_M(w) \geq \sum_{i=1}^{m} p(i) \geq \sum_{i=1}^{m}\frac{i-2}{2} = \Omega(m^2).$$

So the average running time of M on wcw is $\Omega(m^2)$ implying that there exists an input wcw of length $2*m+1$ on which M runs $\Omega(m^2)$ steps.

Therefore, we have proved a lower bound of the worst case complexity of the language-decision problem for $L = \{wcw : w \in \{0,1\}^*\}$ on one-tape Turing machine. This lower bound is $\Omega(m^2)$ for $O(m)$-long input. On the other hand, this problem is decidable in $O(m)$ time by two-tape TM. Therefore

$$DTime_2(O(n)) \not\subseteq DTime_1(o(n^2)).$$

And so Theorem 3 is tight in the sense that there are functions $t(\cdot)$ such that

$$DTime_2(O(t(\cdot))) \not\subseteq DTime_1(o(t^2(\cdot))).$$