Solutions to Chapter II

Based on *Computational Complexity* [6]

Ricardo Magal

02/02/2026

Contents

Problems

Problem 2.8.6

Statement. Give another proof of Theorem 2.1 by presenting the k strings not next to each other, but on top of each other. Analyze the efficiency of your simulation and compare with the proof of Theorem 2.1. (A single symbol of the simulating machine will encode k symbols of the simulated one. Naturally, the cursors will be all over the place.)

Solution. Let M be a k-tape Turing machine with tape alphabet Γ, and suppose M runs for at most $t(n)$ steps on inputs of length n. We construct a single-tape Turing machine M' that simulates M by encoding the k tapes *on top of each other* (multi-track encoding). This is a standard simulation trick; see, e.g., [9, 3] for related encodings and simulations.

Encoding (stacked tracks). At each tape position $i \geq 0$, M' stores a k-tuple $(a_1, \ldots, a_k) \in \Gamma^k$, where a_j is the symbol in cell i of tape j of M. Thus *one* symbol of M' encodes k symbols of M, exactly as required.

To represent head positions, each track uses a marked version of Γ: for each $a \in \Gamma$ there is a marked symbol $\widehat{a}$. A configuration is encoded so that, in each track j, exactly one cell is marked (the cell currently scanned by tape j's head). The alphabet of M' is therefore a constant-size expansion of Γ^k, since k is fixed.

Simulating one step. Assume M is in some configuration C. To simulate one transition of M, M' must:

(i) locate, on each track, the unique marked symbol (these are the k scanned symbols of M),

(ii) apply M's transition function in its finite control (possible because k is constant),

(iii) update the k written symbols and move each head marker left/right/stay accordingly.

Because the k head positions may be far apart, M' finds them by sweeping over the *active* portion of its tape. Let L be the maximum tape index ever visited by any head of M up to the current time. Then the entire configuration C is represented within cells $0, 1, \ldots, L$ of M''s tape.

A single simulated step can be done with a constant number of full sweeps over $0..L$: one sweep to read the k marked symbols, and one sweep to perform the corresponding updates and move the markers. Therefore one step of M costs $O(L)$ steps of M'.

Running time bound. After r simulated steps, each head of M has moved at most r times, hence $L \leq O(n + r)$. In particular, throughout the simulation of a $t(n)$-step computation, we have $L = O(n + t(n)) = O(t(n))$. Thus M' uses $O(t(n))$ time to simulate each step of M, and the total simulation time is

$$O\left(\sum_{r=1}^{t(n)} L_r\right) \;=\; O\left(\sum_{r=1}^{t(n)} (n + r)\right) \;=\; O\big(t(n)^2 + n\, t(n)\big) \;=\; O\big(t(n)^2\big)\,.$$

Comparison with Theorem 2.1. Theorem 2.1 shows that multitape machines can be simulated by a single-tape machine with at most quadratic slowdown. The stacked-track encoding yields the same asymptotic overhead as the standard "strings side by side" proof: the key cost is that, with one head, M' must sweep over the active region to coordinate k independent head positions. The difference is mainly representational (a constant-factor change in alphabet size and bookkeeping), not asymptotic.

Problem 2.8.7

Statement. Suppose that we have a Turing machine with an infinite two-dimensional tape (blackboard?). There are now moves of the form $\uparrow$ and $\downarrow$, along with $\leftarrow$ and $\rightarrow$. The input is written initially to the right of the initial cursor position.

(a) Give a detailed definition of the transition function of such a machine. What is a configuration?

(b) Show that such a machine can be simulated by a 3-string Turing machine with a quadratic loss of efficiency.

Solution.

(a) **Formal model and transition function.** A two-dimensional Turing machine M is specified by $M = (Q, \Sigma, \Gamma, \delta, s, \mathsf{yes}, \mathsf{no})$, where Q is a finite set of states, $\Sigma \subseteq \Gamma$ is the input alphabet, Γ is the tape alphabet (with a designated blank symbol $u \in \Gamma$), $s \in Q$ is the start state, and $\mathsf{yes}, \mathsf{no} \in Q$ are halting accept/reject states. The two-dimensional tape is indexed by $\mathbb{Z} \times \mathbb{Z}$. A tape *content* is a function $T : \mathbb{Z} \times \mathbb{Z} \to \Gamma$ such that $T(i, j) = u$ for all but finitely many (i, j) (finite nonblank convention). The head position (the cursor) is a pair $(x, y) \in \mathbb{Z} \times \mathbb{Z}$.

The transition function is

$$\delta : (Q \setminus \{\mathsf{yes}, \mathsf{no}\}) \times \Gamma \;\to\; Q \times \Gamma \times \{\leftarrow, \rightarrow, \uparrow, \downarrow\}. \tag{1}$$

If the machine is in state $q \in Q \setminus \{\mathsf{yes}, \mathsf{no}\}$ and the head scans symbol $a = T(x, y)$, then $\delta(q, a) = (q', b, D)$ prescribes:

- write b in the current cell, i.e., set $T(x, y) \leftarrow b$;
- update the state to q';
- move the head one step according to D: $(x, y) \leftarrow (x - 1, y)$ if $D = \leftarrow$; $(x, y) \leftarrow (x + 1, y)$ if $D = \rightarrow$; $(x, y) \leftarrow (x, y + 1)$ if $D = \uparrow$; $(x, y) \leftarrow (x, y - 1)$ if $D = \downarrow$.

The input $w = w_1 \cdots w_n \in \Sigma^*$ is placed initially along the x-axis to the right of the origin: $T(i, 0) = w_i$ for $i = 1, \ldots, n$, all other cells are blank u, and the initial head position is $(0, 0)$ in state s.

A configuration (instantaneous description) is a triple

$$C = (q, T, (x, y)), \tag{2}$$

consisting of the current state q, the full tape function T, and the head coordinates (x, y).

(b) **Simulation by a 3-string machine with quadratic slowdown.** Let M be any two-dimensional machine that halts within $t = t(n)$ steps on inputs of length n. We build a 3-string Turing machine M' that simulates M in $O(t^2)$ time.

Bounding the active region. In t steps, the head of M moves at most t times in each coordinate, hence it never leaves the square $[-t, t] \times [-t, t]$. Therefore, during a t-step computation, only $O(t^2)$ cells can possibly be visited or modified.

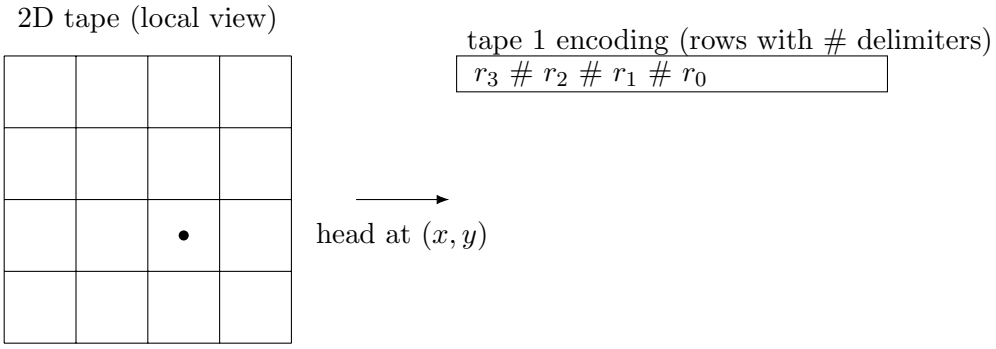
A fixed-grid encoding on one tape. Choose a side length m that is a power of two and satisfies $m \geq 2t + 1$. Encode the $m \times m$ square centered at the origin on tape 1 of M' as a flattened grid:

$$\text{row } (m-1) \# \text{ row } (m-2) \# \cdots \# \text{ row } 0, \tag{3}$$

where each row is a block of exactly m symbols from Γ, and $\# \notin \Gamma$ is a delimiter. One cell in the grid is marked to indicate the simulated head position: use a marked copy $\widehat{\Gamma} = \{\widehat{a} : a \in \Gamma\}$, and the unique marked symbol in the encoding indicates the current head cell and its scanned symbol.

Tape 2 stores m in unary (as 1^m) so that moving up/down by one row can be implemented by counting m steps. Tape 3 is a work tape used for copying during grid expansions (described below).

A schematic picture.



Simulating one transition. Assume tape 1's head is positioned on the unique marked symbol (the simulated head cell). If M is in state q and the marked cell carries $\widehat{a}$, then M' computes $\delta(q, a) = (q', b, D)$ in its finite control, replaces $\widehat{a}$ by b, and then moves the mark according to D:

- If $D \in \{\leftarrow, \rightarrow\}$, move one symbol left/right within the current row (if a delimiter # is encountered, a resize is triggered).
- If $D = \uparrow$, move left by $m+1$ positions on tape 1 (crossing exactly one delimiter #), using the unary counter 1^m on tape 2 to count m steps, plus one step to cross #.
- If $D = \downarrow$, move right by $m + 1$ positions analogously.

When the new cell is reached, M' marks it by replacing its symbol $c \in \Gamma$ with $\widehat{c}$.

Cost per simulated step. Left/right moves cost $O(1)$ tape-head moves. Up/down moves cost $O(m)$ tape-head moves to traverse one row-length. Thus each simulated step costs $O(m)$ time.

Handling boundary growth by doubling. If a move would leave the represented $m \times m$ grid, M' doubles the grid size: rebuild tape 1 into a $2m \times 2m$ blank grid and copy the old $m \times m$ contents into the centered sub-square, preserving the marked head cell. This rebuild is done by a linear scan/copy using tape 3 as scratch and takes $O(m^2)$ time. Tape 2 is updated from 1^m to 1^{2m} in $O(m)$ time.

Because m doubles, at most $\lceil \log_2(2t+1) \rceil$ rebuilds occur in a t-step computation.

Total running time. Throughout the simulation, $m = O(t)$. Simulating t steps costs $t \cdot O(m) = O(t^2)$. The rebuild costs form a geometric series $O(1^2)+O(2^2)+O(4^2)+\cdots+O(t^2) = O(t^2)$, so they do not change the asymptotic bound. Therefore the overall simulation runs in $O(t^2)$ time, i.e., a quadratic loss of efficiency.

Problem 2.8.8

Statement. Suppose that Turing machines can delete and insert symbols in their string, instead of only overwriting.

(a) Define carefully the transition function and the computation of such machines.

(b) Show that such a machine can be simulated by an ordinary Turing machine with a quadratic loss of efficiency.

Solution. We formalize an edit operation-TM (an "insert/delete" machine) and then simulate it by an ordinary overwrite-only machine. The simulation follows the standard theme that reasonable model variations preserve polynomial time, and here yields a quadratic overhead; see, e.g. [9, 3].

(a) **Definition of the model.** An insert/delete Turing machine M is a tuple $M = (Q, \Sigma, \Gamma, u, \delta, s, \mathsf{yes}, \mathsf{no})$ where Q is a finite set of states, $\Sigma \subseteq \Gamma$ is the input alphabet, Γ is the tape alphabet, $u \in \Gamma$ is the blank symbol, $s \in Q$ is the start state, and $\mathsf{yes}, \mathsf{no} \in Q$ are halting states.

The "string" and head position. At any moment the tape content is required to be a finite string $w \in \Gamma^*$ written contiguously starting at cell 1 (to the right of the left end marker), followed by blanks u forever. The head position is an index $i \in \{1, \ldots, |w|+1\}$, where $i = |w|+1$ means the head is scanning the first blank immediately after the string. Thus the scanned symbol is $a = w_i$ if $1 \le i \le |w|$, and $a = u$ if $i = |w|+1$.

Transition function. The transition function has the form

$$\delta : (Q \setminus \{\mathsf{yes}, \mathsf{no}\}) \times \Gamma \;\to\; Q \times \Gamma \times \{\mathsf{write}, \mathsf{ins}, \mathsf{del}\} \times \{L, R, S\}. \tag{4}$$

Given $\delta(q, a) = (q', b, \mathsf{op}, D)$, one step of computation from state q scanning a proceeds as follows.

- write: if $1 \le i \le |w|$, replace $w_i \leftarrow b$; if $i = |w|+1$, this appends b at the end (so the new string becomes wb).
- ins (insert): insert b *at position* i and shift the suffix right by one: the new string is $w' = w_1 \cdots w_{i-1}\, b\, w_i \cdots w_{|w|}$. In particular, if $i = |w|+1$ this is again append.

- del (delete): if $1 \le i \le |w|$, delete w_i and shift the suffix left by one: $w' = w_1 \cdots w_{i-1}\, w_{i+1} \cdots w_{|w|}$. If $i = |w| + 1$ (scanning the first blank), deletion is a no-op.

After applying op, the head index is updated by the move D: $i \leftarrow i - 1$ if $D = L$ and $i > 1$, otherwise $i \leftarrow i$; $i \leftarrow i + 1$ if $D = R$ and $i \le |w'|$, otherwise $i \leftarrow i$; and $i \leftarrow i$ if $D = S$. The state updates to q'. A configuration is therefore a triple (q, w, i) consisting of the current state, the current string, and the current head index.

(b) **Simulation by an ordinary TM with quadratic overhead.** Let M be an insert/delete machine that runs for at most $t = t(n)$ steps on inputs of length n. We construct an ordinary overwrite-only single-tape Turing machine M' that simulates M.

Representation invariant. M' stores the current string w contiguously on its tape (starting at cell 1), followed by blanks. It also uses a marked version of the alphabet to indicate the simulated head position: for each $a \in \Gamma$, M' has a symbol $\widehat{a}$. Exactly one cell among $1, \ldots, |w| + 1$ is marked: $\widehat{a}$ if the head is scanning $a \in \Gamma$, and $\widehat{u}$ if the head is scanning the first blank after w. This makes the scanned symbol and position locally identifiable during a sweep.

Simulating operations. Given q and the marked symbol $\widehat{a}$, M' computes $\delta(q, a) = (q', b, \mathsf{op}, D)$ in its finite control.

- write: replace the marked cell by b (or, if it is $\widehat{u}$, write b there, extending the string by one), then re-mark the appropriate neighbor according to D. This costs $O(1)$ time.
- ins: to insert b at the marked position, M' shifts the suffix one cell to the right. Concretely, it sweeps right to find the first blank after the string, then sweeps left performing the standard right-shift: carry the symbol in a control register and move it one cell right, until reaching the insertion point; finally write b at the insertion point. This is a linear-time shift in the current string length.
- del: to delete the symbol under the mark (if any), M' shifts the suffix one cell to the left. It overwrites the marked cell with the symbol to its right, then continues sweeping right, copying each symbol one cell left, until it reaches the final blank, which it leaves blank. Again this is linear-time in the current string length.

After the edit, M' moves the mark one step left/right/stay as specified by D, using local changes plus (if needed) a short sweep to find the new mark position. The key point is that ins and del cost $O(m)$ time where m is the current string length, while write costs $O(1)$.

Quadratic time bound. Let m_r be the string length after r steps of M. Each step can change the length by at most 1, hence $m_r \le n + r$ for all r. The simulation cost of step r is $O(m_r)$, so the total simulation time is

$$\sum_{r=0}^{t-1} O(m_r) \le \sum_{r=0}^{t-1} O(n + r) = O\left(tn + \sum_{r=0}^{t-1} r\right) = O\big(tn + t^2\big) = O(t^2).$$

Thus M' simulates M with at most a quadratic slowdown.

Conclusion. Allowing insert and delete does not increase computational power beyond ordinary Turing machines; it only changes the cost model by at most a quadratic factor under straightforward simulation.

Problem 2.8.9

Statement. Show that any k-tape Turing machine operating within time $f(n)$ can be simulated by a 2-string Turing machine within time $f(n) \log f(n)$. (This is a clever simulation, from [2]. The trick is to keep all strings on top of one another in a single string—that is, a single symbol will encode k, as in Problem 2.8.6 above—so that the locations of all cursors coincide; the strings are unbounded in both directions, say. Cursor motions are simulated by moving blocks of symbols around, into spaces that were intentionally written sparsely; the sizes of the blocks form a geometric progression. The second string is only used for *fast copying*.) This result is a useful substitute for Theorem 2.1, when a Turing machine with a fixed number of strings has to be constructed, as it achieves better time bounds.

Solution. Let M be a k-tape Turing machine with a fixed number k of work tapes, running in time $t = f(n)$ on inputs of length n. We describe a 2-string Turing machine M' that simulates M in $O(t \log t)$ time. The construction is based on the classical two-tape simulation of multitape machines by Hennie and Stearns [2]; see also [9].

1) Goal: align all heads at one physical position

We encode all k simulated tapes on top of each other on tape 1 of M': one tape-1 cell encodes a k-tuple of symbols, one per simulated tape. We maintain the invariant that the k simulated head positions always correspond to the same tape-1 cell of M' (the "shared cursor"). Thus reading the k scanned symbols is constant-time (up to fixed k) and applying M's transition is done in the finite control.

The difficulty is simulating independent head moves when all heads are forced to coincide. This is handled by representing, for each simulated tape, the tape content as two strings split at the head (left-of-head and right-of-head), and maintaining these strings in a sparse block layout that supports efficient shifts.

2) Sparse block layout with geometric block sizes

Fix a block hierarchy of sizes $1, 2, 4, 8, \ldots, 2^m$, where $m = \lceil \log_2 t \rceil$. For each simulated tape i, we maintain two sequences of blocks on tape 1:

- a left stack encoding the symbols to the left of the head (in reverse order), and
- a right stack encoding the symbol under the head and those to its right (in forward order).

Each stack is partitioned into consecutive blocks whose sizes follow the geometric progression $1, 2, 4, \ldots$, where each block of size 2^j is stored with padding (blank gaps) so it can be shifted or rebuilt locally without immediately colliding with neighboring blocks. Intuitively, the layout is "mostly blank"; actual symbols occupy only a sparse subset of cells, leaving room to move blocks around.

Tape 2 is used only as a work tape for copying/rebuilding blocks quickly ("fast copying").

3) Simulating a move by local rebalancing

A single simulated head move for tape i corresponds to popping one symbol from one stack and pushing it onto the other:

- move right: pop the head symbol from the right stack and push it onto the left stack;
- move left: pop the top symbol from the left stack and push it onto the right stack.

If the needed symbol is present in the top (small) blocks, this is done by constant local edits near the shared cursor.

When a small block becomes empty (or overfull), we rebalance by merging/splitting blocks at increasing sizes, exactly like carrying in binary. For example, if the size-2^0 block on the right stack is empty, we rebuild it by taking one symbol from the size-2^1 block; if that is empty too, we go to size-2^2, and so on. A rebuild at level j involves moving $\Theta(2^j)$ symbols, which costs $\Theta(2^j)$ time on tape 1, but it happens only after $\Theta(2^j)$ simulated moves affecting that stack, so its amortized analysis cost is $O(1)$ per move for that level.

4) Time bound: $O(t \log t)$

Each simulated step of M performs:

- a constant amount of finite-control work to compute the transition, and
- up to k head moves, each implemented by rebalancing in one stack hierarchy.

For a fixed tape i, the stack hierarchy has $m + 1 = O(\log t)$ levels. At each level j, rebuilding a block costs $\Theta(2^j)$ but can be charged to $\Theta(2^j)$ preceding moves that created the imbalance. Thus each level contributes $O(1)$ amortized cost per simulated move, and summing over all levels yields $O(\log t)$ amortized time per simulated step. Since k is fixed, the constant factor does not affect asymptotics.

Therefore the total simulation time is $O(t \log t) = O(f(n) \log f(n))$, as required.

5) Conclusion

A k-tape Turing machine running in time $f(n)$ can be simulated by a 2-string Turing machine within time $O(f(n) \log f(n))$, using sparse geometric blocks on one string and the second string for fast copying, as in [2].

Problem 2.8.10

Statement. Call a Turing machine M oblivious Turing machine if the position of its cursors at the t-th step of its computation on input x depends only on t and $|x|$, not x itself. Problem: Show that the simulating two-string machine above can be made oblivious. (Oblivious machines are useful because they are closer to more "static" models of computation, such as Boolean circuits; see Problem 1.5.22.)

Solution. We refer to the 2-string simulation from Problem 2.8.9 (Hennie–Stearns style [2]) that simulates a fixed-k multitape machine M running in time $t = f(n)$ by a 2-string machine M' in time $O(t \log t)$. We show how to modify M' so that its head trajectories depend only on the simulated time step and the input length, i.e., M' becomes oblivious Turing machine, without changing the $O(t \log t)$ bound by more than a constant factor.

1) What needs to be made input-independent

In the simulation of Problem 2.8.9, tape 1 stores a sparse, geometric block representation of the k simulated tapes around a shared cursor. The simulator performs:

- a *local update* near the shared cursor to apply M's transition, and
- occasional *rebalancing / rebuilding* of blocks of size 2^j when a small block underflows/overflows (triggered by the simulated head motions).

Local updates already happen at the shared cursor, so they do not require data-dependent long travel. The non-obliviousness arises because the times at which block rebuilds are triggered depend on the actual simulated head motions, which depend on the input.

The fix is to *schedule all possible rebuilds in advance*, often enough to handle the worst case, and execute them even if they are not strictly necessary. Since we schedule for the worst case, the simulator remains correct for every input, and the schedule depends only on t (hence on n) and on the step counter.

2) Deterministic rebuild schedule

Let the simulated machine M run for at most t steps. As in Problem 2.8.9, maintain geometric block levels $j = 0, 1, \ldots, m$, where $m = \lceil \log_2 t \rceil$. A level-$j$ rebuild touches $\Theta(2^j)$ symbols per simulated tape and costs $O(2^j)$ time on the simulating machine (using the second tape for copying).

Define a fixed schedule over the t simulated steps:

> At simulated step $r \in \{1, \ldots, t\}$, perform a rebuild at level j for every j such that $2^j \mid r$.

Equivalently, in step r we rebuild exactly those levels j corresponding to the trailing zeros of r in binary; this depends only on r, not on the input.

Why this schedule is sufficient. In 2^j simulated steps, each simulated head can move at most 2^j positions. Thus, regardless of the input, the amount of mass (symbols) that must flow between the level-j and level-$(j{+}1)$ representations is bounded by $O(2^j)$ over any interval of length 2^j. Rebuilding level j at least once every 2^j simulated steps is therefore enough to keep the level-j invariant valid in the worst case. If the simulated heads happened to move less, the rebuild simply reconstructs the same blocks (a no-op on the represented strings), but still follows the same head trajectory.

3) Making the simulator's head trajectory fixed

Fix once and for all the physical layout of the level-j blocks on tape 1 (including the padding gaps) and the work area on tape 2. In step r, the simulating machine performs the following phases in a fixed order:

1. **Local phase (always).** Stay at the shared cursor region, read the scanned symbols (a constant-size neighborhood because k is fixed), compute M's next move in the finite control, and locally update the encoded symbols that represent the scanned cells and the next state.
2. **Rebuild phase (scheduled).** For each j with $2^j \mid r$ (in increasing j), execute the level-j rebuild by scanning exactly the predetermined tape-1 segment that stores the level-j blocks (for all k tapes) and using tape 2 for the standard copy/repack procedure. Each such rebuild is implemented by the same left-to-right and right-to-left scans of fixed-length segments, independent of contents.

Because the list of j values is determined by r, and the scan boundaries for each level are fixed by the layout chosen from t and n, the positions of the simulator's two heads at every internal step are a deterministic function of r and n only. Hence the modified simulator is oblivious Turing machine.

4) Running-time bound remains $O(t \log t)$

We bound the extra work caused by rebuilding on a fixed schedule.

A level-j rebuild costs $O(2^j)$ time. By the schedule, level j is rebuilt exactly $\lfloor t/2^j \rfloor$ times. Therefore the total time spent on level j rebuilds is

$$O(2^j) \cdot \left\lfloor \frac{t}{2^j} \right\rfloor = O(t).$$

Summing over all levels $j = 0, 1, \ldots, m$ gives total rebuild time $O(t(m+1)) = O(t \log t)$. The local phase costs $O(1)$ per simulated step (for fixed k), hence $O(t)$ overall. Thus the entire simulation still runs in $O(t \log t) = O(f(n) \log f(n))$ time.

5) Conclusion

By replacing data-dependent rebuild triggers with a worst-case periodic rebuild schedule, the two-string simulating machine can be made oblivious Turing machine while preserving the $O(f(n) \log f(n))$ time bound up to constant factors.

Problem 2.8.11

Statement. Regular languages. Suppose that a language L is decided by a Turing machine within space k, and that k is a constant.

(a) Show that there is an equivalent Turing machine deciding L with only a read-only input string (acceptance and rejection is signaled by states "yes" and "no", say). Such a machine is called a two-way finite automaton. A one-way finite automaton is a two-way finite automaton with only $\rightarrow$ cursor moves (a "yes" or "no" state is entered when the first blank is encountered). The language decided by a one-way finite automaton is called regular language.

(b) Show that the following language is regular language:

$$L = \{\, x \in \{0,1\}^* : x \text{ contains at least two 0s but not two consecutive 0s} \,\}.$$

For more on regular language languages see the book by Lewis and Papadimitriou cited above, as well as Problems 3.4.2 and 20.2.13.

(c) Let $L \subseteq \Sigma^*$ be a language, and define the following equivalence relation on Σ^*: $x \equiv_L y$ if for all $z \in \Sigma^*$, $xz \in L$ iff $yz \in L$. Show that L is regular language if and only if there are finitely many equivalence classs in $\equiv_L$. How many equivalence classes are there in the case of the language in (b) above?

(d) Show that if a language L is decided by a two-way finite automaton, then it is regular language. (This is a quite subtle argument from [7].)

Show that $x \equiv_L y$ if and only if x and y "effect the same behavior" on the two-way finite automaton. Here by "behavior" one should understand a certain mapping from state to state (cf. part (c)). We conclude that SPACE(k), where k is any integer, is precisely the class of regular language languages (part (c)).

(e) Show that, if L is infinite and regular language, then there are $x, y, z \in \Sigma^*$ such that $y \neq \epsilon$, and $xy^i z \in L$ for all $i \geq 0$. (Since there are finitely many states in the machine, one long enough string one must repeat.)

(f) Recall that $b(i)$ stands for the binary representation of integer i, with no leading zeros. Show that the language of "arithmetic progression"

$$L = \{\, b(1);\ b(2);\ \cdots;\ b(n) : n \geq 1 \,\}$$

is not regular language. (Use part (e) above.)

(g) Show that L in part (f) above can be decided in space $\log \log n$.

Solution.

(a) SPACE($O(1)$) **implies a two-way finite automaton.** Let M be a decider for L that uses at most k work-tape cells, where k is a fixed constant. Assume the standard space-complexity model: M has a read-only input tape, a read/write work tape, and halting accept/reject states.

Because the work tape has only k cells, a complete description of the work tape consists of:

- the current control state $q \in Q$,
- the contents of the k work-tape cells (an element of Γ^k), and
- the work-tape head position (an integer in $\{1, \ldots, k\}$).

There are only finitely many such triples. Let Q' be the set of all these triples, and view them as the states of a new machine A that has *no* work tape at all. On input symbol $a \in \Sigma \cup \{\sqcup\}$ and state $(q, \tau, p) \in Q'$, A looks up M's transition on (q, a, τ_p), updates (q, τ, p) accordingly, and moves only its input head $L/R/S$ as M would. Accept and reject are represented by dedicated halting states.

Thus A is a two-way finite automaton (read-only input, finite control) deciding L.

(b) **A DFA for the given language.** Let $L = \{x \in \{0,1\}^* : x$ has at least two 0s and no substring $00\}$. A one-way finite automaton can track (i) whether we have seen 0 zeroes, exactly 1 zero, or at least 2 zeroes, and (ii) whether the previous symbol was 0.

Define states:

- q_0: seen 0 zeroes so far, previous symbol is not 0 (start);
- $q_{1,0}$: seen exactly 1 zero, previous symbol is 0;
- $q_{1,1}$: seen exactly 1 zero, previous symbol is 1;
- $q_{2,0}$: seen at least 2 zeroes, previous symbol is 0 (accepting);
- $q_{2,1}$: seen at least 2 zeroes, previous symbol is 1 (accepting);
- $q_\times$: dead state (the string has a substring 00).

Transitions are the obvious ones (reading 0 from a state with previous 0 goes to $q_\times$, otherwise it increases the zero-counter class and sets "previous 0"). This DFA accepts exactly L, so L is regular language.

(c) **Myhill–Nerode theorem and counting classes for (b).** Define $x \equiv_L y$ iff for all $z \in \Sigma^*$, $xz \in L \Leftrightarrow yz \in L$. This is a right-invariant equivalence relation (a right congruence).

($\Rightarrow$) If L is regular, then $\equiv_L$ has finitely many classes. Let $A = (Q, \Sigma, \delta, q_0, F)$ be a one-way finite automaton for L. For any $x \in \Sigma^*$, let $\hat{\delta}(q_0, x)$ be the state reached after reading x. If $\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$, then for every z, $xz \in L \Leftrightarrow \hat{\delta}(q_0, xz) \in F \Leftrightarrow \hat{\delta}(q_0, yz) \in F \Leftrightarrow yz \in L$. Hence $x \equiv_L y$. Therefore the number of $\equiv_L$-classes is at most $|Q|$, finite.

($\Leftarrow$) If $\equiv_L$ has finitely many classes, then L is regular. Assume $\equiv_L$ has finitely many equivalence classes. Construct a DFA whose states are the classes $[x]$, with start state $[\epsilon]$, accepting states $\{[x] : x \in L\}$, and transition $\delta([x], a) = [xa]$. This is well-defined because $x \equiv_L y \Rightarrow xa \equiv_L ya$. By construction, the DFA accepts exactly L. Thus L is regular language. (This is the Myhill–Nerode theorem; see [5, 4, 3, 9].)

Number of classes for the language in (b). For L from (b), $\equiv_L$ has exactly 6 classes, corresponding to the DFA above:

- strings with no 0 yet (q_0);
- exactly one 0, previous symbol 0 ($q_{1,0}$);
- exactly one 0, previous symbol 1 ($q_{1,1}$);
- at least two 0s, previous symbol 0 ($q_{2,0}$);
- at least two 0s, previous symbol 1 ($q_{2,1}$);
- the dead class containing a substring 00 ($q_\times$).

These classes are pairwise distinct (for instance, appending 0 distinguishes $q_{1,0}$ from $q_{1,1}$, and distinguishes $q_{2,0}$ from $q_{2,1}$).

(d) **Two-way DFA implies regular.** Let A be a two-way finite automaton deciding L, with state set Q. The standard theorem is that two-way deterministic finite automata recognize exactly the regular language languages; equivalently, every two-way finite automaton has an equivalent one-way finite automaton. A proof can be given by associating to each string x a finite "behavior" object that records how A moves across the boundary between x and a suffix, and showing that there are only finitely many such behaviors (bounded by a function of $|Q|$), so $\equiv_L$ has finite index; then apply part (c). This construction and its correctness are due to Shepherdson [7]; modern presentations appear in standard texts (e.g. [3, 9]). Therefore L is regular language.

(e) **Pumping lemma for regular languages.** Let L be infinite and regular language, and let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA for L. Let $p = |Q|$. Since L is infinite, there exists $w \in L$ with $|w| \geq p$. Consider the run of A on w. Among the first $p+1$ visited states, some state repeats; thus $w = xyz$ with $|xy| \leq p$, $|y| \geq 1$, and $\hat{\delta}(q_0, xy^i z) \in F$ for all $i \geq 0$. Hence $xy^i z \in L$ for all $i \geq 0$. (This is the pumping lemma; see [3, 9].)

(f) **The "arithmetic progression" language is not regular.** Let $L = \{\, b(1);\, b(2);\, \cdots ;\, b(n) : n \geq 1 \,\}$ over the alphabet $\{0, 1, ;\}$. Assume, for contradiction, that L is regular language. Let p be its pumping length from (e).

Choose k such that the fixed prefix $P = b(1);\, b(2);\, \cdots ;\, b(k)$ has length at least p. Now take $w = b(1); \cdots ;\, b(n) \in L$ with $n \geq k$, so P is a prefix of w, and the first p symbols of w lie entirely inside P.

By the pumping lemma, write $w = xyz$ with $|xy| \leq p$ and $|y| \geq 1$. Then y lies within the first p symbols of w, hence within the fixed prefix P. Consider $w' = xy^0z = xz$, obtained by deleting y. Because $|y| \geq 1$, the first p symbols of w' differ from the first p symbols of w, hence w' does *not* have prefix P.

But every string in L of the form $b(1); \cdots ; b(m)$ with $m \geq k$ necessarily begins with the same prefix P. Also, w' is longer than $|P|$ (it still contains the suffix of w beyond P), so w' cannot equal any string $b(1); \cdots ; b(m)$ with $m < k$. Therefore $w' \notin L$, contradicting the pumping lemma conclusion that $xy^iz \in L$ for all $i \geq 0$. Hence L is not regular language.

(g) **Deciding the language in space** $\log \log n$**.** Let the input length be n. We sketch a decider using $O(\log \log n)$ work space.

The input is a sequence of binary blocks separated by the delimiter ";". The algorithm checks:

(a) the first block is exactly $b(1) = 1$;

(b) every subsequent block is the binary successor of the previous block;

(c) the string ends exactly after the last block.

Space bound. At any point, the algorithm stores only:

- the current block length $\ell \leq \lfloor \log_2 n \rfloor + 1$, using $O(\log \ell) = O(\log \log n)$ bits;
- an index $j \in \{0, \ldots, \ell\}$ while comparing bits from right to left, using $O(\log \ell)$ bits;
- a constant number of flags (carry bit, etc.).

Successor test using two-way input head. To check that a block B is the successor of block A, the machine repeatedly scans to the end of each block (the next ";" or end-of-input) and then moves left j positions to read the j-th bit from the right. By iterating $j = 0, 1, \ldots, \ell - 1$ and maintaining a one-bit carry, it verifies the addition-by-one rule without storing the full value of A or B. The special case where $|B| = \ell + 1$ is handled by checking that A is all ones and B is 1 followed by ℓ zeros.

Because the input head may move both directions, the machine can re-scan blocks as many times as needed; time is not constrained in a space bound. Thus the decider uses $O(\log \log n)$ space.

Problem 2.8.12

Statement. Show that if a Turing machine uses space that is smaller than $c \log \log n$ for all $n > 0$, then it uses constant space. (This is from [1].)

Consider a "state" of a machine with input to also include the string contents of the work strings. Then the behavior of the machine on an input prefix can be characterized as a mapping from states to states. Consider now the shortest input that requires space $S > 0$; all its prefixes must exhibit different behaviors—otherwise a shorter input requires space S. But the number of behaviors is doubly exponential in S. Notice that in view of this result, language L in (g) of Problem 2.8.11 above requires $O(\log \log n)$ space.

Solution. Fix a deterministic Turing machine M in the usual space model (read-only input tape plus a constant number of work tapes). Let $s(n)$ be the maximum number of work-tape cells M ever scans on any input of length n. Assume $s(n) < c \log\log n$ for all $n > 0$, where $c > 0$ is a constant to be chosen later. We prove that $s(n)$ is bounded by an absolute constant (depending only on M and c), i.e., M uses constant space.

1) Extended states and prefix behaviors

For an integer $S \geq 0$, define an extended state of M at space bound S to consist of:

- the control state $q \in Q$,
- the contents of all work tapes restricted to the first S visited cells (an element of $\Gamma^{O(S)}$), and
- the positions of all work-tape heads (each in $\{1, \ldots, S\}$, or a special value if $S = 0$).

The number of such extended states satisfies

$$N(S) \;=\; 2^{O(S)}. \tag{5}$$

(The constant hidden in $O(S)$ depends on M: number of tapes, $|Q|$, $|\Gamma|$.)

Now fix an input alphabet Σ and consider any string $x \in \Sigma^*$. Define the *behavior* of x at space bound S to be the function $B_x : \{1, \ldots, N(S)\} \to \{1, \ldots, N(S)\}$ mapping an extended state σ to the extended state σ' obtained as follows: start M in extended state σ with the input head positioned *immediately to the right of* x (at the boundary between x and the suffix), and run M until the next time its input head again returns to that boundary position with the same orientation convention (for definiteness: the first time it is at the boundary and about to move to the right). If such a return happens, let σ' be the extended state then; if it never happens (e.g., the machine halts first), encode that outcome by a designated sink value. Because M is deterministic and we restrict attention to computations that stay within S work cells, this defines a well-defined function B_x on a finite domain.

The crucial property is *compositionality*: for any x, y,

$$B_{xy} = B_y \circ B_x. \tag{6}$$

Intuitively, the effect of first exposing the machine to x and then to y is the composition of the two boundary-transfer behaviors.

2) Counting behaviors: doubly exponential in S

Since B_x is a function on a set of size $N(S)$, the number of possible behaviors is at most

$$\#\mathrm{Beh}(S) \;\leq\; N(S)^{N(S)} \;=\; 2^{N(S)\log N(S)} \;=\; 2^{2^{O(S)}}, \tag{7}$$

using (5). This is the "doubly exponential" bound from the hint.

3) Minimal hard input implies all prefixes have distinct behaviors

Assume there exists an input on which M uses space $S > 0$. Let w be a *shortest* such input, and write $n = |w|$. For each prefix $w_{\leq i}$ of length $i \in \{0, 1, \ldots, n\}$, consider its behavior $B_{w_{\leq i}}$ at space bound S.

We claim that these $n+1$ behaviors are all distinct. Indeed, if $B_{w_{\le i}} = B_{w_{\le j}}$ for some $0 \le i < j \le n$, write $w = uvz$ where $u = w_{\le i}$ and $uv = w_{\le j}$, so $v \neq \epsilon$. By (6), $B_{uv} = B_v \circ B_u$. If $B_u = B_{uv}$, then for any continuation z the machine's boundary interactions with the suffix after u are identical whether or not v is present: the composition law forces the same transfer on extended states at the $u \,|\, z$ boundary. Consequently, the computation of M on uz reproduces the same sequence of extended states at the boundary as on $uvz = w$, and in particular reaches a configuration using S work cells as well. Thus uz is a strictly shorter input requiring space S, contradicting the choice of w. Hence all prefix behaviors are distinct.

Therefore,

$$n+1 \;\le\; \#\mathrm{Beh}(S) \;\le\; 2^{2^{O(S)}}. \tag{8}$$

4) Inverting the bound and concluding constant space

From (8) there exists a constant $a > 0$ (depending only on M) such that for all $S \ge 1$,

$$\log\log n \;\le\; aS + a. \tag{9}$$

(For example, if $n \le 2^{2^{aS}}$ then $\log\log n \le aS$, and additive constants can absorb the "+1" in $n+1$.)

But on the minimal hard input w of length n, we also have $S \le s(n) < c\log\log n$. Combining with (9) gives

$$S \;<\; c(aS+a) \;=\; (ac)S + ac.$$

If $c < 1/a$, then $(1-ac)S < ac$, hence $S < \dfrac{ac}{1-ac}$, a constant depending only on a and c. Thus S cannot be arbitrarily large: every computation of M uses at most a fixed constant amount of work space. Equivalently, M uses constant space.

5) Remark

This is one of the standard "lower-than-$\log\log n$" collapse phenomena for space: once the allowed space is below a small enough constant multiple of $\log\log n$, the machine cannot support unboundedly many distinct prefix behaviors, forcing the space usage to be bounded. This explains why the language in Problem 2.8.11(g) sits naturally at the $O(\log\log n)$ threshold.

Problem 2.8.13

Statement. Show that the language of palindromes cannot be decided in less than $\log n$ space. (The argument is a hybrid between Problems 2.8.5 and 2.8.12.)

Solution. Let $\Sigma = \{0,1\}$ and let

$$\mathrm{PAL} = \{\, w \in \Sigma^* : w = w^R \,\}$$

be the language of palindromes, where w^R is the reversal of w. We show that any deterministic Turing machine deciding PAL must use $\Omega(\log n)$ work space on some inputs of length n.

1) Exponentially many distinguishable prefixes

Consider the Myhill–Nerode theorem equivalence for a language $L \subseteq \Sigma^*$:

$$x \equiv_L y \quad\Longleftrightarrow\quad \forall z \in \Sigma^*,\; xz \in L \Leftrightarrow yz \in L.$$

Fix $m \geq 1$. For each $x \in \Sigma^m$, define the suffix $z_x = x^R$ (also length m). Then $xz_x = xx^R \in \text{PAL}$. If $y \in \Sigma^m$ and $y \neq x$, then $yz_x = yx^R \notin \text{PAL}$ (because yx^R is a palindrome iff $y = x$). Hence $x \not\equiv_{\text{PAL}} y$ whenever $x \neq y$. Therefore $\equiv_{\text{PAL}}$ has at least 2^m distinct equivalence classs among length-m prefixes.

2) A space-S decider yields at most $2^{2^{O(S)}}$ prefix types

Let M be a deterministic Turing machine deciding some language L and using at most S work-tape cells on all inputs of length $2m$. As in Problem 2.8.12, define an extended state at space bound S to include: the control state, the contents of the bounded work area of all work tapes, and the work-head positions. Let $N(S)$ be the number of such extended states. Then $N(S) = 2^{O(S)}$.

Fix a boundary immediately to the right of a prefix $x \in \Sigma^m$. Define the *boundary behavior* of x at space bound S as the function

$$B_x : \{1, \ldots, N(S)\} \to \{1, \ldots, N(S)\} \cup \{\text{HALT}\}$$

defined as follows: start M with the input head at the boundary and with extended state σ, and run M until the first time the input head returns to the boundary and is about to move to the right; if this happens, output the resulting extended state σ', otherwise output HALT. This depends only on x, because we stop at the first attempted move from the boundary into the suffix.

Key implication (the same mechanism behind the two-way-to-one-way reduction): if $B_x = B_y$, then $x \equiv_L y$. Intuitively, from the perspective of the suffix, the prefix is a finite-state transducer on extended states at the boundary; if two prefixes induce the same transducer, then substituting one for the other cannot change accept/reject on any continuation.

Hence the number of $\equiv_L$-classes among length-m prefixes is at most the number of possible behaviors B_x. But B_x is a function on a domain of size $N(S)$, so the total number of behaviors is at most

$$(N(S)+1)^{N(S)} \;=\; 2^{O(N(S)\log N(S))} \;=\; 2^{2^{O(S)}}.$$

3) Conclude the $\Omega(\log n)$ lower bound

Apply the previous bound to $L = \text{PAL}$. Part 1 gives at least 2^m distinct $\equiv_{\text{PAL}}$-classes among length-m prefixes. Part 2 gives at most $2^{2^{O(S)}}$ such classes for any decider using space S on length $2m$ inputs. Therefore

$$2^m \;\leq\; 2^{2^{O(S)}}.$$

Taking $\log_2$ twice yields $O(S) \geq \log_2 m - O(1)$, i.e. $S = \Omega(\log m)$. Since $n = 2m$, we obtain $S = \Omega(\log n)$.

Thus the language of palindromes cannot be decided in space $o(\log n)$. (For background on Myhill–Nerode theorem and related equivalences, see [9].)

Problem 2.8.14

Statement. Give a detailed proof of Theorem 2.3. That is, give an explicit mathematical construction of the simulating machine in terms of the simulated machine (assume the latter has one string, besides the input string).

Solution. We recall the theorem (as stated in the book [6]):

> **Theorem 2.3.** Let L be a language in SPACE($f(n)$). Then, for any $\varepsilon > 0$, $L \in$ SPACE($2 + \varepsilon f(n)$).

We give an explicit space compression construction for the case in which the original decider has a read-only input string and a single work string.

Model

Let $M = (Q, \Sigma, \Gamma, \delta, s, \text{yes}, \text{no})$ be a deterministic Turing machine with:

- a read-only input tape over Σ, and
- one read/write work tape over Γ, with a distinguished blank symbol in Γ.

Let $f : \mathbb{N} \to \mathbb{N}$. Assume that on every input x of length n, during its computation M scans at most $f(n)$ distinct work-tape cells.

Fix $\varepsilon > 0$. We construct a machine M_ε that decides the same language as M and, on all inputs of length n, scans at most $2 + \varepsilon f(n)$ work-tape cells.

Construction: pack blocks of work symbols

Let $b = \left\lceil \frac{1}{\varepsilon} \right\rceil$. Then $1/b \le \varepsilon$.

The idea is to store b consecutive work-tape cells of M inside a single work-tape cell of M_ε. Define the packed alphabet $\Gamma' = \Gamma^b$. A symbol of Γ' is a b-tuple $T = (T[0], \ldots, T[b-1])$ of symbols in Γ.

Work-tape encoding. Fix an input x of length n. At any time, let the work tape of M (restricted to the $f(n)$ scanned cells) be the string

$$w = w_0 w_1 \cdots w_{f(n)-1} \in \Gamma^{f(n)}.$$

We encode w as a string $W \in (\Gamma')^*$ on the single work tape of M_ε by grouping into blocks: for each block index $j \ge 0$,

$$W[j] \;=\; \big(w_{jb+0},\; w_{jb+1},\; \ldots,\; w_{jb+(b-1)}\big),$$

where missing positions beyond $f(n)-1$ are padded with blanks. Thus $|W| = \left\lceil \frac{f(n)}{b} \right\rceil$ packed cells suffice.

Head position encoding. If M's work head is currently at position $i \in \{0, \ldots, f(n)-1\}$, write $i = jb + r$ with $0 \le r < b$. Then M_ε's work head points to packed cell j, and the offset r is stored in the finite control. Concretely, the state set of M_ε is $Q \times \{0, 1, \ldots, b-1\}$, plus accepting/rejecting states.

Explicit transition function of M_ε

Let the current state of M_ε be (q, r), where $q \in Q$ is the simulated state of M and r is the offset. Let the current input symbol be $a \in \Sigma \cup \{\text{blank}\}$. Let the current packed work symbol be $T = (T[0], \ldots, T[b-1]) \in \Gamma'$, so the simulated work symbol is $T[r]$.

Assume δ has the form

$$\delta(q, a, T[r]) \;=\; (q', a', \gamma', D_{\text{in}}, D_{\text{wk}}),$$

where $D_{\text{in}}, D_{\text{wk}} \in \{\leftarrow, \rightarrow, \text{stay}\}$ are the input/work head moves.

Then M_ε performs:

1. **Write on the packed work tape.** Let

$$T' = (T[0], \ldots, T[r-1], \gamma', T[r+1], \ldots, T[b-1]).$$

   Write T' in the current packed work cell.

2. **Move the input head.** Move according to D_{in}.

3. **Update (q, r) and move the packed work head.**

   - If $D_{\text{wk}} = \text{stay}$: keep r and do not move the packed work head.
   - If $D_{\text{wk}} = \rightarrow$:
     - if $r < b-1$, set $r := r+1$ (packed head stays);
     - if $r = b-1$, set $r := 0$ and move the packed head right by one cell.
   - If $D_{\text{wk}} = \leftarrow$:
     - if $r > 0$, set $r := r-1$ (packed head stays);
     - if $r = 0$, set $r := b-1$ and move the packed head left by one cell.

4. **Update the simulated control state.** Set $q := q'$. Map $q' = \text{yes}$ and $q' = \text{no}$ to the corresponding halting states of M_ε.

This defines the transition function of M_ε purely in terms of δ. Correctness follows by induction on the number of simulated steps: the packed tape and the offset encode exactly the work tape and work-head position of M, and the update rule applies exactly the same local rewrite and head movement as δ.

Space bound

On inputs of length n, M scans at most $f(n)$ work cells. The encoding uses at most $\lceil f(n)/b \rceil$ packed cells. Reserve a constant number of extra packed cells for bookkeeping; absorb this into 2.

Using $\lceil u \rceil \le u + 1$ and $1/b \le \varepsilon$,

$$\left\lceil \frac{f(n)}{b} \right\rceil \;\le\; \frac{f(n)}{b} + 1 \;\le\; \varepsilon f(n) + 1.$$

Hence M_ε scans at most $2 + \varepsilon f(n)$ packed work cells on inputs of length n, proving Theorem 2.3 in the one-work-tape case.

Remark

If the original machine has a constant number of work tapes, one can first combine them into one tape using tracks (a constant-factor blowup) and then apply the same packing argument, preserving the statement $\mathrm{SPACE}(f(n)) \subseteq \mathrm{SPACE}(2 + \varepsilon f(n))$.

Problem 2.8.16

Statement. Modify the nondeterministic Turing machine for reachability in Example 2.10 so that it uses the same amount of space, but always halts.

It turns out that all space-bounded computation machines, even those that use very little space and that they cannot count steps, can be modified so that they always halt; for the tricky construction see [8].

Solution. We describe the standard "clocking by configuration bound" modification, specialized to the reachability machine of Example 2.10.

1) The nondeterministic reachability machine and why it may not halt

In Example 2.10, the nondeterministic Turing machine decides reachability (reachability in a configuration graph) by simulating a walk in the graph: it starts from the source configuration c_s, repeatedly nondeterministically chooses a legal successor configuration, updates the current configuration, and accepts if it ever reaches the target configuration c_t. If c_t is not reachable from c_s, some computation branches may loop forever by cycling among configurations.

2) Key observation: a space bound implies finitely many configurations

Fix an input instance of length n and suppose the simulated machine is guaranteed to use at most S work cells on inputs of length n. A full configuration of the simulated machine can be encoded using:

- $O(1)$ bits for the finite control state,
- $O(\log n)$ bits for the input-head position, and
- $O(S)$ bits for the work tape contents and work-head position(s).

Hence the number of distinct configurations that can ever appear under the space bound S is at most

$$N \;\le\; 2^{O(S+\log n)} \;=\; n^{O(1)} \cdot 2^{O(S)}. \tag{10}$$

In particular, if $S \ge \log n$ (the usual regime for nontrivial space bounds), then $\log N = O(S)$.

3) There is always a short accepting path if any path exists

If c_t is reachable from c_s in the configuration graph, then there exists a simple path from c_s to c_t that never repeats a configuration. Such a path has length strictly less than N, because it visits at most one new configuration per step and there are at most N configurations.

4) The halting modification

We modify the reachability nondeterministic Turing machine as follows. On input of length n with space bound S, let $N(n, S)$ be a fixed explicit upper bound satisfying (10) (for example $N(n, S) = 2^{c(S+\lceil \log n \rceil)}$ for a large enough constant c determined by the encoding).

The new machine M' maintains:

- the current simulated configuration c (as in Example 2.10), and
- a counter T initialized to $N(n, S)$, stored in binary.

Then M' repeats:

1. If $c = c_t$, **accept**.
2. If $T = 0$, **reject**.
3. Otherwise, nondeterministically choose a legal successor configuration c' of c, set $c := c'$, and decrement T.

This machine halts on every branch after at most $N(n, S) + 1$ iterations.

5) Space usage

Relative to the machine in Example 2.10, the only additional work tape storage is the counter T, which needs $O(\log N)$ bits. By (10), $\log N = O(S + \log n)$. In the standard setting where the reachability simulation already stores a full configuration (including an input-head position) and $S \geq \log n$, this is $O(S)$ extra bits, i.e., only a constant-factor increase and the same asymptotic space bound. If $S < \log n$, then either the model for the simulated configuration already forces an $O(\log n)$ term, or the computation collapses to constant space (cf. Problem 2.8.12), in which case halting can be enforced by an absolute constant-space wrapper; in all cases, the modification does not increase the space bound beyond that of the original reachability construction.

6) Correctness

- If c_t is reachable from c_s, then there exists a simple path of length $< N$. The nondeterministic choices can follow this path, so some branch reaches c_t before the counter expires and accepts.
- If c_t is not reachable from c_s, then no branch can ever reach c_t. Every branch either exhausts the counter and rejects, or would otherwise loop; but looping is prevented by the counter, so all branches reject.

Thus M' decides reachability, always halts, and uses the same asymptotic amount of space as the machine in Example 2.10.

Problem 2.8.17

Statement. Show that any language decided by a k-string nondeterministic Turing machine within time $f(n)$ can be decided by a 2-string nondeterministic Turing machine also within time $f(n)$. (Discovering the simple solution is an excellent exercise for understanding nondeterminism. Compare with Theorem 2.1 and Problem 2.8.9 for deterministic machines.)

Needless to say, any k-string nondeterministic Turing machine can be simulated by a single-string nondeterministic Turing machine with a quadratic loss of efficiency, exactly as with deterministic machines.

Solution. Let M be a k-string nondeterministic Turing machine deciding L in time $t = f(n)$ on inputs of length n. Write k for the number of work strings (besides the read-only input string).

(a) **From k strings to 2 strings in the same time bound (up to a constant factor).**

The key point is that *time-bounded nondeterminism is about the existence of an accepting computation path of length $\leq t$.* A 2-string nondeterministic Turing machine can therefore proceed by *guessing* such a path and verifying it step-by-step while maintaining only a constant amount of additional bookkeeping per simulated step.

Formally, define the instantaneous description (configuration) of M at any moment to consist of: (i) the control state, (ii) the input-head position, (iii) for each of the k work strings: its head position and the symbol stored in each work cell ever visited so far. Because M runs for at most t steps, across the entire computation it can visit at most t new work cells *per string*, hence at most kt distinct work cells in total.

Construct a 2-string nondeterministic Turing machine M_2 that operates as follows on input x of length n:

1. Initialize an encoding of the current configuration of M on work string #1 (the second string of M_2); initially all work strings are blank and all work heads are at their start cells. Also initialize a step counter $s := 0$ (in binary) as part of the same encoding.
2. Repeat while $s < t$:
   (i) From the encoding, read the current control state of M, the input-head position, and the k symbols currently scanned by the k work heads.
   (ii) Nondeterministically choose one transition of M consistent with these scanned symbols (this is exactly where nondeterminism is used).
   (iii) Update the configuration encoding accordingly: change the control state; update the input-head position; and for each work string, update the scanned symbol (write) and head move $(\leftarrow, \rightarrow, \text{stay})$. If a work head moves onto a work cell not yet represented in the encoding, extend the encoding by adding that cell as blank.
   (iv) If the simulated state is accepting, **accept**.
   (v) Increment s.
3. If the loop finishes without accepting, **reject**.

Correctness. If $x \in L$, then M has an accepting computation path of length $\leq t$; along that path, the nondeterministic choices of M_2 can select exactly the same transitions, and all updates are consistent by construction, so M_2 accepts. If $x \notin L$, then no accepting computation path of length $\leq t$ exists, hence every sequence of nondeterministic choices yields either a rejecting/halting path or fails to reach an accepting state within t steps, so M_2 rejects.

Running time. Each simulated step performs only constant local updates to the configuration encoding plus counter maintenance, and k is a constant of the machine model. Thus the simulation runs in $O(t)$ time; by standard constant-factor normalization of the time measure (absorbing fixed overhead into f), this is "within time $f(n)$" as required. See also [10] for robustness of time-bounded nondeterministic models.

(b) **From k strings to 1 string: quadratic slowdown.**

By Theorem 2.1 (the standard single-string simulation of a multi-string machine), any 2-string nondeterministic Turing machine running in time t can be simulated by a single-string nondeterministic Turing machine in time $O(t^2)$. Applying this to the machine M_2 from part (a) yields a single-string nondeterministic machine deciding L in time $O(f(n)^2)$, exactly as in the deterministic case.

Problem 2.8.18

Statement.

(a) Show that a nondeterministic one-way finite automaton (recall 2.8.11) can be simulated by a deterministic one, with possibly exponentially many states. (Construct a deterministic automaton each state of which is a set of states of the given nondeterministic one.)

(b) Show that the exponential growth in (a) is inherent. (Consider the language $L = \{x\sigma : x \in (\Sigma \setminus \{\sigma\})^*\}$, where the alphabet Σ has n symbols.)

Solution.

(a) **Subset construction (nondeterministic finite automaton (NFA) $\to$ deterministic finite automaton (DFA)).** Let $N = (Q, \Sigma, \delta, q_0, F)$ be a (one-way) nondeterministic finite automaton (NFA). Here $\delta(q, a) \subseteq Q$ is the set of possible next states from q on input symbol a. (If ε-moves are allowed, replace the start set below by the ε-closure of q_0.)

Define a deterministic finite automaton (DFA) $D = (Q', \Sigma, \delta', q_0', F')$ by:

- $Q' = 2^Q$ (the set of all subsets of Q);
- $q_0' = \{q_0\}$;
- for $S \subseteq Q$ and $a \in \Sigma$,
$$\delta'(S, a) = \bigcup_{q \in S} \delta(q, a); \tag{11}$$
- $F' = \{S \subseteq Q : S \cap F \neq \emptyset\}$.

Let $\hat{\delta}$ and $\hat{\delta}'$ be the extended transition functions of N and D, respectively. We prove by induction on $|w|$ that for every $w \in \Sigma^*$,

$$\hat{\delta}'(\{q_0\}, w) = \{\, q \in Q : q \text{ is reachable in } N \text{ from } q_0 \text{ by reading } w \,\}. \tag{12}$$

Base case. For $w = \epsilon$, $\hat{\delta}'(\{q_0\}, \epsilon) = \{q_0\}$, which matches the right-hand side.

Inductive step. Write $w = ua$ with $a \in \Sigma$. Assuming (12) holds for u, we have

$$\hat{\delta}'(\{q_0\}, ua) = \delta'(\hat{\delta}'(\{q_0\}, u), a) = \bigcup_{q \in \hat{\delta}'(\{q_0\}, u)} \delta(q, a)$$

by (11). By the induction hypothesis, $\hat{\delta}'(\{q_0\}, u)$ is exactly the set of states reachable in N after reading u, so the union above is exactly the set of states reachable after reading ua. Thus (12) holds for ua as well.

Therefore D accepts w iff $\hat{\delta}'(\{q_0\}, w) \cap F \neq \emptyset$, which is equivalent to N having at least one accepting path on w. Hence $L(D) = L(N)$, and $|Q'| \leq 2^{|Q|}$. This is the classical subset construction (see [3, 10]).

(b) **The exponential blow-up can be necessary.** To avoid ambiguity in the hint's notation, interpret the language as

$$L \;=\; \{\, x\sigma : \sigma \in \Sigma,\; x \in (\Sigma \setminus \{\sigma\})^* \,\}, \tag{13}$$

i.e., $w \in L$ iff the last symbol of w does not appear earlier in w. Assume $|\Sigma| = n$.

An nondeterministic finite automaton (NFA) of size $O(n)$ for L. Build an nondeterministic finite automaton (NFA) N that nondeterministically guesses the final symbol σ. Use states: one start state q_{start}, one accepting state q_{acc}, and for each $\sigma \in \Sigma$ a state q_σ. From q_{start}, add an ε-transition to every q_σ. In state q_σ, on any input symbol $a \neq \sigma$ stay in q_σ; on input σ go to q_{acc}. From q_{acc}, on any further input symbol move to a dead sink; acceptance occurs only if the input ends immediately after reading σ. This nondeterministic finite automaton (NFA) has $n + O(1)$ states and recognizes L.

Any equivalent deterministic finite automaton (DFA) needs at least 2^n states. We use the Myhill–Nerode theorem theorem (Problem 2.8.11(c)). For each subset $A \subseteq \Sigma$, choose a string x_A that lists exactly the symbols of A once each (in any fixed order), and no other symbols. We claim that if $A \neq B$ then $x_A \not\equiv_L x_B$.

Pick $a \in A \triangle B$ and assume $a \in A \setminus B$. Let $z = a$. Then:

- $x_A a \notin L$, because the last symbol a already appears earlier in x_A;
- $x_B a \in L$, because $a \notin B$ implies $x_B \in (\Sigma \setminus \{a\})^*$, hence $x_B a \in (\Sigma \setminus \{a\})^* a \subseteq L$.

Thus x_A and x_B are distinguished by the continuation $z = a$, so they lie in different $\equiv_L$-classes. Therefore $\equiv_L$ has at least 2^n equivalence classes, one per subset $A \subseteq \Sigma$, and any deterministic finite automaton (DFA) for L needs at least 2^n states.

Combining the $O(n)$-state nondeterministic finite automaton (NFA) with the 2^n deterministic finite automaton (DFA) lower bound shows that the exponential blow-up in part (a) is inherent in general.

Glossary

amortized analysis A method for bounding the average cost per operation over a sequence, by charging occasional expensive operations to many cheap steps. 8

arithmetic progression Here: the concatenation $b(1);\ b(2);\ \cdots;\ b(n)$ of successive binary integers separated by a delimiter. 11

Boolean circuit A directed acyclic graph of logic gates computing a Boolean function; a more "static" computation model than a Turing machine. 8

configuration An instantaneous description of a Turing machine: the current state, the full tape contents, and the head positions. 2, 3, 6, 19

configuration graph A directed graph whose vertices are configurations of a machine on a fixed input and whose edges represent legal single-step transitions. 19

cursor The tape head position; for a two-dimensional tape it is a pair of coordinates $(x, y) \in \mathbb{Z}^2$. 3, 8

delete Operation that removes the symbol at the current head position (if any) and shifts the suffix one cell to the left. 5, 7

deterministic finite automaton (DFA) A finite automaton with exactly one next state for each state and input symbol. 22, 23

edit operation A primitive tape update that can write, insert, or delete a symbol at the head position. 5

equivalence class For an equivalence relation $\sim$, the set $[x] = \{y : y \sim x\}$. 10, 16

equivalence relation A relation that is reflexive, symmetric, and transitive. 10, 11

extended state A machine state augmented with the contents of the bounded work tapes and the work-head positions; essentially a work-space-bounded configuration with the input head position abstracted out. 14, 16

insert Operation that inserts a symbol at the current head position and shifts the suffix one cell to the right. 5, 7

instantaneous description A complete snapshot of a Turing machine at a moment in time: state, head positions, and work-tape contents within the space used so far. 21

k-string nondeterministic Turing machine A nondeterministic Turing machine with a read-only input string and k additional work strings (tapes), each with its own head. 20, 21

k-tape Turing machine A Turing machine equipped with k tapes, each with its own head; one global transition depends on the k scanned symbols. 2, 7, 8

Myhill–Nerode theorem A language is regular iff the right congruence $\equiv_L$ has finitely many equivalence classes. 15, 16, 23

nondeterminism A computation model in which a transition may branch into multiple possible moves; acceptance means that at least one branch accepts. 20, 21

nondeterministic finite automaton (NFA) A finite automaton that may have multiple possible next states for a given state and input symbol; it accepts if some path ends in an accepting state. 22, 23

nondeterministic Turing machine A Turing machine whose transition function may offer multiple possible moves; it accepts if at least one computation branch accepts. 19, 20

oblivious Turing machine A Turing machine whose head positions at time t depend only on t and the input length n, not on the specific input. 8, 10

one-way finite automaton A finite automaton that reads the input strictly left-to-right (only $\rightarrow$ moves). 10, 11, 12

palindrome A string that equals its reversal. 15, 16

pumping lemma A property of regular languages guaranteeing that sufficiently long strings can be decomposed and "pumped". 12

reachability Given a directed graph and two vertices s, t, decide whether there is a directed path from s to t; here, reachability in a configuration graph. 19, 20

regular language A language recognized by a one-way finite automaton (equivalently, by a regular expression). 10, 11, 12, 13

simple path A path in a graph that does not repeat vertices. 19

single-string nondeterministic Turing machine A nondeterministic Turing machine with a read-only input string and a single additional work string. 20, 22

single-tape Turing machine A Turing machine with one work tape and one head; it can simulate multitape machines with polynomial overhead. 2

space-bounded computation A computation whose work-tape usage is bounded by a function $S(n)$ of the input length. 19

space compression A transformation that reduces the work-space usage of a Turing machine by a constant factor by packing multiple tape symbols into one. 17

subset construction The powerset transformation that converts an NFA with state set Q into a DFA with states 2^Q recognizing the same language. 22

tape alphabet The finite set Γ of symbols a Turing machine may write on its tape, including a designated blank symbol. 2

track In a packed encoding of several tapes on one tape, a track is one component of a tuple stored in each cell, representing one simulated tape cell. 2

transition function The mapping δ that determines, from the current state and scanned symbol(s), the next state, symbol(s) written, and head move(s). 3, 5

Turing machine A finite-control abstract machine with an unbounded tape and a read/write head; in this chapter, the tape has a left-end marker and a blank symbol. 2, 3, 5, 6, 7, 8, 10, 13, 14, 15, 16, 17, 24

two-way finite automaton A finite automaton whose input head may move left and right on a read-only input (typically with endmarkers). 10, 11, 12

two-dimensional tape A tape indexed by $\mathbb{Z} \times \mathbb{Z}$ with a head that can move left/right/up/down. 3

2-string nondeterministic Turing machine A nondeterministic Turing machine with a read-only input string and two additional strings (tapes) total, i.e., one work string besides the input. 20, 21, 22

References

[1] Juris Hartmanis, Philip M. Lewis II, and Richard E. Stearns. Hierarchies of memory-limited computations. *Proceedings of the 6th Annual IEEE Symposium on Switching Circuit Theory and Logical Design*, pages 179–190, 1965.

[2] Fred C. Hennie and Richard E. Stearns. Two-tape simulation of multitape Turing machines. *Journal of the ACM*, 13(4):533–546, 1966.

[3] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison–Wesley, 1979.

[4] John Myhill. Finite automata and the representation of events. *WADC Technical Report*, 1957.

[5] Anil Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.

[6] Christos H. Papadimitriou. *Computational Complexity.* Addison–Wesley, 1994.

[7] J. C. Shepherdson. The reduction of two-way automata to one-way automata. *IBM Journal of Research and Development*, 3:198–200, 1959.

[8] Michael Sipser. Halting space-bounded computations. *Theoretical Computer Science*, 10:335–338, 1980.

[9] Michael Sipser. *Introduction to the Theory of Computation.* Cengage Learning, 3 edition, 2012.

[10] Michael Sipser. *Introduction to the Theory of Computation.* Cengage Learning, 3 edition, 2013.