Chapter 1

Efficient Computation

1.1 Problems and Algorithms

A *computation* receives an input, executes a sequence of elementary steps, and produces an output. To unpack this concept, let's separate two questions:

(i) What is the computation supposed to accomplish?
(ii) How does the computation accomplish it?

An answer to *(i)* is a *computational problem*, or *problem* for short: a relationship between inputs and outputs, specifying what the output should be for each possible input. An answer to *(ii)* is an *algorithm*: instructions for going from input to output by elementary steps that could be performed by a computer program or patient human. A problem is a goal, and an algorithm is a way to achieve that goal. This is like "interface versus implementation" in programming. A problem may have multiple algorithms that are *correct*, meaning they *solve* the problem: For every possible input, the algorithm produces output as specified by the problem.

This book adheres to some stylistic conventions: Names of computational problems are written in Small Capitals. Problems are in rectangles. Algorithms are in rounded rectangles.

Problems and algorithms are the central characters in the story of computational complexity. Let's introduce some running examples.

1.1.1 Multiplying Numbers

Multiplication	(1.1)
Input: Nonnegative integers x and y	
Output: The product $x \cdot y$	

A "brute force" algorithm solves this problem, assuming we know how to add numbers:

Brute force algorithm for Multiplication (1.2)

```
initialize z ← 0
repeat y times:
    update z ← z + x
output z
```

Pseudocode in this book uses "←" for variable assignments, "=" for definitions and comparisons, and Python-style indentation for bodies of loops, conditionals, and subroutines.

In primary school, we learned a "better" algorithm for when x and y are large. Let's review this using base two (binary) instead of the traditional base ten. Let $y = \cdots y_2 y_1 y_0$ in binary, so

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

$y = \sum_j (y_j \cdot 2^j)$, where $y_j \in \{0,1\}$ for each index j. The primary-school algorithm exploits the distributive law:

$$x \cdot y \;=\; x \cdot \textstyle\sum_j (y_j \cdot 2^j) \;=\; \sum_j (x \cdot y_j \cdot 2^j)$$

The algorithm sums the rows of a table where the jth row (the top row being the 0th) is $x \cdot y_j \cdot 2^j$ in binary, which is obtained by shifting x left by j many bit positions to get $x \cdot 2^j$ and multiplying it by the individual bit y_j of y. Denote the output $x \cdot y$ as $z = \sum_i (z_i \cdot 2^i)$.

Example. Suppose x is twenty-seven and y is twenty-two. In base ten, $x = 27$ and $y = 22$, while in base two:

$$\begin{aligned} x = 11011 \quad \text{since} \quad 27 \;&=\; 16 + 8 \quad\;\; + 2 + 1 \;=\; 1\cdot 2^4 + 1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 \\ y = 10110 \quad \text{since} \quad 22 \;&=\; 16 \quad + 4 + 2 \quad\;\; =\; 1\cdot 2^4 + 0\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0 \end{aligned}$$

Figure 1.1's left side computes $27 \cdot 22 = 594$ using the primary-school algorithm. The 0th and 3rd rows are "zeroed out" since $y_0 = y_3 = 0$. The 1st, 2nd, and 4th rows are shifted copies of $x = 11011$ since $y_1 = y_2 = y_4 = 1$. Blanks are equivalent to 0s. The usual addition algorithm sums the rows by processing the columns from right to left. Each individual column's bits are summed to produce a bit of the output and a "carry" value that feeds into the next column. Figure 1.1's right side shows how the output and carries are computed. The output is:

$$z = 1001010010 \quad \text{since} \quad 594 \;=\; 512 + 64 + 16 + 2 \;=\; 2^9 + 2^6 + 2^4 + 2^1 \qquad \blacklozenge$$

We now describe this algorithm in general. Assume x and y are both n bits long (possibly with leading 0s), so $x < 2^n$ and $y < 2^n$, which implies $x \cdot y < 2^{2n}$, so $x \cdot y$ is $2n$ bits long. The pseudocode uses three variables: i is the current column's index, j is the current row's index (as we go down the entire current column, treating blanks as 0s), and c is the carry that gets updated from one column to the next.

Primary-school algorithm for Multiplication (1.3)

view the input as binary numbers $x = x_{n-1} \cdots x_1 x_0$ and $y = y_{n-1} \cdots y_1 y_0$
initialize $c \leftarrow 0$
for $i \leftarrow 0, 1, \ldots, 2n-1$:
    for $j \leftarrow 0, 1, \ldots, n-1$:
        update $c \leftarrow c + (x_{i-j} \cdot y_j)$ (where $x_{i-j} = 0$ if $i - j < 0$ or $i - j \geq n$)
    let $z_i \leftarrow c \bmod 2$
    update $c \leftarrow \lfloor c/2 \rfloor$
output the binary number $z = z_{2n-1} \cdots z_1 z_0$

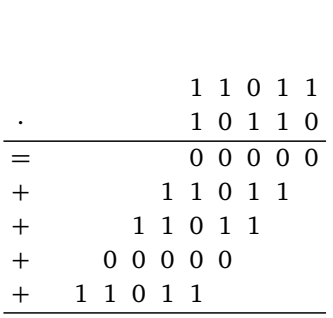
How is this better than the brute force algorithm? Though the primary-school algorithm is more complicated to describe, it's more *efficient* to run, as we discuss in §1.2.

A *decision problem*'s output is always a single bit. Such a problem asks a yes/no or true/false question, where 1 means yes/true and 0 means no/false. Multiplication isn't a decision problem, because the output has multiple bits. Here's a decision problem version of Multiplication:

Multiplication Decision (1.4)

Input: Nonnegative integers x, y, and i
Output: The ith bit of $x \cdot y$ in binary

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003



column index i	carry in	+	bits of column		=	carry out z_i
0			(0	)	=	0 0
1	0	+	(0 + 1	)	=	0 1
2	0	+	(0 + 1 + 1	)	=	1 0
3	1	+	(0 + 0 + 1 + 0	)	=	1 0
4	1	+	(0 + 1 + 0 + 0 + 1)		=	1 1
5	1	+	(1 + 1 + 0 + 1)		=	1 0 0
6	1 0	+	(1 + 0 + 0)		=	1 1
7	1	+	(0 + 1)		=	1 0
8	1	+	(1)		=	1 0
9	1				=	1

Figure 1.1: Example of running the primary-school algorithm for Multiplication

To phrase the output as a yes/no question, we could write "Does the ith bit of $x \cdot y$ equal 1?" (The 0th bit is the least significant.) It's tempting to think, "This asks for less than Multiplication does, so the primary-school algorithm already solves this." Strictly speaking, it doesn't solve this because it doesn't even output the right number of bits. We can modify the algorithm to solve Multiplication Decision by outputting just z_i rather than all of z (or outputting 0 if $i \geq 2n$). Technically, that small modification yields a different algorithm. This may seem pedantic, but such distinctions matter in computational complexity theory. We should be careful and precise when discussing problems and algorithms.

We'll revisit the primary-school algorithm for Multiplication, but next let's introduce our other running example.

1.1.2 Reachability in Graphs

Many problems involve paths in graphs. Navigation systems suggest how to get from one location to another. Computer networks route packets from one host to another. Let's pare these goals down to a decision problem that asks whether it's even possible to go from here to there.

In a directed graph, node t is *reachable* from node s when there exists a walk from s to t. There exists a walk from s to t (which may repeat nodes and edges) iff there exists a path from s to t (which doesn't repeat nodes or edges), by Lemma A.5.

Directed Reachability	(1.5)
Input: Directed graph G, nodes s and t	
Output: Is t reachable from s in G?	

Assume G has no multi-edges or self-loops, which would be irrelevant.

Every path has length (number of edges) at most $n-1$, where n is the number of nodes in G. A "brute force" algorithm tries all potential paths of length up to $n-1$ and checks whether any is actually a path from s to t:

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

Brute force algorithm for Directed Reachability (1.6)

```
for ℓ ← 0, 1, ..., n − 1:
    for every tuple of distinct nodes (v_1, v_2, ..., v_{ℓ+1}) with v_1 = s and v_{ℓ+1} = t:
        if G has an edge from v_i to v_{i+1} for every i ∈ {1, 2, ..., ℓ}: halt and output "yes"
output "no"
```

We design a better algorithm with this insight: If we've deduced that a node u is reachable from s, and u has an edge to node v, then we deduce that v is also reachable from s, via a path from s to u followed by the edge (u, v). Upon deducing that u is reachable from s, we mark u as "discovered," and then eventually we "process" u by considering each outgoing edge (u, v) and marking v as discovered (if it wasn't already), and later we process v, and so on. This kind of algorithm explores outward from s, discovering all reachable nodes. If t is reachable from s via a path $s \to v_2 \to v_3 \to \cdots \to t$, then t eventually gets discovered: v_2 gets discovered when we process s, and v_3 gets discovered when we process v_2 (if not sooner), and so on. Conversely, if t gets discovered, t must be reachable from s.

To fully specify the algorithm, we must choose which node to process in each iteration. We use a data structure to remember which nodes have been discovered but not yet processed. The *breadth-first search (BFS)* algorithm uses a *queue* data structure, which supports *enqueueing* (inserting a node) and *dequeueing* (removing the oldest remaining node). In short, enqueue = discover, and dequeue = process. After v is discovered, v waits its turn while we process nodes that have been waiting longer.

Breadth-first search algorithm for Directed Reachability (1.7)

```
if s = t: halt and output "yes"
mark s as discovered and initialize the queue with s
while the queue is not empty:
    dequeue a node u
    for each edge (u, v):
        if v is undiscovered:
            if v = t: halt and output "yes"
            mark v as discovered and enqueue v
output "no"
```

Example. We demonstrate BFS on the input in Figure 1.2. In the first iteration, we dequeue and process node 4, whose outgoing edges lead to the undiscovered nodes 1 and 2. We mark 1 and 2 as discovered and enqueue them for later processing. It doesn't matter which order they're enqueued in, but say we enqueue 1 first, so we dequeue and process 1 in the next iteration. Since 1's only outgoing edge leads to the already-discovered node 2, nothing needs to be done. (We don't enqueue 2 again, since it should only get processed once.) Now the queue only contains 2, so we dequeue and process 2, whose outgoing edges lead to 4 (which was already discovered) and to 5. After marking and enqueueing 5, we promptly dequeue and process 5. The outgoing edges lead to 1 (already discovered) and 3 (which is t) and 4 (already discovered). We encountered t, so we celebrate without bothering to mark and enqueue it. ♦

Directed Reachability may seem inadequate because the output doesn't say how to get from s to t (if t is indeed reachable from s). The following variant is more practical:

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

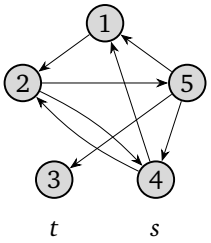


Figure 1.2: Example input for Directed Reachability

Directed Reachability Search	(1.8)
Input: Directed graph G, nodes s and t	
Output: A path from s to t in G, or report that no such path exists	

This is a *search problem*: Each input has an associated set of acceptable "solutions," and the goal is to output any acceptable solution (it doesn't matter which one) or report that the set is empty.

We can solve Directed Reachability Search with a trivial modification of brute force or with an augmented version of BFS called *extended BFS*: When a node v (other than s) gets discovered, we record which node u "caused" the discovery of v, and we deem u the "predecessor" of v. When t is encountered, we record its predecessor and begin a second phase to reconstruct a path from s to t in reverse: If v_2 is t's predecessor, and v_3 is v_2's predecessor, and so on, and s is v_ℓ's predecessor, then we output the path $s \to v_\ell \to \cdots \to v_2 \to t$.

Example. On our earlier example, extended BFS outputs the path $4 \to 2 \to 5 \to 3$. Another acceptable solution is $4 \to 1 \to 2 \to 5 \to 3$, but any algorithm only needs to find one of them. ♦

Extended BFS also solves a more stringent version of Directed Reachability Search where the output must be a shortest path (with as few edges as possible), rather than any old path, from s to t. We call this problem Directed Shortest Path. This is an *optimization problem*, which is a type of search problem where the goal is to output a "best" solution.

Extended BFS finds a shortest path because the set of discovered nodes is like a wave emanating steadily from s: Define any node v's *distance* (from s) as the length of a shortest path from s to v (or ∞ if v is unreachable from s). In the first iteration, all nodes at distance 1 are discovered, with s as their predecessor. While processing nodes at distance 1, the algorithm discovers exactly the nodes at distance 2: Every node discovered by a node at distance 1 has distance 2 (no more, and no less since otherwise it would've been discovered earlier), and conversely, every node at distance 2 is discovered by a node at distance 1 (since it has an edge from a node at distance 1 but no edge from s). In general, while dequeueing and processing nodes at distance $d-1$, the algorithm discovers and enqueues exactly the nodes at distance d. The queue's "first in, first out" property ensures that all nodes at distance $d-1$ get processed before any node at distance d does. If t has distance d, then t's predecessor has distance $d-1$, and that node's predecessor has distance $d-2$, and so on, so the path constructed by extended BFS indeed has length d.

1.2 Efficiency

How should we compare different algorithms for the same problem? We may prefer an algorithm that's easy to understand, implement, analyze, modify, or generalize. Computational complexity

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

theory is about *efficiency*: An algorithm is more efficient than another if it uses less of some computational *resource*. The most fundamental resources are *time* and *space*.

1.2.1 Time

An algorithm's *running time* on a particular input is the number of elementary steps the algorithm executes before it halts. We discuss the precise meaning of "elementary step" in §1.4, but an informal intuition suffices for now. In practice, we care about "wall clock time" (seconds, minutes, hours), which depends on various factors extrinsic to the algorithm, such as processor speed. The number of elementary steps is a good measure of running time because it's approximately proportional to wall clock time but is intrinsic to the algorithm.

Let's examine the running times of algorithms for Multiplication. Our first instinct might be, "In a programming language such as Java, we can multiply integers `x` and `y` with a single instruction `x * y`, so the running time is 1." That's fine if `x` and `y` are `int` or `long` values, which fit in single machine words and can be multiplied with a hardware instruction. In computational complexity, we study how running time depends on the input's "size." We discuss the general meaning of "size" in §1.3, but for Multiplication it's the number of bits in x and y. Suppose x and y are both n bits. We impose no limit on n, so x and y could be arbitrarily large (a hundred or a million bits, or more). This is what Java's `BigInteger` class is for.

The brute force Multiplication algorithm adds x to a running sum, y times. A loop's running time is at most "number of iterations" times "steps per iteration." We won't even worry about how many steps the addition takes inside the loop, because that's "a drop in the bucket" compared to the number of iterations. Since y's value is as large as 2^n, the loop could have 2^n iterations—an exponential dependence on n. What if $n = 300$? Though 300 bits is a pitifully small amount of data, 2^{300} is a staggeringly huge number—greater than the number of atoms in the known universe. With a billion steps per second, the wall clock time would be far greater than the universe's age since the Big Bang. In practice, cryptography uses huge number arithmetic: Multiplying thousand-bit numbers (as large as 2^{1000} in value) is common, so there better be a faster algorithm!

In the primary-school Multiplication algorithm, the outer loop has $2n$ iterations, and the inner loop has n iterations. The maximum number of steps inside the inner loop doesn't depend on n: subtraction $i - j$, comparisons $i - j < 0$ and $i - j \geq n$, lookups x_{i-j} and y_j, multiplication $x_{i-j} \cdot y_j$, addition $c + (x_{i-j} \cdot y_j)$, assignment to c, increment to j, and comparison $j \geq n$ to see whether the inner loop is done. This is perhaps 10 steps. The outer loop has perhaps 5 steps that aren't part of the inner loop. So the running time is at most $2n \cdot (n \cdot 10 + 5) = 20n^2 + 10n$ steps—a polynomial dependence on n. With $n = 300$ and a billion steps per second, the wall clock time would be a fraction of a second. Running the primary-school algorithm by hand, with a few steps per minute, we'd drastically outperform a computer running brute force.

The preceding paragraph was too much detail. Using big-O notation, we summarize the running time $20n^2 + 10n = O(n^2)$. At a glance, the algorithm has $O(n)$ outer iterations, $O(n)$ inner iterations, and $O(1)$ steps per iteration; multiplying these gives $O(n^2)$. Can we improve the running time? Skipping the table's blank entries decreases the running time by perhaps a factor of 2, but there would still be n^2 nonblanks, so big-O wouldn't be affected. A factor 2 speedup can be a big deal in practice, but it can also be eclipsed by the effect of larger inputs: Doubling the input size could increase a $O(n^2)$ running time by a factor of 4. Big-O captures how running time scales with the input size. The relative importance of hidden constant factors and lower-order terms diminishes as the input grows. Even with a constant factor in the billions, a $O(n^2)$-time

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

MULTIPLICATION algorithm would vastly outperform brute force on 300-bit numbers.

In summary: On large inputs, an algorithm's big-O running time matters much more than the execution speed or hidden constant factor.

For DIRECTED REACHABILITY, suppose the graph has n nodes and m edges, so two parameters describe the input's size. Brute force tries all tuples of distinct nodes to see if any is a path from s to t. There are more than $n!$ (n factorial) possibilities to try, which is even worse than 2^n. Brute force is totally impractical for graphs with hundreds of nodes.

Breadth-first search's running time depends on how the graph is represented and how the data structures are implemented. One common graph representation is the *adjacency matrix*, which lets algorithms look up whether (u, v) is an edge (for any nodes u and v) in $O(1)$ steps. Here and in most of this book, the *adjacency list* representation is more convenient: Each node u has a list of all nodes to which u has an edge. This enables a simple implementation of "for each edge (u, v)" when processing u, since finding the next outgoing edge only takes $O(1)$ steps. What about the queue and the set of all discovered nodes? Using simple arrays, we can implement these data structures so the following operations only take $O(1)$ steps: enqueueing, dequeueing, checking whether a node is undiscovered, and marking a node as discovered.

Let's analyze BFS's running time. For the outer loop, "number of iterations" times "steps per iteration" doesn't give the most detailed bound: There are up to n iterations (each node is processed at most once), and the number of steps in an iteration is proportional to the outdegree of the node u being processed. Since $\text{outdeg}(u) < n$, the total running time is $O(n) \cdot O(n) \cdot O(1) = O(n^2)$. We do a more careful analysis: Different outer iterations contribute different amounts to the running time. The outer iteration for processing u contributes at most $c \cdot (\text{outdeg}(u) + 1)$ steps, for a constant c. Why the $+1$? Because if $\text{outdeg}(u) = 0$, then processing u still takes a few steps (dequeueing u and seeing that u has no outgoing edges). Summing over outer iterations, the running time is $\leq \sum_u c \cdot (\text{outdeg}(u) + 1) = c \cdot \left(\sum_u \text{outdeg}(u) + \sum_u 1\right) \leq c \cdot (m + m + 1) = O(m)$ since the sum of the outdegrees counts each of the m edges once, and since the number of processed nodes is at most $m + 1$.

In computational complexity theory, we generally bound the running time in terms of a single parameter N, which is the input's "total size." For MULTIPLICATION, the input has $N = 2n$ bits, so the primary-school algorithm has running time $O(n^2) = O(N^2)$. For DIRECTED REACHABILITY, the input's adjacency list representation has size $N = n + m$, so BFS has running time $O(m) = O(N)$. For some graph algorithms, the running time depends on both size parameters, such as $O(mn)$, but we should be mindful that there are multiple ways to define big-O for functions of several variables.

1.2.2 Space

An algorithm's *memory space* on a particular input is the number of bits it uses to store intermediate results ("scratch work") during the computation. This corresponds to the amount of RAM (random-access memory) a program uses. An algorithm can store whatever data it wants in memory and retrieve that data later. It can overwrite old intermediate results (that are no longer needed) with new ones. A fundamental distinction between space and time is that space can be reused, but time cannot (alas).

The brute force MULTIPLICATION algorithm only needs $O(n)$ memory space (assuming x and y are n bits) if implemented prudently: The running sum z never exceeds $x \cdot y$, so it fits in $2n$ bits of memory. As we add another copy of x to z, we construct z's new value in another $2n$ bits of memory. After the addition, we safely forget z's old value (since we won't use it again) and

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

overwrite it with z's new value. We also reuse the memory where we constructed z's new value for the same purpose in the next iteration, so we need not allocate any more memory for successive iterations. Remembering which iteration we're on (so we can stop after y iterations) takes n bits. We also need $O(\log n)$-bit pointers (indices) for running the addition algorithm, moving z after the addition, and incrementing the iteration counter. So we only need $O(n + \log n) = O(n)$ bits of memory at a time. We don't count the space to store the program, since that space doesn't depend on the input.

A straightforward implementation of the primary-school Multiplication algorithm uses $O(n^2)$ bits of memory for the n-by-$2n$ table. Why should we care about space? Suppose x and y each have $n =$ one million bits, and our computer executes one billion instructions per second and has 32 gigabytes of RAM. With $O(n^2)$ running time (with a reasonable hidden constant), if we start the computation before lunch, it might finish before we get back. However, n^2 bits of memory is over 100 gigabytes, so we might see an "out of RAM" error instead. Time is more often in the limelight, but sometimes space is the bottleneck.

The primary-school algorithm need not store the whole table all at once. One way to get the memory space down to $O(n)$ is to sum the rows one by one (like the brute force algorithm). It seems impossible to do better than $O(n)$ space, because the input and output are already more than n bits. But we'll show that the primary-school algorithm can get by with much less than n bits of memory if we disregard the space to store the input and output.

In computational complexity theory, the input is in a separate read-only memory segment, and the output goes in a separate write-only memory segment, and these special segments don't count toward the space. Intermediate data is stored in the read-write *work memory*. The space is the number of bits of work memory. If an algorithm's work memory is larger than its input and output, then the input, output, and intermediate data might as well be in one common memory. The segments idea is useful when the work memory is smaller than the input or output. This is motivated by practical situations where the input or output is on an external storage device, or available over a network, or not explicitly stored anywhere. If the input's individual parts can be retrieved on demand, and the output's individual parts can be sent to their destination, we don't include them in "what the algorithm is doing with its memory."

How much work memory does the primary-school Multiplication algorithm need? It stores i (index into z), j (index into y), $i - j$ (index into x), n (length of x and y), and c (carry). The inner loop reads the bits x_{i-j} and y_j from the input segment. The outer loop writes the bit z_i to the output segment. The indices and n take $O(\log n)$ bits, but the carry is more subtle to analyze. The space to store c is the log of the maximum value c could attain throughout the computation. We analyze this using a loop invariant:

Claim 1.1. *The value of c never exceeds $2n$ in the primary-school* Multiplication *algorithm.*

Proof. First, we prove that at the boundaries between outer iterations, c doesn't exceed n. This invariant holds at the beginning since $c = 0$. To see that the invariant is maintained, suppose it holds at the beginning of outer iteration i. In each inner iteration, c goes up by $x_{i-j} \cdot y_j \leq 1$. There are $\leq n$ inner iterations, so c goes up by $\leq n$ throughout the inner loop. Since $c \leq n$ before the inner loop (by assumption), we have $c \leq n + n$ after the inner loop. Outer iteration i concludes with $c \leftarrow \lfloor c/2 \rfloor \leq \lfloor 2n/2 \rfloor = n$, so the invariant indeed holds again.

The largest c could ever be is $2n$, right after some execution of the inner loop. ∎

Hence, c only needs $\lceil \log(2n + 1) \rceil = O(\log n)$ bits. (In this book, logs are base 2 by default.) Summing the memory requirements for all the variables, the primary-school algorithm uses only $O(\log n)$ work space—a logarithmic dependence on n.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

Now, we analyze the memory space of BFS for Directed Reachability. Suppose the graph has n nodes and m edges. The computation remembers the queue's contents and which nodes have been discovered. Each node needs one bit to record whether it's been discovered, which is n bits for all n nodes. The queue's space is the maximum number of items ever in the queue simultaneously, times the number of bits per item. Each node gets enqueued at most once, so the queue never holds more than n items. A node's label is $\lceil \log n \rceil$ bits. In total, the two data structures take $n + n \cdot \lceil \log n \rceil = O(n \log n)$ bits of memory. The computation also remembers other node labels and pointers, which only contribute $+O(\log n)$ bits and don't affect the overall space usage of $O(n \log n)$.

As with running time, we generally bound the memory space in terms of the total input size N. For Multiplication with $N = 2n$, the primary-school algorithm uses memory space $O(\log n) = O(\log N)$. For Directed Reachability with $N = n + m$, BFS uses memory space $O(n \log n) = O(N \log N)$. For *dense* graphs with $m \approx n^2$, we have $n \log n \approx \sqrt{N}$, so the actual memory space is much less than $O(N \log N)$. But the final analysis should hold for all inputs.

Outside this book, some analyses count "words" rather than bits of memory. Some references say the primary-school algorithm uses space $O(1)$ and BFS uses space $O(n) = O(N)$ because in practice, counters/indices/pointers fit in single machine words. In computational complexity theory, such variables take space $O(\log N)$ because bits (not words) are the unit of space, and the input size N is theoretically unlimited. We examine this issue in §1.3 and §1.4.

1.2.3 Complexity

With respect to a resource such as time or space:

- An algorithm's *efficiency* is the function that maps N to the maximum, over all inputs of size exactly N, of the resource usage on that input. Perhaps confusingly, a larger efficiency function means "less efficient."
- A problem's *complexity* is the efficiency of the "most efficient" algorithm that solves it. This is only an informal definition, because different algorithms could be more efficient on different input sizes, but this generally isn't a significant issue (see Exercise 1.20).

This isn't just about known algorithms, or about algorithms that could be devised using known techniques. A problem's complexity is determined by the "best" algorithm that exists mathematically, regardless of whether the algorithm has been found. "What is problem A's time complexity?" isn't a well-defined question, but it represents a collection of well-defined questions such as "Does A have an algorithm with time efficiency $O(N^2)$ or not?"

To show a problem has low complexity, we prove an *upper bound*—some algorithm is both correct and efficient. To show a problem has high complexity, we prove a *lower bound*—every algorithm is either incorrect or inefficient. *Correct* means for every input, the output is as it should be. *Efficient* means for every input, the resource usage is at most some target. Because of these $\forall$ quantifiers over inputs, correctness and efficiency are *worst-case* notions: An algorithm should perform well for every input, no matter how "bad" the input may be.

To summarize our upper bounds so far: Multiplication has time complexity $O(N^2)$ and space complexity $O(\log N)$ where $N = 2n$. Directed Reachability has time complexity $O(N)$ and space complexity $O(N \log N)$ where $N = n + m$. Since big-O is like "less than or equal to" for functions, these upper bounds leave open the possibility of more efficient algorithms—the problems might have even lower complexities.

In general, what does a lower bound look like? Suppose for some function T, we have a

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

correct algorithm with time efficiency $O(T(N))$ for a problem A, and we suspect it's impossible to design a faster algorithm for A. That is, we know an upper bound of $O(T(N))$ on A's time complexity, and we conjecture a matching lower bound.

- One formulation uses little-o: No algorithm for A has time efficiency $o(T(N))$. This means for every correct algorithm, there exists $c > 0$ such that for infinitely many inputs, the running time is $\geq c \cdot T(N)$.
- A stronger formulation uses big-Omega: Every algorithm for A has time efficiency $\Omega(T(N))$. This means for every correct algorithm, there exists $c > 0$ such that for all but finitely many N, the running time is $\geq c \cdot T(N)$ for at least one input of size N.

The distinction between "not little-o" and "big-Ω" is that for big-Ω, the infinitely many inputs on which an algorithm is inefficient must be "regularly occurring"—at least one per input size. This matters in some abstract corners of computational complexity theory. When studying specific, natural problems, the distinction is usually insignificant because nothing about particular sizes of inputs makes them more difficult than other sizes. Caveat: Some problems only have inputs of certain sizes. For example, suppose we insist that in Multiplication, both x and y are exactly n bits, so the input size $N = 2n$ must be even. Then it's unreasonable to want a big-Ω lower bound, which requires algorithms to be slow for both odd and even N despite there being nothing to do for odd N. Three ways to iron out this wrinkle are:

- Change the definition of big-Ω to quantify only over valid input sizes for the problem at hand.
- Change the definition of the problem so it has valid inputs of every size N. For example, Multiplication could say x has $\lceil N/2 \rceil$ bits and y has $\lfloor N/2 \rfloor$ bits.
- Change the definition of time efficiency at N to be the maximum running time over all valid inputs of size $\leq N$ rather than size exactly N.

Ideally, we want matching upper and lower bounds for a problem. For example, if we prove that some algorithm has space efficiency $O(S(N))$ and every algorithm has space efficiency $\Omega(S(N))$, we've pinned down the problem's space complexity (within a constant factor) at $\Theta(S(N))$. Typically, lower bounds are tougher to prove than upper bounds, because an upper bound just involves exhibiting a single algorithm, whereas a lower bound involves ruling out all possible algorithms. For many problems, a $\Omega(N)$ time lower bound and $\Omega(\log N)$ space lower bound hold because with less time or space, an algorithm can't even access the entire input. Superlinear time lower bounds and superlogarithmic space lower bounds tend to be drastically harder to prove, and open questions abound. The time and space complexities of countless problems remain unresolved by a wide margin.

What is Multiplication's time complexity? It's tempting to conjecture that no algorithm beats the primary-school algorithm's $O(N^2)$ time efficiency. If we expand

$$x \cdot y \;=\; \left(\sum_k x_k \cdot 2^k\right) \cdot \left(\sum_j y_j \cdot 2^j\right) \;=\; \sum_{k,j} (x_k \cdot y_j) \cdot 2^{k+j}$$

then it feels necessary to multiply each pair of bits $x_k \cdot y_j$, and there are $\Omega(N^2)$ such pairs. However, there are Multiplication algorithms with time efficiency pretty close to $O(N)$, using sophisticated mathematical tricks to bypass the explicit enumeration of all pairs of input bits.

What is Directed Reachability's space complexity? We showed a $O(N \log N)$ upper bound. The log factor can be shaved off because it arose from remembering the order of nodes in the queue, which is unnecessary (when we don't care about a path's length). We may be pessimistic about the prospect of going below $O(N)$, because it feels necessary to record at least one bit

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

per node, to avoid endless loops of rediscovering nodes. However, we prove a $O(\log^2 N) = O((\log N)^2)$ upper bound in Chapter 3. For Undirected Reachability (in which the input graph is *un*directed), the space complexity is $O(\log N)$ by a sophisticated algorithm that's beyond the scope of this book. The latter algorithm's discovery was predated by reasons to suspect it was possible, as we discuss in Chapter 7 and Chapter 11. For Directed Reachability, no such reasons are known, and most complexity theorists conjecture the space complexity is $\Omega(\log^2 N)$. A proof that Directed Reachability's space complexity is not $O(\log N)$ would already be an astounding breakthrough.

These examples show that intuition is unreliable when speculating about computational complexity. When we find an algorithm, there's usually a lingering question: Is a more efficient algorithm waiting to be discovered?

Lower bound proofs tend to be elusive because we can't make any assumptions whatsoever about the inner workings of algorithms. We must view any potential algorithm as a completely arbitrary tuple of instructions, which might have no apparent "rhyme or reason." To even conceive of rigorously proving lower bounds, we must first formalize the meanings of *problem* (§1.3) and *algorithm* (§1.4).

In the research community, the prevailing sentiment is that our goal is to understand *why* a theorem is true, not just ascertain *whether* it's true. We prefer proofs that yield insight. If Directed Reachability has no $O(\log N)$-space algorithm, we really want to learn *why not?* Or perhaps it has such an algorithm—who knows? The theory of computational complexity harbors many surprises, some of which we explore in this book.

1.3 Input/Output Encodings

Computers represent all information with bits. When we define a computational problem, we should specify how to represent the input and output with bits. For Multiplication this is simple—binary numbers already consist of bits. For Directed Reachability it's less immediate: A node's label from $\{1, 2, \ldots, n\}$ is approximately $\log n$ many bits. For the adjacency list representation, each node u has a list of the labels of nodes to which u has an edge, along with a way of locating the beginning and end of u's list. We'll see more details later, but a key takeaway of this section is that details don't matter much.

An *encoding* is an unambiguous way of representing each possible input and output value as a string of bits. *Unambiguous* means different values have different representations, so we can determine the value that any bit string represents. There isn't just one encoding for any problem—there can be many encodings that differ in how they handle various details.

- Technically, a problem's specification should include an encoding. Changing the encoding changes the problem.
- Practically, the encoding details are often left unspecified because they don't affect the problem's complexity very much. An algorithm for one reasonable encoding can be adapted to handle another reasonable encoding, without much overhead in time or space.

For example, one encoding for Multiplication has bits in most-to-least-significant order, while another encoding uses the opposite order. Formally, these are two distinct computational problems. But if an algorithm works with one order, we can tweak the algorithm to work with the other order, by changing where it looks for x's least significant bit in the input, and so on. There's not much point distinguishing between these encodings—intuitively, both are incarnations of a single problem called Multiplication.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

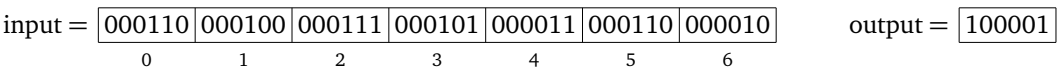


Figure 1.3: Example of Small Iterated Addition encoding

1.3.1 Word Size

In a computer, bits of data are organized into *words* of a common size. How many bits per word? This question exposes a dissonance between theory and practice. In real architectures, the word size is a constant such as 64 bits. But to express problems and algorithms naturally, individual words should fit simple data such as counters, indices, and pointers, which depend on the (arbitrarily large) input size. So the word size should depend on the input size. Let's see an example to demonstrate this clash.

Small Iterated Addition	(1.9)
Input:	Nonnegative integers $x_1, x_2, \ldots, x_N$ such that $x_i \le N$ for each i
Output:	$x_1 + x_2 + \cdots + x_N$

In this problem's name, Small means restricting each value x_i to a modest range $\{0, 1, \ldots, N\}$ (unlike Multiplication's big numbers), and Iterated means allowing many summands.

If we encounter this problem "in the field," we might whip up a Java method `public int sum(int[] x)`. The x_i values and return value (output) should be like "`int`"s in Java, which fit in single words. But x_i in binary is $\approx \log N$ bits, and the output $x_1 + x_2 + \cdots + x_N$ could be as large as N^2, so it's $\approx \log N^2 = 2 \log N$ bits. If these must fit in 64-bit words, the input size N would be limited to $\approx 2^{32}$. That's fairly generous, which is why the word size issue isn't typically a concern in practice. But for big-O analysis, N must go to infinity, necessitating a word size with at least a logarithmic dependence on N.

Example. Suppose the input to Small Iterated Addition is $(6, 4, 7, 5, 3, 6, 2)$ and the output is 33 (in base ten). The input size is $N = 7$, and the word size is $\lceil \log(N^2 + 1) \rceil = 6$. Figure 1.3 shows our encoding as a tuple of words, where each word is a bit string viewed as a nonnegative binary number with most-to-least-significant order. The little numbers underneath are addresses that algorithms will use to access the input's words: x_i is at address $i - 1$. ♦

From a practical standpoint, letting the word size depend on N seems preposterous, since hardware won't adjust its word size on the fly. For Small Iterated Addition, we could modify the encoding to use a constant word size by splitting each number's binary representation across multiple words, but this wouldn't match the spirit of the problem. Letting the word size grow with N is a standard compromise to reconcile the finiteness of real computers with the infinity of big-O analysis. We just have to get used to it.

We don't want the word size to grow too fast with N, since then N wouldn't be a reasonable measure of the amount of data in the input. We only allow $O(\log N)$ word size, which is generous enough for sensible encodings.

If the word size is W, a word of data is an element of $\{0, 1\}^W$ (the set of all bit strings of length W). If the input size is N, an (encoded) input is an element of $(\{0, 1\}^W)^N$ (the set of all N-tuples of words). We should write W_N to make it explicit that different values of N can have different associated word sizes, but we often tidy up the notation by just writing W with the implicit understanding that it depends on N.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

1.3.2 Computational Problems

For some problem encodings, not every element of $(\{0,1\}^W)^N$ represents a valid input. For example, in Small Iterated Addition, if some word is greater than N, then the whole input is invalid according to the problem's definition. Two ways to deal with this are:

- Restrict our attention to encodings where every element of $(\{0,1\}^W)^N$ represents a valid input. For example, in Small Iterated Addition, we could treat any input number greater than N as zero. An input may have multiple different representations, which is fine, but an algorithm would need to be correct and efficient on all of them. Multiple representations of an input are technically distinct inputs that "mean" the same thing.
- Allow elements of $(\{0,1\}^W)^N$ that don't represent valid inputs, but disallow giving them to algorithms. An algorithm has no burden to check whether its input is valid, and on an invalid input there's no correctness or efficiency requirement (even running forever is allowed). This is computational complexity's version of "garbage in, garbage out." We *promise* that the input will be valid (and don't care about "what if it's not"). The problem is called a *promise problem*. We specify the input-output relationship using a *partial function*, which may be undefined for some elements of the domain $(\{0,1\}^W)^N$. A *total function* is a partial function that's defined on the whole domain, so every element is a valid input.

The choice between these conventions is often immaterial but sometimes important. We allow promise problems since they simplify specifying encodings and implementing algorithms.

We're ready for our official definition of *problem*. For search problems, the definition involves *power sets*: If S is a set then $\text{pow}(S) = \{T : T \subseteq S\}$ is the set of all subsets of S.

Definition. A *computational problem* associates to each positive integer *input size* N a pair (W_N, F_N), abbreviated (W, F), such that:

- W is a positive integer *word size*, which must be $O(\log N)$.
- F is a partial function with domain $(\{0,1\}^W)^N$. The inputs on which F is defined are *valid*.
  - *Decision problem*: F has codomain $\{0,1\}$. On input x, the goal is to output $F(x)$. If $F(x) = 1$ then x is a 1-*input* or *yes-input*. If $F(x) = 0$ then x is a 0-*input* or *no-input*.
  - *Function problem*: F has codomain $(\{0,1\}^W)^M$ for some M depending on N. On input x, the goal is to output $F(x)$.
  - *Search problem*: F has codomain $\text{pow}\big((\{0,1\}^W)^M\big)$ for some M depending on N. On input x, the goal is to output an element of the set $F(x)$ or report that $F(x) = \emptyset$.

If A is a problem and x is a valid input of size N, then $A(x)$ means $F_N(x)$. ◄

For search problems, $F(x)$ is the set of acceptable solutions. Function problems (such as Multiplication) are equivalent to search problems where each valid input has exactly one acceptable solution. For function and search problems, M is the output size, and the output is encoded using the same word size as the input.

Every problem has an encoding with word size 1 (though perhaps not the most natural encoding). We could define a decision problem as a single partial function $F : \{0,1\}^+ \to \{0,1\}$ encompassing all input sizes, since $\{0,1\}^+$ is the set of all nonempty bit strings. This is handy when discussing general concepts, without a particular problem in mind. The point of bigger word sizes is that the two-level structure (break input into words, break words into bits) lets us express specific problems naturally, without fussing about delineating pieces of data.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

$(x, y) = (10110, 01100)$

0	1	1	0	1	0	0	1	1	0
0	1	2	3	4	5	6	7	8	9

Figure 1.4: Example of Multiplication encoding

1.3.3 Example Encodings

We describe particular input/output encodings for Multiplication and Directed Reachability. We use these encodings when implementing the primary-school algorithm and breadth-first search after seeing our official definition of *algorithm* in §1.4.

For Multiplication, we use word size $W_N = 1$ and have each number's bits in least-to-most-significant order (from lower to higher addresses). We promise that the input size N is even, and we view the first $N/2$ bits as x and the last $N/2$ bits as y. The output is a bit string of length $M = N$. When N is even, $F_N \colon \{0,1\}^N \to \{0,1\}^N$ is a total function. When N is odd, F_N is a partial function that's undefined on all inputs. Figure 1.4 shows an example, with addresses represented by little numbers underneath.

Multiplication's complexity is robust with respect to the encoding details. Other reasonable encodings differ in the bit order, the case when N is odd, and so on. We chose our encoding to make the algorithm implementation as simple as we can.

We could consider an encoding with word size > 1 and cram more input/output bits into each word. This technique is handy in practice for improving efficiency, but it doesn't capture the cleanest way to view the problem. We intended the input size N to be the number of bits in the binary representation.

For Directed Reachability, we use a larger word size and view each word as a binary number (with most-to-least-significant bit order) representing either a node label from $[n] = \{1, 2, \ldots, n\}$, a count of some number of nodes or edges, or a pointer (address of another input word). Every node u gets a contiguous list of words holding the labels of u's outneighbors (nodes to which u has an edge) in increasing order (though the order won't matter). The encoding also needs some structure to help algorithms find the beginning and end of each node's list: Address 0 gets n (the number of nodes). For each $u \in [n]$, address u gets a pointer to the beginning of node u's list. Then starting from address $n + 1$, we concatenate the lists for nodes $1, 2, \ldots, n$. Node u's list consists of a word holding $\text{outdeg}(u)$, followed by $\text{outdeg}(u)$ many words holding u's outneighbors' labels. After node n's list, two more words hold the labels of s and t (the start and destination nodes).

Example. Figure 1.5 shows the encoding of an example input, with annotations to make the structure apparent. Words are shown in base ten for readability. The first bracket on top illustrates that the initial word 5 counts how many pointers follow. For each of those 5 words, the arrow underneath shows which address it points to. Address 6 is the start of node 1's list, address 8 is the start of node 2's list, and so on. For each node's list, the bracket on top illustrates that the first word counts how many node labels follow. For example, address 15 contains 3 because node 5 has edges to three nodes (namely nodes $1, 3, 4$, which are the words under the bracket). Node 3's bracket has no words since $\text{outdeg}(3) = 0$. ♦

We view Directed Reachability as a promise problem: Any tuple of words that doesn't conform to our encoding (with node labels not in $[n]$, or inconsistent pointers, for example) is invalid. An algorithm can assume its input legitimately represents some (G, s, t), without checking for formatting mistakes.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

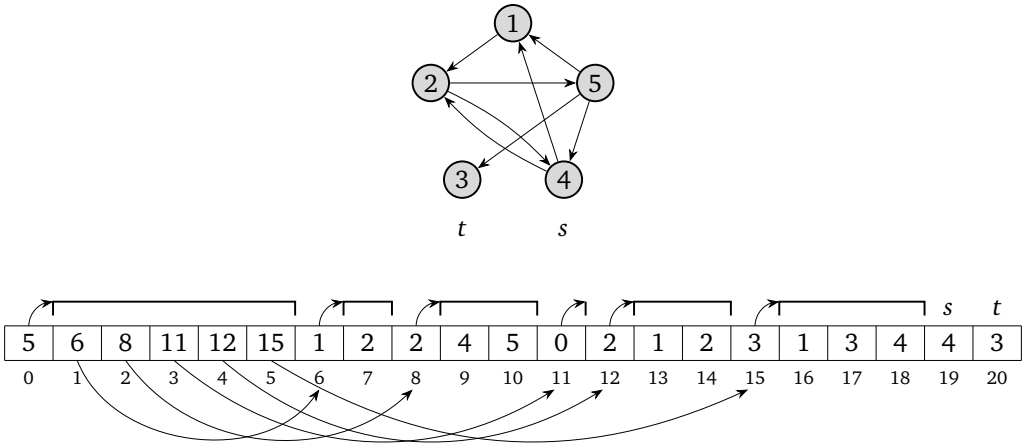


Figure 1.5: Example of Directed Reachability encoding

If G has n nodes and m edges, then the number of words is $N = 2n + m + 3$ because each node has two words (the pointer and outdegree), each edge (u, v) is accounted for once (in u's list), and n, s, t are the other three words. Technically, we should express algorithm efficiency and problem complexity in terms of this N, which isn't exactly the $N = n + m$ from earlier. This doesn't affect our big-O analysis, since these values of N are within a constant factor.

What word size is appropriate? Each node label and each count only needs $\lceil \log(n+1) \rceil$ bits since the possible values are $\{0, 1, \ldots, n\}$. A pointer could contain an address near the end of the input ($N - 1$ is the last address), so we can use $\lceil \log N \rceil$ bits per pointer. Since $n + 1 \leq N$, word size $W_N = \lceil \log N \rceil$ is enough.

As usual, there's flexibility in designing the encoding. The outdegree words are redundant since an algorithm could calculate an outdegree by subtracting adjacent pointers. We could modify the encoding by eliminating the outdegree words, at the cost of an extra implementation step and a special case for node n, but this wouldn't affect our big-O analysis. We adopt our chosen encoding for convenience, but the details don't matter much.

Throughout this book, when a problem's input contains a graph, we assume the encoding is as above, unless specified otherwise. The same encoding works for undirected graphs, by viewing an undirected edge $\{u, v\}$ as a pair of directed edges (u, v) and (v, u).

1.4 The Model of Computation

Informally, an algorithm is a tuple of instructions specifying elementary steps of a computation. What's an "elementary step"? Many ways to flesh out the definition are essentially equivalent as long as they have certain core features. A *model of computation* is a formal definition of what constitutes algorithms. The details don't matter much because they have little impact on the complexities of problems—an algorithm in one model can be translated to another reasonable model without much loss in efficiency.

Though various models of computation would be fine, we must designate an "official" one for this book. This is a balancing act: For upper bounds, we'd prefer a feature-rich model like a high-level programming language, to facilitate specifying algorithms. For lower bounds, we'd prefer a bare-bones model, to minimize the variety of possible algorithms to rule out. The model

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

we elaborate in this section is an idealized assembly language. It imitates how real computers work but is tidier. Algorithms are like software, and the model of computation is like hardware.

There's no formal distinction between *algorithm* and *program*. "There exists an algorithm such that . . . " has literally the same meaning as "There exists a program such that . . . " But by convention, *algorithm* means high-level pseudocode as in §1.1, and *program* means actual code in our official model, and *implementation* means turning an algorithm into a program.

1.4.1 Model Definition

Memory. The memory is organized into five parts:

- The read-only *input segment*.
- The write-only *output segment*.
- The read-write *work segment*.
- The general-purpose *registers*.
- The *program counter*.

The input, output, and work segments each have an unlimited number of words, with addresses starting from 0. (Many real architectures split words into bytes, and addresses refer to bytes. In our model, addresses refer to words.) Registers and the program counter are single words.

The number of registers depends on the program but not on the input. A program can use any number of registers. This may seem unrealistic since real architectures have a fixed number of registers, but we'll see in §1.5 that we can adapt every program to use just two registers.

The input segment's first N words are initialized with the input. The 0th register is initialized with the input size N. All other words of memory are initialized to all-0s.

Code. A program has a tuple of instructions. There are four types of instructions:

- **Data processing.** An instruction can get the contents of one or two registers, perform an arithmetic or bitwise logical operation on them, and put the result in a register. The instruction specifies which registers and which operation. It can also use a hardcoded constant ("immediate value") in place of either operand register. In a real processor, the ALU (arithmetic-logic unit) does these operations.
- **Control flow.** An instruction can make the computation *branch* (jump) to another instruction, possibly conditional on a comparison between two register (or constant) values. The program counter holds the current instruction's *line number* (an address in the *code segment* containing the instructions). The program counter is incremented after every instruction except when a branch is taken.
- **Memory access.** An instruction can *load* a word from the work segment into a register, or *store* a word from a register into the work segment. The instruction specifies two registers (or constants): the word of data being copied, and the address within the work segment. (Real architectures have fancier addressing modes. Our model only has "direct" and "register indirect" modes.) To get a word from the input segment or put a word in the output segment, we use instructions called *read* and *write* (to distinguish them from the analogous *load* and *store*). Output segment addresses must be written to consecutively.
- **Termination.** Three instructions make a computation halt: *accept* and *reject* mean "output 1" and "output 0" respectively (for solving decision problems), and *halt* means "output the output segment's contents up through the largest address that was written to" (for solving

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

function and search problems). For search problems, *reject* means "no acceptable solution." If none of these instructions is executed, the computation runs forever. A program's last instruction (in the tuple of instructions) must be a termination or unconditional branch.

We flesh out the instruction set in §1.4.2, but the details aren't very important.

Many instructions can use hardcoded constants in lieu of register values. A program can use arbitrarily large constants. This may seem unrealistic since real architectures have a fixed number of bits for immediate values, but we'll see in §1.5 that we can adapt every program to use just single-bit constants.

Word size. The program counter has its own special word size $\lceil \log L \rceil$ where L is the number of instructions (lines of code). Each other word of memory has size W_N (depending on the input size N), which must be large enough to hold N itself and any hardcoded constant (not including register indices and line numbers). A program Π (capital Greek pi) is a pair (I, W) where I is the tuple of instructions (code) and $W : \mathbb{N}_+ \to \mathbb{N}_+$ is the word size function, where $W(N)$ means W_N (and $\mathbb{N}_+$ is the set of positive integers).

Correctness. A program *solves* a problem iff for every valid input, it terminates with a correct output. The program and problem might not have the same word size. If the program has smaller word size than the problem, then it can't solve the problem since the input words don't fit. If the program has larger word size than the problem, then the input segment's words are the problem's input words left-padded with 0s, and the output segment's words (for function/search problems) should be the problem's output words left-padded with 0s.

Efficiency. A program's *running time* on a particular input is the number of instruction executions up through termination, which could be ∞. (A pitfall of modeling computation with a high-level programming language would be that instructions may conceal mini-algorithms, so counting instructions wouldn't accurately reflect the running time.)

A program's *memory space* on a particular input is the total number of bits in all registers and the work segment from address 0 up through the largest address accessed by a load or store (including any in-between addresses that aren't accessed). The program counter's size doesn't depend on the input, so it's not included in the memory space. If a program has R registers, accesses at most the first M words of the work segment, and has word size W, then the memory space is $\leq (R+M)W$. Since R is a constant, the memory space is $O(MW)$ if the work segment is accessed, and is $O(W)$ if not. It would be reasonable to define memory space as the number of bits in the words that are actually accessed (regardless of the largest address accessed), but this wouldn't make a big difference (§1.7.3). Our definition is like allocating a contiguous block of M words of work memory for a program to manage on its own. Space can't be ∞ even if time is ∞: Addresses must fit in registers, so they can't exceed $2^W - 1$.

It might seem unfair that, for example, adding two single-word numbers is only 1 time step but $O(W)$ bits of space. This is an artifact of the "W depends on N" discord between theory and practice. Bits have always been the unit of space in computational complexity, but it's still sensible to count time steps that resemble real machine instructions.

With respect to a problem, a program's *time efficiency* and *space efficiency* are the functions $T : \mathbb{N}_+ \to \mathbb{N}$ and $S : \mathbb{N}_+ \to \mathbb{N}$ where $T(N)$ and $S(N)$ are the maximum running time and maximum memory space over all inputs of size N that are valid for the problem (and $T(N) = S(N) = 0$ if there are no valid inputs of size N). We don't care about time or space on invalid inputs.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

1.4.2 Instruction Set

On the spectrum of RISC (reduced instruction set computer) versus CISC (complex instruction set computer), our model is at the RISC end. The word in register i is $\mathrm{Reg}[i]$. The words at address i in the work/input/output segments are $\mathrm{Mem}[i]$ / $\mathrm{In}[i]$ / $\mathrm{Out}[i]$. (A program can't access its "code segment.") Hardcoded constants—which may be operands in data processing or control flow instructions, or addresses or data in memory access instructions—are nonnegative binary integers, treated as whole words (left-padded with 0s).

Data processing. These instructions begin with "let". The simplest is copying one register to another:

$$\text{let } \mathrm{Reg}[i] \leftarrow \mathrm{Reg}[j]$$

This overwrites $\mathrm{Reg}[i]$ with the contents of $\mathrm{Reg}[j]$, so the old contents of $\mathrm{Reg}[i]$ are lost and $\mathrm{Reg}[j]$ is unchanged. In lieu of $\mathrm{Reg}[j]$, the instruction could assign a hardcoded constant to $\mathrm{Reg}[i]$. Other instructions are "let $\mathrm{Reg}[i] \leftarrow \ldots$" with arithmetic or bitwise logical operations on $\mathrm{Reg}[j]$ and $\mathrm{Reg}[k]$:

$\mathrm{Reg}[j] + \mathrm{Reg}[k]$	(sum)	$\neg\mathrm{Reg}[j]$	(not)
$\mathrm{Reg}[j] - \mathrm{Reg}[k]$	(difference)	$\mathrm{Reg}[j] \wedge \mathrm{Reg}[k]$	(and)
$\mathrm{Reg}[j] \cdot \mathrm{Reg}[k]$	(product)	$\mathrm{Reg}[j] \vee \mathrm{Reg}[k]$	(or)
$\lfloor \mathrm{Reg}[j] / \mathrm{Reg}[k] \rfloor$	(quotient)	$\mathrm{Reg}[j] \oplus \mathrm{Reg}[k]$	(xor)
$\mathrm{Reg}[j] \bmod \mathrm{Reg}[k]$	(remainder)	$\mathrm{Reg}[j] \gg \mathrm{Reg}[k]$	(shift right)
$\lfloor \log_2(\mathrm{Reg}[k]) \rfloor$	(logarithm)	$\mathrm{Reg}[j] \ll \mathrm{Reg}[k]$	(shift left)

The register indices i, j, k need not be distinct. These instructions can have hardcoded constants in lieu of $\mathrm{Reg}[j]$ and/or $\mathrm{Reg}[k]$.

For arithmetic instructions, a word in $\{0,1\}^W$ is a nonnegative integer in $\{0, 1, \ldots, 2^W - 1\}$. Sum/difference/product are done "mod 2^W" so there's no overflow or negative numbers. (Bits that would be "too significant" to fit in a word are dropped.) Quotient/remainder/logarithm yield $\mathrm{Reg}[i] \leftarrow 0$ if $\mathrm{Reg}[k] = 0$. (A program is responsible for anticipating divide-by-0 and log-of-0, since our model has no runtime errors.)

To do signed arithmetic using two's complement to represent negative numbers (as real computers do), a program can treat the positive integers $2^{W-1}, \ldots, 2^W - 1$ as the negative integers $-2^{W-1}, \ldots, -1$ respectively. The "mod 2^W" arithmetic automatically handles signs correctly for sum/difference/product (but division would require consideration), so signed arithmetic is available even though the model officially only has unsigned numbers.

A bitwise shift instruction obtains the new contents of $\mathrm{Reg}[i]$ from $\mathrm{Reg}[j]$ by dropping $\mathrm{Reg}[k]$ many bits off one end and filling in the other end with 0s. For example, $1 \ll \mathrm{Reg}[k]$ is $2^{\mathrm{Reg}[k]}$ in binary if $\mathrm{Reg}[k] < W$. The logarithm instruction might feel out of place, but it's tasteful to include because it's the inverse of the exponentiation provided by shift left: $\lfloor \log_2(\mathrm{Reg}[k]) \rfloor$ is the position of the leftmost 1 in $\mathrm{Reg}[k]$ (where position 0 is the rightmost bit).

Control flow. An unconditional branch updates the program counter to point to another instruction, and execution continues from there:

$$\text{goto line } \ell$$

The value of ℓ is hardcoded in the instruction. We number lines from 0, so $0 \le \ell < L$ where L is the number of lines of code. A conditional branch compares the contents of two registers (or a

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

constant in lieu of either register). If the condition is true, it updates the program counter to ℓ. If the condition is false, it increments the program counter as usual.

$$\text{if } \mathrm{Reg}[i] = \mathrm{Reg}[j] \text{ then goto line } \ell$$

We can also use $\neq$, $<$, $\leq$, $>$, $\geq$ for the comparison. These treat words as unsigned integers, so a program must compensate if it uses two's complement.

Memory access. Addresses are nonnegative binary integers.

Get word from work segment address Reg[j] into Reg[i]:	load Reg[i] ← Mem[Reg[j]]
Put word from Reg[i] into work segment address Reg[j]:	store Mem[Reg[j]] ← Reg[i]
Get word from input segment address Reg[j] into Reg[i]:	read Reg[i] ← In[Reg[j]]
Put word from Reg[i] into next output segment address:	write Reg[i]

We allow $i = j$. There can be a constant in lieu of Reg[i] (the data) and/or Reg[j] (the address), except that load and read can't have a constant in lieu of Reg[i].

A read instruction with address $\geq N$ gets an all-0 word. The first write instruction executed writes to Out[0], the second write instruction executed writes to Out[1], and so on. We disallow "write Out[Reg[j]] ← Reg[i]" because it's susceptible to abuse: Writing to a high address while leaving gaps of "unwritten-to" addresses would let a program produce a large output without being charged for the time to write each word—it could get all-0 words in the output for free. Also, writing to the same output address more than once would let a program use it as a kind of work memory without being charged for the space—it could put a word there, then later decide whether to leave that word or overwrite it. Our write instruction forces programs to write each output word exactly once.

Termination. As explained in §1.4.1, three instructions make the computation stop and announce the output:

accept reject halt

1.4.3 Example Programs

For "hello world," we write a program solving Small Iterated Addition from §1.3.1. A loop adds each input number (one by one) to a running sum. We don't use the work segment; the registers are enough space.

- Reg[0] is initialized to N, and we won't change it.
- Reg[1] is the loop counter and an address (pointer) into the input.
- Reg[2] is the current number from the input.
- Reg[3] is the running sum.

Reg[1], Reg[2], Reg[3] are initialized to 0, though for Reg[2] it doesn't matter.

Program for Small Iterated Addition (1.10)

0. read Reg[2] ← In[Reg[1]]
1. let Reg[3] ← Reg[3] + Reg[2]
2. let Reg[1] ← Reg[1] + 1
3. if Reg[1] < Reg[0] then goto line 0
4. write Reg[3]
5. halt

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

The loop is line 0 through line 3. Line 0 reads the next input number. Line 1 adds it to the running sum. Line 2 increments the loop counter. Line 3 branches to the top for another iteration if more input remains. Line 4 puts the running sum at address 0 in the output segment. Since this is the only output word accessed, it's the entire output when the program halts on line 5.

The program works with word size $\lceil \log(N^2+1) \rceil$ since Reg[0], Reg[1], Reg[2] never exceed N and Reg[3] never exceeds N^2, assuming the input is valid. (The problem promises $x_i \leq N$ for each i.) This illustrates that a program's word size should depend on N, so words can hold simple data that would be `int` variables in Java, say. The program has time efficiency $O(N)$ and space efficiency $O(\log N)$ since each register holds $O(\log N)$ bits. Since the program has no load or store instructions, only registers contribute to the space.

Next, we implement the primary-school algorithm for Multiplication and the breadth-first search algorithm for Directed Reachability with the input/output encodings from §1.3.3. To improve readability of these programs, we use human-friendly aliases for registers, and indentation to show the correspondence with the pseudocode's nested loops.

The top of Figure 1.6 shows our program for Multiplication. The input segment has x's bits followed by y's bits (with each bit in a separate word) in least-to-most-significant order, so $\text{In}[k] = x_k$ and $\text{In}[n+k] = y_k$ for each $k \in \{0, \ldots, n-1\}$. We use aliases for Reg[0], ..., Reg[9]:

$$\begin{aligned} N &= \text{input size} \\ n &= N/2 \\ c &= \text{carry value, and running sum for each column} \\ i &= \text{outer loop counter} \\ j &= \text{inner loop counter} \\ ptr &= \text{pointer for input addresses } i-j \text{ (to read } x_{i-j}\text{) and } n+j \text{ (to read } y_j\text{)} \\ xbit &= x_{i-j} \\ ybit &= y_j \\ zbit &= z_i \\ prod &= \text{temporary holder for } x_{i-j} \cdot y_j \end{aligned}$$

All registers except N are initialized to 0, but we only rely on this for c and i. We must reinitialize j each time the inner loop starts (line 2). The heart is $c \leftarrow c + (x_{i-j} \cdot y_j)$ (line 7 through line 11), but before that we check if $i - j < 0$ (line 4) or $i - j \geq n$ (line 6) and skip updating c if either holds. For line 4, why not compare $ptr < 0$ after computing $ptr \leftarrow i - j$, similar to line 6? Since our model only has unsigned arithmetic, that comparison would always be false—negative numbers wrap around to large positive numbers. (This is a common bug in real-world programs that use unsigned integers.) Comparing $i < j$ sidesteps the issue. The bits of z are written to separate words of the output segment, so $\text{Out}[i] = z_i$. The first $N = 2n$ output segment words are the output upon termination.

The program works with word size $\lceil \log(N+1) \rceil$ (which is greater than the problem's word size of 1) since each register never exceeds N (see Claim 1.1 regarding c). The program has time efficiency $O(N^2)$ and space efficiency $O(\log N)$, as we claimed earlier.

The bottom of Figure 1.6 shows our program for Directed Reachability. The work segment layout is:

- Address 0 goes to waste since this slightly simplifies the implementation.
- Addresses $1, \ldots, n$ record which nodes have been discovered: Mem[u] is 1 or 0 to indicate whether node u is discovered or undiscovered, respectively. The initialization automatically makes all nodes undiscovered at the beginning.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

Implementation of the primary-school algorithm for MULTIPLICATION (1.11)

```
0.  let n ← ⌊N/2⌋
1.  if i ≥ N then goto line 19
2.      let j ← 0
3.      if j ≥ n then goto line 14
4.          if i < j then goto line 12
5.          let ptr ← i − j
6.          if ptr ≥ n then goto line 12
7.          read xbit ← In[ptr]
8.          let ptr ← n + j
9.          read ybit ← In[ptr]
10.         let prod ← xbit · ybit
11.         let c ← c + prod
12.         let j ← j + 1
13.         goto line 3
14.     let zbit ← c mod 2
15.     write zbit
16.     let c ← ⌊c/2⌋
17.     let i ← i + 1
18.     goto line 1
19. halt
```

Implementation of the breadth-first search algorithm for DIRECTED REACHABILITY (1.12)

```
0.  let ptr ← N − 2
1.  read s ← In[ptr]
2.  let ptr ← N − 1
3.  read t ← In[ptr]
4.  if s ≠ t then goto line 6
5.      accept
6.  store Mem[s] ← 1
7.  read head ← In[0]
8.  let head ← head + 1
9.  store Mem[head] ← s
10. let tail ← head + 1
11. if head = tail then goto line 28
12.     load u ← Mem[head]
13.     let head ← head + 1
14.     read ptr ← In[u]
15.     read outdeg ← In[ptr]
16.     let end ← ptr + outdeg
17.     if ptr = end then goto line 11
18.         let ptr ← ptr + 1
19.         read v ← In[ptr]
20.         load disc ← Mem[v]
21.         if disc = 1 then goto line 17
22.         if v ≠ t then goto line 24
23.             accept
24.         store Mem[v] ← 1
25.         store Mem[tail] ← v
26.         let tail ← tail + 1
27.         goto line 17
28. reject
```

Figure 1.6: Implementations of algorithms

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

- Addresses $n+1, \ldots, 2n$ are the queue: The label of the first node to be enqueued goes at address $n+1$, the second to be enqueued goes at address $n+2$, and so on. The queue's current contents are at addresses *head*, *head* + 1, ..., *tail* − 1. The next place to dequeue is *head*. The next place to enqueue is *tail*. The queue is empty iff *head* = *tail*. Since each node is enqueued at most once, the queue need not "wrap around" the allocated space.

We use aliases for Reg[0], ..., Reg[10]:

$$\begin{aligned} N &= \text{input size} \\ ptr &= \text{pointer for accessing various input words} \\ s &= \text{start node} \\ t &= \text{destination node} \\ head &= \text{queue's head pointer} \\ tail &= \text{queue's tail pointer} \\ u &= \text{node being processed} \\ outdeg &= \text{node } u\text{'s outdegree} \\ end &= \text{pointer to the last entry in } u\text{'s list} \\ v &= \text{other endpoint of an edge out of } u \\ disc &= \text{indicator of whether } v \text{ is discovered} \end{aligned}$$

Line 0 through line 5 gets s and t and accepts if $s = t$. Line 6 marks s as discovered, and line 7 through line 10 sets up the queue to contain s (with $head = n+1$ and $tail = n+2$). The outer loop (line 11 through line 27) iterates until the queue is empty. Line 12 and line 13 dequeue u. Line 14 through line 16 finds the beginning and end of u's list. The inner loop (line 17 through line 27) iterates through the list's entries (pointed to by *ptr*). Line 19 through line 21 gets v and continues to the next inner iteration if v was already discovered. Then, the program accepts if $v = t$, and otherwise marks v as discovered (line 24) and enqueues v (line 25 and line 26).

The program works with word size $\lceil \log(N+1) \rceil$ since each register never exceeds N: For *ptr* and *end* this is clear since they point into the input; $s, t, u, v, outdeg$ never exceed the number of nodes $n \leq N$; *head* and *tail* never exceed $2n+1 \leq N$; *disc* is always 0 or 1. The program has time efficiency $O(N)$ and space efficiency $O(N \log N)$, as we claimed earlier.

Computational complexity theory is mostly *not* about dealing with such code. There are much bigger fish to fry, which transcend the low-level details. But the formalism of the model of computation is always behind the scenes, supporting the theory.

1.5 Robustness of the Model

There's a lot of leeway in defining a model of computation. As long as the model has sufficient representation of the four essential ingredients—data processing, control flow, memory access, and termination—we can add (and sometimes remove) various instructions without significantly changing the complexities of problems. This is because any program that uses the extra instructions can be adapted to accomplish the same thing, almost as efficiently, without those instructions. We illustrate this in §1.5.1.

Programs can use as many registers as they want, and hardcoded constants as large as they want. We prove in §1.5.2 that essentially nothing is lost if programs can only use two registers and one-bit constants.

§1.5.3 has some observations about memory access, which will simplify some things later.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

Another issue has to do with word sizes. A program's word size should just provide appropriate space to store basic units of data in individual words. A computation might be sensitive to the exact word size, but in §1.5.4 we show that increasing the word size doesn't mess things up if we impose restrictions to preclude abuse of the word size.

Upshot: Don't fret about the model. An intuitive understanding of computation is good enough, because the formal underpinning is very resilient.

1.5.1 Adding and Removing Instructions

Base+offset addressing. Instead of specifying a memory address with just a register ("indirect mode") or constant ("direct mode"), we could specify it as a sum of two values, each of which is a register or constant. This is handy for accessing a data structure in a contiguous block of memory: The *base* points to the start of the data structure, and *offsets* refer to components of the data structure. This feature is merely a convenience because we can split a "base+offset" memory access into two official instructions: calculate the absolute address, then do the access with register indirect mode.

For "load", since we're about to overwrite $\text{Reg}[i]$, we might as well use it for the address:

$$\begin{array}{ll} \text{load } \text{Reg}[i] \leftarrow \text{Mem}[\text{Reg}[j] + \text{Reg}[k]] & \rightsquigarrow \quad \text{let } \text{Reg}[i] \leftarrow \text{Reg}[j] + \text{Reg}[k] \\ & \qquad\quad \text{load } \text{Reg}[i] \leftarrow \text{Mem}[\text{Reg}[i]] \end{array}$$

This also works with constants in lieu of $\text{Reg}[j]$ or $\text{Reg}[k]$.

For "store", it's more subtle. We must handle all possible combinations of i, j, k being equal and constants in lieu of any registers. A worrisome example is "store $\text{Mem}[\text{Reg}[i] + c] \leftarrow \text{Reg}[i]$" for a constant c: Which register should we put the address $\text{Reg}[i] + c$ into? Using $\text{Reg}[i]$ would mess up the data being stored, and commandeering another register might derail the computation. Fortunately, our model lets the modified program use more registers than the original program. Pulling a new "register h" out of thin air to hold the address:

$$\begin{array}{ll} \text{store } \text{Mem}[\text{Reg}[j] + \text{Reg}[k]] \leftarrow \text{Reg}[i] & \rightsquigarrow \quad \text{let } \text{Reg}[h] \leftarrow \text{Reg}[j] + \text{Reg}[k] \\ & \qquad\quad \text{store } \text{Mem}[\text{Reg}[h]] \leftarrow \text{Reg}[i] \end{array}$$

We need not feel guilty about introducing a new register, because we can always go down to two registers (§1.5.2). This also works with constants in lieu of $\text{Reg}[i]$, $\text{Reg}[j]$, or $\text{Reg}[k]$.

After applying these substitutions, target line numbers of branches may be messed up. Let $\lambda(\ell)$ be the new line number of the instruction from line ℓ of the original program. (λ is lower-case Greek lambda.) We update the target of each conditional and unconditional branch:

$$\text{goto line } \ell \quad \rightsquigarrow \quad \text{goto line } \lambda(\ell)$$

Time and space only go up by constant factors.

Indirect branching. This means the target line number is in a register (and if the register points to a nonexistent line, then the indirect branch instruction is ignored). To simulate this:

$$\begin{array}{ll} \text{goto line } \text{Reg}[i] & \rightsquigarrow \quad \text{if } \text{Reg}[i] = 0 \text{ then goto line } \lambda(0) \\ & \qquad\quad \text{if } \text{Reg}[i] = 1 \text{ then goto line } \lambda(1) \\ & \qquad\quad \text{if } \text{Reg}[i] = 2 \text{ then goto line } \lambda(2) \\ & \qquad\quad \vdots \end{array}$$

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

To simulate a conditional indirect branch, we precede the block of "if"s with a conditional branch that jumps over the block if the condition is false. $\lambda(\ell)$ is the new line number (after doing all the substitutions) of the instruction from line ℓ of the original program. We also update the target line numbers of preexisting "direct" branches.

The original line numbers become hardcoded constants as comparison operands. We need not feel guilty about introducing large new constants, because we can always go down to one-bit constants (§1.5.2).

A technical nuisance: Operand constants—including the ones we just introduced—must fit in the word size for every N. Our modified program might be illegal for the silly reason that these "line number constants" take more than W_N bits when N is small. To bypass this, we apply a different substitution using a fresh "register h":

$$\begin{array}{lcl} \text{goto line Reg}[i] & \rightsquigarrow & \text{let Reg}[h] \leftarrow \text{Reg}[i] \\ & & \text{if Reg}[h] = 0 \text{ then goto line } \lambda(0) \\ & & \text{let Reg}[h] \leftarrow \text{Reg}[h] - 1 \\ & & \text{if Reg}[h] = 0 \text{ then goto line } \lambda(1) \\ & & \text{let Reg}[h] \leftarrow \text{Reg}[h] - 1 \\ & & \text{if Reg}[h] = 0 \text{ then goto line } \lambda(2) \\ & & \vdots \end{array}$$

The modified program is legal and equivalent to the original. (For small N, if some original line numbers don't fit in a word, then the original program couldn't indirectly branch to those lines.)

Time and space only go up by constant factors depending on the original program's size.

Subroutine calls. We use base+offset addressing and indirect branching to create a subroutine call mechanism. ("Subroutine" has fewer connotations than similar terms like "method" and "procedure.") Part of the work memory is designated as a *call stack*. (Real architectures sometimes deem the call stack a separate segment.) Each item on the call stack is a *stack frame*, which is a contiguous block of memory holding data associated with a subroutine invocation: local variables and arguments, return address (line number where execution resumes when the subroutine returns), and previous values of registers the subroutine uses. One register is designated as the *frame pointer*, which points to the start of the currently executing invocation's stack frame.

- A subroutine call *pushes* a frame associated with the new invocation. This involves initializing local variables, saving the return address, saving the contents of registers (including the frame pointer), updating the frame pointer, and branching to the subroutine.
- During execution of the subroutine, the program uses the frame pointer as the base, and uses different offsets to access local variables and arguments within the frame.
- A return statement *pops* the frame. This involves putting the return value at a designated place, restoring saved register contents, and doing an indirect branch to the saved return address, so the caller picks up where it left off.

There are various details to iron out, but we can do it all with official instructions and features we already added to the model.

Adding other instructions. Whipping up new registers is helpful for augmenting the instruction set. For example, we simulate a hypothetical instruction "let $\text{Reg}[i] \leftarrow \lfloor\sqrt{\text{Reg}[j]}\rfloor$" by inserting a mini-algorithm to compute $\lfloor\sqrt{\text{Reg}[j]}\rfloor$ using auxiliary registers. The time overhead for "square root" wouldn't be a constant factor—it would depend on the word size—but would be a small

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

price to pay.

For floating point arithmetic, "just do it in software." Treating two registers as the mantissa and exponent, we simulate arithmetic operations with mini-algorithms using auxiliary registers, incurring a low time overhead depending on the word size.

When writing high-level pseudocode, we assume all these features are available. In programming language parlance, these are syntactic sugar—features that sweeten the programming process but can be "de-sugared" down to core instructions.

Removing instructions. We can remove some instructions without crippling the model. For example, we express "xor" with "and", "or", and "not" by replacing "let $\text{Reg}[i] \leftarrow \text{Reg}[j] \oplus \text{Reg}[k]$" with instructions evaluating $(\text{Reg}[j] \wedge (\neg \text{Reg}[k])) \vee ((\neg \text{Reg}[j]) \wedge \text{Reg}[k])$ using auxiliary registers to hold intermediate results. The same goes for the logarithm instruction—we can compute $\lfloor \log_2(\text{Reg}[k]) \rfloor$ efficiently using auxiliary registers.

To completely purge "xor" and "log" from the model, these instructions shouldn't be reintroduced if we modify a program to use two registers and one-bit constants. The proof in § 1.5.2 creates no "xor" or "log" instruction that wasn't already there, so "xor" and "log" are indeed superfluous and can be safely removed from our instruction set.

We won't identify a "theoretically minimal" instruction set—that's off topic. The point is that we can push and pull the model definition in reasonable directions without significantly altering its capabilities. The concept of efficient computation is a sturdy facet of reality.

1.5.2 Number of Registers and Size of Constants

A *tight* program has only two registers and one-bit hardcoded constants (except target line numbers of branches).

Theorem 1.2. *For every program, there exists a tight program such that for every input with the original program's word size:*

- *Correctness: The programs produce the same output as each other (or both run forever).*
- *Efficiency: The new program's time and space are only constant factors larger than the original's, and the new word size is only an additive constant larger (where these constants depend on the original program).*

Proof idea. To get by with just two registers, we designate some words of work memory to hold the contents of the original program's registers, and just load or store a couple at a time, whenever the program needs to access them.

To get by with just one-bit constants, we use a bundle of instructions to construct any large constant in a register, one bit at a time, and then use the register in lieu of the constant.

These two issues seem independent but are interrelated: Decreasing the number of registers introduces large constants. Building a constant bit-by-bit requires an available register.

The modified program might need to access work segment addresses beyond the reach of the original word size. So we increase the word size. But that might change the program's behavior: If some instructions make 1s spill over into a word's new most significant bits, then the original program might have relied on those bits disappearing under the original word size. To preserve the program's behavior and output, we use extra instructions to "zero out" the new most significant bits whenever this complication might arise. Thus every word of memory (except the program counter) effectively has the original word size. ■

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

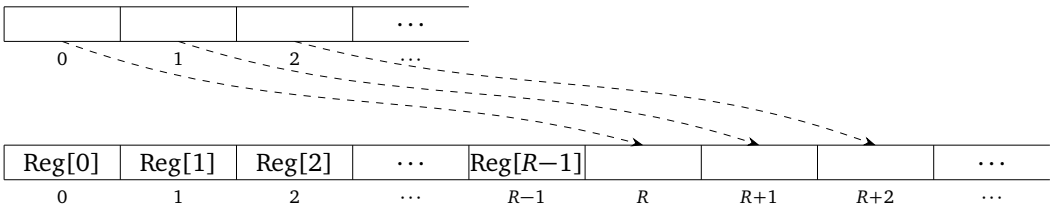


Figure 1.7: Scooting the work segment over to make room for registers

The formal proof is slightly laborious and skippable by anyone comfortable with the idea.

Proof (ancillary). First, we deal with the number of registers. Suppose the original program uses $R > 2$ many registers. The new program keeps those registers' contents in its work segment's first R words, and scoots the original program's work segment over to occupy the rest of the new program's work segment (Figure 1.7). The original Reg[i] resides at the new Mem[i] for each $i \in \{0, 1, \ldots, R-1\}$, and the original Mem[$i$] resides at the new Mem[$i + R$].

Since Reg[0] is initialized with the input size N, the new program's first instruction is "store Mem[0] ← Reg[0]". Then, each original instruction gets replaced as follows. (We only illustrate one data processing instruction and one control flow instruction; the rest are similar.)

let Reg[i] ← Reg[j] + Reg[k]	⇝	load Reg[0] ← Mem[j] load Reg[1] ← Mem[k] let Reg[0] ← Reg[0] + Reg[1] store Mem[i] ← Reg[0]
if Reg[i] = Reg[j] then goto line ℓ	⇝	load Reg[0] ← Mem[i] load Reg[1] ← Mem[j] if Reg[0] = Reg[1] then goto line $\lambda(\ell)$
load Reg[i] ← Mem[Reg[j]]	⇝	load Reg[0] ← Mem[j] let Reg[0] ← Reg[0] + R load Reg[0] ← Mem[Reg[0]] store Mem[i] ← Reg[0]
store Mem[Reg[j]] ← Reg[i]	⇝	load Reg[0] ← Mem[j] let Reg[0] ← Reg[0] + R load Reg[1] ← Mem[i] store Mem[Reg[0]] ← Reg[1]
read Reg[i] ← In[Reg[j]]	⇝	load Reg[0] ← Mem[j] read Reg[0] ← In[Reg[0]] store Mem[i] ← Reg[0]
write Reg[i]	⇝	load Reg[0] ← Mem[i] write Reg[0]

$\lambda(\ell)$ is the new line number (after all substitutions and the new first instruction) of the instruction from line ℓ of the original program. It's straightforward to handle hardcoded constants in lieu of registers. For example, in "let Reg[i] ← Reg[j] + Reg[k]" if there's a constant c instead of

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

Substitution	Constant	Construct in which register?
let	j, or in lieu of Reg[j]	Reg[0]
	k, or in lieu of Reg[k]	Reg[1]
	i	Reg[1]
if	i, or in lieu of Reg[i]	Reg[0]
	j, or in lieu of Reg[j]	Reg[1]
load	j, or in lieu of Reg[j]	Reg[0]
	R	Reg[1]
	i	Reg[1]
store	j, or in lieu of Reg[j]	Reg[0]
	R	Reg[1]
	i, or in lieu of Reg[i]	Reg[1]
read	j, or in lieu of Reg[j]	Reg[0]
	i	Reg[1]
write	i, or in lieu of Reg[i]	Reg[0]

Figure 1.8: Where to construct constants

Reg[j], then the substitution's third line becomes "let Reg[0] $\leftarrow c +$ Reg[1]" and we remove the first line "load Reg[0] $\leftarrow$ Mem[j]". The new program only uses two registers.

Now, we deal with the size of constants. We can construct any hardcoded constant c in a register using bitwise-or and shift left. First, we place c's most significant 1 in the register's least significant position. Then, we shift the register left until we're ready to insert c's next most significant 1 in the register's least significant position, by bitwise-or-ing with 1, and so on. Thus c "slides in" from the right. Since new constants can only be one bit, shifting left by more than one position is accomplished by repeatedly shifting left by one. Here's the example of constructing the binary number 11010 (26 in base ten):

$$
\begin{array}{lcl}
\text{let Reg[0]} \leftarrow 26 & \rightsquigarrow & \text{let Reg[0]} \leftarrow 1 \\
 & & \text{let Reg[0]} \leftarrow \text{Reg[0]} \ll 1 \\
 & & \text{let Reg[0]} \leftarrow \text{Reg[0]} \vee 1 \\
 & & \text{let Reg[0]} \leftarrow \text{Reg[0]} \ll 1 \\
 & & \text{let Reg[0]} \leftarrow \text{Reg[0]} \ll 1 \\
 & & \text{let Reg[0]} \leftarrow \text{Reg[0]} \vee 1 \\
 & & \text{let Reg[0]} \leftarrow \text{Reg[0]} \ll 1
\end{array}
$$

We do this for each hardcoded constant (except target line numbers) right before where it's used—including the constants i, j, k, R in our earlier substitutions such as "load Reg[0] $\leftarrow$ Mem[i]" and "let Reg[0] $\leftarrow$ Reg[0] $+ R$". We must be careful about which register to construct the constant in, since we only have two registers to work with (Figure 1.8). Now, each hardcoded constant is only one bit. We update the target line numbers in the branch substitutions after including all these instructions.

The original program's hardcoded constants fit in single words (by assumption), but the

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

constants i, j, k, R might not fit when N is small. Also, even if R fits, the address Reg[0] + R (in the load and store substitutions) might not fit, if the original program accesses addresses near the top of its range. So we increase the word size from W_N to $W_N + r$ where $r = \lceil \log(R+1) \rceil$.

To faithfully simulate the original program, the new program zeroes out the r most significant bits of each word in its work segment, leaving only the W_N least significant bits (as the original program expects for its registers and work segment). In the "let" substitution, after "let Reg[0] ← Reg[0] + Reg[1]" but before constructing i in Reg[1], we create a mask in Reg[1] that has 0s in the most significant r bits and 1s in the least significant W_N bits, and we bitwise-and the mask with the data word that's being assigned:

$$\begin{array}{l} \text{let Reg[1]} \leftarrow \neg 0 \\ \left.\begin{array}{l} \text{let Reg[1]} \leftarrow \text{Reg[1]} \gg 1 \\ \vdots \\ \text{let Reg[1]} \leftarrow \text{Reg[1]} \gg 1 \end{array}\right\} r \text{ times} \\ \text{let Reg[0]} \leftarrow \text{Reg[0]} \wedge \text{Reg[1]} \end{array}$$

Masking is unnecessary in the memory access substitutions—the word being stored/written was already masked since it came from the new program's work segment or input segment. (Each input word fits in W_N bits.)

This replicates the original program's register contents and work segment, since operations that might overflow are done "mod 2^W" anyway. (If the original program did two's complement for negative numbers, the new program does "two's complement in the least significant W_N bits.") The new computation proceeds as the original would have with word size W_N: Each comparison (in conditional branches) has the same outcome as in the original, and the output is the same, except each output word is left-padded with 0s.

Time only goes up by a constant factor since each instruction is replaced with a constant-size bundle of instructions. Space only goes up by a constant factor since the word size and number of words of work memory both go up by additive constants. ■

1.5.3 Memory Access

We observe three simple lemmas that will be handy later:

- A program might as well never read past the end of the input, because it would just get an all-0 word.
- If a program uses only a constant number of words of the work segment, then it might as well use none, since it could use extra registers instead.
- If a program uses multiple work segments, then it might as well use just one, which combines all the work segments.

Lemma 1.3. *For every program, there exists a program that never reads past the input, and such that for every input:*

- *Correctness: The programs produce the same output as each other (or both run forever).*
- *Efficiency: The new program's time and space are only constant factors larger than the original's, and the word size is unchanged.*

Proof. We introduce an auxiliary register nicknamed N and insert "let N ← Reg[0]" before the first instruction, to remember the input size in case the program overwrites Reg[0]. We do the following substitution (and update the target line numbers of all branches):

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

read Reg[i] ← In[Reg[j]]	⇝	if Reg[j] < N then goto 3 lines below let Reg[i] ← 0 goto 2 lines below read Reg[i] ← In[Reg[j]]

Time and space only go up by constant factors. ■

A *register-only* program has no load/store instructions and no work segment.

Lemma 1.4. *For every program and every constant M, there exists a register-only program such that for every input on which the original program doesn't access any work segment address $\geq M$:*

- *Correctness: The programs produce the same output as each other (or both run forever).*
- *Efficiency: The new program's time and space are only constant factors larger than the original's (where these constants depend on M), and the word size is unchanged.*

Proof. Suppose the original program has R registers. To modify it to a register-only program, we relocate the contents of Mem[0], ..., Mem[$M-1$] to new registers Reg[R], ..., Reg[$R+M-1$]. (This is the inverse of the idea behind Theorem 1.2, where we eliminated registers by relocating their contents to the work segment.) For $m \in \{0, 1, \ldots, M-1\}$, loading/storing Mem[$m$] corresponds to copying data from/to Reg[$R+m$]. Since the address m may be indirect (held in a register), we need "indirect indexing for registers," which we de-sugar like indirect branching:

load Reg[i] ← Mem[Reg[j]]	⇝	let Reg[i] ← Reg[R + Reg[j]]
	⇝	if Reg[j] ≠ m then goto 3 lines below let Reg[i] ← Reg[$R+m$] goto line after this whole substitution } ∀m
store Mem[Reg[j]] ← Reg[i]	⇝	let Reg[R + Reg[j]] ← Reg[i]
	⇝	if Reg[j] ≠ m then goto 3 lines below let Reg[$R+m$] ← Reg[i] goto line after this whole substitution } ∀m

Then, we update the target line numbers of preexisting branch instructions.

The addresses $m \in \{0, 1, \ldots, M-1\}$ become hardcoded constants as comparison operands. If $M-1$ doesn't fit in the word size when N is small, we get around this—without increasing the word size—using the same trick as for indirect branching (§1.5.1). Time and space only go up by constant factors depending on M. The new program faithfully simulates the original program if the latter doesn't access any work segment address $\geq M$. ■

Sometimes it's convenient to have multiple work segments Mem$_1$, Mem$_2$, ..., Mem$_K$, which are accessed using instructions like "load Reg[i] ← Mem$_k$[Reg[j]]". This feature helps us organize data, but it doesn't increase the model's power:

Lemma 1.5. *For every program with multiple work segments, there exists a program with one work segment, such that for every input with the original program's word size:*

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

$\text{Mem}_1[0]$	$\text{Mem}_2[0]$	$\cdots$	$\text{Mem}_K[0]$	$\text{Mem}_1[1]$	$\text{Mem}_2[1]$	$\cdots$	$\text{Mem}_K[1]$	$\text{Mem}_1[2]$	$\cdots$
0	1	$\cdots$	$K-1$	K	$K+1$	$\cdots$	$2K-1$	$2K$	$\cdots$

Figure 1.9: Interleaving K work segments into one

- *Correctness: The programs produce the same output as each other (or both run forever).*
- *Efficiency: The new program's time and space are only constant factors larger than the original's, and the new word size is only an additive constant larger (where these constants depend on the original number of work segments).*

Proof. A simple way to combine K work segments into one is to interleave them (Figure 1.9). For any $k \in \{1, \ldots, K\}$, an original load or store to $\text{Mem}_k[m]$ instead goes to $\text{Mem}[K \cdot m + k - 1]$, using an auxiliary register to calculate the new address. The catch is that although the original word size can hold m, it might not hold $K \cdot m + k - 1$. Adding $\lceil \log(K+1) \rceil$ to the word size fixes this. To preserve the program's behavior, we vigilantly zero out the new most significant bits after every "let" instruction (except those for calculating the address) as in the proof of Theorem 1.2. Time and space only go up by constant factors: The number of words of the new work segment is at most K times the maximum number of words of any original work segment. ■

1.5.4 Specifying and Changing the Word Size

A program's word size should provide appropriate "breathing room"—enough space per word to store basic data such as counters and pointers. If a program needs M words of work memory, then it needs word size $\geq \log M$ to fit load/store addresses in registers. Allowing a constant factor leeway in the word size is reasonable. For example, if a program uses $2^{\sqrt{N}}$ words of work memory, then word size $O(\sqrt{N})$ is appropriate.

Exactly which word size functions $W : \mathbb{N}_+ \to \mathbb{N}_+$ should we allow? An arbitrary function W contains infinite information since there are infinitely many input sizes N. But a program should have a finite specification, including the word size function. We restrict programs to use "simple" word sizes, so they can't smuggle hard-to-compute information via W. For example, $O(\log N)$ includes bizarre functions with no finite description, but we may require a program to use a very simple word size $c \log N$ for a positive integer constant c. Technically, $c \log N$ is shorthand for $c \cdot \max(\lceil \log_2(N) \rceil, 1)$. We allow arbitrarily large c so programs aren't constrained to use smaller words than would be natural.

In general, what is simple enough? Our definition will permit reasonable word sizes and prohibit exotic word sizes that let contraband information seep into a computation. The idea is that $W(N)$ should be efficiently computable from N. Informally, W is (T, S)-*constructible* iff some *constructor* program computes W in time T and space S. There's a "chicken and egg" conundrum: What word size do we use when computing the word size? We require a constructor to be oblivious to its own word size, so if we run it with a bigger word size, it does the same thing and still produces $W(N)$. We imagine each register has unlimited size, but only bits that are "used" count toward the space. This precludes computations that find a register's left end by exploiting modular arithmetic or bitwise-not.

Definition. W is (T, S)-*constructible* iff there exists a program such that:

- There are no load/store/read/write/accept/reject instructions.
- Each register can hold any nonnegative integer.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

- With Reg[0] initialized to N and all other registers initialized to 0:
  - It halts with $W(N)$ in Reg[1].
  - The running time is at most $T(N)$.
  - At every step, each register's value fits in the least significant $S(N)$ bits (and so does any hardcoded constant). ◀

If we run such a program using registers of any finite size $\geq S(N)$, it "won't notice" and will still construct $W(N)$.

Example. $W = \max\big(2\lceil\log(N+1)\rceil, 7\big)$ is $(O(1), O(\log N))$-constructible by the following constructor program. Since our model has $\lfloor\log_2\rfloor$ but not $\lceil\log_2\rceil$, line 3 checks whether $N+1$ is a power of 2, and if not, then line 4 adds 1 to the floor to get the ceiling. Line 6 does the max.

Example constructor (1.13)

```
0. let Reg[2] ← Reg[0] + 1
1. let Reg[1] ← ⌊log2(Reg[2])⌋
2. let Reg[3] ← 1 ≪ Reg[1]
3. if Reg[3] = Reg[2] then goto line 5
4. let Reg[1] ← Reg[1] + 1
5. let Reg[1] ← 2 · Reg[1]
6. if Reg[1] ≥ 7 then goto line 8
7. let Reg[1] ← 7
8. halt
```

The time is $O(1)$. The space is $O(\log N)$ since no register exceeds $N+1$, and hardcoded constants are 3 bits. ⧫

Example.

- N^d is $(O(1), O(\log N))$-constructible for every integer d.
- $\lceil\sqrt{N}\rceil$ is $(O(\log N), O(\log N))$-constructible by binary search (comparing squared candidate values to N).
- $\lceil N^d\rceil$ is $(O(\log N), O(\log N))$-constructible for every rational d, similarly.
- 2^N is $(O(1), O(N))$-constructible.
- 3^N is $(O(\log N), O(N))$-constructible by the well-known "repeated squaring" algorithm. ⧫

The definition of constructibility is somewhat complicated but is fortunately "self-obviating": Any program's constructible word size can be rounded up to a very simple word size with little impact on the program's efficiency. Increasing the word size is even relevant in practice: A 64-bit machine can't "just run" a 32-bit program—it needs a "compatibility mode," which incurs some overhead. Here is "compatibility mode for theory," showing that our model is robust with respect to increasing the word size.

Lemma 1.6. *For every program with (T, S)-constructible word size W, and every $U \geq \max(W, S)$, there exists a program with word size U such that for every input with word size W:*

- *Correctness: The programs produce the same output as each other (or both run forever).*
- *Efficiency: If the original program's time and space are T^* and S^*, then the new program's time and space are only constant factors larger than $T^* + T$ and $S^* \cdot U/W$.*

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

Proof idea. Constructibility of W lets the new program create a mask to "zero out" all but the least significant W bits of each word as it runs the original program. So although the word size is $U \geq W$, the program gets executed as if the word size were W. ■

Proof. Abbreviate $W(N)$ as W, and so on. The new program first runs W's constructor program using separate registers, after copying N from Reg[0] to the register corresponding to the constructor's Reg[0]. The constructor works since $U \geq S$. After constructing W, the new program evaluates $(1 \ll W) - 1$ to create a mask in an auxiliary Reg[h], with 1s in the least significant W bits and 0s in the most significant $U - W$ bits. Then, the new program runs the original program but after every "let Reg[i] $\leftarrow \ldots$", it inserts "let Reg[i] $\leftarrow$ Reg[i] $\wedge$ Reg[h]" to zero out the bits that wouldn't exist under the original word size W. This makes the word size effectively W, so the original program's behavior and output are preserved.

The new time is $\leq T + 3 + 2T^*$. The original program contributes $S^* \cdot U/W$ to the new space because the new program uses the same number of words of memory as the original, but now each word has U bits instead of W bits. The constructor's registers and Reg[h] take $O(U)$ space, which doesn't dominate the overall space, since $U \leq S^* \cdot U/W$ follows from $S^* \geq W$. ■

Takeaway: A program's exact word size doesn't matter, as long as it's not a covert source of information. The cumbersome definition of constructibility renders itself unnecessary.

1.6 Complexity Classes

Complexity classes "classify" computational problems according to how efficiently they can be solved. A complexity class is the set of all problems solvable within a certain efficiency constraint. If a particular problem is an element of a particular complexity class, then the problem has low complexity in a sense. If the problem isn't in the class, it has high complexity. Most complexity classes only contain decision problems, to capture the essence of efficient computation while avoiding minor hassles associated with function/search problems.

We use SANS SERIF CAPITALS for complexity class names. TIME[T] contains every decision problem with a $O(T(N))$-time program, and SPACE[S] contains every decision problem with a $O(S(N))$-space program. We'll see the formal definitions soon.

A program's word size should be small and constructible, so the program can't hide too much computation in individual instructions and in the word size specification. We combine these requirements into a streamlined definition: W is *U-bounded* iff $W \leq U$ and W is (U, U)-constructible (meaning $W(N)$ is computable from N in time U and space U).

A program's U-bounded word size can be rounded up to U with little overhead in time and space (Lemma 1.6). A common case is that a program's *log-bounded* (which means $O(\log N)$-bounded) word size can be rounded up to $c \log N$ for an integer c: Time only goes up by a constant factor plus $O(\log N)$. Space only goes up by a constant factor since the word size was already $\geq \log N$ (since N fits in a register).

All logarithms are base 2 unless specified otherwise, and we usually omit ceilings and floors when they're unimportant, such as pretending $\log N$ is always an integer.

1.6.1 Time Classes

For time T, we allow word size $\geq \log T$ so words can hold values up to T, such as counters and pointers. A constant factor leeway in the word size is reasonable, so we allow $O(\log T)$-bounded

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

word size. If $T < N$, we allow $O(\log N)$-bounded word size so words can hold pointers to the input, and Reg[0] can hold N. For example, binary search (Exercise 1.2) takes $O(\log N)$ time but needs word size $O(\log N)$, not $O(\log(\log N))$. A problem's word size need not be constructible but must be $O(\log N)$ and at most the program's word size.

Definition. For every function $T(N)$, $\mathsf{TIME}[T]$ is the set of all decision problems for which there exists a correct program with $O(T)$ time efficiency and $O(\log \max(T, N))$-bounded word size. ◄

We showed Multiplication Decision $\in \mathsf{TIME}[N^2]$ and Directed Reachability $\in \mathsf{TIME}[N]$. Sometimes people get sloppy and write Multiplication $\in \mathsf{TIME}[N^2]$, but technically that's wrong since Multiplication isn't a decision problem.

$\mathsf{TIME}[\text{a set of functions}]$ is the union of $\mathsf{TIME}[T]$ over all T in the set. Let

$$\operatorname{poly} N = \left\{T : \text{there exists a constant } d \text{ such that } T = O(N^d)\right\}$$

be the set of functions that only grow polynomially fast. This includes functions that aren't polynomials per se but are at most a polynomial, such as $N \log N \le N^2$.

$$\mathsf{TIME}[\operatorname{poly} N] = \bigcup_{T \in \operatorname{poly} N} \mathsf{TIME}[T] = \bigcup_d \mathsf{TIME}[N^d]$$
$$\mathsf{TIME}[2^{\operatorname{poly} N}] = \bigcup_{T \in \operatorname{poly} N} \mathsf{TIME}[2^T] = \bigcup_d \mathsf{TIME}[2^{O(N^d)}]$$

These two classes have special names:

Definition.

- $\mathsf{P} = \mathsf{TIME}[\operatorname{poly} N]$ (Polynomial time)
- $\mathsf{EXP} = \mathsf{TIME}[2^{\operatorname{poly} N}]$ (Exponential time) ◄

We often shorten "polynomial" to "poly": $A \in \mathsf{P}$ iff A is a decision problem with a poly-time program—which has $O(N^d)$ time efficiency (for a constant d) and log-bounded word size (which we may assume is $c \log N$ for an integer c). Likewise, $A \in \mathsf{EXP}$ iff A is a decision problem with an exponential-time program—which has $2^{O(N^d)}$ time efficiency (for a constant d) and poly-bounded word size (which we may assume is cN^e for integers c and e). Not all decision problems are in EXP: Some problems require superexponential time, and some can't be solved by any program at all, regardless of efficiency.

If A is a function/search problem, we can't say $A \in \mathsf{P}$ or $A \in \mathsf{EXP}$, but we can say A has a poly-time program (with log-bounded word size) or exponential-time program (with poly-bounded word size).

An *inclusion* is a complexity class being a subset of another. A *separation* is a complexity class being *not* a subset of another. The inclusion $\mathsf{P} \subseteq \mathsf{EXP}$ holds because any program showing $A \in \mathsf{P}$ also shows $A \in \mathsf{EXP}$, since polynomials grow slower than exponentials. (We'll see many inclusions in this book, but proving them usually involves modifying the program.) We prove the separation $\mathsf{EXP} \not\subseteq \mathsf{P}$ in §4.2 by proving a superpolynomial time lower bound for a problem in EXP. Thus the *strict inclusion* $\mathsf{P} \subsetneq \mathsf{EXP}$ holds.

Another named complexity class is $\mathsf{E} = \mathsf{TIME}[2^{O(N)}]$ (*linear exponential* time). Though $\mathsf{TIME}[T]$ has an implicit big-O, we make the big-O explicit in the exponent because $2^{O(N)}$ is not $O(2^N)$. We won't study E often in this book. Let

$$\operatorname{polylog} N = \left\{T : \text{there exists a constant } d \text{ such that } T = O(\log^d N)\right\}$$

be the set of functions that only grow polylogarithmically fast. These are subpolynomial since $\log^d N = o(N^e)$ for all constants d (however large) and $e > 0$ (however small). We generally deem polylog time factors insignificant, so we dub $\mathsf{TIME}[N \operatorname{polylog} N]$ *quasi-linear* time.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

1.6.2 Why P?

P is the most important class—the central hub of complexity theory. A *tractable* problem has an "efficient" algorithm, and the most common definition of tractability is having a poly-time algorithm. Multiplication Decision and Directed Reachability are tractable. One reason to focus on P is because it's more robust than the TIME[T] classes. TIME[T] is sensitive to:

- The input/output encoding: For example, a graph problem might be in TIME[N] with adjacency matrix encoding but not with adjacency list encoding.
- The model of computation: For example, a problem might be in TIME[N] when the model has a log instruction but not otherwise.

P is far less finicky: Expelling a problem from P would require drastic changes to the encoding or the model. For example, only polynomial slowdown is incurred by converting adjacency list to adjacency matrix, or by computing logs using other instructions. EXP is similarly robust.

Like any celebrity, P has detractors. Criticisms leveled at P include:

"Allowing arbitrarily high-degree polynomials is too generous." A $O(N^{100})$-time algorithm is hopelessly impractical even on tiny inputs, but the problem it solves is in P. A counterpoint is that such algorithms are rarely the fastest in practice: Almost every useful problem known to be in P is also known to be in TIME[N^6] or better. There's a big theoretical chasm between polynomial and superpolynomial time. If a $O(N^{100})$-time algorithm is discovered for a problem previously only known to be in EXP, then a deep insight about the problem's structure must have been revealed. Once that cat is out of the bag, researchers typically capitalize on the new understanding to significantly improve the time efficiency. A similar criticism is that even with $O(N)$ time, big-O could conceal a terrible constant like 2^{100}. But it's rarely so bad in practice.

"Requiring worst-case performance is too restrictive." A poly-time algorithm must be correct and fast on every single valid input. But an algorithm that's wrong or slow on some contrived inputs might work great on real-world inputs. By declaring bad inputs invalid, we'd have a promise problem in P, but that's not very satisfactory because a problem shouldn't be a moving target. Instead, *average-case complexity* (Chapter 10) is about algorithms that work on "typical" inputs, without specifying which inputs those are.

"It doesn't account for randomness or other physical resources." Randomized algorithms (Chapter 7) produce correct output with high probability, which may be good enough in practice. But it's conjectured that every randomized poly-time algorithm can be converted to a deterministic (non-randomized) poly-time algorithm (Chapter 11). A bigger threat to P's status as the definition of tractability comes from bizarre properties of quantum physics. Counterintuitive behavior of subatomic particles could be harnessed to solve some problems much faster than any deterministic or randomized algorithm. Practical realization of quantum computing remains a major engineering challenge. Quantum complexity is beyond the scope of this book.

1.6.3 Space Classes

For space S, we allow word size $\geq \log S$ so words can hold values up to S, such as counters and pointers. A constant factor leeway in the word size is reasonable, so we allow $O(\log S)$-bounded word size. If $S < N$, we allow $O(\log N)$-bounded word size so words can hold pointers to the input, and Reg[0] can hold N.

Definition. For every function $S(N)$, SPACE[S] is the set of all decision problems for which there exists a correct program with $O(S)$ space efficiency and $O(\log \max(S, N))$-bounded word size. ◄

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

We showed Multiplication Decision $\in$ SPACE$[\log N]$ and Directed Reachability $\in$ SPACE$[N \log N]$. We prove Directed Reachability $\in$ SPACE$[\log^2 N]$ in Chapter 3.

Definition.

- $\mathsf{L} = \mathsf{SPACE}[\log N]$ (Logarithmic space)
- $\mathsf{PSPACE} = \mathsf{SPACE}[\mathrm{poly}\, N]$ (Polynomial space) ◀

$A \in \mathsf{L}$ iff A is a decision problem with a log-space program—which has $O(\log N)$ space efficiency and log-bounded word size (which we may assume is $c \log N$ for an integer c). Likewise, $A \in \mathsf{PSPACE}$ iff A is a decision problem with a poly-space program—which has $O(N^d)$ space efficiency (for a constant d) and log-bounded word size (which we may assume is $c \log N$ for an integer c). We won't study $\mathsf{EXPSPACE} = \mathsf{SPACE}[2^{\mathrm{poly}\, N}]$.

$\mathsf{SPACE}[S]$ is meaningful when $S \geq \log N$, since a program's word size is necessarily $\geq \log N$. A log-space program can only use a constant number of words of the work segment (on valid inputs) and therefore might as well be register-only (with no work segment and no load/store instructions) by Lemma 1.4. This clean characterization is worth highlighting:

Observation 1.7. L *is the set of all decision problems for which there exists a correct register-only program with log-bounded word size.* ■

(This ■ means no separate proof is needed.)

Like with P and EXP: If A is a function/search problem, we can't say $A \in \mathsf{L}$ or $A \in \mathsf{PSPACE}$, but we can say A has a log-space or poly-space program (with log-bounded word size). The strict inclusion $\mathsf{L} \subsetneq \mathsf{PSPACE}$ holds because the inclusion $\mathsf{L} \subseteq \mathsf{PSPACE}$ is trivial and we prove the separation $\mathsf{PSPACE} \not\subseteq \mathsf{L}$ in §4.2. The classes L and PSPACE are robust with respect to the encoding and the model, though for L the reason is less clear and we postpone discussion to §1.9.

Now, we consider time and space together. $\mathsf{TISP}[T, S]$ contains every decision problem with a program that uses $O(T)$ time and $O(S)$ space. For example, Multiplication Decision $\in \mathsf{TISP}[N^2, \log N]$. To contrast $\mathsf{TISP}[T, S]$ with $\mathsf{TIME}[T] \cap \mathsf{SPACE}[S]$:

- $A \in \mathsf{TISP}[T, S]$ means some program for A uses $O(T)$ time and $O(S)$ space.
- $A \in \mathsf{TIME}[T] \cap \mathsf{SPACE}[S]$ means some program for A uses $O(T)$ time and some possibly different program for A uses $O(S)$ space.

Presumably $\mathsf{TIME}[T] \cap \mathsf{SPACE}[S] \not\subseteq \mathsf{TISP}[T, S]$: We can't always have the best of both worlds. But $\mathsf{TISP}[T, S] \subseteq \mathsf{TIME}[T] \cap \mathsf{SPACE}[S]$ should hold: $\mathsf{TISP}[T, S] \subseteq \mathsf{TIME}[T]$ by ignoring space, and $\mathsf{TISP}[T, S] \subseteq \mathsf{SPACE}[S]$ by ignoring time. To ensure this, the word size should be both $O(\log \max(T, N))$-bounded and $O(\log \max(S, N))$-bounded.

Definition. For all functions $T(N)$ and $S(N)$, $\mathsf{TISP}[T, S]$ is the set of all decision problems for which there exists a correct program with $O(T)$ time efficiency, $O(S)$ space efficiency, and $O(\log \max(\min(T, S), N))$-bounded word size. ◀

If $T \in \mathrm{poly}\, N$ and $S \in \mathrm{poly}\, N$, we may assume the word size is $c \log N$ for an integer c. If sets of functions stand in for T and S, then the TISP class is the union of $\mathsf{TISP}[T, S]$ over all pairs (T, S) from the respective sets. For example,

$$\mathsf{TISP}[\mathrm{poly}\, N, \mathrm{polylog}\, N] = \bigcup_{(T,S) \in (\mathrm{poly}\, N) \times (\mathrm{polylog}\, N)} \mathsf{TISP}[T, S]$$

which equals $\bigcup_d \mathsf{TISP}[N^d, \log^d N]$ by rounding both exponents up to the same constant d.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

1.7 Time Versus Space

In this section, we prove inclusions that relate time and space:

- We convert space to time with an exponential blowup: $\mathsf{SPACE}[S] \subseteq \mathsf{TIME}[2^{O(S)}]$, so $\mathsf{L} \subseteq \mathsf{P}$ and $\mathsf{PSPACE} \subseteq \mathsf{EXP}$.
- We convert time to space with a log factor blowup: $\mathsf{TIME}[T] \subseteq \mathsf{SPACE}[T \log T]$ if $T \geq N$, so $\mathsf{P} \subseteq \mathsf{PSPACE}$.

The four main robust classes form complexity theory's "backbone":

$$\mathsf{L} \subseteq \mathsf{P} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP}$$

Each of these three inclusions is widely believed to be strict, but these conjectures all remain unproven. The separation $\mathsf{P} \neq \mathsf{EXP}$ leaves a frustrating but amusing state of affairs:

- Most researchers conjecture that both $\mathsf{P} \neq \mathsf{PSPACE}$ and $\mathsf{PSPACE} \neq \mathsf{EXP}$ hold.
- We know for sure at least one of them holds, but with no indication of which one.
- Proving either one would be an earth-shattering achievement. ("Jaw-dropping" is too mild a description.) Nobody is aware of any viable approach that might work any time soon.

An analogous situation comes from $\mathsf{L} \neq \mathsf{PSPACE}$ implying that either $\mathsf{L} \neq \mathsf{P}$ or $\mathsf{P} \neq \mathsf{PSPACE}$.

In §4.1 we prove another result about the backbone: If $\mathsf{L} = \mathsf{P}$ then $\mathsf{PSPACE} = \mathsf{EXP}$. The converse is not known to hold, though.

1.7.1 From Space to Time

Theorem 1.8. $\mathsf{SPACE}[S] \subseteq \mathsf{TIME}[2^{O(S)}]$ *for every* $S \geq \log N$.

Proof idea. For any program that uses $O(S)$ space, if it terminates at all, then it must terminate within $2^{O(S)}$ steps. We show the contrapositive. At any moment, the computation is in one of only $2^{O(S)}$ possible "configurations." If it hasn't halted within $2^{O(S)}$ steps, then by the pigeonhole principle, it must have repeated some configuration. Once the program experiences such déjà vu, it's doomed to repeat history forever: Upon reaching some configuration a second time, the computation will undergo the same sequence of steps as it did after the first time, until it reaches that configuration a third time ("déjà vu all over again"), and so on, ad infinitum. ■

The concept of configurations is useful throughout this book:

Definition. Consider a program with L lines of code, R registers, and word size W. Suppose on a particular input, the program never accesses any work segment address $\geq M$, so the memory space is $\leq (R+M)W$. A *configuration* consists of a line number and contents of all R registers and the work segment's first M words:

$$\big(\ell, \mathrm{Reg}[0], \ldots, \mathrm{Reg}[R-1], \mathrm{Mem}[0], \ldots, \mathrm{Mem}[M-1]\big) \in \{0,1\}^{\log L} \times (\{0,1\}^W)^R \times (\{0,1\}^W)^M$$

We assume L is a power of 2 (by appending dummy instructions if necessary), so $\{0,1\}^{\log L}$ is the set of all line numbers. ◀

A configuration is a snapshot of the state of a computation (including the program counter) at a moment in time. It's like "full debug info": If we pause a particular program running on a particular input, then the current configuration contains all information needed to resume the

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

execution where we left off. The next configuration just depends on the program, input, and current configuration. (The program and input aren't part of a configuration because they don't change during the execution.)

Proof of Theorem 1.8. Suppose $A \in \mathsf{SPACE}[S]$ by a program Π with $O(S)$ space efficiency and $O(\log \max(S, N))$-bounded word size. Say Π has L lines of code, R registers, and word size W. Consider any valid input x of size N, and let $M-1$ be the largest address accessed by a load or store on input x (or $M = 0$ if no load or store is executed). The memory space is $(R+M)W \leq cS$ for some constant c. Since a configuration has $\log L + (R+M)W \leq \log L + cS$ bits, there are $T = L2^{cS}$ possible configurations.

Since Π is correct on input x, it eventually halts. We claim that it halts within $T-1$ steps. Suppose for contradiction it executes at least T steps. Let $C_0, C_1, C_2, \ldots$ be the sequence of configurations the computation goes through, where C_0 is the initial configuration and C_t is the configuration after step t. Since this sequence has length $> T$ but each configuration comes from a set of size $\leq T$, we have $C_{t'} = C_t$ for some $t' > t$. Since the computation knows only the current configuration and not how it got there, it proceeds after step t' exactly as it did after step t. That is, $C_{t'+1} = C_{t+1}$ and then $C_{t'+2} = C_{t+2}$ and so on, up to $C_{t'+(t'-t)} = C_{t+(t'-t)} = C_{t'} = C_t$. Thus after step $t' + (t'-t)$ it again repeats the same cycle of configurations, returning to $C_{t'+2(t'-t)} = C_t$. Repeating this reasoning shows that $C_{t+k(t'-t)} = C_t$ for every integer $k \geq 0$ and thus Π runs forever on x, which is a contradiction.

Π has $2^{O(S)}$ time efficiency since it halts within $L2^{cS} - 1$ steps for all valid inputs of size N. Since W is trivially also $O(S)$-bounded (as $S \geq \log N$), we conclude $A \in \mathsf{TIME}[2^{O(S)}]$. ■

1.7.2 From Time to Space

Since load/store addresses must fit in single words, a program with word size $O(\log T)$ can only access work segment addresses up to $2^{O(\log T)} = \text{poly}\, T$. This implies $\mathsf{TIME}[T] \subseteq \mathsf{SPACE}[\text{poly}\, T]$ if $T \geq N$, which already yields $\mathsf{P} \subseteq \mathsf{PSPACE}$. We prove a stronger inclusion:

Theorem 1.9. $\mathsf{TIME}[T] \subseteq \mathsf{SPACE}[T \log T]$ *for every* $T \geq N$.

Proof idea. A program can load/store only one word per step, so if it runs in time $O(T)$, then it accesses only $O(T)$ addresses. This is a total of $O(T \log T)$ bits of memory since the word size is $O(\log T)$. If it accessed only the first $O(T)$ work segment addresses, we'd be done. But it may access some higher addresses, leaving gaps of never-accessed addresses. Since our definition of memory space is based on the largest address accessed, the program's space efficiency might not be $O(T \log T)$. We modify the program so it solves the same problem but is frugal with space. We consolidate the accessed work memory so there's no unused space. This is like fragmentation issues that operating systems contend with. We "defragment" by rearranging the accessed memory words into a contiguous block. When the original program wants to load/store a word, the modified program figures out the new address where that word is kept. This is a job for a *dictionary* data structure, which maintains a set of (key, value) pairs such that each possible key is in at most one pair, and which supports these operations:

- Lookup: Given a key k, find k's associated value v, or report that k isn't present.
- Update: Given a key k and a value v, change k's associated value to v, or insert (k, v) if k wasn't already present.

For our defragmentation, a key is a work segment address, and its associated value is the word that would reside there for the original program. Load = lookup, and store = update.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

Time efficiency doesn't matter for this theorem, so for the dictionary, we just use an unordered list of (key, value) pairs. Each operation iterates through the list looking for the desired key. If an update doesn't find the key, then it appends the (key, value) pair to the list. ■

Proof. Suppose $A \in \mathsf{TIME}[T]$ by a program Π with $O(T)$ time efficiency and $O(\log T)$-bounded word size (since $T \geq N$). We turn Π into a program Π' with the same code except for load/store substitutions (and updated target line numbers of branches). Π' views its work segment as a list of *items*, each of which is a pair of words: a work segment address (*key*) and what Π would have at that address (*value*). The keys are ordered according to the first time at which Π stores to that address. For example, after Π stores $\mathrm{Mem}[8] \leftarrow 1$, then $\mathrm{Mem}[6] \leftarrow 2$, then $\mathrm{Mem}[8] \leftarrow 3$, then $\mathrm{Mem}[7] \leftarrow 4$, then $\mathrm{Mem}[6] \leftarrow 5$, the work segment of Π' would contain $(8, 3, 6, 5, 7, 4, \ldots)$.

Π' uses three extra registers: the length of the list, a pointer for iterating through the list, and a key from the list (to compare against the desired key). For each "load $\mathrm{Reg}[i] \leftarrow \mathrm{Mem}[\mathrm{Reg}[j]]$" and "store $\mathrm{Mem}[\mathrm{Reg}[j]] \leftarrow \mathrm{Reg}[i]$" in Π, Π' instead iterates through all items until it finds a key that equals $\mathrm{Reg}[j]$, and then:

- "load": Π' loads the associated value into $\mathrm{Reg}[i]$, from the address after where the key is. If no key equals $\mathrm{Reg}[j]$ (Π is loading from an address it never stored to), Π' assigns 0 to $\mathrm{Reg}[i]$.
- "store": Π' stores $\mathrm{Reg}[i]$ to the associated value, at the address after where the key is. If no key equals $\mathrm{Reg}[j]$, Π' stores the words $(\mathrm{Reg}[j], \mathrm{Reg}[i])$ as a new item at the end of the list, and updates the "list length" register.

Π' solves A because it faithfully simulates Π. It has $O(T \log T)$ space efficiency because it only accesses the first $O(T)$ work segment addresses (at most two addresses per original store) and the word size is $O(\log T)$.

There's a slight catch: Since Π' accesses twice as many words of work memory as Π, the addresses might not quite fit in the word size. But if Π' notices that they don't fit, then only the first $O(T)$ words of work memory are addressable anyway, in which case Π' can run Π unmodified, knowing that it already uses $O(T \log T)$ space. So it's not necessary to change the word size.

Of course the $O(\log T)$-bounded word size is also $O(\log(T \log T))$-bounded, so we conclude $A \in \mathsf{SPACE}[T \log T]$. ■

If $T < N$, the same proof yields $\mathsf{TIME}[T] \subseteq \mathsf{SPACE}[T \log N]$ since the word size is $O(\log N)$.

1.7.3 Simultaneous Time and Space Bounds

$\mathsf{SPACE}[S] \subseteq \mathsf{TISP}[2^{O(S)}, S]$ because in the proof of Theorem 1.8, the program doesn't change. $\mathsf{TIME}[T] \subseteq \mathsf{TISP}[T^2, T \log T]$ because in the proof of Theorem 1.9, for each of the $O(T)$ original loads and stores, the modified program may look through $O(T)$ items to find the desired key, resulting in $O(T^2)$ time efficiency. This can be improved to $\mathsf{TIME}[T] \subseteq \mathsf{TISP}[T \log T, T \log T]$ using a more sophisticated dictionary (a self-balancing search tree). We prove a slightly weaker version with time $O(T \log^2 T)$ instead of $O(T \log T)$. This is good enough for our purposes (since we usually won't care about polylog time factors), and the proof is elementary and self-contained.

Theorem 1.10. $\mathsf{TIME}[T] \subseteq \mathsf{TISP}[T \log^2 T, T \log T]$ *for every* $T \geq N$.

Proof idea. Suppose in the proof of Theorem 1.9, we change the dictionary to keep the list of items sorted in increasing order of the keys. A load takes $O(\log T)$ time, using binary search to

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

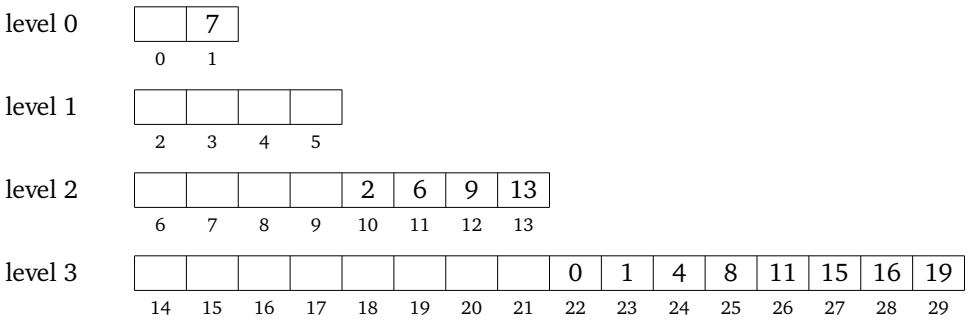


Figure 1.10: Levels of slots

locate the desired key. The time for a store depends on whether the store is *fresh*, meaning it's the original program's first store to this address. A nonfresh store takes $O(\log T)$ time. A fresh store still takes $O(T)$ time because after determining where the key should be, we must shift over all subsequent items (with larger keys) to make room to insert the new item: We need "insertion sort" to maintain the invariant that the keys are sorted.

A sorted list is good for loads and nonfresh stores, while an unsorted list has an advantage for fresh stores—appending to the list only takes $O(1)$ time. We combine and balance these approaches to make all loads and stores fast. We use a "list of lists" where each inner list is sorted but the outer list has no particular order. For example, we could have $O(\sqrt{T})$ inner lists, each of length $O(\sqrt{T})$. Then a load or nonfresh store takes $O(\sqrt{T}\log T)$ time (using binary search on each inner list), and a fresh store also takes $+\,O(\sqrt{T})$ time to insert into one of the inner lists. Thus $O(T)$ loads and stores take total time $O(T^{1.5}\log T)$ and space $O(T\log T)$.

We do even better with fewer inner lists, of varying lengths, and imitating "merge sort" (rather than insertion sort) to maintain sortedness after an insertion. This doesn't achieve $O(\log^2 T)$ time for every load/store, but it achieves $O(\log^2 T)$ time averaged over all loads/stores—this style of reasoning is called *amortized analysis*. ■

Proof (ancillary). We turn any program Π with $O(T)$ time efficiency and $O(\log T)$-bounded word size into an equivalent program Π' with $O(T\log^2 T)$ time efficiency, $O(T\log T)$ space efficiency, and the same word size. We just change the loads and stores.

We partition the work segment of Π' into *slots*, where slot $i \in \mathbb{N}$ is the pair of addresses $2i$ and $2i+1$. A slot can either be empty or hold an item, which is a (key, value) pair representing an address and the word Π would have at that address in its work segment. We worry later about how to tell whether a slot is empty.

We partition the sequence of slots into *levels*, where level $\ell \in \mathbb{N}$ contains $2\cdot 2^\ell$ many slots (specifically, slots $2^{\ell+1}-2,\ldots,2^{\ell+2}-3$). Π' maintains the invariant that for each level, either all slots are empty, or the first half of the slots are empty and the second half are full and have keys in increasing order. Thus if level ℓ is nonempty, it has exactly 2^ℓ items. Figure 1.10 shows an example, where the little numbers underneath are slot numbers, and each nonempty slot shows only a key (with no value).

The invariant ensures that Π' uses only the first $O(T)$ words of its work segment. This is because if L is the greatest level that's ever nonempty, then the number of keys is $\geq 2^L$ but also $\leq O(T)$, so $2^L \leq O(T)$. The highest slot number is $2^{L+2}-3$, and the highest work segment address is $2(2^{L+2}-3)+1 \leq O(T)$. Since the word size is $O(\log T)$, the space is $O(T\log T)$.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

Π' uses a register to remember how many keys are currently in the dictionary. This number's bits indicate which levels are nonempty: If the ℓth bit (0th = least significant) is 1 then level ℓ is half-full, and if the ℓth bit is 0 then level ℓ is empty. Thus we can tell whether a particular slot is empty in $O(1)$ time by examining a bit of this register. Junk can be left in empty slots.

To accomplish "load Reg[i] ← Mem[Reg[j]]" or "store Mem[Reg[j]] ← Reg[i]", Π' first locates the key Reg[j] by doing binary search on each nonempty level. If the key is found, its associated value is copied to or from Reg[i]. Otherwise, a load assigns 0 to Reg[i], and a fresh store inserts the item (Reg[j], Reg[i]) into the dictionary somehow. The number of nonempty levels is $\le L + 1 \le O(\log T)$, and each binary search takes $O(\log T)$ time since each level has $\le O(T)$ keys. So the time per original load/store is $O(\log^2 T)$ except for a fresh store's insertion.

To insert a (k, v) pair (assuming the key k isn't already present), Π' uses *cascade insertion*. If level 0's second half is empty, we put (k, v) there. Otherwise, we put (k, v) in level 0's first half, which breaks the invariant, and then we rearrange items to restore the invariant. This pseudocode moves keys around, but their associated values are implicitly moved with the keys.

Cascade insertion algorithm (1.14)

```
if level 0 is empty: put k in slot 1 and stop
put k in slot 0
for ℓ ← 0, 1, 2, ... until level ℓ + 1 is empty:
    merge the two halves of level ℓ into the first half of level ℓ + 1:
        initialize i ← 1; let a_1 < a_2 < ··· < a_{2^ℓ} denote the keys of the first half of level ℓ
        initialize j ← 1; let b_1 < b_2 < ··· < b_{2^ℓ} denote the keys of the second half of level ℓ
        for h ← 1, 2, ..., 2^{ℓ+1}:
            if i ≤ 2^ℓ and either j > 2^ℓ or a_i < b_j:
                move a_i to the hth slot of level ℓ + 1; increment i
            else:
                move b_j to the hth slot of level ℓ + 1; increment j
merge the two halves of level ℓ into the second half of level ℓ + 1
```

Figure 1.11 shows an example.

For intuition, when we increment the "number of keys" register, the least significant 1s all get flipped to 0s until we encounter a 0, which becomes a 1. So when rearranging items, levels $0, 1, 2, \ldots$ should become empty until we encounter an already-empty level, which becomes half-full with all the items from previous levels along with the fresh item. (Recall $2^{\ell+1} = (2^0 + 2^1 + \cdots + 2^\ell) + 1$.) Making level 0 full triggers a cascade where a full level gets moved (and sorted in the process) to the first half of the next level, thus making that level full, and so on. For each round of the cascade, both halves of the full level are already sorted, so we just have to merge them (like in merge sort): We repeatedly compare the smallest remaining first-half key with the smallest remaining second-half key, and move the smaller of the two to the next slot in the next level (and when one half is exhausted, we just move the rest of the other half).

A cascade insertion can be very time-consuming if the first empty level is far down, so a fresh store doesn't always take $O(\log^2 T)$ time. But we prove that the total time of the cascade insertions, over all fresh stores, is $O(T \log T)$: There are $O(T)$ keys and $O(\log T)$ levels. Each key progresses through the levels but never goes backward to a previous level. Thus the total amount of time spent moving any particular key is $O(\log T)$. The key might be compared to many other keys, but we "charge" a comparison to whichever key gets moved. More precisely, we partition the time steps of all cascade insertions into phases, using "key moves" as boundaries. Each phase

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

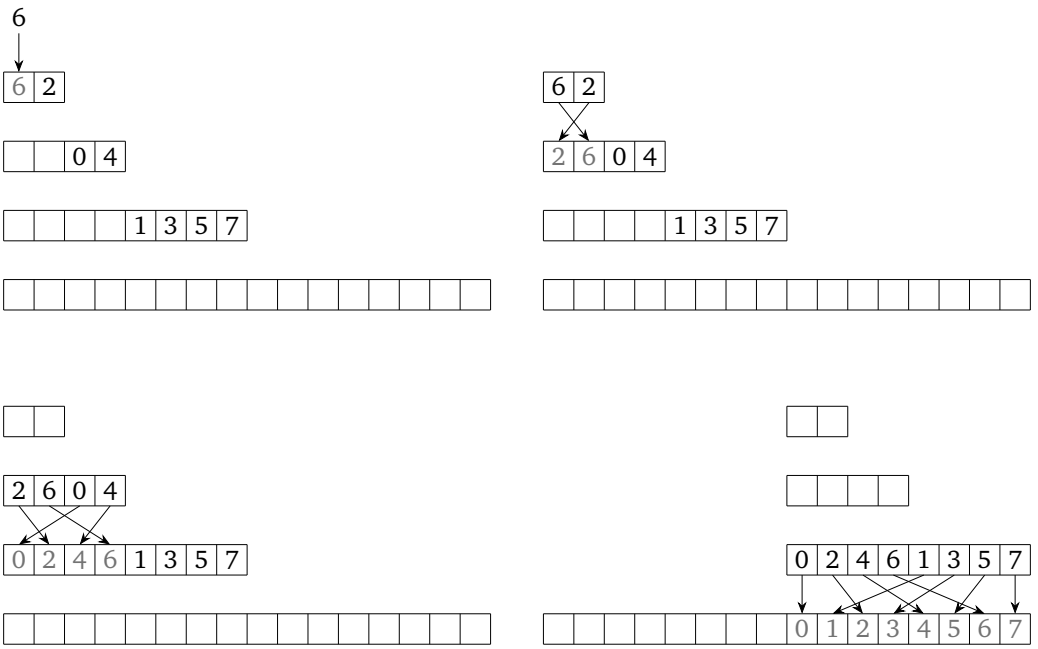


Figure 1.11: Cascade insertion example

is $O(1)$ steps, which get charged to the key that's moved at the end of the phase. So the total time for insertions is (*O*(*T*) keys) · (*O*(log *T*) phases per key) · (*O*(1) steps per phase).

The overall time efficiency of Π' is: $O(T)$ for original non-load/non-store instructions, plus $O(T \log^2 T)$ for all load/store binary searches, plus $O(T \log T)$ for all fresh store insertions.

As in the proof of Theorem 1.9, Π' can abandon the dictionary approach if it discovers the word size is too small, and just run Π using $O(T)$ time and $O(T \log T)$ space in that case. ■

If $T < N$, the same proof yields $\mathsf{TIME}[T] \subseteq \mathsf{TISP}[T \log^2 T, T \log N]$ since the word size is $O(\log N)$.

We mentioned in §1.4.1 that it wouldn't make a big difference if we defined memory space as the number of bits in words that are actually accessed (regardless of the biggest address). This is because the proof of Theorem 1.10 shows that for every program running in time $O(T)$ and space $O(S)$ under this alternative definition of space, there exists an equivalent program running in time $O(T \log^2 S)$ and space $O(S)$ under our official definition of space.

1.8 Time-Efficient Reductions

Reductions relate the complexities of different computational problems. A reduction from problem *A* to problem *B* shows that "if *B* has an efficient algorithm, then so does *A*," and the contrapositive, "if *A* doesn't have an efficient algorithm, then neither does *B*." It turns an upper bound for *B* into an upper bound for *A*, and turns a lower bound for *A* into a lower bound for *B*.

This is "reducing *A* to *B*." We should dispel preconceptions about the word "reduce": In ordinary English it means making something smaller, but that's not a great analogy here. Computational complexity appropriated the word "reduce" and assigned it a new, technical meaning.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

1.8.1 Oracle Reductions

Informally, a reduction from A to B is an algorithm that solves A by making use of any algorithm that solves B. To illustrate, consider a more general version of Directed Reachability:

Directed Subset Reachability	(1.15)
Input:	Directed graph G, subsets S and T of G's set of nodes
Output:	Do there exist nodes $s \in S$ and $t \in T$ such that t is reachable from s?

Instead of designing an algorithm from scratch, we can take advantage of any algorithm for Directed Reachability: To solve Directed Subset Reachability on input (G, S, T), we loop through all pairs $(s, t) \in S \times T$ and run any Directed Reachability algorithm on (G, s, t) to see whether t is reachable from s, and we accept iff at least one run of the Directed Reachability algorithm accepts. We just reduced $A =$ Directed Subset Reachability to $B =$ Directed Reachability. The reduction itself was simple because we didn't have to "reinvent the wheel."

Using BFS, which takes $O(N)$ time, our algorithm for Directed Subset Reachability would take $O(N^3)$ time because the loop might try $O(N^2)$ pairs (s, t). We'll soon see a more efficient reduction between these problems, but first let's elaborate on the concept of reductions.

Note that our example reduction treated the Directed Reachability algorithm as a black box whose inner workings were irrelevant. We could plug in any Directed Reachability algorithm (even the brute force one) and the reduction would still solve Directed Subset Reachability correctly (albeit less efficiently).

Reductions are common in real-world programming. Suppose a software library provides functionality that's relevant but not exactly what we need. Without modifying the library, we might be able to use it by writing wrapper code around it. Rather than "roll our own," we reduce from our desired functionality to the library's. We need not know or care how the library is implemented—we just assume it somehow accomplishes what it's supposed to.

We can reduce A to B without being aware of any algorithm for B at all. A reduction is not a complete algorithm for A—it's an algorithm with a "B-shaped hole." If an efficient algorithm for B becomes available, we could plug it into the reduction to get a complete, efficient algorithm for A. Reductions are like thought experiments: "If we had an algorithm for B, what could we do with it?" Even if we had no idea how to solve Directed Reachability, we could design an efficient reduction from Directed Subset Reachability, relating these problems' complexities.

We view a hypothetical algorithm for B as a subroutine we can call despite the lack of any code for the subroutine's body. (In C/C++ we would say "declared but not defined.") Running the hypothetical algorithm means we call this stub, pass it arguments that form an input to B, and pretend it instantaneously solves B and returns the correct output. In real programming, there's nothing fishy about making calls to a subroutine stub and assuming a correct body for the subroutine will be written eventually.

In complexity theory lingo, to *query the oracle* means "call the subroutine stub representing the hypothetical algorithm." Dubbing it "the oracle for B" evokes a magical entity that answers any question of the form: "Here's y, what's $B(y)$?" A reduction translates an A-question into any number of B-questions whose answers help to answer the A-question efficiently. In other words, a reduction solves A by delegating some work to the oracle for B. This explains the etymology of "reduce": A reduction does part of the work of solving A, thereby "decreasing" the amount of work remaining—all that's left is to solve B.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

A *query* is any particular input to B that a reduction asks the oracle about. A reduction can make as many queries as it wants, and each query can depend on the answers received from the oracle for previous queries. This general kind of reduction is named an *oracle reduction*, to distinguish it from restricted kinds. We'll see a formal definition in §1.8.5, but for now:

Definition. An *oracle reduction* from problem A to problem B is an algorithm that solves A with access to an *oracle* that solves B: The reduction can write any valid input y to B in a designated part of its memory and do a *query* to get $B(y)$ in one time step. If B is a search problem, the reduction must solve A no matter which acceptable solution the oracle returns for each query.

The notation $A \leq_p^o B$ means there exists a polynomial-time oracle reduction from A to B. In particular, on an input of size N, such a reduction makes at most $\text{poly}\,N$ many queries, and the size of each query is at most $\text{poly}\,N$. ◀

In "$\leq_p^o$" the subscript stands for "poly-time" and the superscript stands for "oracle." The "$\leq$" notation is not ideal but has prevailed historically. It doesn't mean A is "less than or equal to" B, but it suggests that A's complexity can't be much higher than B's complexity.

Figure 1.12 illustrates the flow of execution of a reduction interacting with an oracle. The oracle for B is shown as a dashed box because it's a hypothetical algorithm. Given input x, the reduction computes and eventually produces a query y_1. Control is transferred to the oracle, which magically returns $B(y_1)$. Control is transferred back to the reduction, which computes and may produce another query y_2, and so on. Figure 1.12 depicts three queries, but it could be any number, including zero. After getting the answer to a query, the reduction decides whether to make another query and if so, what the next query should be, and if not, what to output.

A reduction can ignore the oracle (make no queries), so a self-contained poly-time algorithm for A is automatically an oracle reduction to literally any problem B. Reductions are useful when we don't already have an efficient algorithm for A.

Our earlier reduction was:

Oracle reduction from Directed Subset Reachability to Directed Reachability (1.16)

```
on input (G, S, T):
    for each node s ∈ S:
        for each node t ∈ T:
            query the oracle on input (G, s, t)
            if the oracle accepted: accept
    reject
```

1.8.2 Mapping Reductions

A *mapping reduction* is an oracle reduction that always makes exactly one query and outputs the same answer the oracle returns. Since the only work is to go from the input to the query, we usually just specify a mapping reduction as a function that *maps* an input x to a query y such that $A(x) = B(y)$ (Figure 1.13). We can combine a mapping reduction with any algorithm for B to obtain an algorithm for A: Run the mapping reduction on x to get y, then run the algorithm for B to get $B(y)$, then output the same thing for $A(x)$. A mapping reduction translates an A-question into one equivalent B-question.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

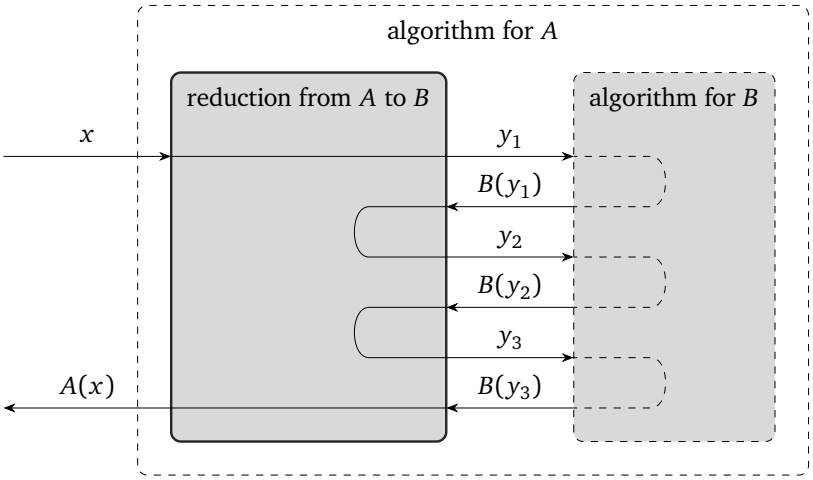


Figure 1.12: Oracle reduction

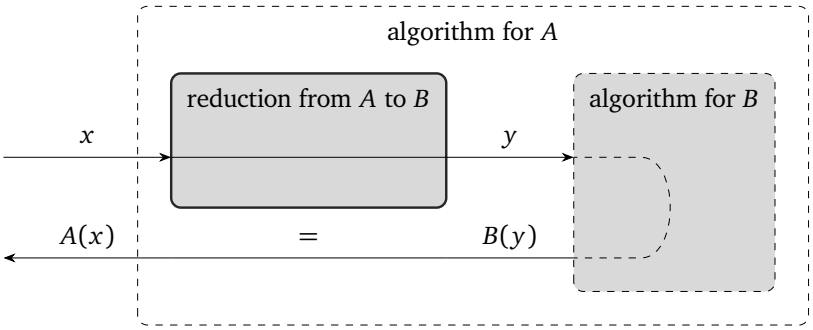


Figure 1.13: Mapping reduction

Definition. Assume A and B are both decision problems or both function problems. A *mapping reduction* from A to B is an algorithm that computes a function F: (A's valid inputs) $\to$ (B's valid inputs) such that $A(x) = B(F(x))$ for every valid input x to A.

The notation $A \leq_p^m B$ means there exists a polynomial-time mapping reduction from A to B. In particular, on an input of size N, such a reduction produces a query of size at most poly$N$. ◄

In "$\leq_p^m$" the subscript stands for "poly-time" and the superscript stands for "mapping."

We could define mapping reductions for search problems, but we won't need them.

Our oracle reduction from A = Directed Subset Reachability to B = Directed Reachability in §1.8.1 wasn't a mapping reduction, since it made many queries. We now show $A \leq_p^m B$ by efficiently mapping (transforming) an input of A to an equivalent input of B:

Mapping reduction from Directed Subset Reachability to Directed Reachability (1.17)

map (G, S, T) to (G', s', t') where:
  G' contains G along with:
    a new node s' with edges to all nodes of S
    a new node t' with edges from all nodes of T

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

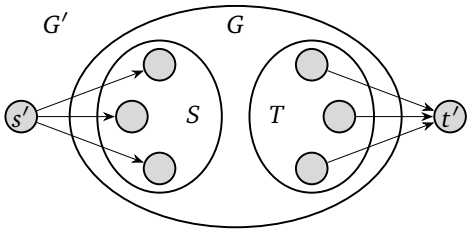


Figure 1.14: Mapping reduction from Directed Subset Reachability to Directed Reachability

See Figure 1.14. The oracle internally renames the arguments (G', s', t') as (G, s, t), which are like parameter names in a Directed Reachability subroutine. The reduction uses the "primed" names to avoid a name clash with the G, s, t that appear in the definition of Directed Subset Reachability. It's simple to see the reduction is correct, but let's formalize this to illustrate the format of such proofs.

Claim 1.11. $A(G, S, T) = B(G', s', t')$.

Proof. We show that "there exist $s \in S$ and $t \in T$ such that t is reachable from s in G" iff "t' is reachable from s' in G'."

$\Rightarrow$: Assume G has a path $s \to \cdots \to t$ for some $s \in S$ and $t \in T$. Then $s' \to s \to \cdots \to t \to t'$ is a path in G'.

$\Leftarrow$: Assume G' has a path $s' \to s \to \cdots \to t \to t'$. Then $s \to \cdots \to t$ is a path in G (it doesn't use any new nodes or new edges of G'), and $s \in S$ since $s' \to s$ is an edge, and $t \in T$ since $t \to t'$ is an edge. (If S and T overlap then we could have $s = t$, which is fine since that node would be a path of length 0 from S to T.) ■

This mapping reduction runs in time $O(N)$ and produces a query of size $O(N)$. Using BFS to answer the query in $O(N)$ time, we get a Directed Subset Reachability algorithm with time efficiency $O(N)$.

By the way, in the opposite direction, Directed Reachability $\leq_p^m$ Directed Subset Reachability by mapping (G, s, t) to $(G, \{s\}, \{t\})$.

For an example involving function problems, consider a more general version of Small Iterated Addition:

Iterated Addition	(1.18)
Input:	Nonnegative integers $x_1, x_2, \ldots, x_n$ represented in binary with m bits each
Output:	$x_1 + x_2 + \cdots + x_n$ in binary

Each x_i is a separate number (not an individual bit of a single number x). The input encoding has n and m (two words) and $x_1, \ldots, x_n$ (one word per bit), so the input size is $N = 2 + nm$.

We can decompose the primary-school Multiplication algorithm into a mapping reduction from Multiplication to Iterated Addition, together with an Iterated Addition algorithm.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

Mapping reduction from Multiplication to Iterated Addition (1.19)

map (x, y) to $(q_0, q_1, \ldots, q_{n-1})$ where:
  n is the number of bits in each of $x = x_{n-1} \cdots x_0$ and $y = y_{n-1} \cdots y_0$
  $m = 2n - 1$
  $q_j = x \cdot y_j \cdot 2^j$ represented with m bits, for each $j \in \{0, \ldots, n-1\}$

To avoid a name clash, the arguments $(q_0, \ldots, q_{n-1})$ passed to the oracle have different names than the corresponding input parameters $(x_1, \ldots, x_n)$ in the definition of Iterated Addition. (We used q to stand for "query.") The reduction is correct since $x \cdot y = x \cdot \sum_{j=0}^{n-1}(y_j \cdot 2^j) = \sum_j q_j$.

The primary-school Multiplication algorithm contains the standard "carry algorithm" for Iterated Addition. The latter algorithm runs in linear time, but we should be careful: If we say $O(N)$ time, it's unclear whether N is the size of the input to Multiplication or the size of the query to Iterated Addition. It's best to give distinct names to the input sizes of the two problems. Let's use N for Multiplication and M for Iterated Addition. The reduction runs in time $O(N^2)$ and produces a query of size $M = O(N^2)$, and the carry algorithm runs in time $O(M) = O(N^2)$. Thus the primary-school Multiplication algorithm has time efficiency $O(N^2)$.

1.8.3 Decision Versus Search

As a warmup, let's consider decision versus function problems. Reducing a decision problem to a related function problem is often trivial. For example, if we can compute the product of two numbers, then of course we can compute any particular bit of the product:

Reduction from Multiplication Decision to Multiplication (1.20)

on input (x, y, i):
  query the oracle on input (x, y) to get $z = x \cdot y$
  output z_i, or output 0 if z has at most i bits

In the opposite direction, we can compute the product by computing its bits one by one:

Reduction from Multiplication to Multiplication Decision (1.21)

on input (x, y), where x and y each have $N/2$ bits:
  for $i \leftarrow 0, 1, \ldots, N-1$:
    query the oracle on input (x, y, i) to get $z_i \in \{0, 1\}$
  output the binary number $z = z_{N-1} \cdots z_0$

This isn't the most efficient way to solve Multiplication. We could ignore the oracle and solve Multiplication from scratch. If we didn't know how to solve Multiplication efficiently—or if a Multiplication Decision implementation were conveniently available—this reduction would exploit the decision problem black box to solve the function problem without hassle.

This reduction is *nonadaptive*, which means the queries don't depend on the oracle's answers to previous queries: The reduction knows in advance which queries it will make. The answers to the queries are only used to form the output, not to form other queries.

We turn to decision versus search. Reductions from search problems to related decision problems are handy for several results in this book. Let's illustrate the idea with this search problem (introduced in §1.1.2) and a related decision problem:

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

Directed Shortest Path	(1.22)
Input:	Directed graph G, nodes s and t
Output:	A shortest path from s to t in G, or report that t is not reachable from s

Directed Shortest Path Decision	(1.23)
Input:	Directed graph G, nodes s and t, nonnegative integer k
Output:	Does there exist a path of length $\leq k$ from s to t in G?

Pretend we don't know Directed Shortest Path can be solved in linear time by the extended BFS algorithm, but we can make calls to a hypothetical algorithm for Directed Shortest Path Decision. How can we construct a shortest path from s to t with the aid of an oracle that only gives one bit of information at a time?

For starters, we determine the distance (length of a shortest path) from s to t by querying the oracle on inputs (G, s, t, k) for different values of k, to find the smallest k for which the oracle accepts. If the actual distance is d, the oracle accepts when $k \geq d$ and rejects when $k < d$. We can do binary search to find d in about $\log n$ queries (where n is the number of nodes) or just try all possibilities from 0 to $n-1$, which would still be poly time.

After finding d, we figure out which edge our path of length d should start with. For each edge (s, v), we formulate the question "Is it safe to have (s, v) be the first edge?" as the query "Does there exist a path of length $\leq d-1$ from v to t?" After identifying a safe edge (s, v) with the oracle's help, we repeat this with v's outgoing edges to pick the path's next edge, and so on. This is a *greedy* approach—the reduction commits to each edge of the path in sequence, and never changes its mind. Suppose we've built a path from s to some node u so far. To determine whether an edge (u, v) is safe to append to our path, we have the oracle "smell" whether v is in the right direction—whether the distance from v to t is less than the distance from u to t.

Reduction from Directed Shortest Path to Directed Shortest Path Decision (1.24)

```
on input (G, s, t):
    find the smallest d ∈ {0, 1, ..., n−1} for which the oracle accepts input (G, s, t, d)
    if no such d exists: halt and report that t is not reachable from s
    let u_0 = s
    for ℓ = 1, 2, ..., d:
        for each edge (u_{ℓ−1}, v):
            query the oracle on input (G, v, t, d − ℓ)
            if the oracle accepted: let u_ℓ = v and continue to the next outer iteration
    output u_0 → u_1 → ··· → u_{d−1} → u_d
```

This first finds the distance $d = \text{dist}(s, t)$. Then, assuming t is reachable from s, the outer loop invariant is $\text{dist}(u_\ell, t) \leq d - \ell$. (This distance is exactly $d - \ell$, but we only need $\leq$ for the proof.) The invariant holds for $\ell = 0$ by the definition of d. To see that the invariant is maintained, suppose it holds for $\ell - 1$: $\text{dist}(u_{\ell-1}, t) \leq d - \ell + 1$. The oracle accepts at least one of the queries $(G, v, t, d - \ell)$ in outer iteration ℓ, because removing the first edge $(u_{\ell-1}, v)$ of a path from $u_{\ell-1}$ to t of length $\leq d - \ell + 1$ yields a path from v to t of length $\leq d - \ell$. Multiple outgoing edges of $u_{\ell-1}$ might make the oracle accept, but whichever one the reduction finds first, the invariant will hold for ℓ: $\text{dist}(u_\ell, t) \leq d - \ell$. Since the invariant holds at the end when $\ell = d$, we have

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

$u_d = t$ since $\text{dist}(u_d, t) \le d - d = 0$. Thus the output is correct: $u_0 \to \cdots \to u_d$ is a path from s to t of length d. Though the reduction doesn't explicitly avoid repeating nodes (as required by the definition of *path*), there indeed won't be any repeats since otherwise G would have an even shorter path from s to t (by removing cycles as in Lemma A.5), contradicting the definition of d.

This reduction is *adaptive* because the answer to a query $(G, v, t, d - \ell)$ affects subsequent queries: If the oracle accepts, then the next queries correspond to outgoing edges of this v, otherwise they don't.

This illustrates the style of search-to-decision reductions in a familiar context. Compared to using extended BFS to solve the problem from scratch, this reduction has no advantage in time efficiency but has an advantage in space efficiency (§1.9.3).

1.8.4 Properties of Reductions

Lemma 1.12. *If $A \le_p^o B$ and B has a poly-time program, then so does A. In particular, for all decision problems A and B, if $A \le_p^o B$ and $B \in \mathsf{P}$ then $A \in \mathsf{P}$.*

Proof idea. We create a program for A that runs the reduction and uses the program for B to answer each oracle query. This is like what a linker does: Given compiled code that makes calls to an external subroutine whose implementation is in another file, a linker combines them into a complete, executable program.

The time efficiency of B's program is a function of B's input size, not of A's input size. If we run the reduction on an input of size N and it produces a query of size M, then B's program takes $\text{poly}\, M$ time to answer the query. Since the reduction runs in $\text{poly}\, N$ time, it makes only $\text{poly}\, N$ many queries, each of $\text{poly}\, N$ size. So A's program has time efficiency:

$$\underbrace{\text{poly}\, N}_{\substack{\text{time of}\\ \text{reduction}}} + \underbrace{\text{poly}\, N}_{\substack{\text{number}\\ \text{of queries}}} \cdot \underbrace{\text{poly}}_{\substack{\text{time of } B\text{'s}\\ \text{program}}} \Big(\underbrace{\text{poly}\, N}_{\substack{\text{query}\\ \text{size}}} \Big) = \text{poly}\, N$$

■

The next lemma is the *transitive property* of reductions.

Lemma 1.13. *If $A \le_p^o B$ and $B \le_p^o C$ then $A \le_p^o C$. If $A \le_p^m B$ and $B \le_p^m C$ then $A \le_p^m C$.*

Proof idea. If we can translate A-questions to B-questions, and we can translate B-questions to C-questions, then by chaining these together we can translate A-questions to C-questions. An A-to-C reduction can run the A-to-B reduction but use the B-to-C reduction to answer each B oracle query. This solves A by making C oracle queries, and it still runs in poly time. See Figure 1.15 and Figure 1.16. ■

We formally define reductions in §1.8.5 and then formally prove these two lemmas in §1.8.6. But an intuitive understanding of reductions is usually good enough.

1.8.5 Formal Definition

We likened oracles to subroutines, but instead of representing oracles with the subroutine call mechanism from § 1.5.1, we add dedicated features to our model of computation. We use pseudocode for high-level "reduction algorithms" and implement them as low-level "reduction programs" in our model.

First, consider reductions to a decision problem B. We have a write-only *query segment* and two new instructions:

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

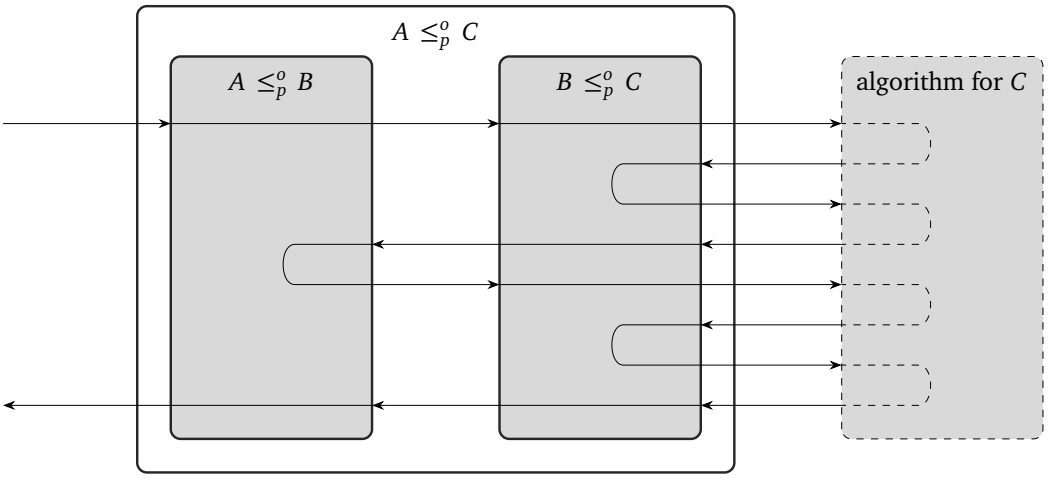


Figure 1.15: Chaining oracle reductions together

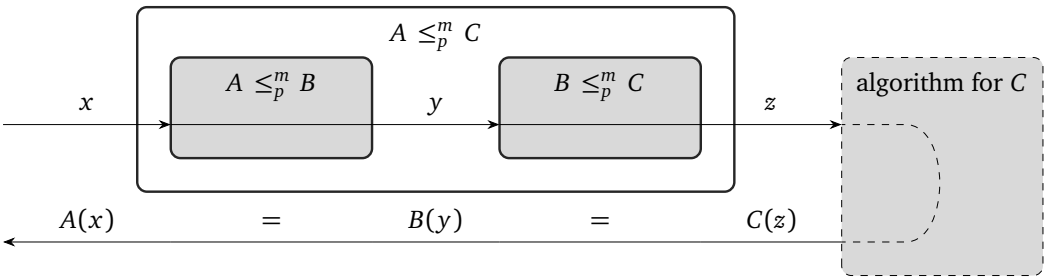


Figure 1.16: Chaining mapping reductions together

- "write-query Reg[i]" puts the word from Reg[i] (or a hardcoded constant) into the next query segment address (like the "write" instruction for the output segment).
- "query Reg[i] ← Answer" treats the query segment's contents (up through the largest address written to) as an input y to B, puts $B(y)$ (which is 0 or 1) into Reg[i], and erases the query segment (resetting every word to all-0). This counts as one time step.

After a query step, subsequent write-query steps will write to addresses $0, 1, 2, \ldots$ again (in that order), and the next query step will let y be the words written by those subsequent write-query steps. There are subtleties in defining a query step's behavior:

- If y is an invalid input to B, the reduction "freezes" by keeping the program counter on the query instruction for all future steps. This represents that a hypothetical program for B might not halt on an invalid input. Thus, on a valid input to A, a reduction from A to B can't make an invalid query to B, because then the reduction wouldn't halt.
- If B's word size (depending on the size of y) is greater than the reduction's word size (depending on the size of the reduction's input), then each word of y is left-padded with 0s to form the input to B. If B's word size is less than the reduction's word size, then any y with a 1 in the "extra" most significant bits is necessarily an invalid input to B.

Now, suppose B is a function problem. We have a read-only *answer segment* for holding each oracle query's result. We still have "write-query Reg[i]", but instead of "query Reg[i] ← Answer"

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

we have two new instructions:

- "query" erases the answer segment, then puts $B(y)$ into the answer segment (where y is the query segment's contents), then erases the query segment.
- "read-answer Reg[i] ← Ans[Reg[j]]" puts the word from answer segment address Reg[j] (or a hardcoded constant address) into Reg[i].

For a query step: The reduction freezes if y is an invalid input to B. If B's word size doesn't equal the reduction's word size, then the words of $B(y)$ are either left-padded with 0s or truncated to the least significant bits to fit the answer segment.

If B is a search problem, the instruction "query Reg[i] ← Found?" erases the answer segment, then puts 1 into Reg[i] and an arbitrary acceptable solution of $B(y)$ into the answer segment if such a solution exists and otherwise just puts 0 into Reg[i], then erases the query segment. To solve A, a reduction must generate correct output no matter which acceptable solution appears in the answer segment after each query.

$A \leq_p^o B$ means there exists a reduction to B that solves A with poly$N$ time efficiency and log-bounded word size, where N is A's input size.

$A \leq_p^m B$ means $A \leq_p^o B$ via a reduction that, for every valid input to A, performs exactly one query and outputs the same answer. For function problems, we might as well assume the answer segment is the output segment, and the "query" instruction causes termination. For decision problems, we might as well assume that instead of "query Reg[i] ← Answer" we just have a "query" instruction that's equivalent to "accept" or "reject" depending on the oracle's answer. In these cases, the accept/reject/halt instructions are out of commission.

1.8.6 Formal Proofs of Properties of Reductions

Warmup for proof of Lemma 1.12. We prove the important special case of mapping reductions between decision problems: If $A \leq_p^m B$ and $B \in \mathsf{P}$ then $A \in \mathsf{P}$.

Let $\Pi_{A \leq B}$ be a poly-time mapping reduction from A to B. Let Π_B be a poly-time program for B. It suffices to design a poly-time program Π_A for A using three work segments (Lemma 1.5):

- Mem_1 corresponds to $\Pi_{A \leq B}$'s work segment.
- Mem_2 corresponds to $\Pi_{A \leq B}$'s query segment and Π_B's input segment.
- Mem_3 corresponds to Π_B's work segment.

Π_A's registers are:

- $\Pi_{A \leq B}$'s registers.
- Π_B's registers, renamed $\text{Reg}_B[\,]$ to distinguish them from $\Pi_{A \leq B}$'s registers.

Π_A contains $\Pi_{A \leq B}$'s code with $\text{Mem}_1[\,]$ instead of Mem[ ] in load/store instructions, followed by Π_B's code with $\text{Reg}_B[\,]$ instead of Reg[ ] and with $\text{Mem}_3[\,]$ instead of Mem[ ]. We do these substitutions in $\Pi_{A \leq B}$'s code:

write-query Reg[i]	⇝	store $\text{Mem}_2[\text{Reg}_B[0]] \leftarrow \text{Reg}[i]$ let $\text{Reg}_B[0] \leftarrow \text{Reg}_B[0] + 1$
query	⇝	goto 0th line of Π_B's code

When the execution gets to Π_B, $\text{Reg}_B[0]$ has been initialized with the query size. We do this substitution in Π_B's code:

read Reg[i] ← In[Reg[j]]	⇝	load $\text{Reg}_B[i] \leftarrow \text{Mem}_2[\text{Reg}_B[j]]$

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

We also update the target line numbers of all branches.

Π_A solves A because $\Pi_{A\le B}$ solves A assuming the query is answered correctly, and Π_B answers the query correctly. $\Pi_{A\le B}$ and Π_B might not agree on the word size, but we deal with this snag momentarily.

Suppose $\Pi_{A\le B}$ has time efficiency $O(N^c)$ and Π_B has time efficiency $O(M^d)$ where M is the query size. For Π_A's time efficiency: $\Pi_{A\le B}$'s contribution is still $O(N^c)$ since it's at most twice the unmodified $\Pi_{A\le B}$'s time (after accounting for the write-query substitution). Since $M \le O(N^c)$ (as each write-query is a time step of $\Pi_{A\le B}$), the execution of Π_B contributes $O(O(N^c)^d) = O(N^{cd})$ time. Thus Π_A runs in time $O(N^c) + O(N^{cd}) = O(N^{cd})$ (assuming $d \ge 1$).

Back to the word size issue: We may assume $\Pi_{A\le B}$ uses word size $c' \log N$ and Π_B uses word size $d' \log M$, for integers c' and d'. Since $M \le eN^c$ for an integer e, the intended word size for Π_B is $\le d' \log(eN^c)$. Thus, the log-bounded word size $\max\big(c' \log N, d' \log(eN^c)\big)$ is big enough for Π_A. As usual, Π_A zeroes out the most significant bits of each word, leaving $c' \log N$ bits for $\Pi_{A\le B}$'s words and $d' \log M$ bits for Π_B's words. This only increases the time by a constant factor. ■

For function/search problems, we deal with the answer segment. To handle multiple queries, we reinitialize all of Π_B's memory for each query. The details are unenlightening.

Proof of Lemma 1.12 (ancillary). First, assume B is a function problem.

Let $\Pi_{A\le B}$ be a poly-time oracle reduction from A to B. Let Π_B be a poly-time program for B. It suffices to design a poly-time program Π_A for A using four work segments (Lemma 1.5):

- Mem_1 corresponds to $\Pi_{A\le B}$'s work segment.
- Mem_2 corresponds to $\Pi_{A\le B}$'s query segment and Π_B's input segment.
- Mem_3 corresponds to Π_B's work segment.
- Mem_4 corresponds to Π_B's output segment and $\Pi_{A\le B}$'s answer segment.

Π_A's registers are:

- $\Pi_{A\le B}$'s registers.
- Π_B's registers, renamed $\text{Reg}_B[\,]$ to distinguish them from $\Pi_{A\le B}$'s registers.
- Three more registers, nicknamed *qsize*, *bsize*, *asize*, to remember the sizes of the contents of Mem_2, Mem_3, Mem_4 respectively.

Π_A runs $\Pi_{A\le B}$ using $\text{Mem}_1[\,]$ instead of $\text{Mem}[\,]$, and with these substitutions:

write-query $\text{Reg}[i]$	$\rightsquigarrow$	store $\text{Mem}_2[qsize] \leftarrow \text{Reg}[i]$ let $qsize \leftarrow qsize + 1$
query	$\rightsquigarrow$	erase Mem_4 erase Mem_3 let $\text{Reg}_B[0] \leftarrow qsize$ let $\text{Reg}_B[i] \leftarrow 0$ } $\forall i \ne 0$ a copy of Π_B's code erase Mem_2
read-answer $\text{Reg}[i] \leftarrow \text{Ans}[\text{Reg}[j]]$	$\rightsquigarrow$	load $\text{Reg}[i] \leftarrow \text{Mem}_4[\text{Reg}[j]]$
erase Mem_2	$\rightsquigarrow$	if $qsize = 0$ then goto 4 lines below let $qsize \leftarrow qsize - 1$ store $\text{Mem}_2[qsize] \leftarrow 0$ goto 3 lines above

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

Similarly, "erase Mem_3" uses *bsize*, and "erase Mem_4" uses *asize*. We update Π_B's code to use $\text{Reg}_B[\,]$ instead of $\text{Reg}[\,]$, and $\text{Mem}_3[\,]$ instead of $\text{Mem}[\,]$, and with these substitutions:

read $\text{Reg}[i] \leftarrow \text{In}[\text{Reg}[j]]$	$\rightsquigarrow$	load $\text{Reg}_B[i] \leftarrow \text{Mem}_2[\text{Reg}_B[j]]$
write $\text{Reg}[i]$	$\rightsquigarrow$	store $\text{Mem}_4[asize] \leftarrow \text{Reg}_B[i]$ let $asize \leftarrow asize + 1$
store $\text{Mem}[\text{Reg}[j]] \leftarrow \text{Reg}[i]$	$\rightsquigarrow$	if $\text{Reg}_B[j] < bsize$ then goto 2 lines below let $bsize \leftarrow \text{Reg}_B[j] + 1$ store $\text{Mem}_3[\text{Reg}_B[j]] \leftarrow \text{Reg}_B[i]$
halt	$\rightsquigarrow$	goto next "erase Mem_2"

If B is a decision problem, the "query $\text{Reg}[i] \leftarrow$ Answer" substitution is the "query" substitution but without "erase Mem_4" (there's no answer segment), and this copy of Π_B's code has:

accept	$\rightsquigarrow$	let $\text{Reg}[i] \leftarrow 1$ goto next "erase Mem_2"
reject	$\rightsquigarrow$	let $\text{Reg}[i] \leftarrow 0$ goto next "erase Mem_2"

If B is a search problem, the "query $\text{Reg}[i] \leftarrow$ Found?" substitution is the "query" substitution for function problems, but this copy of Π_B's code has:

halt	$\rightsquigarrow$	let $\text{Reg}[i] \leftarrow 1$ goto next "erase Mem_2"
reject	$\rightsquigarrow$	let $\text{Reg}[i] \leftarrow 0$ erase Mem_4 goto next "erase Mem_2"

We also update the target line numbers of all branches.

(We duplicate Π_B's code for every query instruction, which is poor programming style, but that doesn't matter here. To avoid the duplication, we could branch to a unique copy of Π_B, then indirectly branch back to the calling point when Π_B is done.)

Π_A solves A because $\Pi_{A \le B}$ solves A assuming each query is answered correctly, and Π_B answers each query correctly. We deal with the word size issue just like in the warmup proof, though now different runs of Π_B may have different query sizes and thus need different effective word sizes.

Suppose $\Pi_{A \le B}$ has time efficiency $O(N^c)$ and Π_B has time efficiency $O(M^d)$ where $M = qsize$ is the query size. We bound Π_A's time efficiency by partitioning the steps into five groups, and bounding each group's contribution to the time. The five groups are: $\Pi_{A \le B}$, the executions of Π_B, and the "erase Mem_2", "erase Mem_3", and "erase Mem_4" phases.

$\Pi_{A \le B}$'s contribution is still $O(N^c)$ after accounting for the substitutions. If $M_1, \ldots, M_p$ are the sizes of queries made by $\Pi_{A \le B}$ on a particular valid input of A, then $M_1 + \cdots + M_p \le O(N^c)$, so Π_B's total contribution is $O(M_1^d + \cdots + M_p^d) \le O((M_1 + \cdots + M_p)^d) \le O(N^{cd})$, since expanding $(M_1 + \cdots + M_p)^d$ yields $M_1^d + \cdots + M_p^d$ plus other nonnegative terms.

The total contribution of all "erase Mem_2" phases is $O(N^c)$ since each iteration of the erase loop corresponds to a write-query step of $\Pi_{A \le B}$. The total contribution of all "erase Mem_4" phases is $O(N^{cd})$ since each iteration of the erase loop corresponds to a write step of Π_B.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

The contribution of one "erase Mem_3" phase is proportional to the highest address stored to by the previous execution of Π_B, which is poly N since the word size is $O(\log N)$. We can improve the total contribution to $O(N^{cd})$ by using an extra work segment to record which addresses Π_B stores to, and just erasing those addresses.

So Π_A has time efficiency $O(N^{cd})$ with the "erase Mem_3" modification. ■

Proof of Lemma 1.13 (ancillary). We just outline the proof; it's nearly identical to Lemma 1.12.

First, we show this for oracle reductions. The A-to-C reduction has work segments corresponding to the A-to-B reduction's work/query/answer segments and the B-to-C reduction's work segment. The A-to-C reduction's query/answer segments correspond to the B-to-C reduction's query/answer segments. The A-to-C reduction has registers corresponding to the registers of the A-to-B and B-to-C reductions, and extra registers to remember the sizes of work segments it needs to erase. The A-to-C reduction runs the A-to-B reduction and replaces each B query with a run of the B-to-C reduction, using the same instruction substitutions as in the proof of Lemma 1.12. This solves A assuming each C query is answered correctly. The analysis of the time efficiency and word size is as in Lemma 1.12. (The proof of Lemma 1.5 also works for reductions, provided we zero out the new most significant bits after each "read-answer".)

For mapping reductions, we observe that the proof for oracle reductions preserves the "mapping" property: Assume A, B, C are all decision problems or all function problems. On input x, the A-to-B mapping reduction makes one query y, such that $A(x) = B(y)$. This query is answered by the B-to-C mapping reduction, which makes one query z, such that $B(y) = C(z)$. So on input x, the A-to-C reduction makes one query z, such that $A(x) = C(z)$, and is thus a mapping reduction. ■

1.9 Space-Efficient Reductions

1.9.1 Log-Space Mapping Reductions

The notation $A \leq_\ell^m B$ means there exists a log-space mapping reduction from A to B (which are both decision problems or both function problems). As usual, the space includes the registers and work segment but not the input or output segments. It doesn't include the query segment. (If it did, log-space reductions could only produce queries with $O(\log N)$ bits, which is very restrictive.) The mapping reduction examples from §1.8.2 only need log space:

Example. Multiplication $\leq_\ell^m$ Iterated Addition: We map $(x, y) = (x_{n-1} \cdots x_0, y_{n-1} \cdots y_0)$ to $(q_0, \ldots, q_{n-1})$ where $q_j = x \cdot y_j \cdot 2^j$ has $m = 2n - 1$ bits. As we write the encoding of $(q_0, \ldots, q_{n-1})$, we hold $j \in \{0, \ldots, n-1\}$ and $i \in \{0, \ldots, m-1\}$ in registers and use more registers to compute $q_{j,i} = x_{i-j} \cdot y_j$ (the ith least significant bit of the query's jth number) for each pair (j, i) with $0 \leq i - j < n$, and $q_{j,i} = 0$ for other pairs (j, i). Each register has $O(\log N)$ bits, so the reduction has space efficiency $O(\log N)$ even though the query is at least $nm = \Omega(N^2)$ bits. ♦

Example. Directed Subset Reachability $\leq_\ell^m$ Directed Reachability: We map (G, S, T) (where G has n nodes) to (G', s', t') where G' contains G and nodes $s' = n + 1$ (with edges to all nodes of S) and $t' = n + 2$ (with edges from all nodes of T). As we write the encoding of (G', s', t'), we mostly just copy G, using a register to remember where we are and a register to hold each word as it's copied. For each node $u \in [n]$, we check whether $u \in T$ and if so, append t' to u's adjacency list and shift all subsequent lists over. Then, s' has adjacency list S, and t' has an empty adjacency list. At the beginning, as we go through the list of pointers to the adjacency lists, we remember how many nodes so far are in T, and update each pointer accordingly. ♦

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

Every log-space program is poly-time, since there are only poly many possible configurations and the computation wouldn't halt if it ever repeated a configuration (Theorem 1.8). This also holds for mapping reductions if configurations don't include the query segment:

Lemma 1.14. *Every log-space mapping reduction is poly-time, so if $A \leq_\ell^m B$ then $A \leq_p^m B$.* ■

Poly-time mapping reductions transfer time complexity bounds from one problem to another: If $A \leq_p^m B$ and B has a poly-time program, then so does A (Lemma 1.12). There's an analogue for space:

Lemma 1.15. *If $A \leq_\ell^m B$ and B has a log-space program, then so does A. In particular, for all decision problems A and B, if $A \leq_\ell^m B$ and $B \in \mathsf{L}$ then $A \in \mathsf{L}$.*

Now we see why L is robust with respect to the problem encoding, as mentioned in §1.6.3: Different encodings of a problem are technically different problems, but we can convert between reasonable encodings with log-space mapping reductions, so either all versions of the problem are in L or none are (Lemma 1.15). Also, L is robust with respect to the model of computation because the adaptations in §1.5 only involve new registers and a constant number of work segment words.

Proof idea for Lemma 1.15. In the poly-time setting (Lemma 1.12), we combined a reduction and a program for B into a program for A, in which the reduction's query segment became a work segment. In the log-space setting, we can't do that, because we'd suddenly be charged for the previously "free" query segment.

For example, to get a log-space Multiplication algorithm from a log-space Iterated Addition algorithm, we can't afford to store the entire query and then run the Iterated Addition algorithm on it. The primary-school Multiplication algorithm instead calculates each bit of the query on the fly when it's needed—and then forgets it—rather than calculating them all up front. This blurs the boundary between the reduction and the algorithm it's interacting with.

In general, the idea is to never store the whole query at once. Whenever B's program wants to read a word of its input (the query), we pause B's program and compute the desired word by running the reduction and ignoring the query it generates, until it gets to the address being read by B's program. We reuse the same memory every time we run the reduction. If B's program reads the same word multiple times, we resist the temptation to store the word for future lookup—we just recompute it by rerunning the reduction, as many times as needed. This program for A might not be impressively fast, but it just needs the space of the reduction plus the space of B's program, both of which are $O(\log N)$. ■

This illustrates a paradigm for designing space-efficient algorithms:

Recompute intermediate results to avoid wasting space storing them.

Here's the complementary paradigm for designing time-efficient algorithms:

Store intermediate results to avoid wasting time recomputing them.

Before we formally prove Lemma 1.15, we precisely define *log-space mapping reduction*. Recall that a mapping reduction to B can use these instructions:

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

- "write-query Reg[i]" puts the word from Reg[i] (or a hardcoded constant) into the next query segment address.
- "query" treats the query segment's contents (up through the largest address written to) as an input y to B, and terminates the execution with $B(y)$ as the output. If B is a decision problem, this is equivalent to "accept" or "reject". If B is a function problem, this is equivalent to "halt" where the answer segment and output segment are one and the same.

$A \leq_\ell^m B$ means there exists a mapping reduction to B that solves A with $O(\log N)$ space efficiency and log-bounded word size (where N is A's input size), where the input, query, and answer/output segments don't count toward the space. We may assume such a reduction is register-only, as in Observation 1.7.

Proof of Lemma 1.15. Let $\Pi_{A\leq B}$ be a register-only mapping reduction from A to B, and let Π_B be a register-only program for B, both with log-bounded word size. We design a register-only program Π_A for A using these registers:

- $\Pi_{A\leq B}$'s registers.
- Π_B's registers, renamed Reg$_B$[ ] to distinguish them from $\Pi_{A\leq B}$'s registers.
- Two more registers, nicknamed N and *qaddr*.

Π_A first has "let $N \leftarrow$ Reg[0]", which puts A's input size in the N register for safekeeping, in case $\Pi_{A\leq B}$ overwrites Reg[0]. Then, Π_A initializes Reg$_B$[0] to the query size using a copy of $\Pi_{A\leq B}$'s code with these substitutions:

write-query Reg[i]	$\rightsquigarrow$	let Reg$_B$[0] $\leftarrow$ Reg$_B$[0] + 1
query	$\rightsquigarrow$	goto 0th line of Π_B's code

Then, Π_A has Π_B's code with Reg$_B$[ ] instead of Reg[ ] and with this substitution:

read Reg[i] $\leftarrow$ In[Reg[j]]	$\rightsquigarrow$	let *qaddr* $\leftarrow$ 0 let Reg[0] $\leftarrow N$ let Reg[k] $\leftarrow$ 0 } $\forall k \neq 0$ another copy of $\Pi_{A\leq B}$'s code

This new copy of $\Pi_{A\leq B}$'s code (there's one copy per "read" line in Π_B) computes the word that would be at address Reg$_B$[j] of the query segment, and puts it in Reg$_B$[i]. To accomplish this, we modify the code to use the *qaddr* register to remember how many words have been "written" to the (nonexistent) query segment so far, and stop when it reaches the desired word:

write-query Reg[k]	$\rightsquigarrow$	if *qaddr* $\neq$ Reg$_B$[j] then goto 3 lines below let Reg$_B$[i] $\leftarrow$ Reg[k] goto next line of Π_B's code let *qaddr* $\leftarrow$ *qaddr* + 1

We assume Π_B never reads past its input (Lemma 1.3), so this copy of $\Pi_{A\leq B}$ won't reach a "query" instruction—we must have Reg$_B$[j] < query size.

We also update the target line numbers of all branches.

Π_A solves A because it faithfully simulates Π_B on $\Pi_{A\leq B}$'s query. Though $\Pi_{A\leq B}$ and Π_B might not agree on which word size to use, we handle this as in Lemma 1.12: Since $\Pi_{A\leq B}$ runs in time poly$N$ (Lemma 1.14), the query has size $M \leq$ poly$N$, and Π_B expects word size $O(\log M) = O(\log N)$. We choose a log-bounded word size large enough for running both $\Pi_{A\leq B}$

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

and Π_B, and we persistently zero out all but the intended number of bits in the registers for both computations. This only uses a few extra registers for the masks. Since Π_A is register-only and has word size $O(\log N)$, it has space efficiency $O(\log N)$. ■

The transitive property (like Lemma 1.13) holds by the same proof as Lemma 1.15, except using a B-to-C reduction in place of the program for B:

Lemma 1.16. *If $A \leq_\ell^m B$ and $B \leq_\ell^m C$ then $A \leq_\ell^m C$.* ■

1.9.2 Log-Space Oracle Reductions

The notation $A \leq_\ell^o B$ means there exists a log-space oracle reduction from A to B, where the query segment doesn't count toward the space. We only let B be a decision problem, to avoid the issue of whether the answer segment counts toward the space. (For mapping reductions, the answer segment is the output segment, which doesn't count anyway.) Such a reduction can use "write-query Reg[i]" and "query Reg[i] ← Answer", which puts the bit $B(y)$ into Reg[i] and erases the query segment. Such a reduction has log-bounded word size, and is register-only without loss of generality.

Lemma 1.17. *Every log-space oracle reduction to a decision problem is poly-time, so if B is a decision problem and $A \leq_\ell^o B$ then $A \leq_p^o B$.*

Proof. This is more subtle than for $\leq_\ell^m$ (Lemma 1.14). A configuration is the contents of the program counter and registers (assuming the reduction is register-only), which is $O(\log N)$ bits, so there are only $2^{O(\log N)} = \operatorname{poly} N$ possible configurations on an input of size N. But the reduction isn't necessarily doomed to run forever if it repeats a configuration, since the future execution can depend on the query segment, which isn't part of the configuration.

Consider any valid input. Partition the computation steps into phases delimited by queries. The last phase ends with a termination step, and each other phase ends with a query step.

If two phases start with the same configuration, then the reduction wouldn't halt: The query segment is empty at these steps, so the execution after a repeated configuration would be the same as after the first occurrence, leading to an endless loop (as in Theorem 1.8). Since the reduction halts, it has only $\operatorname{poly} N$ many phases.

Within a phase (not including the configuration after the phase's query step, which is the first configuration of the next phase), if two configurations were the same, then the reduction wouldn't halt: The query segment may be different at these two steps, but it doesn't affect the execution until the query. Thus, a repeated configuration within a phase would lead to an endless loop (as in Theorem 1.8), possibly writing an infinitely long query.

In summary, the reduction runs in $\operatorname{poly} N$ time $= (\operatorname{poly} N$ phases$) \cdot (\operatorname{poly} N$ time/phase$)$. ■

Lemma 1.18. *If $A \leq_\ell^o B$ and $B \in \mathsf{L}$ then A has a log-space program.*

Proof idea. Like in Lemma 1.15, whenever B's program wants to read a word of the query, we fetch the word by rerunning the reduction and discarding the other query words as we go. But with adaptive queries, we shouldn't rerun the reduction from the beginning, which would involve redoing earlier queries. We just rerun the reduction's latest phase, starting after the most recent query (or the beginning if we're working on the first query). At the start of each phase, we save the reduction's current configuration in extra registers, so we can restore this configuration to kick off each rerun during the next query. To restore the program counter, we do an indirect branch (§1.5.1). ■

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

Proof (ancillary). Let $\Pi_{A\le B}$ be a register-only oracle reduction from A to B, and let Π_B be a register-only program for B, both with log-bounded word size. We design a register-only program Π_A for A using these registers:

- Three copies of $\Pi_{A\le B}$'s registers, named $\mathrm{Reg}[\,]$, $\mathrm{Reg}_{\mathrm{snapshot}}[\,]$, and $\mathrm{Reg}_{\mathrm{replay}}[\,]$.
- Π_B's registers, renamed $\mathrm{Reg}_B[\,]$.
- Two more registers, nicknamed *qsite* and *qaddr*.

Π_A has "let $\mathrm{Reg}_{\mathrm{snapshot}}[0] \leftarrow \mathrm{Reg}[0]$" followed by $\Pi_{A\le B}$'s code with these substitutions:

write-query $\mathrm{Reg}[i]$	$\rightsquigarrow$	let $\mathrm{Reg}_B[0] \leftarrow \mathrm{Reg}_B[0]+1$

And for the qth "query" instruction (qth in terms of lines of code, not computation steps):

query $\mathrm{Reg}[h] \leftarrow$ Answer	$\rightsquigarrow$	a copy of Π_B's code	
		let $\mathrm{Reg}_B[k] \leftarrow 0$	$\}\ \forall k$
		let $\mathrm{Reg}_{\mathrm{snapshot}}[k] \leftarrow \mathrm{Reg}[k]$	$\}\ \forall k$
		let *qsite* $\leftarrow q$	

Here, $q \in \{1,2,\ldots\}$ is a hardcoded constant. This copy of Π_B's code uses $\mathrm{Reg}_B[\,]$ and the following substitutions:

accept	$\rightsquigarrow$	let $\mathrm{Reg}[h] \leftarrow 1$ goto line after this copy of Π_B's code
reject	$\rightsquigarrow$	let $\mathrm{Reg}[h] \leftarrow 0$ goto line after this copy of Π_B's code

And for the rth "read" instruction in this copy of Π_B's code:

read $\mathrm{Reg}[i] \leftarrow \mathrm{In}[\mathrm{Reg}[j]]$	$\rightsquigarrow$	let *qaddr* $\leftarrow 0$	
		let $\mathrm{Reg}_{\mathrm{replay}}[k] \leftarrow \mathrm{Reg}_{\mathrm{snapshot}}[k]$	$\}\ \forall k$
		if *qsite* $= s$ then goto line $\ell_{q,r,s}$	$\}\ \forall s$
		another copy of $\Pi_{A\le B}$'s code	

Here, each $s \in \{1,2,\ldots\}$ is a hardcoded constant, and $\ell_{q,r,s}$ is the number of the line following the sth "query" in this copy of $\Pi_{A\le B}$'s code (which is the rth copy within the qth copy of Π_B's code). If *qsite* $= 0$ then the block of "if"s falls through to the 0th line of the copy.

This copy of $\Pi_{A\le B}$'s code computes the word that would be at address $\mathrm{Reg}_B[j]$ of the query segment, and puts it in $\mathrm{Reg}_B[i]$. To do this, it replays $\Pi_{A\le B}$'s computation from just after the previously performed query (which was the *qsite*th "query" instruction), or from the beginning if no query was previously performed (*qsite* $= 0$). The $\mathrm{Reg}[\,]$ contents from that moment were saved in $\mathrm{Reg}_{\mathrm{snapshot}}[\,]$, and they now get recalled into $\mathrm{Reg}_{\mathrm{replay}}[\,]$, which the copy of $\Pi_{A\le B}$'s code uses. It uses *qaddr* to detect when the desired word becomes available, with the same substitution as in the proof of Lemma 1.15 (assuming Π_B never reads past its input):

write-query $\mathrm{Reg}[k]$	$\rightsquigarrow$	if *qaddr* $\neq \mathrm{Reg}_B[j]$ then goto 3 lines below let $\mathrm{Reg}_B[i] \leftarrow \mathrm{Reg}_{\mathrm{replay}}[k]$ goto line after this copy of $\Pi_{A\le B}$'s code let *qaddr* $\leftarrow$ *qaddr* $+1$

We remove any "write" instructions from this copy of $\Pi_{A\le B}$'s code.

We also update the target line numbers of all branches.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

Π_A solves A because $\Pi_{A\le B}$ solves A assuming each query is answered correctly, and Π_B answers each query correctly since each "read" gets the desired word. We handle the word size as in Lemma 1.15, using the fact that each query has size at most $\text{poly}N$ since $\Pi_{A\le B}$ runs in $\text{poly}N$ time (Lemma 1.17). We also ensure the word size is big enough for the q and s constants. Since Π_A is register-only and has word size $O(\log N)$, it has space efficiency $O(\log N)$. ■

The transitive property holds by the same proof as Lemma 1.18, except using a B-to-C reduction in place of the program for B:

Lemma 1.19. *If $A \le_\ell^o B$ and $B \le_\ell^o C$ then $A \le_\ell^o C$, assuming B and C are decision problems.* ■

1.9.3 Decision Versus Search

This chapter's finale is a log-space search-to-decision reduction for Directed Reachability.

Lemma 1.20. Directed Shortest Path $\le_\ell^o$ Directed Shortest Path Decision.

Proof. The reduction in § 1.8.3 only needs log space: When finding the distance d from s to t, it only needs registers to remember the current guess for d and copy the input to the query segment. When finding a path, it only needs registers to remember the loop counter ℓ and latest node $u_{\ell-1}$, iterate through $u_{\ell-1}$'s adjacency list, and copy things to the query segment. It can output the path as it goes along. The previous nodes $u_0, \dots, u_{\ell-2}$ need not be remembered since they were already written to the output segment. ■

Lemma 1.21. Directed Shortest Path Decision $\le_\ell^m$ Directed Reachability.

Proof. Map (G, s, t, k) to (G', s', t') where G' is a "layered" version of G. There are $k+1$ layers in G', numbered $0, 1, \dots, k$. Each layer is a copy of G but with the edges going to the next layer, and layer k has no outgoing edges. Each copy of t in G', except the one in layer k, also has a "horizontal" edge to the next layer's copy of t (as if t had a self-loop in G). Node s' corresponds to s in layer 0, and node t' corresponds to t in layer k. Non-s' nodes in layer 0 and non-t' nodes in layer k (and all their edges) are optional because they can't be on a path from s' to t'.

Figure 1.17 shows an example yes-input. The bold edges show how a path from s to t of length $2 \le k = 3$ corresponds to a path from s' to t'. The gray nodes and edges are optional.

Formally, from $G = (V, E)$ define $G' = (V', E')$ where $V' = V \times \{0, 1, \dots, k\}$ and for each $j \in \{0, \dots, k-1\}$, we have $\big((u, j), (v, j+1)\big) \in E'$ for each $(u, v) \in E$, and $\big((t, j), (t, j+1)\big) \in E'$. Define $s' = (s, 0)$ and $t' = (t, k)$.

We argue that G has a path from s to t of length $\le k$ iff G' has a path from s' to t':

$\Rightarrow$: Assume G has a path from s to t of length $d \le k$. Follow the corresponding sequence of nodes from s' through the successive layers of G' to reach the copy of t in layer d, then follow $k-d$ horizontal edges to reach t'. Formally, a path $s = u_0 \to u_1 \to \cdots \to u_d = t$ becomes a path $s' = (u_0, 0) \to (u_1, 1) \to \cdots \to (u_d, d) \to (u_d, d+1) \to \cdots \to (u_d, k) = t'$.

$\Leftarrow$: Assume G' has a path from s' to t'. The corresponding sequence of nodes in G—stopping as soon as we reach t, and trimming out any cycles—is a path from s to t of length $\le k$. Formally, $s' = (v_0, 0) \to (v_1, 1) \to \cdots \to (v_k, k) = t'$ becomes a walk $s = v_0 \to v_1 \to \cdots \to v_d = t$ where $d \le k$ is the first layer with $v_d = t$.

The reduction can be implemented in log space because while writing G', it only needs to remember the current node (u, j) and some pointers for consulting u's adjacency list in G. We may assume $k \le n-1$ (where G has n nodes), so each layer index j fits in one word. ■

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

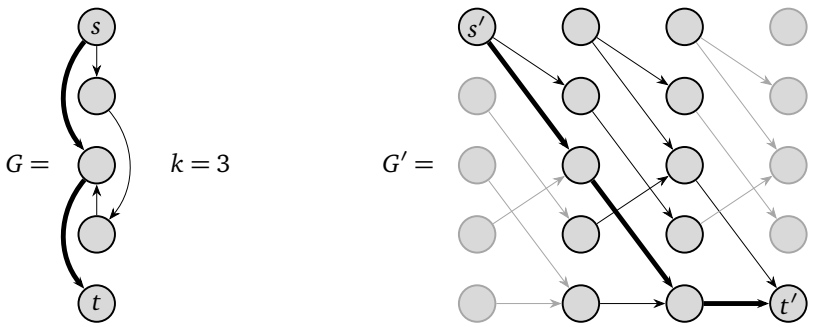


Figure 1.17: Layered version of a graph

Theorem 1.22. Directed Reachability Search $\leq_\ell^o$ Directed Reachability.

Proof. Directed Reachability Search is less stringent than Directed Shortest Path, and:

$$\begin{aligned} \text{Directed Shortest Path} &\leq_\ell^o \text{Directed Shortest Path Decision} && \text{(Lemma 1.20)} \\ &\leq_\ell^m \text{Directed Reachability} && \text{(Lemma 1.21)} \end{aligned}$$

■

It's unknown whether Directed Reachability has a log-space program. But if it does, then so does Directed Reachability Search (Theorem 1.22 and Lemma 1.18).

Exercises

1.1 Implement an algorithm in our official model for finding the minimum and maximum of a list of numbers with $O(N)$ time efficiency and $O(\log N)$ space efficiency.

Min and Max	(1.25)
Input:	Nonnegative integers $x_1, x_2, \ldots, x_N$ such that $x_i \leq N$ for each i
Output:	$\min(x_1, x_2, \ldots, x_N)$ and $\max(x_1, x_2, \ldots, x_N)$

1.2 Show Containment in Sorted List $\in \mathsf{TISP}[\log N, \log N]$ by implementing the binary search algorithm in our official model.

Containment in Sorted List	(1.26)
Input:	Nonnegative integers $x_1 \leq x_2 \leq \cdots \leq x_n$ and y such that $x_n \leq n$ and $y \leq n$
Output:	Does there exist an i such that $x_i = y$?

The input size is $N = n + 1$, the problem's word size is $\lceil \log N \rceil$, the input encoding is $x_1, x_2, \ldots, x_n, y$ with one value per word, and it's promised that $x_1 \leq x_2 \leq \cdots \leq x_n \leq n$ and $y \leq n$.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

1.3 Implement an algorithm in our official model showing Greater Than $\in \mathsf{TISP}[N, \log N]$, where the input encoding is the same as for Multiplication in §1.3.3.

Greater Than	(1.27)
Input:	Nonnegative integers x and y represented in binary
Output:	Is $x > y$?

1.4 Show Element Distinctness $\in \mathsf{TIME}[N] \cap \mathsf{SPACE}[\log N]$ and implement both algorithms in our official model.

Element Distinctness	(1.28)
Input:	Nonnegative integers $x_1, x_2, \ldots, x_N$ such that $x_i \leq N^2$ for each i
Output:	Are $x_1, x_2, \ldots, x_N$ all distinct (that is, no two are equal)?

1.5 Implement the extended BFS algorithm for Directed Reachability Search (and Directed Shortest Path) in our official model. For convenience, assume the output encoding is the labels of nodes on a path backward from t to s (so output segment address 0 gets t). In particular, the output size may depend on the input, not just on the input size. Your program should reject if there's no path from s to t, and use the halt instruction otherwise.

1.6 This is about dags and topological orders (§A.2.3). Implement an algorithm solving the following search problem with time efficiency $O(N)$ and word size $O(\log N)$ in our official model. Assume the same input encoding as Directed Reachability except there's no s or t.

Topological Order	(1.29)
Input:	Directed graph G
Output:	A topological order of G, or report that none exists

For $u \in \{1, 2, \ldots, n\}$, let Mem$[u]$ be node u's indegree. Use Mem$[n+1], \ldots,$ Mem$[2n]$ for a queue of nodes that have indegree 0. When a node u is dequeued and processed, the algorithm puts u next in the topological order and effectively deletes u by decrementing the indegree of each node v to which u has an edge, and enqueueing v if its indegree became 0. Your program should reject if G is not a dag, and use the halt instruction otherwise. It's OK if your program executes some write instructions before rejecting.

1.7 Show Triangle Detection $\in \mathsf{TISP}[N^{1.5}, \log N]$. Write pseudocode (not actual code in our model) and analyze the time and space efficiencies. Assume the graph has no multi-edges or self-loops.

Triangle Detection	(1.30)
Input:	Undirected graph G encoded as an adjacency matrix
Output:	Does G have a cycle of length 3?

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

1.8 **(a)** Show Most Significant Bit of Median $\in \mathsf{TISP}[\sqrt{N}, \log N]$ using pseudocode.

Most Significant Bit of Median	(1.31)
Input:	Nonnegative integers $x_1, \ldots, x_n$ where each x_i is represented in binary with n bits (so $x_i < 2^n$) and n is odd
Output:	The most significant bit (position $n-1$) of the median of $x_1, \ldots, x_n$

(b) Show Least Significant Bit of Median $\in \mathsf{TIME}[N] \cap \mathsf{SPACE}[\log N]$ using pseudocode.

1.9 Design an algorithm (using pseudocode) for General Iterated Addition with $O(N)$ time efficiency and $O(\log N)$ word size.

General Iterated Addition	(1.32)
Input:	Nonnegative integers $x_1, x_2, \ldots, x_n$ represented in binary
Output:	$x_1 + x_2 + \cdots + x_n$ in binary

The input numbers may have different bit-lengths from each other (unlike in Iterated Addition), so the input size N is essentially the sum of the bit-lengths. The specific input encoding isn't important, but assume it's n, then pointers to the beginnings of the individual x_i numbers, then the numbers themselves with one word per bit (reminiscent of our Directed Reachability encoding), so $N = 1 + n + \sum_{i=1}^{n}(\text{bit-length of } x_i)$.

1.10 A *forest* is an undirected graph with no cycles—that is, each connected component is a tree. Show Forest Reachability $\in \mathsf{L}$ using pseudocode. (It's considerably harder to prove the more general result that Undirected Reachability $\in \mathsf{L}$.)

Forest Reachability	(1.33)
Input:	Forest G, nodes s and t
Output:	Is t reachable from s in G?

Hint: You can escape a maze by keeping your left hand on the wall, without making a map.

1.11 Show that each of the following instructions can be added to the model of computation without increasing its power. Specifically, show that every program using the instruction can be converted to a program that's equivalent (as in Theorem 1.2) but doesn't use the instruction. Do not rely on auxiliary registers for this exercise—develop substitutions that only employ the registers the original program was already using. Handle all possible combinations of register indices being equal and constants standing in for any registers.

(a) Bitwise-iff: "let $\text{Reg}[i] \leftarrow \text{Reg}[j] \iff \text{Reg}[k]$"

(b) Bitwise-implication: "let $\text{Reg}[i] \leftarrow \text{Reg}[j] \Rightarrow \text{Reg}[k]$"

1.12 In Java, >> denotes *signed shift right*, meaning the leftmost bits get filled in with copies of the leftmost bit of the number being shifted, while >>> denotes unsigned shift right (which always fills in with 0s). Since our model uses $\gg$ for unsigned shift right, let's do the opposite of Java by using $\ggg$ for signed shift right. Show that every program using

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

⋙ can be converted to a program that's equivalent (as in Theorem 1.2) but doesn't use ⋙. You may employ an auxiliary register.

1.13 For each of the following instructions, show that it can be removed from the model of computation without decreasing the model's power. (This is not asking you to remove all three simultaneously.) Specifically, show that every program using the instruction can be converted to an equivalent tight program (as in Theorem 1.2) that doesn't use the instruction. This involves adjusting the proof of Theorem 1.2 to avoid reintroducing the instruction.

(a) Bitwise-and

(b) Bitwise-or

(c) Bitwise-not

1.14 Prove that for every log-space program that uses "write Out[Reg[j]] ← Reg[i]" instructions (and doesn't use our official "write" instruction), there exists a log-space program that always produces the same output as the original program, has the same word size, and uses only our official "write" instruction.

1.15 Prove that $\mathsf{P} = \mathsf{L}$ (where the classes contain promise problems) iff $\mathsf{P} = \mathsf{L}$ with the classes redefined to contain total problems only.

1.16 For a directed graph in which each edge is labeled with a nonnegative "height" value, define a path's height as the height of a highest edge on the path. Design an algorithm (using pseudocode) for the following function problem with $O(N \log N)$ time efficiency and $O(\log N)$ word size, by giving an oracle reduction to Directed Reachability.

Directed Min Path Height	(1.34)
Input:	Directed graph G in which each edge has a height label in $\{1, \ldots, m\}$ where m is the number of edges, nodes s and t
Output:	Minimum height of any path from s to t in G, or report that no such path exists

1.17 Implement the mapping reduction from Multiplication to Iterated Addition from §1.8.2 in our official model.

1.18 **(a)** Explain why for all decision problems A and B, if $A \leq_p^o B$ and $B \in \mathsf{EXP}$ then $A \in \mathsf{EXP}$.

(b) Let $\mathsf{QUASIP} = \mathsf{TIME}[2^{\text{polylog}\,N}]$, and let $A \leq_{qp}^o B$ mean there exists an oracle reduction from A to B with $2^{\text{polylog}\,N}$ time efficiency and polylog-bounded word size. Explain why for all decision problems A and B, if $A \leq_{qp}^o B$ and $B \in \mathsf{QUASIP}$ then $A \in \mathsf{QUASIP}$.

1.19 **(a)** Prove that Undirected Connectivity $\leq_\ell^o$ Undirected Reachability using only $n-1$ oracle queries, where n is the number of nodes in the input graph.

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

Undirected Connectivity	(1.35)
Input:	Undirected graph G
Output:	Is G connected?

(b) Prove that Undirected Connectivity $\leq_\ell^m$ Undirected Reachability.

1.20 This exercise justifies why it's generally safe to assume a problem has well-defined time and space complexities: If different programs are more efficient on different inputs, we can combine them to get "the best of both worlds." Let Π_1 and Π_2 be programs that solve the same decision problem and have the same word size. Prove that there exist programs Π_3 and Π_4 that also solve that problem and have word sizes larger by only an additive constant, and such that the following hold for every valid input x. (Feel free to use Lemma 1.5.)

(a) The running time of $\Pi_3(x)$ is at most a constant factor higher than the minimum of the running time of $\Pi_1(x)$ and the running time of $\Pi_2(x)$. Hint: This is like how an operating system creates the illusion of parallelism (on a single-core processor) by switching back and forth between different processes.

(b) The memory space of $\Pi_4(x)$ is at most a constant factor higher than the minimum of the memory space of $\Pi_1(x)$ and the memory space of $\Pi_2(x)$.

(c) Update your programs Π_3 and Π_4 to handle function and search problems.

Notes

Now that we're "off to the races," here are references for further reading on traditional complexity theory: Texts surveying much of the field (like this book does) include [407, 149, 557, 319, 196, 28, 376, 559]. Texts developing complexity theory in the historical context of automata and computability include [252, 334, 497, 249, 117].

An input to a computational problem is also called an "instance." A decision problem is also called a "language" because it can be viewed as a set of strings—the encodings of all 1-inputs. A search problem is also called a "relation."

An algorithm's efficiency is also sometimes called the algorithm's "complexity." We prefer to reserve the word "complexity" for assessing computational problems (best efficiency of any correct algorithm), to draw a clear line between these two concepts.

Like Multiplication, the Division problem can be solved in $O(\log N)$ space [124], but that's much more difficult to prove. Some sophisticated Multiplication and Division algorithms have time efficiencies pretty close to $O(N)$ [472, 178, 235].

In the computational complexity literature, algorithms (or programs) are often called "machines" since the original general-purpose model of computation is called "Turing machines" [525]. Our model from §1.4 is called "Word RAM" (standing for Random-Access Machine, not Random-Access Memory). This way of expressing computations dates back to the genesis of digital computers, but credit for introducing the Word RAM model in the theory literature usually goes to [173]. This was predated by other RAM models where words can store arbitrarily large values [140].

Computability theory is about which computational problems can be solved by algorithms at all, regardless of efficiency. Our model is unsuitable for studying computability, because the word

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003

size automatically limits the space (since an address must fit in a word), which in turn limits the time (Theorem 1.8). For computability theory (with no bounds on a computation's resources) we need a model with a mechanism for accessing arbitrarily large memory addresses, such as sequential access, or pointers spanning multiple words, or letting the word size grow during the computation. Our model is suitable for complexity theory since we can select the word size large enough to accommodate whatever resource bounds we have in mind.

Sometimes "easy" and "hard" are used as synonyms for tractable and intractable, respectively. The idea that "polynomial time" corresponds to "tractable" is typically attributed to [131, 153, 421, 154]. The TIME and SPACE complexity classes are sometimes called "DTIME" and "DSPACE" where D stands for "deterministic." TISP[poly N, polylog N] is also called SC, which stands for "Steve's class" in honor of Stephen Cook.

We defined complexity classes such as P to contain promise problems, but lots of the literature uses names like prP ("promise P") for this and reserves names such as P for classes of total problems.

Oracle reductions are also called "Turing reductions" (after [526]) and "Cook reductions" (after [136]). Mapping reductions are also called "many-one reductions" and "Karp reductions" (after [292]).

Least Significant Bit of Median $\in$ L (Exercise 1.8.b) appears in [404].

Downloaded from https://www.cambridge.org/core. UNIVERSIDAD AUTONOMA METROPOLITANA, on 09 Nov 2026 at 18:51:08, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009752329.003