Mathematics of the impossible: Computational Complexity

Emanuele "Manu" Viola

January 26, 2023

Contents

0.1 Conventions, choices, and caveats . . . 1

1 A teaser **5**

2 The alphabet of Time **7**

2.1 Tape machines (TMs) . . . 8

2.1.1 You don't need much to have it all . . . 9

2.1.2 Time complexity, P, and EXP . . . 10

2.1.3 The universal TM . . . 12

2.2 Multi-tape machines (MTMs) . . . 12

2.3 Circuits . . . 13

2.4 Rapid-access machines (RAMs) . . . 16

2.5 A challenge to the computability thesis . . . 18

2.5.1 Robustness of BPP: Error reduction and tail bounds for the sum of random variables . . . 19

2.5.2 Does randomness buy time? . . . 20

2.5.3 Polynomial identity testing . . . 20

2.5.4 Simulating BPP by circuits . . . 22

2.5.5 Questions raised by randomness . . . 23

2.6 Inclusion extend "upwards," separations downwards . . . 23

2.7 Problems . . . 24

0.1 Conventions, choices, and caveats

I write this section before the work is complete, so some of it may change.

This book covers basic results in complexity theory and can be used for a course on the subject. At the same time, it is perhaps *sui generis* in that it also tells a story of the quest for impossibility results, includes some personal reflections, and makes some non-standard choices about topics and technical details. Some of this is discussed next.

To test your understanding of the material... this book is interspersed with mistakes, some subtle, some blatant, some not even mistakes but worrying glimpses into the author's mind. Please send all bug reports and comments to *(my five-letter last name)@ccs.neu.edu* to be included in the list of heroes.

The c notation. The mathematical symbol c has a special meaning in this text. Every *occurrence* of c denotes a real number > 0. There exist choices for these numbers such that the claims in this book are (or are meant to be) correct. This replaces, is more compact than, and is less prone to abuse than the big-Oh notation (sloppiness hides inside brackets).

Example 0.1. "For all sufficiently large n" can be written as $n \geq c$.

"For every ϵ and all sufficiently large n" can be written as $n \geq c_\epsilon$.

The following are correct statements:

"It is an open problem to show that some function in NP requires circuits of size cn." At the moment of this writing, one can replace this occurrence with 5. Note such a claim will remain true if someone proves a $6n$ lower bounds. One just needs to "recompile" the constants in this book.

"$c > 1 + c$", e.g. assign 2 to the first occurrence, 1 to the second.

"$100n^{15} < n^c$", for all large enough n. Assign $c = 16$.

The following are not true:

"$c < 1/n$ for every n". No matter what we assign c to, we can pick a large enough n. Note the assignment to c is absolute, independent of n.

More generally, when subscripted this notation indicates a function of the subscript. There exist choices for these functions such that the claims in this book are (or are meant to be) correct. Again, each occurrence can indicate a different function.

For the reader who prefers the big-Oh notation a quick an dirty fix is to replace every occurrence of c in this book with $O(1)$.

The alphabet of TMs. I define TMs with a fixed alphabet. This choice slightly simplifies the exposition (one parameter vs. two), while being more in line with common experience (it is more common experience to increase the length of a program than its alphabet). This choice affects the proof of Theorem **??**. But it isn't clear that the details are any worse.

Partial vs. total functions (a.k.a. on promise problems).

> Recall that *promise problems offer the most direct way of formulating natural computational problems. [...]* In spite of the foregoing opinions, we adopt the convention of focusing on standard decision and search problems. *[3]*

I define complexity w.r.t. *partial* functions whereas most texts consider *total* functions, i.e. we consider computing functions with arbitrary domains rather than any possible string. This is sometimes called "promise problems." This affects many things, for example the hierarchy for BPTime (Exercise **??**).

References and names. I decided to keep references in the main text to a minimum, just to avoid having a long list later with items "Result X is due to Y," but relegate discussion to bibliographic notes. I have also decided to not spell out names of authors, which is increasingly awkward. Central results, such as the PCP theorem, are co-authored by five or more people.

Polynomial. It is customary in complexity theory to bound quantities by a polynomial, as in polynomial time, when in fact the only terms that matters is the leading time. This also lends itself to confusion since polynomials with many terms are useful for many other things. I use *power* instead of polynomial, as in power time.

Random-access machines. "Random access" also leads to strange expressions like "randomized random-access" [2].

Reductions. Are presented as an implication.

Randomness and circuits. While randomness and circuits are everywhere in current research, and seem to be on everyone's mind, they are sometimes still relegated to later chapters, almost as an afterthought. This book starts with them right away, and attempts to weave them through the narrative.

Data structures Their study, especially negative results, squarely belongs to complexity theory. Yet data structures are strangely omitted in common textbooks. Results on data structures even tend to miss main venues for complexity theory to land instead on more algorithmic venues! We hope this book helps to revert this trend.

Algorithms & Complexity ...are of course two sides of the same coin. The rule of thumb I follow is to present algorithms that are *surprising* and *challenge our intuition of computation*, and ideally match lower bounds, even though they may not be immediately deployed.

Exercises and problems. Exercises are interspersed within the narrative and serve as "concept check." They are not meant to be difficult or new, though some are. Problems are collected at the end and tend to be harder and more original, though some are not.

Summary of some terminological and not choices. Here it is:

Some other sources	this book	acronym
$O(1)$, $\Omega(1)$	c	
Turing machine	tape machine	TM
random-access machine	rapid-access machine	RAM
polynomial time	power time	P
mapping reduction (sometimes)	A reduces to B in P means $B \in \mathrm{P} \Rightarrow A \in \mathrm{P}$	
Extended Church-Turing thesis	Power-time computability thesis	
pairwise independent	pairwise uniform	
FP, promise-P	P	
TM with any alphabet	TM with fixed alphabet	
classes have total functions	classes have partial functions	

Copyright 2022-present by Emanuele Viola

Chapter 1

A teaser

Consider a computer with *three* bits of memory. There's also a clock, beating $1, 2, 3, \ldots$ In one clock cycle the computer can read one bit of the input and update its memory arbitrarily based on the value of the bit and the current memory, or stop and return a value.

Let's give a few examples of what such computer can do.

First, it can compute the And function on n bits:

```
Computing And of (x_1,x_2,...,x_n)

For i = 1,2,... until n
   Read x_i
   If x_i = 0 return 0
Return 1
```

We didn't really use the memory. Let's consider a slightly more complicated example. A word is *palindrome* if it reads the same both ways, like *racecar*, *non*, *anna,* and so on. Similarly, example of palindrome bit strings are $11, 0110$, and so on.

Let's show that the computer can decide if a given string is palindrome quickly, in n steps

```
Deciding if (x_1, x_2, ..., x_n) is palindrome:

For i = 1,2,... until i > n/2
   Read x_i and write it in memory bit m
   If m ≠ x_{n-i} return 0
Return 1
```

That was easy. Now consider the Majority function on n bits, which is 1 iff the sum of the input bits is $> n/2$ and 0 otherwise. Majority, like any other function on n bits, can be computed on such a computer in time *exponential* in n. To do that, you do a pass on the input and check if it's all zero, using the program for And given above. If it is, return 0. If it is not, you do another pass now checking if it's all zero except the last bit is 1. If it is, return 0. You continue this way until you exhausted all the $2^n/2$ possible inputs with Majority equals to 0. If you never returned 0 you can now safely return 1.

As we said, this works for any function, but it's terribly inefficient. Can we do better for Majority? Can we compute it in time which is just a power of n?

Exercise 1.1. Convince yourself that this is impossible. Hint: If you start counting bits, you'll soon run out of memory.

If you solved the exercise, you are not alone.

And yet, we will see the following shocking result:

Shocking theorem:
Majority can be computed on such a computer in time n^c.

And this is not a trick tailored to majority. Many other problems, apparently much more complicated, can also be solved in the same time.

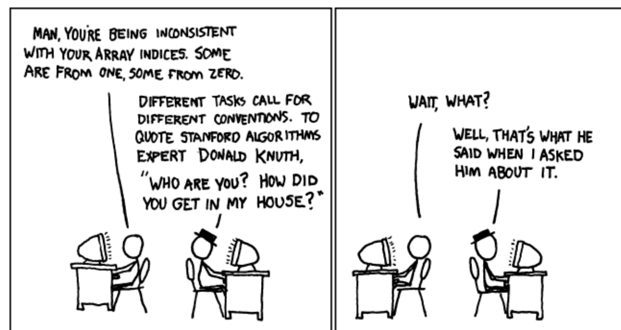
But, there's something possibly even more shocking.

Shocking situation:
It is consistent with our state of knowledge that every "textbook algorithm" can be solved in time n^c on such a computer! Nobody can disprove that. (Textbook algorithms include sorting, maxflow, dynamic programming algorithms like longest common subsequence etc., graph problems, numerical problems, etc.)

The **Shocking theorem** gives some explanation for the **Shocking situation**. It will be hard to rule out efficient programs on this model, since they are so powerful and counterintuitive. In fact, we will see later that this can be formalized. Basically, we will show that the model is so strong that it can compute functions that provably escape the reach of current mathematics... if you believe certain things, like that it's hard to factor numbers. This now enters some of the *mysticism* that surrounds complexity theory, where different beliefs and conjectures are pitted against each other in a battle for ground truth.

Chapter 2

The alphabet of Time



https://xkcd.com/163/

The details of the model of computation are not too important if you don't care about power differences in running times, such as the difference between solving a problem on an input of length n in time cn vs. time cn^2. But they matter if you do.

The fundamentals features of computation are two:

- **Locality.** Computation proceeds in small, local steps. Each step only depends on and affects a small amount of "data." For example, in the grade-school algorithm for addition, each step only involves a constant number of digits.

- **Generality.** The computational process is general in that it applies to many different problems. At one extreme, we can think of a single algorithm which applies to an infinite number of inputs. This is called *uniform* computation. Or we can design algorithms that work on a finite set of inputs. This makes sense if the description of the algorithm is much smaller than the description of the inputs that can be processed by it. This setting is usually referred to as *non-uniform* computation.

Keep in mind these two principles when reading the next models.

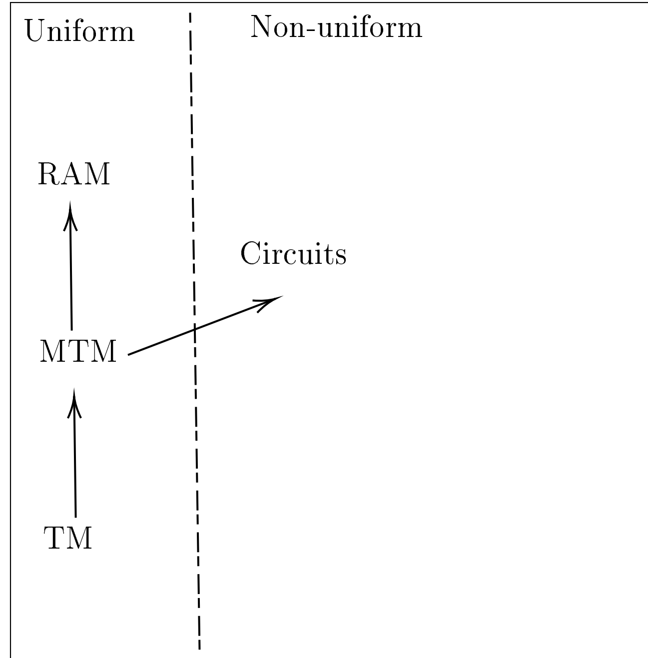


Figure 2.1: Computational models for Time. An arrow from A to B means that B can simulate A efficiently (from time t to $t \log^c t$).

2.1 Tape machines (TMs)

Tape machines are equipped with an infinite tape of cells with symbols from the *tape alphabet* A, and a *tape head* lying on exactly one cell. The machine is in one of several states, which you can think of as lines in a programming language. In one step the machine writes a symbol where the head points, changes state, and moves the head one cell to the right or left. Alternatively, it can stop. Such action depends only on the state of the machine and the tape symbol under the head.

We are interested in studying the *resources* required for computing. Several resources are of interest, like time and space. In this chapter we begin with time.

Definition 2.1. *[9]A tape machine* (TM) with s states is a map (known as *transition* or *step*)

$$\sigma : \{\underline{1}, \underline{2}, \ldots, \underline{s}\} \times A \to A \times \{\text{Left}, \text{Right}, \text{Stop}\} \times \{\underline{1}, \underline{2}, \ldots, \underline{s}\},$$

where $A := \{0, 1, \#, -, _\}$ is the *tape alphabet.* The alphabet symbol _ is known as *blank.*

A *configuration* of a TM encodes its tape content, the position of the head on the tape, and the current state. It can be written as a triple $(M, i, \underline{j})$ where M maps the integers to A and specifies the tape contents, i is an integer indicating the position of the head on the tape, and $\underline{j}$ is the state of the machine.

A configuration $(\mu, i, \underline{j})$ *yields* $(\mu', i+1, \underline{j}')$ if $\sigma(\underline{j}, \mu[i]) = (a, \text{Right}, \underline{j}')$ and $\mu'[i] = a$ and $\mu' = \mu$ elsewhere, and similarly it yields $(\mu', i-1, \underline{j}')$ if $\sigma(\underline{j}, \mu[i]) = (a, \text{Left}, \underline{j}')$ and $\mu'[i] = a$ and $\mu' = \mu$ elsewhere, and finally it yields itself if $\sigma(\underline{j}, \mu[i]) = (a, \text{Stop}, \underline{j}')$.

We say that a TM computes $y \in \{0,1\}^*$ on input $x \in \{0,1\}^*$ in *time* t (or in t steps) if, starting in configuration $(\mu, 0, \underline{1})$ where $x = \mu[0]\mu[1]\cdots\mu[|x|-1]$ and μ is blank elsewhere, it yields a sequence of t configurations where the last one is $(\mu, i, \underline{j})$ where $\sigma(\mu[i], \underline{j})$ has a Stop instruction, and $y = \mu[i]\mu[i+1]\cdots\mu[i+|y|-1]$ and μ is blank elsewhere.

Describing TMs by giving the transition function quickly becomes complicated and uninformative. Instead, we give a high-level description of how the TM works. The important points to address are how the head moves, and how information is moved across the tape.

Example 2.1. On input $x \in \{0,1\}^*$ we wish to compute $x+1$ (i.e., we think of x as an integer in binary, and increment by one). This can be accomplished by a TM with c states as follows. Move the head to the least significant bit of x. If you read a 0, write a 1, move the head to the beginning, and stop. If instead you read a 1, write a 0, move the head by one symbol, and repeat. If you reach the beginning of the input, shift the input by one symbol, append a 1, move the head to the beginning and stop.

The TM only does a constant number of passes over the input, so the running time is $c|x|$.

Example 2.2. On an input $x \in \{0,1\}^*$ we wish to decide if it has the the same number of zeros and ones. This can be done as follows. Do a pass on the input, and cross off one 0 and one 1 (by replacing them with tape symbol #). If you didn't find any 0 or or 1, accept (that is, write 1 on the tape and stop). If only find a 0 but not a 1, or vice versa, reject.

Since every time we do a pass we cross at least two symbols, the running time is cn^2.

Exercise 2.1. Describe a TM that decides if a string $x \in \{0,1\}^*$ is palindrome, and bound its running time.

Exercise 2.2. Describe a TM that on input $x \in \{0,1\}^*$ computes $n := |x|$ in binary in time $cn \log n$.

TMs can compute any function if they have sufficiently many states:

Exercise 2.3. Prove that every function $f : \{0,1\}^n \to \{0,1\}$ can be computed by a TM in time n using 2^{n+1} states.

2.1.1 You don't need much to have it all

How powerful are tape machines? Perhaps surprisingly, they are all-powerful.

Power-time computability thesis. For any "realistic" computational model C there is $d > 0$ such that: Anything that can be computed on C in time t can also be computed on a TM in time t^d.

This is a *thesis*, not a *theorem*. The meaning of "realistic" is a matter of debate, and one challenge to the thesis is discussed in section §2.5.

However, the thesis can be proved for many standard computational models, which include all modern programming languages. The proofs aren't hard. One just tediously goes through each instruction in the target model and gives a TM implementation. We prove a representative case below (Theorem 2.6) for *rapid-access machines* (RAMs), which are close to how computers operate, and from which the jump to a programming language is short.

Given the thesis, why bother with TMs? Why not just use RAMs or a programming language as our model? In fact, we will basically do that. Our default for complexity will be RAMs. However, some of the benefits of TMs remain

- TMs are easier to define – just imagine how more complicated Definition 2.1 would be were we to use a different model. Whereas for TMs we can give a short self-contained definition, for other models we have to resort to skipping details. There is also some arbitrariness in the definition of other models. What operations exactly are allowed?

- TMs allow us to pinpoint more precisely the limits of computation. Results such as Theorem **??** are easier to prove for TMs. A proof for RAM would first go by simulating RAM by a TM.

- Finally, TMs allow us to better pinpoint the limits of our knowledge about computation; we will see several examples of this.

In short, RAMs and programming languages are useful to carry computation, TMs to analyze it.

2.1.2 Time complexity, P, and EXP

We now define our first complexity classes. We are interested in solving a variety of computational tasks on TMs. So we make some remarks before the definition.

- We often need to compute *structured* objects, like tuples, graphs, matrices, etc. One can encode such objects in binary by using multiple bits. We will assume that such encodings are fixed and allow ourselves to speak of such structures objects. For example, we can encode a tuple $(x_1, x_2, \ldots, x_t)$ where $x_i \in \{0,1\}^*$ by repeating each bit in each x_i twice, and separate elements with 01.

- We can view machines as *computing functions*, or *solving problems,* or *deciding sets*, or *deciding languages*. These are all equivalent notions. For example, for a set A, the problem of deciding if an input x belongs to A, written $x \in A$, is equivalent to computing the boolean characteristic function f_A which outputs 1 if the input belongs to A, and 0 otherwise. We will use this terminology interchangeably. In general, "computing a function" is more appropriate terminology when the function is not boolean.

- We allow *partial functions,* i.e., functions with a domain X that is a strict subset of $\{0,1\}^*$. This is a natural choice for many problems, cf. discussion in section §0.1.

- We measure the running time of the machine in terms of the *input length,* usually denoted n. Input length can be a coarse measure: it is often natural to express the running time in terms of other parameters (for example, the time to factor a number could be better expressed in terms of the number of factors of the input, rather than its bit length). However for most of the discussion this coarse measure suffices, and we will discuss explicitly when it does not.

- We allow non-boolean outputs. However the running time is still only measured in terms of the input. (Another option which sometimes makes sense, it to bound the time in terms of the output length as well, which allows us to speak meaningfully of computing functions with very large outputs, such as exponentiation.)

- More generally, we are interested in computing not just functions but *relations*. That is, given an input x we wish to compute some y that belongs to a *set* $f(x)$. For example, the problem at hand might have more than one solution, and we just want to compute any of them.

- We are only interested in sufficiently large n, because one can always hard-wire solutions for inputs of fixed size, see Exercise 2.3. This allows us to speak of running times like $t(n) = n^2/1000$ without worrying that it is not suitable when n is small (for example, $t(10) = 100/1000 < 1$, so the TM could not even get started). This is reflected in the $n \geq c_M$ in the definition.

With this in mind, we now give the definition.

Definition 2.2. [Time complexity classes – boolean] Let $t : \mathbb{N} \to \mathbb{N}$ be a function. ($\mathbb{N}$ denotes the natural numbers $\{0, 1, 2, \ldots\}$.) TM-Time(t) denotes the functions f that map bit strings x from a subset $X \subseteq \{0,1\}^*$ to a set $f(x)$ for which there exists a TM M such that, on any input $x \in X$ of length $\geq c_M$, M computes y within $t(|x|)$ steps and $y \in f(x)$.

$$\mathrm{P} := \bigcup_{d \geq 1} \text{TM-Time}(n^d),$$

$$\mathrm{Exp} := \bigcup_{d \geq 1} \text{TM-Time}(2^{n^d}).$$

We will not need to deal with relations and partial functions until later in this text.

Also, working with boolean functions, i.e., functions f with range {0,1} slightly simplifies the exposition of a number of results we will see later. To avoid an explosion of complexity classes, we adopt the following convention.

Convention about complexity classes:
Unless specified otherwise, inclusions and separations among complexity classes refer to boolean functions. For example, an expression like P $\subseteq$ NP means that every boolean function in P is in NP.

As hinted before, the definition of P is robust. In the next few sections we discuss this robustness in more detail, and also introduce a number of other central computational models.

2.1.3 The universal TM

Universal machines can simulate any other machine on any input. These machines play a critical role in some results we will see later. They also have historical significance: before them machines were tailored to specific tasks. One can think of such machines as epitomizing the victory of *software* over *hardware:* A single machine (hardware) can be programmed (software) to simulate any other machine.

Lemma 2.1. There is a TM U that on input (M, t, x) where M is a TM, t is an integer, and x is a string:
-Stops in time $|M|^c \cdot t \cdot |t|$,
-Outputs $M(x)$ if the latter stops within t steps on input x.

Proof. To achieve the desired speed, the idea is to maintain the invariant that M and t are always next to the tape head. After the simulation of each step of M the tape of U will contain

$$(x, M, \underline{i}, t', y)$$

where M is in state $\underline{i}$, the tape of M contains xy and the head is on the left-most symbol of y. The integer t' is the counter decreased at every step. Computing the transition of M takes time $|M|^c$. Decreasing the counter takes time $c|t|$. To move M and t next to the tape head takes $c|M||t|$ time. **QED**

2.2 Multi-tape machines (MTMs)

Definition 2.3. A k-TM is like a TM but with k tapes, where the heads on the tapes move independently. The input is placed on the first tape, and all other tapes are initialized to _. The output is on the first tape. k-TM-Time is defined analogously to TM-Time.

Exercise 2.4. Prove that Palindromes is in 2-TM-Time(cn). Compare this to the run-time from the the TM in Exercise 2.1.

The following result implies in particular that P is unchanged if we define it in terms of TMs or k-TMs.

Theorem 2.1. k-TM-Time$(t(n)) \subseteq$ TM-Time$(c_k t^2(n))$ for any $t(n) \geq n$ and k.

Exercise 2.5. Prove this. Note: Use TMs as we defined them, this may make a step of the proof less obvious than it may seem.

A much less obvious simulation is given by the following fundamental result about MTMs. It shows how to reduce the number of tapes two *two*, at very little cost in time. Moreover, the head movements of the simulator are restricted in a sense that at first sight appears too strong.

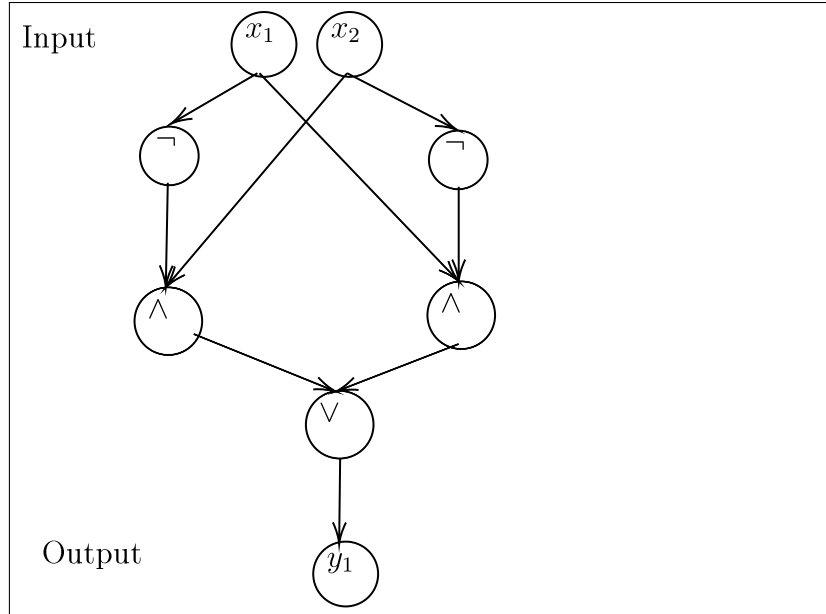


Figure 2.2: A circuit computing the Xor of two bits.

Theorem 2.2. [4, 7]k-TM-Time$(t(n)) \subseteq$ 2-TM-Time$(c_k t(n) \log t(n))$, for every function $t(n) \geq n$. Moreover, the 2-TM is *oblivious:* the movement of each tape head depends only on the length of the input.

Using this results one can prove the existence of universal MTMs similar to the universal TMs in Lemma 2.1.

2.3 Circuits

We now define circuits. It may be helpful to refer to figure 2.2 and figure 2.3.

Definition 2.4. A *circuit,* abbreviated Ckt, is a directed acyclic graph where each node is one of the following types: an input variable (fan-in 0), an output variable (fan-in 1), a negation gate $\neg$ (fan-in 1), an And gate $\wedge$ (fan-in 2), or an Or gate $\vee$ (fan-in 2). The *fan-in* of a gate is the number of edges pointing to the gate, the *fan-out* is the number of edges pointing away from the gate.

An *alternating circuit*, abbreviated AltCkt, is a circuit with unbounded fan-in Or and And gates arranged in alternating layers (that is, the gates at a fixed distance from the input all have the same type). For each input variable x_i the circuit has both x_i and $\neg\, x_i$ as input.

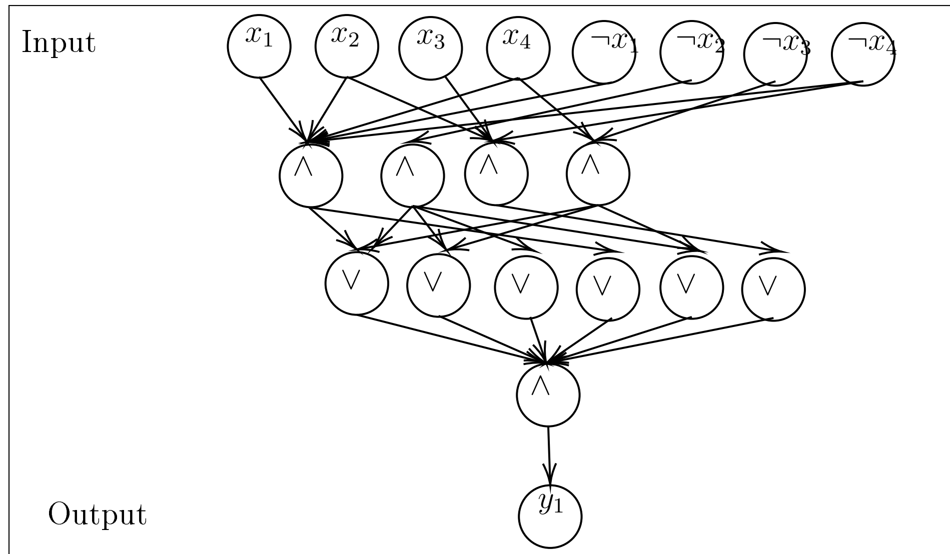


Figure 2.3: An alternating circuit.

A DNF (resp. CNF) is an AltCkt whose output is Or (resp. And). The non-ouput gates are called *terms* (resp. *clauses*).

CktGates($g(n)$) denotes the set of function $f : \{0,1\}^* \to \{0,1\}^*$ that, for all sufficiently large n, on inputs of length n have circuits with $g(n)$ gates; input and output gates are not counted.

The *size* of a circuit is the number of gates.

Exercise 2.6. [Pushing negation gates at the input] Show that for any circuit $C : \{0,1\}^n \to \{0,1\}$ with g gates and depth d there is a monotone circuit C' (that is, a circuit without Not gates) with $2g$ gates and depth d such that for any $x \in \{0,1\}^n : C(x_1, x_2, \ldots, x_n) = C'(x_1, \neg x_1, x_2, \neg x_2 \ldots, x_n, \neg x_n)$.

Often we will consider computing functions on *small inputs*. In such cases, we can often forget about details and simply appeal to the following result, which gives exponential-size circuits which are however good enough if the input is really small. In a way, the usefulness of the result goes back to the locality of computation. The result, which is a circuit analogue of Exercise 2.3, will be extensively used in this book.

Theorem 2.3. Every function $f : \{0,1\}^n \to \{0,1\}$ can be computed by

(1) [6] circuits of size $\leq (1 + o(1))2^n/n$, and

(2) A DNF or CNF with $\leq 2^n + 1$ gates (in particular, circuits of size $\leq n2^n$).

Exercise 2.7. Prove that the Or function on n bits has circuits of size cn. Prove Item (2) in Theorem 2.3. Prove a weaker version of Item (1) in Theorem 2.3 with bound $cn2^n$.

Exercise 2.8. Prove that the sum of two n-bit integers can be computed by circuits with cn gates, and by alternating circuits of depth c and size n^c.

We now show that circuits can simulates TMs. We begin with a simple but instructive simulation which incurs a quadratic loss, then discuss a more complicated and sharper one.

Theorem 2.4. Suppose an s-state TM computes $f : \{0,1\}^* \to \{0,1\}$ in time $t \geq n$. Then $f \in \text{CktGates}(c_s t^2(n))$.

For this proof it is convenient to represent a configuration of a TM in a slightly different way, as a string over the alphabet $A \times \{0, 1, \ldots, s\}$. String

$$(a_1, 0)(a_2, 0) \ldots (a_{i-1}, 0)(a_i, j)(a_{i+1}, 0) \ldots (a_m, 0)$$

with $j > 0$ indicates that (1) the tape content is $a_1 a_2 \ldots a_m$ with blanks on either side, (2) the machine is in state $\underline{j}$, and (3) the head of the machine is on the i tape symbol a_i in the string.

Locality of computation here means that one symbol in a string only depends on the symbols corresponding to the same tape cell i in the previous step and its two neighbors – three symbols total – because the head only moves in one step.

Proof of Theorem 2.4. Given a TM M with s states consider a $t \times (2t + 1)$ matrix T, a.k.a. the *computation table*, where row i is the configuration at time i. The starting configuration in Row 1 has the head in the middle cell. Note we don't need more than t cells to the right or left because the head moves only by one cell in one step. Next we claim that Row $i + 1$ can be computed from Row i by a Ckt with $c_s t$ gates. This follows by locality of computation, where note each entry in Row $i + 1$ can be computed by a Ckt of size c_s, by Theorem 2.3.

Stacking t such circuits we obtain a circuit of size $c_s t^2$ which computes the end configuration of them TM.

There remains to output the value of the function. Had we assumed that the TM writes the output in a specific cell, we could just read it off by a circuit of size c. Without the assumption, we can have a circuit $C : A \times \{0, 1, \ldots, s\} \to \{0,1\}$ which outputs 1 on (x, y) iff $y \neq 0$ and $x = 1$ (i.e., if x is a 1 that is under the TM's head). Taking an Or such circuits applied to every entry in the last row of T concludes the proof. **QED**

The simulation in Theorem 2.4. incurs a quadratic loss. However, a better simulation exists. In fact, this applies even to k-TMs.

Theorem 2.5. [7] Suppose an s-state k-TM computes $f : \{0,1\}^* \to \{0,1\}$ in time $t(n) \geq n$. Then $f \in \text{CktGates}(c_{s,k} t(n) \log t(n))$.

Exercise 2.9. Prove Theorem 2.5 assuming Theorem 2.2.

However, we won't need it, as we will work with another model which is closer to how computers operate. And for this model we shall give a different and simpler "simulation" by circuits in Chapter **??**.

In the other direction, TMs can simulate circuits if they have enough states. In general, allowing for the number of states to grow with the input length gives models with "hybrid uniformity."

Exercise 2.10. Suppose that $f : \{0,1\}^n \rightarrow \{0,1\}$ has circuits with s gates. Show that f can be computed by a TM with s^c states in time s^c.

2.4 Rapid-access machines (RAMs)

> "In some sense we are therefore merely making concrete intuitions that already pervade the literature. A related model has, indeed, been treated explicitly" [...] *[2]*

The main feature that's missing in all models considered so far is the ability to access a memory location in one time step. One can augment TMs with this capability by equipping them with an extra *addressing tape* and a special "jump" state which causes the head on a tape to move in one step to the address on the address tape. This model is simple enough to define, and could in a philosophical sense be the right model for how hardware can scale, since we charge for the time to write the address.

However, other models are closer to how computers seem to operate, at least over small inputs. We want to think of manipulating small integers and addresses as constant-time operations, as one typically has in mind when programming. There is a variety of such models, and some arbitrariness in their definition. Basically, we want to think of the memory as an array μ of s *cells* of w bits and allow for typical operations of them, including addressing arithmetic and *indirect addressing*: reading and writing the cell indexed by another cell.

One issue that arises is how much memory the machine should have and consequently how big w should be. There are two main options here. For "typical programming," we have a fixed memory size s and time bound t in mind, for example $S = n^3$ and $t = n^2$. A good choice then is to set $w := \lceil \log_2 s \rceil$ bits.

This however makes it harder to compare machines with different memory bounds. Also in some scenarios the memory size and the time bound are not fixed. This occurs for example when simulating another machine. To handle such scenarios we can start with a memory of $s = n + c$ cells,and a cell size of $w = \lceil \log_2 s \rceil$ bits, enough to access the input. We then equip machines with the operation MAlloc which increases the memory (i.e., s) by one, and always sets $w := \lceil \log_2 s \rceil$. Note the latter operation may increase w by 1. The MAlloc operation is akin to the TM's tape head wandering into unknown cells.

There are also two options for how the input is given to the machine. The difference doesn't matter if you don't care about w factors in time, but it matters if you do. For many problems, like sorting, etc. we think of the input and the output as coming in n cells of w bits. (Typically, $w = c \log n$, and one can simulate such cells with c cells with $\log n$ bits.) In this case, the RAM is computing a function $f : (\{0,1\}^w)^n \rightarrow (\{0,1\}^w)^m$ and the input to the RAM is given accordingly. This is what one often has in mind when writing programs that involve numbers. For other problems, it is natural to just give one bit of the input in each cell. That is, the RAM is computing $f : \{0,1\}^n \rightarrow \{0,1\}^m$ and bit i of the input is placed in the i input cells. We will not be concerned too much with small factors and so we pick the second choice for simplicity.

Definition 2.5. A *w-bit ℓ-line rapid-access machine* (RAM) *with s cells* consists of a memory array $\mu[1..s]$ of s cells of w bits, c *registers* $r_1, r_2, \ldots$ of w bits, and a program of ℓ lines.

Each line of the program contains an instruction among the following:

- Standard arithmetical and logical operations, such as $r_i = r_j + r_k$ etc.
- $r_i := \mu[r_j]$, called a Read operation, which reads the r_j memory cell and copies its content into r_i,
- $\mu[r_i] := r_j$, called a Write operation, which writes r_j into memory cells r_i, memory cell and copies its content into r_i,
- MAlloc which increases s by 1 and, if $s \geq 2^w$ also increases w by 1,
- Stop.

Read and write operations out of boundary indices have no effect.

On an input $x \in \{0,1\}^n$, the RAM starts the computation with $s := n+1$ cells of memory. The input is written in cells $1..n$, while $\mu[0]$ contains the length n of the input.

The output is written starting in cell 1.

We use RAMs as our main model for time inside P.

Definition 2.6. Time($t(n)$) is defined as TM-Time($t(n)$) but for RAMs instead of TMs.

Theorem 2.6. Time($t(n)$) $\subseteq$ TM-Time($t^c(n)$), for any $t(n) \geq n$.

Exercise 2.11. Prove it.

What is the relationship between circuits and RAMs? If a "description" of the circuit is given, then a RAM can simulate the circuit efficiently. The other way around is not clear. It appears that circuits need a quadratic blow-up to simulate RAMs.

Exercise 2.12. Give a function $f : \{0,1\}^* \to \{0,1\}$ in Time($c \log n$) but which requires circuits of size $\geq cn$.

There are universal RAMs that can simulate any other RAM with only a constant-factor overhead, unlike the logarithmic-factor overhead for tape machines.

Lemma 2.2. There is a RAM U that on input (P, t, x) where P is a RAM, t is an integer, and x is an input

-Stops in time ct,

-Outputs $P(x)$ if the latter stops within t steps on input x.

Proof. Throughout the computation, U will keep track of the memory size s_P and cell-size w_P of P. These are initialized as in the initial configuration of P on input x, whereas U starts with bigger values, since its input also contains P and t. Let h be the first cell where the input x starts. Memory location i of P is mapped to $i+h$ during the simulation. When P performs an operations among registers, U simulates that with its own registers, but discards the data that does not fit into w_P bits.

After each step, U decreases the counter. The counter can be stored in t cells, one bit per cell. The total number of operations to decrease such a counter from t to 0 is $\leq ct$. Alternatively, we can think of the counter as being stored in a single register at the beginning of the simulation. Then decreasing the counter is a single operation. **QED**

2.5 A challenge to the computability thesis

Today, there's a significant challenge to the computability thesis. This challenge comes from... I know what you are thinking: *Quantum computing, superposition, factoring.* Nope. *Randomness.*

The last century or so has seen an explosion of randomness affecting much of science, and computing has been a leader in the revolution. Today, randomness permeates computation. Except for basic "core" tasks, using randomness in algorithms is standard. So let us augment our model with randomness.

Definition 2.7. A *randomized* (or *probabilistic*) RAM, written RRAM, is a RAM equipped with the extra instruction

- $r_i :=$ Rand, which sets r_i to a uniform value, independent of all previous random choices.

For a RRAM and a sequence $R = R_1, R_2, \ldots$ we write $M(x, R)$ for the execution of M on input x where the j-th instruction $r_i :=$ Rand is replaced with $r_i := R_i$.

We refer to BPTime($t(n)$) with error $\epsilon(n)$ as the set of functions f that map bit strings x from a subset $X \subseteq \{0,1\}^*$ to a set $f(x)$ for which there exists a RRAM M such that, on any input $x \in X$ of length $\geq c_M$, M stops within $t(|x|)$ steps and $\mathbb{P}_R[M(x,R) \in f(x)] \geq 1-\epsilon(|x|)$.

If the error ϵ is not specified then it is assumed to be 1/3. Finally, we define

$$\text{BPP} := \bigcup_a \text{BPTime}(n^a).$$

Exercise 2.13. Does the following algorithm show that deciding if a given integer x is prime is in BPP? "Pick a uniform integer $y \leq x$. Check if y divides x."

Today, one usually takes BPP, not P, for "feasible computation." Thus it is natural to investigate how robust BPP is.

2.5.1 Robustness of BPP: Error reduction and tail bounds for the sum of random variables

The error in the definition of BPTime is somewhat arbitrary because it can be reduced. The way you do this is natural. For boolean functions, you repeat the algorithm many times, and take a majority vote. To analyze this you need probability bounds for the sum of random variables (corresponding to the outcomes of the algorithm).

Theorem 2.7. Let $X_1, X_2, \ldots, X_t$ be i.i.d. boolean random variables with $p := \mathbb{P}[X_i = 1]$. Then for $q \geq p$ we have $\mathbb{P}[\sum_{i=1}^t X_i \geq qt] \leq 2^{-D(q|p)t}$, where

$$D(q|p) := q \log\left(\frac{q}{p}\right) + (1-q)\log\left(\frac{1-q}{1-p}\right)$$

is the *divergence.*

Now one can get a variety of bounds by bounding divergence for different settings of parameter. We state one such bound which we use shortly.

Fact 2.1. $D(1/2|1/2-\epsilon) \geq \epsilon^2$.

Exercise 2.14. Plot both sides of Fact 2.1 as a function of ϵ. (Hint: I used https://www.desmos.com/calcul

Using this bound we can prove the error reduction stated earlier.

Theorem 2.8. [Error reduction for BPP] For boolean functions, the definition of BPP (Definition 2.7) remains the same if 1/3 is replaced with $1/2 - 1/n^a$ or $1/2^{n^a}$, for any constant a.

Proof. Suppose that f is in BPP with error $p := 1/2 - 1/n^a$ and let M be the corresponding RRAM. On an input x, let us run $t := n^{2a} \cdot n^b$ times M, each time with fresh randomness, and take a majority vote. The new algorithm is thus

$$\text{Maj}(M(x, R_1), M(x, R_2), \ldots, M(x, R_t)).$$

This new algorithm makes a mistake iff at least $t/2$ runs of M make a mistake. To analyze this error probability we invoke Theorem 2.7 where $X_i := 1$ iff run i of the algorithm makes a mistake, i.e., $M(x, R_i) \neq f(x)$, and $\epsilon := 1/n^a$. By Fact 2.1 we obtain an error bound of

$$2^{-D(1/2|1/2-\epsilon)t} \leq 2^{-\epsilon^2 t} \leq 2^{-n^b},$$

as desired. The new algorithm still runs in power time, for fixed a and b. **QED**

Exercise 2.15. Consider an alternative definition of BPTime, denoted BPTime', which is analogous to BPTime except that the requirement that the machine always stops within $t(|x|)$ steps is relaxed to "the *expected* running time of the machine is $t(|x|)$."

Show that defining BPP with respect to BPTime or BPTime' is equivalent.

Exercise 2.16. Consider *biased* RRAMs which are like RRAMs except that the operation Rand returns one bit which, independently from all previous calls to Rand, is 1 with probability 1/3 and 0 with probability 2/3. Show that BPP does not change if we use biased RRAMs.

2.5.2 Does randomness buy time?

We can always brute-force the random choices in exponential time. If a randomized machine uses r random bits then we simulate it deterministically by running it on each of the 2^r choices for the bits. A RRAM machine running in time $t \geq n$ has registers of $\leq c \log t$ bits. Each Rand operation gives a uniform register, so the machine uses $\leq ct \log t$ bits. This gives the following inclusions.

Theorem 2.9. $\text{Time}(t) \subseteq \text{BPTime}(t) \subseteq \text{Time}(c^{t \log t})$, for any function $t = t(n)$. In particular, $\text{P} \subseteq \text{BPP} \subseteq \text{EXP}$.

Proof. The first inclusion is by definition. The idea for the second was discussed before, but we need to address the detail that we don't know what t is. One way to carry through the simulation is as follows. The deterministic machine initializes a counter r to 0. For each value of r it enumerates over the 2^r choices R for the random bits, and runs the RRAM on each choice of R, keeping track of its output on each choice, and outputting the majority vote. If it ever runs out of random bits, it increases r by 1 and restarts the process.

To analyze the running time, recall we only need $r \leq ct \log t$. So the simulation runs the RRAM at most $ct \log t \cdot 2^{ct \log t} \leq 2^{ct \log t}$ times, and each run takes time ct, where this last bound takes into account the overhead for incrementing the choice of r, and redirecting the calls to Rand to R. **QED**

Now, two surprises. First, BPP $\subseteq$ EXP is the fastest deterministic simulation we can *prove* for RAMs, or even 2-TMs. On the other hand, and that is perhaps the bigger surprise, it appears commonly *believed* that in fact P = BPP! Moreover, it appears commonly believed that the overhead to simulate randomized computation deterministically is very small. Here the mismatch between our ability and common belief is abysmal.

However, we can do better for TMs. A *randomized* TMs has two transition functions σ_0 and σ_1, where each is as in Definition 2.1. At each step, the TM uses σ_0 or σ_1 with probability $1/2$ each, corresponding to tossing a coin. We can define TM-BPTime as BPTime but with randomized TMs instead of RRAMS.

Theorem 2.10. [10] $\text{TM-BPTime}(t) \subseteq \text{Time}(2^{\sqrt{t} \log^c t})$, for any $t = t(n) \geq n$.

2.5.3 Polynomial identity testing

We now discuss an important problem which is in BPP but not known to be in P. In fact, in a sense to be made precise later, this is *the* problem in BPP which is not known to be in P. To present this problem we introduce two key concepts which will be used many times: *finite fields*, and *arithmetic circuits*.

Finite fields A finite field $\mathbb{F}$ is a finite set with elements 0 and 1 that is equipped with operations $+$ and $\cdot$ that behave "in the same way" as the corresponding operations over the reals or the rationals. One example are the integers modulo a prime p. For $p = 2$ this gives

the field with two elements where $+$ is Xor and $\cdot$ is And. For larger p you add and multiply as over the integers but then you take the result modulo p.

The following summarizes key facts about finite fields. The case of prime fields suffices for the main points of this section, but stating things for general finite fields actually simplifies the exposition overall (since otherwise we need to add qualifiers to the size of the field).

Fact 2.2. [Finite fields] A unique finite field of size q exists iff $q = p^t$ where p is a prime and $t \in \mathbb{N}$. This field is denoted $\mathbb{F}_q$.

Elements in the field can be identified with $\{0, 1, \ldots, p-1\}^t$.

[8] Given q, one can compute a *representation* of a finite field of size q in time $(tp)^c$. This representation can be identified with p plus an element of $\{0, 1, \ldots, p-1\}^t$.

Given a representation r and field elements x, y computing $x+y$ and $x \cdot y$ is in $\text{Time}(n \log^c n)$.

Fields of size 2^t are of natural interest in computer science. It is often desirable to have very explicit representations for such and other fields. Such representations are known and are given by simple formulas, and are in particular computable in linear time.

Arithmetic circuits We now move to defining arithmetic circuits, which are a natural generalization of the circuits we encountered in section §2.3.

Definition 2.8. An *arithmetic circuit* over a field $\mathbb{F}$ is a circuit where the gates compute the operations $+$ and $\cdot$ over $\mathbb{F}$, or are constants, or are input variables. Such a circuit computes a polynomial mapping $\mathbb{F}^n \to \mathbb{F}$.

The PIT (polynomial identity testing) problem over $\mathbb{F}$: Given an arithmetic circuit C over $\mathbb{F}$ with n input variables, does $C(x) = 0$ for every $x \in \mathbb{F}^n$?

The PIT problem *over large fields* is in BPP but it is not known to be in P. The requirement that the field be large is critical, see Problem **??**.

Theorem 2.11. [PIT over large fields in BPP] Given an arithmetic circuit C and a field of size $\geq c2^{|C|}$ we can solve PIT in BPP.

To prove this theorem we need the following fundamental fact.

Lemma 2.3. Let p be a polynomial over a finite field $\mathbb{F}$ with n variables and degree $\leq d$. Let S be a subset of $\mathbb{F}$, and suppose $d < |S|$. The following are equivalent:

1. p is the zero polynomial.
2. $p(x) = 0$ for every $x \in \mathbb{F}^n$.
3. $\mathbb{P}_{x_1, x_2, \ldots, x_n \in S}[p(x) = 0] > d/|S|$.

Proof and discussion of Lemma 2.3. The implications 1. $\Rightarrow$ 2. $\Rightarrow$ 3. are trivial, but note that for the latter we need $d < |S|$. The implication 3. $\Rightarrow$ 1. requires more work. It is a multi-variate generalization of the fundamental fact that a non-zero univariate polynomial

of degree d has at most d roots. The fundamental fact can be recovered setting $n := 1$ and $S := \mathbb{F}$.

To prove the multi-variate generalization we proceed by induction on n. The base case $n = 1$ is the fundamental fact (which we will not prove). For larger n write

$$p(x_1, x_2, \ldots, x_n) = \sum_{i=0}^{d} x_1^i p_i(x_2, x_3, \ldots, x_n).$$

If p is not the zero polynomial then there is at least one i such that p_i is not the zero polynomial. Let j be the largest such i. Note that p_j has degree at most $d-j$. By induction hypothesis

$$\mathbb{P}_{x_2,\ldots,x_n \in S}[p_j(x) = 0] \leq (d-j)/|S|.$$

For every choice of $x_2, x_3, \ldots, x_n$ s.t. $p_j(x) \neq 0$, the polynomial p is a non-zero polynomial $q_{x_2,x_3,\ldots,x_n}(x_1)$ only in the variable x_1. Moreover, its degree is at most j by our choice of j. Hence by the $n = 1$ case the probability that q is 0 over the choice of x_1 is $\leq j$.

Overall,

$$\mathbb{P}_{x_1,x_2,\ldots,x_n \in S}[p(x) = 0] \leq (d-j)/|S| + j/|S| = d/|S|.$$

QED

Proof of Theorem 2.11. A circuit C contains at most $|C|$ multiplication gates. Each multiplication gate at most squares the degree of its inputs. Hence C computes a polynomial of degree $\leq 2^{|C|}$. Let S be a subset of size $c \cdot 2^{|C|}$ of $\mathbb{F}$. Assign uniform values from S independently to each variables, and evaluate the circuit. If C evaluates to 0 everywhere then obviously the output will be 0. Otherwise, by Lemma 2.3, the probability we get a 0 is $\leq 2^{|C|}/c2^{|C|} \leq 1/3$. **QED**

Exercise 2.17. Show that the PIT problem over the *integers* is in BPP. (Hint: Use that the number of primes in $\{1, 2, \ldots, t\}$ is $\geq t/\log^c t$, for every $t \geq c$, and that checking if a number is prime is in P.)

The $t/\log^c t$ bound is a weak form of the *prime number theorem.* The weak form typically suffices in computer science, and has a simple and cute encoding proof.

2.5.4 Simulating BPP by circuits

While we don't know if P = BPP, we can prove that, like P, BPP has power-size circuits.

Theorem 2.12. [1] BPP$\subseteq \bigcup_a$ CktGates(n^a).

Proof. Let $f : X \subseteq \{0,1\}^* \to \{0,1\}$ be in BPP. By Theorem 2.8 we can assume that the error is $\epsilon < 2^{-n}$, and let M be the corresponding RRAM. Note

$$\mathbb{P}_R[\exists x \in \{0,1\}^n : M(x,R) \neq f(x)] \leq \sum_{x \in \{0,1\}^n} \mathbb{P}_R[M(x,R) \neq f(x)] \leq 2^n \cdot \epsilon < 1,$$

where the first inequality is a union bound.

Therefore, there is a fixed choice for R that gives the correct answer for every input $x \in \{0,1\}^n$. This choice can be hardwired in the circuit, and the rest of the computation can be written as a circuit by Theorem 2.4. **QED**

Exercise 2.18. In this exercise you will practice the powerful technique of combining tail bounds with union bounds, which was used in the proof of Theorem 2.12.

An *error-correcting code* is a subset $C \subseteq \{0,1\}^n$ s.t. for any distinct $x, y \in C$, x and y differ in at least $n/3$ coordinates. Prove the existence of codes of size $|C| \geq 2^{\epsilon n}$ for a constant ϵ.

2.5.5 Questions raised by randomness

The introduction of randomness in our model raises several fascinating questions. First, does "perfect" randomness exists "in nature?" Second, do we need "perfect" randomness for computation? A large body of research has been devoted to greatly generalize Problem 2.16 to show that, in fact, even imperfect sources of randomness suffices for computation. Third, do we need randomness at all? Is P = BPP?

One of the exciting developments of complexity theory has been the connection between the latter question and the "grand challenge" from the next chapter. At a high level, it has been shown that explicit functions that are hard for circuits can be used to *de-randomize* computation. In a nutshell, the idea is that if a function is hard to compute then its output is "random," so can be used instead of true randomness. The harder the function the less randomness we need. At one extreme, we have the following striking connection:

Theorem 2.13. [5] Suppose for some $a > 0$ there is a function in Time(2^{an}) which on inputs of length n cannot be computed by circuits with $2^{n/a}$ gates, for all large enough n. Then P = BPP.

In other words, either randomness is useless for power-time computation, or else circuits can speed up exponential-time uniform computation!

2.6 Inclusion extend "upwards," separations downwards

To develop intuition about complexity, we now discuss a general technique known as *padding*. In short, the technique shows that if you can trade resource X for Y, then you can also trade *a lot of* X for *a lot of* Y. For a metaphor, if you have a magical device that can turn one pound of sill into gold, you can also use it to turn *two* pounds of sill into gold. The contrapositive is that if you *can't* trade a lot of X for a lot of Y, then you also can't trade a little of X for a little of Y.

We give a first example using the classes that we have encountered so far.

Example 2.3. Suppose that BPTime$(cn) \subseteq$ Time(n^2). Then BPTime$(n^2) \subseteq$ Time(cn^4).

Proof. Let $f : \{0,1\}^* \to \{0,1\}$ be a function in BPTime(n^2). Consider the function f' that on input x of length n equals f computed on the first $\sqrt{n}$ bits of x. Thus, inputs to f' are padded with $n - \sqrt{n}$ useless symbols.

Note that $f' \in$ BPTime(cn), since in linear time we can erase the last $n - \sqrt{n}$ symbols and then just run the algorithm for f which takes time quadratic in $\sqrt{n}$ which is linear in n. (If computing square roots is not an available instruction, one can show that computing $\sqrt{n}$ can be done in linear time, for example using binary search.)

By assumption, $f' \in$ Time(n^2).

To compute f in time cn^4 we can then do the following. Given input x of length n, pad x to an input of length n^2 in time cn^2. Then run the algorithm for f'. This will take time $c(n^2)^2 \leq cn^4$. **QED**

2.7 Problems

Problem 2.1. [Indexing] Describe a TM that on input $(x, i) \in \{0,1\}^n \times \{1, 2, \ldots, n\}$ outputs bit i of x in time $cn \log n$.

Problem 2.2. Show that Palindromes can be solved in time $n \log^c n$ on a randomized TM. (Yes, only one tape.)

Hint: View the input as coefficients of polynomials.

Problem 2.3. Give a function $f : X \subseteq \{0,1\}^* \to \{0,1\}$ that is in BPTime(c) but not in Time($n/100$).

References

[1] Leonard Adleman. Two theorems on random polynomial time. In *19th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 75–83. 1978.

[2] Dana Angluin and Leslie G. Valiant. Fast probabilistic algorithms for hamiltonian circuits and matchings. *J. Comput. Syst. Sci.*, 18(2):155–193, 1979.

[3] Oded Goldreich. *Computational Complexity: A Conceptual Perspective.* Cambridge University Press, 2008.

[4] Fred Hennie and Richard Stearns. Two-tape simulation of multitape turing machines. *J. of the ACM*, 13:533–546, October 1966.

[5] Russell Impagliazzo and Avi Wigderson. *P = BPP* if *E* requires exponential circuits: Derandomizing the XOR lemma. In *29th ACM Symp. on the Theory of Computing (STOC)*, pages 220–229. ACM, 1997.

[6] O. B. Lupanov. A method of circuit synthesis. *Izv. VUZ Radiofiz.*, 1:120–140, 1958.

[7] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. of the ACM*, 26(2):361–381, 1979.

[8] Victor Shoup. New algorithms for finding irreducible polynomials over finite fields. *Mathematics of Computation*, 54(189):435–447, 1990.

[9] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, s2-42(1):230–265, 1937.

[10] Emanuele Viola. Pseudorandom bits and lower bounds for randomized turing machines. *Theory of Computing*, 18(10):1–12, 2022.