Modeling randomized computation

Boaz Barak

**Disclaimer:** The MS-Word version of this text has significant formatting issues compared to the PDF and HTML versions.

# Modeling randomized computation

### 

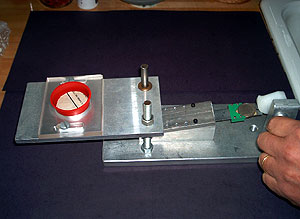
* Formal definition of probabilistic polynomial time: the class .
* Proof that that every function in can be computed by -sized NAND-CIRC programs/circuits.
* Relations between and .
* Pseudorandom generators

*“Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.”* John von Neumann, 1951.

So far we have described randomized algorithms in an informal way, assuming that an operation such as “pick a string ” can be done efficiently. We have neglected to address two questions:

1. How do we actually efficiently obtain random strings in the physical world?
2. What is the mathematical model for randomized computations, and is it more powerful than deterministic computation?

The first question is of both practical and theoretical importance, but for now let’s just say that there are various physical sources of “random” or “unpredictable” data. A user’s mouse movements and typing pattern, (non solid state) hard drive and network latency, thermal noise, and radioactive decay have all been used as sources for randomness (see discussion in modelrandbibnotes). For example, many Intel chips come with a random number generator [built in](http://spectrum.ieee.org/computing/hardware/behind-intels-new-randomnumber-generator). One can even build mechanical coin tossing machines (see coinfig).



A mechanical coin tosser built for Percy Diaconis by Harvard technicians Steve Sansone and Rick Haggerty

In this chapter we focus on the second question: formally modeling probabilistic computation and studying its power. We will show that:

1. We can define the class that captures all Boolean functions that can be computed in polynomial time by a randomized algorithm. Crucially is still very much a *worst case* class of computation: the probability is only over the choice of the random coins of the algorithm, as opposed to the choice of the input.
2. We can *amplify* the success probability of randomized algorithms, and as a result the class would be identical if we changed the required success probability to any number that lies strictly between and (and in fact any number in the range to for any polynomial ).
3. Though, as is the case for and , there is much we do not know about the class , we can establish some relations between and the other complexity classes we saw before. In particular we will show that and .
4. While the relation between and is not known, we can show that if then .
5. We also show that the concept of completeness applies equally well if we use randomized algorithms as our model of “efficient computation”. That is, if a single complete problem has a randomized polynomial-time algorithm, then all of can be computed in polynomial-time by randomized algorithms.
6. Finally we will discuss the question of whether and show some of the intriguing evidence that the answer might actually be *“Yes”* using the concept of *pseudorandom generators*.

## Modeling randomized computation

Modeling randomized computation is actually quite easy. We can add the following operations to any programming language such as NAND-TM, NAND-RAM, NAND-CIRC etc..:

foo = RAND()

where foo is a variable. The result of applying this operation is that foo is assigned a random bit in . (Every time the RAND operation is invoked it returns a fresh independent random bit.) We call the programming languages that are augmented with this extra operation RNAND-TM, RNAND-RAM, and RNAND-CIRC respectively.

Similarly, we can easily define randomized Turing machines as Turing machines in which the transition function gets as an extra input (in addition to the current state and symbol read from the tape) a bit that in each step is chosen at random . Of course the function can ignore this bit (and have the same output regardless of whether or ) and hence randomized Turing machines generalize deterministic Turing machines.

We can use the RAND() operation to define the notion of a function being computed by a randomized time algorithm for every nice time bound , as well as the notion of a finite function being computed by a size randomized NAND-CIRC program (or, equivalently, a randomized circuit with gates that correspond to either the NAND or coin-tossing operations). However, for simplicity we will not define randomized computation in full generality, but simply focus on the class of functions that are computable by randomized algorithms *running in polynomial time*, which by historical convention is known as :

Let . We say that if there exist constants and an RNAND-TM program such that for every , on input , the program halts within at most steps and

where this probability is taken over the result of the RAND operations of .

Note that the probability in BPPdefinitioneq is taken only over the random choices in the execution of and *not* over the choice of the input . In particular, as discussed in randomworstcaseidea, is still a *worst case* complexity class, in the sense that if is in then there is a polynomial-time randomized algorithm that computes with probability at least *on every possible* (and not just random) input.

The same polynomial-overhead simulation of NAND-RAM programs by NAND-TM programs we saw in polyRAMTM-thm extends to *randomized* programs as well. Hence the class is the same regardless of whether it is defined via RNAND-TM or RNAND-RAM programs. Similarly, we could have just as well defined using randomized Turing machines.

Because of these equivalences, below we will use the name *“polynomial time randomized algorithm”* to denote a computation that can be modeled by a polynomial-time RNAND-TM program, RNAND-RAM program, or a randomized Turing machine (or any programming language that includes a coin tossing operation). Since all these models are equivalent up to polynomial factors, you can use your favorite model to capture polynomial-time randomized algorithms without any loss in generality.

Modern programming languages often involve not just the ability to toss a random coin in but also to choose an element at random from a set . Show that you can emulate this primitive using coin tossing. Specifically, show that there is randomized algorithm that on input a set of strings of length , runs in time and outputs either an element or “fail” such that

1. Let be the probability that outputs “fail”, then (a number small enough that it can be ignored).
2. For every , the probability that outputs is exactly (and so the output is uniform over if we ignore the tiny probability of failure)

If the size of is a power of two, that is for some , then we can choose a random element in by tossing coins to obtain a string and then output the -th element of where is the number whose binary representation is .

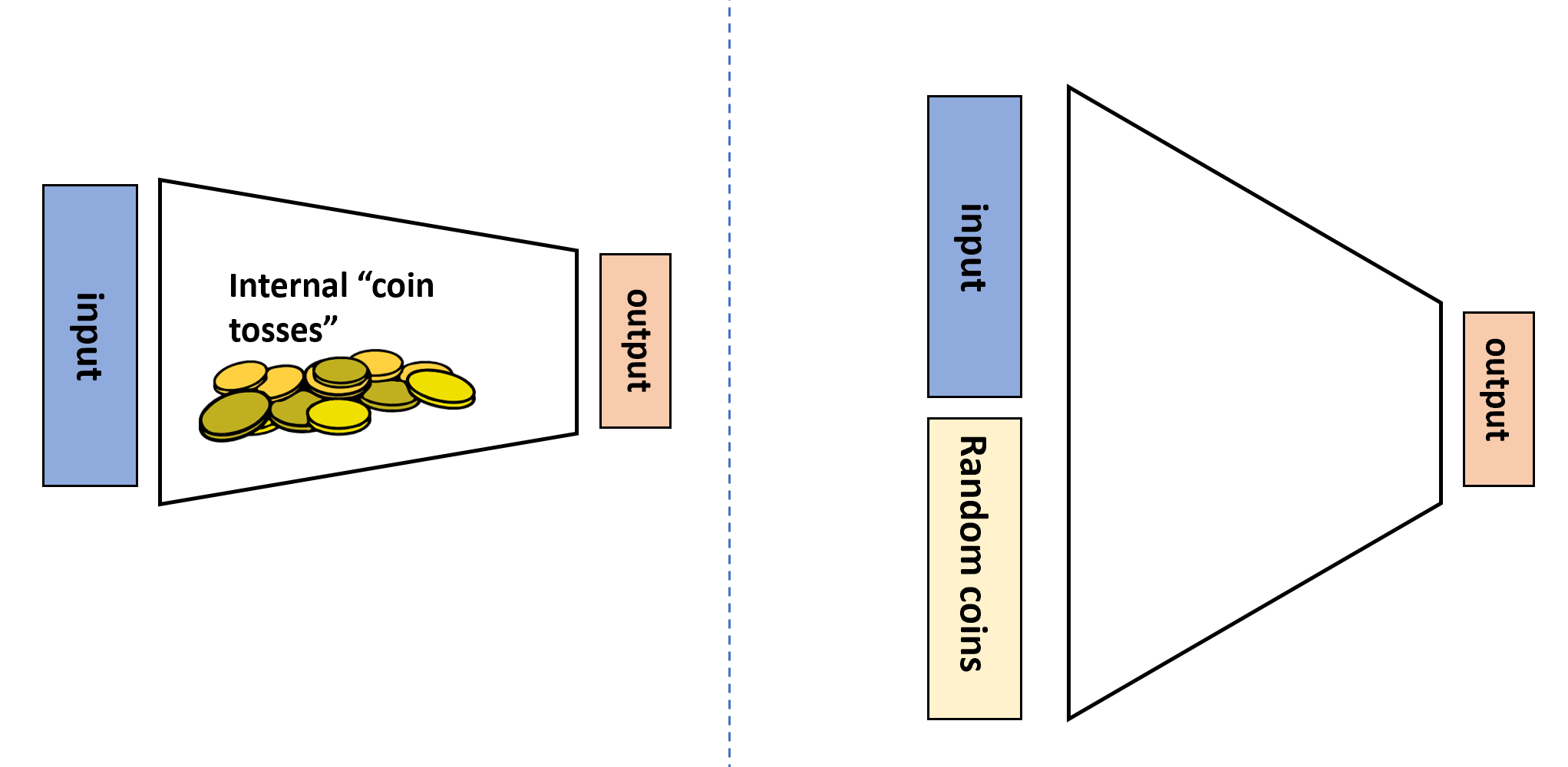
If is not a power of two, then our first attempt will be to let and do the same, but then output the -th element of if and output “fail” otherwise. Conditioned on not outputting “fail”, this element is distributed uniformly in . However, in the worst case, can be almost and so the probability of fail might be close to half. To reduce the failure probability, we can repeat the experiment above times. Specifically, we will use the following algorithm

INPUT: Set $S = \{ x\_0,\ldots, x\_{m-1} \}$ with $x\_i\in \{0,1\}^n$ -for all $i\in [m]$.  
OUTPUT: Either $x\in S$ or "fail"  
  
Let $\ell \leftarrow \lceil \log m \rceil$  
For{$j = 0,1,\ldots,n-1$}  
 Pick $w \sim \{0,1\}^\ell$  
 Let $i\in [2^\ell]$ be number whose binary representation is $w$.  
 If{$i<m$}  
 return $x\_i$  
 Endif  
Endfor  
Return "fail"

Conditioned on not failing, the output of samplefromsetalg is uniformly distributed in . However, since , the probability of failure in each iteration is less than and so the probability of failure in all of them is at most .

### An alternative view: random coins as an “extra input”

While we presented randomized computation as adding an extra “coin tossing” operation to our programs, we can also model this as being given an additional extra input. That is, we can think of a randomized algorithm as a *deterministic* algorithm that takes *two inputs* and where the second input is chosen at random from for some (see randomalgsviewsfig). The equivalence to the BPPdef is shown in the following theorem:



The two equivalent views of randomized algorithms. We can think of such an algorithm as having access to an internal RAND() operation that outputs a random independent value in whenever it is invoked, or we can think of it as a deterministic algorithm that in addition to the standard input obtains an additional auxiliary input that is chosen uniformly at random.

### 

Let . Then if and only if there exists and such that is in and for every ,

### 

The idea behind the proof is that, as illustrated in randomalgsviewsfig, we can simply replace sampling a random coin with reading a bit from the extra “random input” and vice versa. To prove this rigorously we need to work through some slightly cumbersome formal notation. This might be one of those proofs that is easier to work out on your own than to read.

We start by showing the “only if” direction. Let and let be an RNAND-TM program that computes as per BPPdef, and let be such that on every input of length , the program halts within at most steps. We will construct a polynomial-time algorithm that such that for every , if we set , then

where the probability in the righthand side is taken over the RAND() operations in . In particular this means that if we define then the function satisfies the conditions of eqBPPauxiliary.

The algorithm will be very simple: it simulates the program , maintaining a counter initialized to . Every time that makes a RAND() operation, the program will supply the result from and increment by one. We will never “run out” of bits, since the running time of is at most and hence it can make at most this number of RNAND() calls. The output of for a random will be distributed identically to the output of .

For the other direction, given a function satisfying the condition eqBPPauxiliary and a NAND-TM that computes in polynomial time, we can construct an RNAND-TM program that computes in polynomial time. On input , the program will simply use the RNAND() instruction times to fill an array R[] , , R[] and then execute the original program on input where is the -th element of the array R. Once again, it is clear that if runs in polynomial time then so will , and for every input and , the output of on input and where the coin tosses outcome is is equal to .

The characterization of randextrainput is reminiscent of the characterization of in NP-def, with the randomness in the case of playing the role of the solution in the case of . However, there are important differences between the two:

* The definition of is “one sided”: if *there exists* a solution such that and if *for every* string of the appropriate length, . In contrast, the characterization of is symmetric with respect to the cases and .
* The relation between and is not immediately clear. It is not known whether , , or these two classes are incomparable. It is however known (with a non-trivial proof) that if then (see BPPvsNP).
* Most importantly, the definition of is “ineffective,” since it does not yield a way of actually finding whether there exists a solution among the exponentially many possibilities. By contrast, the definition of gives us a way to compute the function in practice by simply choosing the second input at random.

**“Random tapes”.** randextrainput motivates sometimes considering the randomness of an RNAND-TM (or RNAND-RAM) program as an extra input. As such, if is a randomized algorithm that on inputs of length makes at most coin tosses, we will often use the notation (where and ) to refer to the result of executing when the coin tosses of correspond to the coordinates of . This second, or “auxiliary,” input is sometimes referred to as a “random tape.” This terminology originates from the model of randomized Turing machines.

### Success amplification of two-sided error algorithms

The number might seem arbitrary, but as we’ve seen in randomizedalgchap it can be amplified to our liking:

Let be a Boolean function such that there is a polynomial and a polynomial-time randomized algorithm satisfying that for every ,

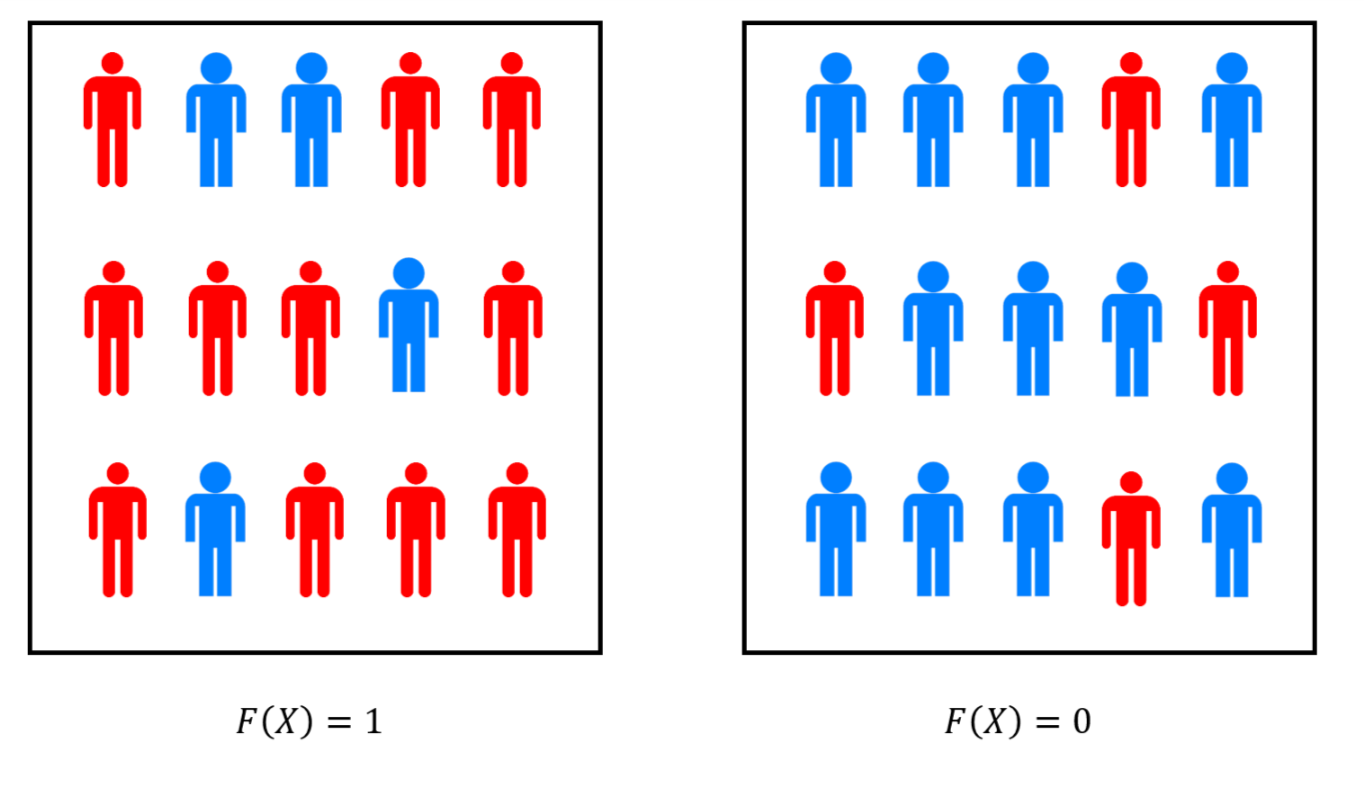
Then for every polynomial there is a polynomial-time randomized algorithm satisfying for every ,

We can *amplify* the success of randomized algorithms to a value that is arbitrarily close to .

### 

The proof is the same as we’ve seen before in the case of maximum cut and other examples. We use the Chernoff bound to argue that if computes with probability at least and we run it times, each time using fresh and independent random coins, then the probability that the majority of the answers will not be correct will be less than . Amplification can be thought of as a “polling” of the choices for randomness for the algorithm (see amplificationfig).

Let be an algorithm satisfying eqbppampassumption. Set and where are the polynomials in the theorem statement. We can run on input for times, using fresh randomness in each execution, and compute the outputs . We output the value that appeared the largest number of times. Let be the random variable that is equal to if and equal to otherwise. The random variables are i.i.d. and satisfy , and hence by linearity of expectation . For the plurality value to be *incorrect*, it must hold that , which means that is at least far from its expectation. Hence by the Chernoff bound (chernoffthm), the probability that the plurality value is not correct is at most , which is smaller than for our choice of .



If then there is randomized polynomial-time algorithm with the following property: In the case two thirds of the “population” of random choices satisfy and in the case two thirds of the population satisfy . We can think of amplification as a form of “polling” of the choices of randomness. By the Chernoff bound, if we poll a sample of random choices , then with probability at least , the fraction of ’s in the sample satisfying will give us an estimate of the fraction of the population within an margin of error. This is the same calculation used by pollsters to determine the needed sample size in their polls.

## and completeness

Since “noisy processes” abound in nature, randomized algorithms can be realized physically, and so it is reasonable to propose rather than as our mathematical model for “feasible” or “tractable” computation. One might wonder if this makes all the previous chapters irrelevant, and in particular if the theory of completeness still applies to probabilistic algorithms. Fortunately, the answer is *Yes*:

### 

Suppose that is -hard and . Then .

Before seeing the proof, note that NPCandBPP implies that if there was a randomized polynomial time algorithm for any -complete problem such as , etc., then there would be such an algorithm for *every* problem in . Thus, regardless of whether our model of computation is deterministic or randomized algorithms, complete problems retain their status as the “hardest problems in .”

### 

The idea is to simply run the reduction as usual, and plug it into the randomized algorithm instead of a deterministic one. It would be an excellent exercise, and a way to reinforce the definitions of -hardness and randomized algorithms, for you to work out the proof for yourself. However for the sake of completeness, we include this proof below.

### 

Suppose that is -hard and . We will now show that this implies that . Let . By the definition of -hardness, it follows that , or that in other words there exists a polynomial-time computable function such that for every . Now if is in then there is a polynomial-time RNAND-TM program such that

for *every* (where the probability is taken over the random coin tosses of ). Hence we can get a polynomial-time RNAND-TM program to compute by setting . By FinBPPeq and since this implies that , which proves that .

Most of the results we’ve seen about hardness, including the search to decision reduction of search-dec-thm, the decision to optimization reduction of optimizationnp, and the quantifier elimination result of PH-collapse-thm, all carry over in the same way if we replace with as our model of efficient computation. Thus if then we get essentially all of the strange and wonderful consequences of . Unsurprisingly, we cannot rule out this possibility. In fact, unlike , which is ruled out by the time hierarchy theorem, we don’t even know how to rule out the possibility that ! Thus a priori it’s possible (though seems highly unlikely) that randomness is a magical tool that allows us to speed up arbitrary exponential time computation.[[1]](#footnote-46) Nevertheless, as we discuss below, it is believed that randomization’s power is much weaker and lies in much more “pedestrian” territory.

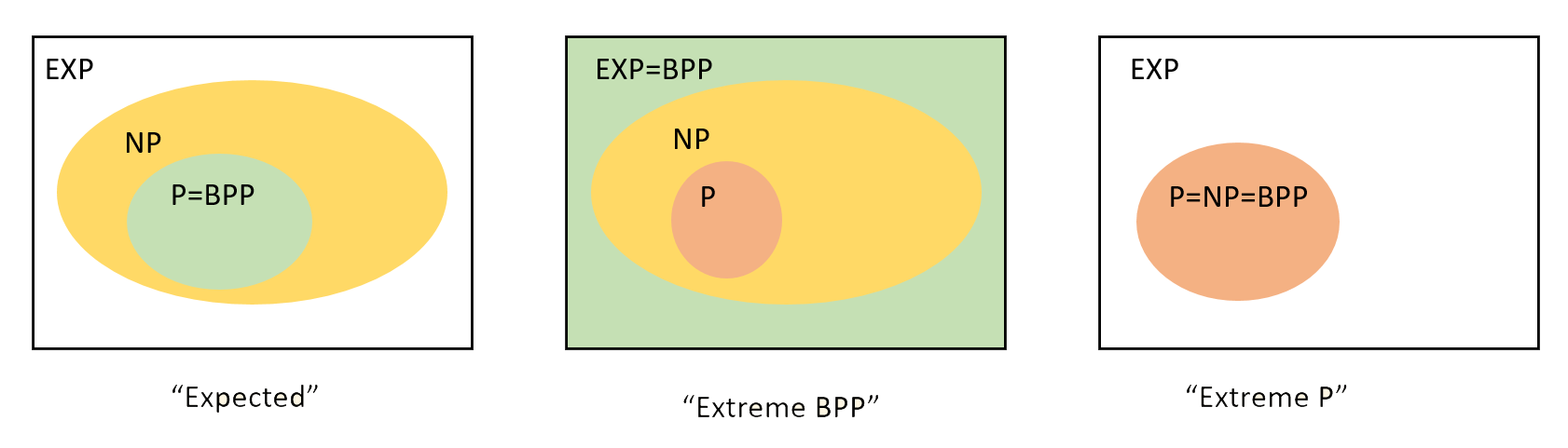
## The power of randomization

A major question is whether randomization can add power to computation. Mathematically, we can phrase this as the following question: does ? Given what we’ve seen so far about the relations of other complexity classes such as and , or and , one might guess that:

1. We do not know the answer to this question.
2. But we suspect that is different than .

One would be correct about the former, but wrong about the latter. As we will see, we do in fact have reasons to believe that . This can be thought of as supporting the *extended Church Turing hypothesis* that deterministic polynomial-time Turing machines capture what can be feasibly computed in the physical world.

We now survey some of the relations that are known between and other complexity classes we have encountered. (See also BPPscenariosfig.)



Some possibilities for the relations between and other complexity classes. Most researchers believe that and that these classes are *not* powerful enough to solve -complete problems, let alone all problems in . However, we have not even been able yet to rule out the possibility that randomness is a “silver bullet” that allows exponential speedup on all problems, and hence . As we’ve already seen, we also can’t rule out that . Interestingly, in the latter case, .

### Solving in exponential time

It is not hard to see that if is in then it can be computed in *exponential* time.

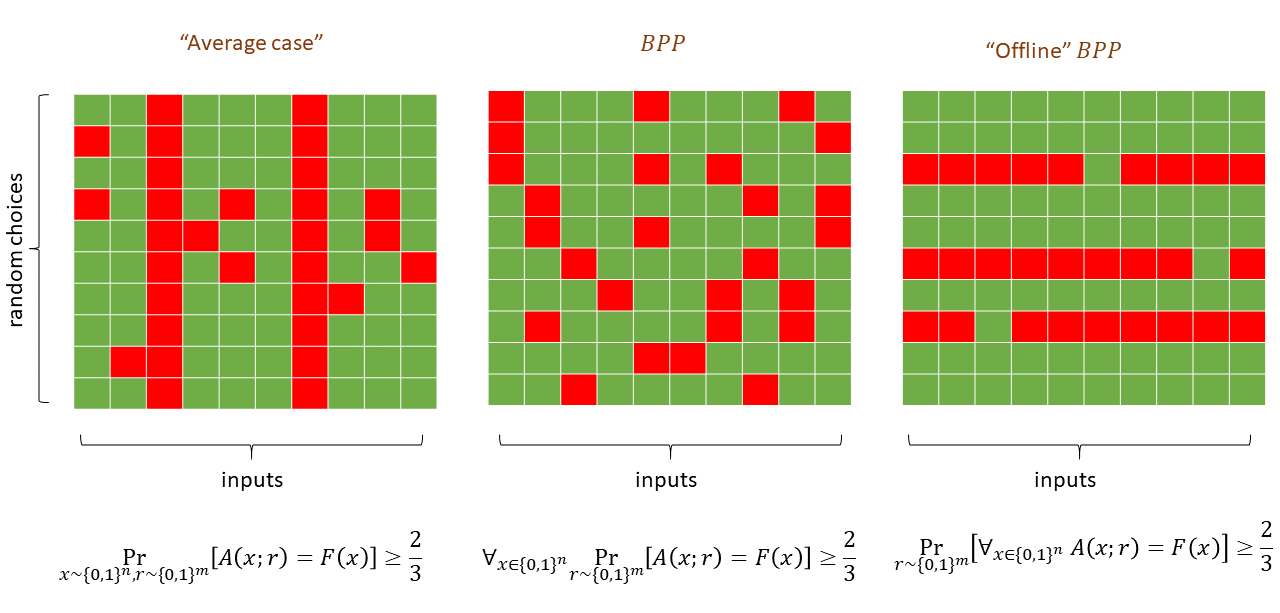
### 

### 

The proof of BPPEXP readily follows by enumerating over all the (exponentially many) choices for the random coins. We omit the formal proof, as doing it by yourself is an excellent way to get comfortable with BPPdef.

### Simulating randomized algorithms by circuits

We have seen in non-uniform-thm that if is in , then there is a polynomial such that for every , the restriction of to inputs is in . (In other words, that .) A priori it is not at all clear that the same holds for a function in , but this does turn out to be the case.



The possible guarantees for a randomized algorithm computing some function . In the tables above, the columns correspond to different inputs and the rows to different choices of the random tape. A cell at position is colored green if (i.e., the algorithm outputs the correct answer) and red otherwise. The standard guarantee corresponds to the middle figure, where for every input , at least two thirds of the choices for a random tape will result in computing the correct value. That is, every column is colored green in at least two thirds of its coordinates. In the left figure we have an “average case” guarantee where the algorithm is only guaranteed to output the correct answer with probability two thirds over a *random* input (i.e., at most one third of the total entries of the table are colored red, but there could be an all red column). The right figure corresponds to the “offline ” case, with probability at least two thirds over the random choice , will be good for *every* input. That is, at least two thirds of the rows are all green. rnandthm () is proven by amplifying the success of a algorithm until we have the “offline ” guarantee, and then hardwiring the choice of the randomness to obtain a nonuniform deterministic algorithm.

### 

.

That is, for every , there exist some such that for every , where is the restriction of to inputs in .

### 

The idea behind the proof is that we can first amplify by repetition the probability of success from to . This will allow us to show that for every there exists a *single fixed choice* of “favorable coins” which is a string of length polynomial in such that if is used for the randomness then we output the right answer on *all* of the possible inputs. We can then use the standard “unravelling the loop” technique to transform an RNAND-TM program to an RNAND-CIRC program, and “hardwire” the favorable choice of random coins to transform the RNAND-CIRC program into a plain old deterministic NAND-CIRC program.

Suppose that . Let be a polynomial-time RNAND-TM program that computes as per BPPdef. Using amplificationthm, we can *amplify* the success probability of to obtain an RNAND-TM program that is at most a factor of slower (and hence still polynomial time) such that for every

where is the number of coin tosses that uses on inputs of length . We use the notation to denote the execution of on input and when the result of the coin tosses corresponds to the string .

For every , define the “bad” event to hold if , where the sample space for this event consists of the coins of . Then by ampeq, for every . Since there are many such ’s, by the union bound we see that the probability that the *union* of the events is at most . This means that if we choose , then with probability at least it will be the case that for *every* , . (Indeed, otherwise the event would hold for some .) In particular, because of the mere fact that the the probability of is smaller than , this means that *there exists* a particular such that

for every .

Now let us use the standard “unravelling the loop” the technique and transform into a NAND-CIRC program of polynomial in size, such that for every and . Then by “hardwiring” the values in place of the last inputs of , we obtain a new NAND-CIRC program that satisfies by hardwirecorrecteq that for every . This demonstrates that has a polynomial sized NAND-CIRC program, hence completing the proof of rnandthm.

## Derandomization

The proof of rnandthm can be summarized as follows: we can replace a -time algorithm that tosses coins as it runs with an algorithm that uses a single set of coin tosses which will be good enough for all inputs of size . Another way to say it is that for the purposes of computing functions, we do not need “online” access to random coins and can generate a set of coins “offline” ahead of time, before we see the actual input.

But this does not really help us with answering the question of whether equals , since we still need to find a way to generate these “offline” coins in the first place. To derandomize an RNAND-TM program we will need to come up with a *single* deterministic algorithm that will work for *all input lengths*. That is, unlike in the case of RNAND-CIRC programs, we cannot choose for every input length some string to use as our random coins.

Can we derandomize randomized algorithms, or does randomness add an inherent extra power for computation? This is a fundamentally interesting question but is also of practical significance. Ever since people started to use randomized algorithms during the Manhattan project, they have been trying to remove the need for randomness and replace it with numbers that are selected through some deterministic process. Throughout the years this approach has often been used successfully, though there have been a number of failures as well.[[2]](#footnote-60)

A common approach people used over the years was to replace the random coins of the algorithm by a “randomish looking” string that they generated through some arithmetic progress. For example, one can use the digits of for the random tape. Using these type of methods corresponds to what von Neumann referred to as a “state of sin”. (Though this is a sin that he himself frequently committed, as generating true randomness in sufficient quantity was and still is often too expensive.) The reason that this is considered a “sin” is that such a procedure will not work in general. For example, it is easy to modify any probabilistic algorithm such as the ones we have seen in #randomizedalgchap, to an algorithm that is *guaranteed to fail* if the random tape happens to equal the digits of . This means that the procedure “replace the random tape by the digits of ” does not yield a *general* way to transform a probabilistic algorithm to a deterministic one that will solve the same problem. Of course, this procedure does not *always* fail, but we have no good way to determine when it fails and when it succeeds. This reasoning is not specific to and holds for every deterministically produced string, whether it obtained by , , the Fibonacci series, or anything else.

An algorithm that checks if its random tape is equal to and then fails seems to be quite silly, but this is but the “tip of the iceberg” for a very serious issue. Time and again people have learned the hard way that one needs to be very careful about producing random bits using deterministic means. As we will see when we discuss cryptography, many spectacular security failures and break-ins were the result of using “insufficiently random” coins.

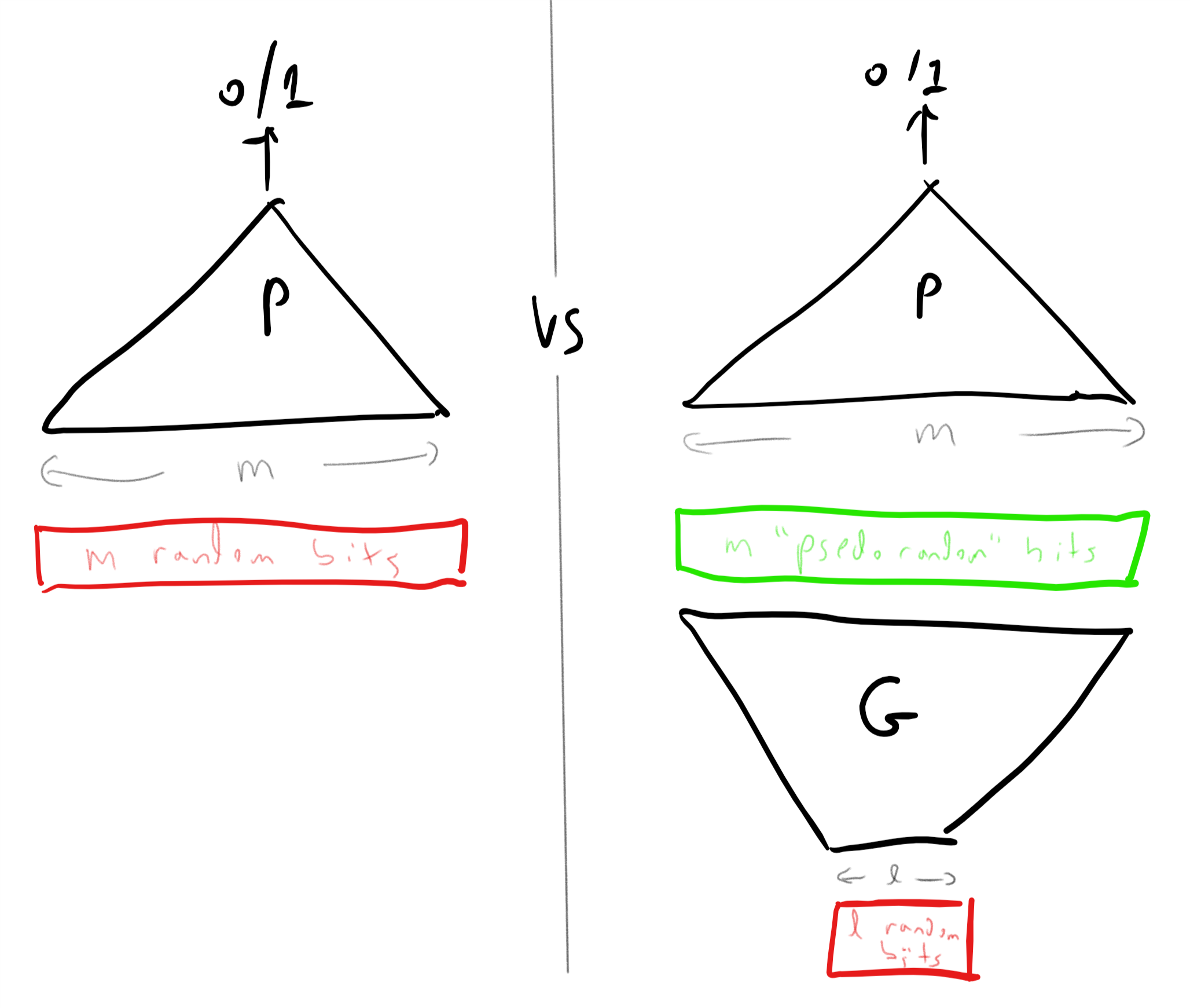
### Pseudorandom generators

So, we can’t use any *single* string to “derandomize” a probabilistic algorithm. It turns out however, that we can use a *collection* of strings to do so. Another way to think about it is that rather than trying to *eliminate* the need for randomness, we start by focusing on *reducing* the amount of randomness needed. (Though we will see that if we reduce the randomness sufficiently, we can eventually get rid of it altogether.)

We make the following definition:

### 

A function is a *-pseudorandom generator* if for every circuit with inputs, one output, and at most gates,



A pseudorandom generator maps a short string into a long string such that an small program/circuit cannot distinguish between the case that it is provided a random input and the case that it is provided a “pseudorandom” input of the form where . The short string is sometimes called the *seed* of the pseudorandom generator, as it is a small object that can be thought as yielding a large “tree of randomness”.

This is a definition that’s worth reading more than once, and spending some time to digest it. Note that it takes several parameters:

* is the limit on the number of gates of the circuit that the generator needs to “fool”. The larger is, the stronger the generator.
* is how close is the output of the pseudorandom generator to the true uniform distribution over . The smaller is, the stronger the generator.
* is the input length and is the output length. If then it is trivial to come up with such a generator: on input , we can output . In this case will simply equal , no matter how many lines has. So, the smaller is and the larger is, the stronger the generator, and to get anything non-trivial, we need .

Furthermore note that although our eventual goal is to fool probabilistic randomized algorithms that take an unbounded number of inputs, prgdef refers to *finite* and *deterministic* NAND-CIRC programs.

We can think of a pseudorandom generator as a “randomness amplifier.” It takes an input of bits chosen at random and expands these bits into an output of *pseudorandom* bits. If is small enough then the pseudorandom bits will “look random” to any NAND-CIRC program that is not too big. Still, there are two questions we haven’t answered:

* *What reason do we have to believe that pseudorandom generators with non-trivial parameters exist?*
* *Even if they do exist, why would such generators be useful to derandomize randomized algorithms?* After all, prgdef does not involve RNAND-TM or RNAND-RAM programs, but rather deterministic NAND-CIRC programs with no randomness and no loops.

We will now (partially) answer both questions. For the first question, let us come clean and confess we do not know how to *prove* that interesting pseudorandom generators exist. By *interesting* we mean pseudorandom generators that satisfy that is some small constant (say ), , and the function itself can be computed in time. Nevertheless, prgexist (whose statement and proof is deferred to the end of this chapter) shows that if we only drop the last condition (polynomial-time computability), then there do in fact exist pseudorandom generators where is *exponentially larger* than .

### 

At this point you might want to skip ahead and look at the *statement* of prgexist. However, since its *proof* is somewhat subtle, I recommend you defer reading it until you’ve finished reading the rest of this chapter.

### From existence to constructivity

The fact that there *exists* a pseudorandom generator does not mean that there is one that can be efficiently computed. However, it turns out that we can turn complexity “on its head” and use the assumed *non existence* of fast algorithms for problems such as 3SAT to obtain pseudorandom generators that can then be used to transform randomized algorithms into deterministic ones. This is known as the *Hardness vs Randomness* paradigm. A number of results along those lines, most of which are outside the scope of this course, have led researchers to believe the following conjecture:

**Optimal PRG conjecture:** There is a polynomial-time computable function that yields an *exponentially secure pseudorandom generator*.

Specifically, there exists a constant such that for every and , if we define as for every and , then is a pseudorandom generator.

The “optimal PRG conjecture” is worth while reading more than once. What it posits is that we can obtain pseudorandom generator such that every output bit of can be computed in time polynomial in the length of the input, where is exponentially large in and is exponentially small in . (Note that we could not hope for the entire output to be computable in , as just writing the output down will take too long.)

To understand why we call such a pseudorandom generator “optimal,” it is a great exercise to convince yourself that, for example, there does not exist a pseudorandom generator (in fact, the number in the conjecture must be smaller than ). To see that we can’t have , note that if we allow a NAND-CIRC program with much more than lines then this NAND-CIRC program could “hardwire” inside it all the outputs of on all its inputs, and use that to distinguish between a string of the form and a uniformly chosen string in . To see that we can’t have , note that by guessing the input (which will be successful with probability ), we can obtain a small (i.e., line) NAND-CIRC program that achieves a advantage in distinguishing a pseudorandom and uniform input. Working out these details is a highly recommended exercise.

We emphasize again that the optimal PRG conjecture is, as its name implies, a *conjecture*, and we still do not know how to *prove* it. In particular, it is stronger than the conjecture that . But we do have some evidence for its truth. There is a spectrum of different types of pseudorandom generators, and there are weaker assumptions than the optimal PRG conjecture that suffice to prove that . In particular this is known to hold under the assumption that there exists a function and such that for every sufficiently large , is not in . The name “Optimal PRG conjecture” is non standard. This conjecture is sometimes known in the literature as the existence of *exponentially strong pseudorandom functions*.[[3]](#footnote-69)

### Usefulness of pseudorandom generators

We now show that optimal pseudorandom generators are indeed very useful, by proving the following theorem:

### 

Suppose that the optimal PRG conjecture is true. Then .

### 

The optimal PRG conjecture tells us that we can achieve *exponential expansion* of truly random coins into as many as “pseudorandom coins.” Looked at from the other direction, it allows us to reduce the need for randomness by taking an algorithm that uses coins and converting it into an algorithm that only uses coins. Now an algorithm of the latter type by can be made fully deterministic by enumerating over all the (which is polynomial in ) possibilities for its random choices.

We now proceed with the proof details.

Let and let be a NAND-TM program and constants such that for every , runs in at most steps and . By “unrolling the loop” and hardwiring the input , we can obtain for every input a NAND-CIRC program of at most, say, lines, that takes bits of input and such that .

Now suppose that is a pseudorandom generator. Then we could deterministically estimate the probability up to accuracy in time where is the time that it takes to compute a single output bit of .

The reason is that we know that will give us such an estimate for , and we can compute the probability by simply trying all possibillites for . Now, under the optimal PRG conjecture we can set or equivalently , and our total computation time is polynomial in . Since , this running time will be polynomial in .

This completes the proof, since we are guaranteed that , and hence estimating the probability to within accuracy is sufficient to compute .

## and vs

Two computational complexity questions that we cannot settle are:

* Is ? Where we believe the answer is *negative*.
* Is ? Where we believe the answer is *positive*.

However we can say that the “conventional wisdom” is correct on at least one of these questions. Namely, if we’re wrong on the first count, then we’ll be right on the second one:

### 

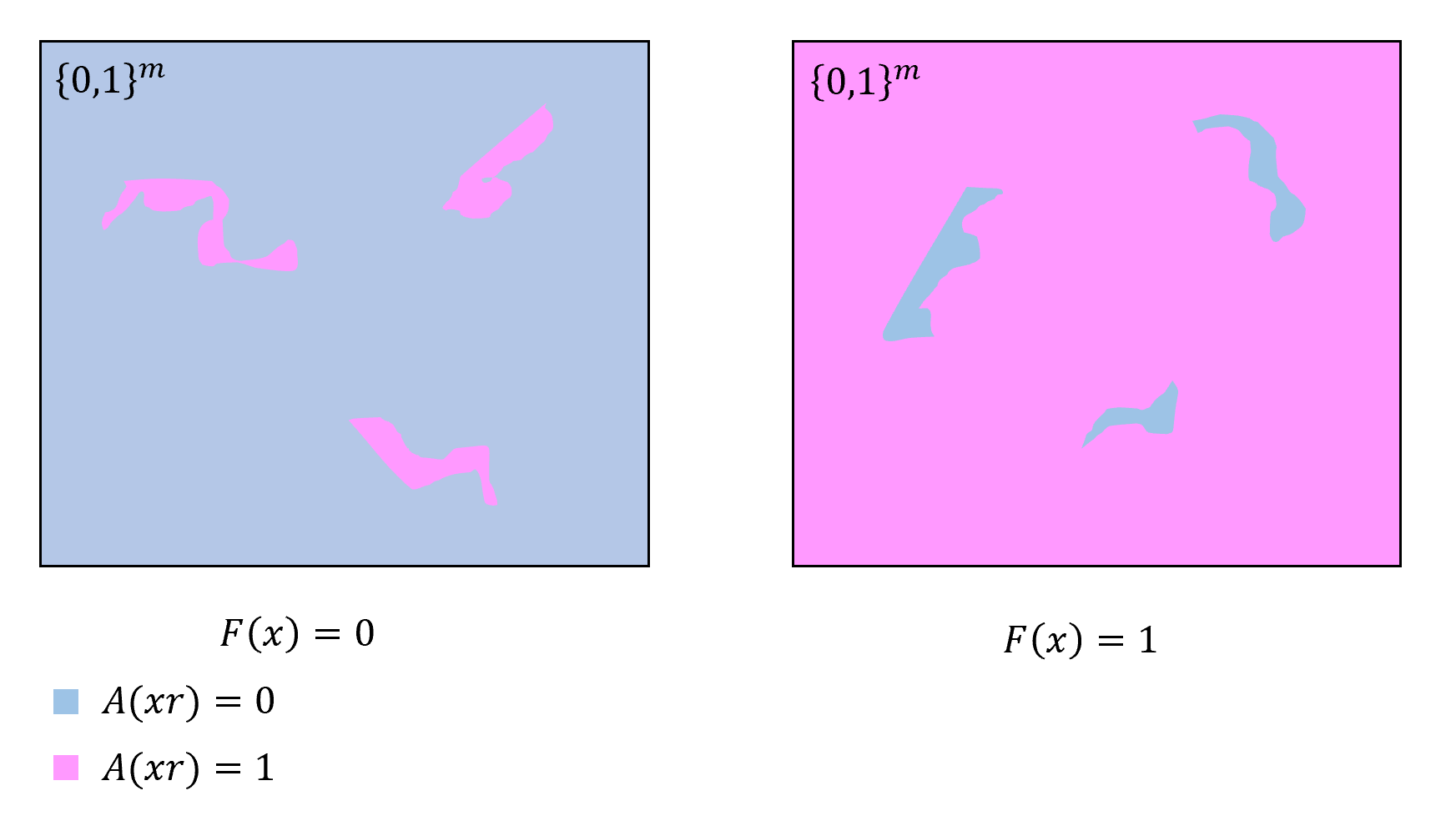
If then .

Before reading the proof, it is instructive to think why this result is not “obvious.” If then given any randomized algorithm and input , we will be able to figure out in polynomial time if there is a string of random coins for such that . The problem is that even if , it can still be the case that even when there exists a string such that .

The proof is rather subtle. It is much more important that you understand the *statement* of the theorem than that you follow all the details of the proof.

The construction follows the “quantifier elimination” idea which we have seen in PH-collapse-thm. We will show that for every , we can reduce the question of some input satisfies to the question of whether a formula of the form is true, where are polynomial in the length of and is polynomial-time computable. By PH-collapse-thm, if then we can decide in polynomial time whether such a formula is true or false.

The idea behind this construction is that using amplification we can obtain a randomized algorithm for computing using coins such that for every , if then the set of coins that make output is extremely tiny, and if then it is very large. Now in the case , one can show that this means that there exists a small number of “shifts” such that the union of the sets (i.e., sets of the form ) covers , while in the case this union will always be of size at most which is much smaller than . We can express the condition that there exists such that as a statement with a constant number of quantifiers.



If then through amplification we can ensure that there is an algorithm to compute on -length inputs and using coins such that . Hence if then almost all of the choices for will cause to output , while if then for almost all ’s. To prove the Sipser–Gács Theorem we consider several “shifts” of the set of the coins such that . If then we can find a set of shifts for which . If then for every such set . We can phrase the question of whether there is such a set of shift using a constant number of quantifiers, and so can solve it in polynomial time if .

Let . Using amplificationthm, there exists a polynomial-time algorithm such that for every , where is polynomial in . In particular (since an exponential dominates a polynomial, and we can always assume is sufficiently large), it holds that

Let , and let be the set . By our assumption, if then and if then .

For a set and a string , we define the set to be where denotes the XOR operation. That is, is the set “shifted” by . Note that . (Please make sure that you see why this is true.)

The heart of the proof is the following two claims:

**CLAIM I:** For every subset , if , then for every , .

**CLAIM II:** For every subset , if then there exist a set of string such that .

CLAIM I and CLAIM II together imply the theorem. Indeed, they mean that under our assumptions, for every , if and only if

which we can re-write as

$$
\exists\_{s\_0,\ldots, s\_{100m-1} \in \{0,1\}^m} \forall\_{w\in \{0,1\}^m} \Bigl( w \in (S\_x \oplus s\_0) \vee w \in (S\_x \oplus s\_1) \vee \cdots w \in (S\_x \oplus s\_{100m-1}) \Bigr)
$$

or equivalently

$$
\exists\_{s\_0,\ldots, s\_{100m-1} \in \{0,1\}^m} \forall\_{w\in \{0,1\}^m} \Bigl( A(x(w\oplus s\_0))=1 \vee A(x(w\oplus s\_1))=1 \vee \cdots \vee A(x(w\oplus s\_{100m-1}))=1 \Bigr)
$$

which (since is computable in polynomial time) is exactly the type of statement shown in PH-collapse-thm to be decidable in polynomial time if .

We see that all that is left is to prove **CLAIM I** and **CLAIM II**. **CLAIM I** follows immediately from the fact that

To prove **CLAIM II**, we will use a technique known as the *probabilistic method* (see the proof of prgexist for a more extensive discussion). Note that this is a completely different use of probability than in the theorem statement, we just use the methods of probability to prove an *existential* statement.

**Proof of CLAIM II:** Let with be as in the claim’s statement. Consider the following probabilistic experiment: we choose random shifts independently at random in , and consider the event that . To prove CLAIM II it is enough to show that , since that means that in particular there must *exist* shifts that satisfy this condition.

For every , define the event to hold if . The event holds if fails for every , and so our goal is to prove that . By the union bound, to show this, it is enough to show that for every . Define the event to hold if . Since every shift is chosen independently, for every fixed the events are mutually independent,[[4]](#footnote-78) and hence

So this means that the result will follow by showing that for every and (as that would allow to bound the righthand side of sipsergacsprodboundeq by ). In other words, we need to show that for every and set with ,

To show this, we observe that if and only if (can you see why). Hence we can rewrite the probability on the lefthand side of sipsergacsprodboundtwoeq as which simply equals ! This concludes the proof of **CLAIM I** and hence of BPPvsNP.

## Non-constructive existence of pseudorandom generators (advanced, optional)

We now show that, if we don’t insist on *constructivity* of pseudorandom generators, then we can show that there exists pseudorandom generators with output that *exponentially larger* in the input length.

### 

There is some absolute constant such that for every , if and , then there is an pseudorandom generator .

### 

The proof uses an extremely useful technique known as the “probabilistic method” which is not too hard mathematically but can be confusing at first.[[5]](#footnote-82) The idea is to give a “non constructive” proof of existence of the pseudorandom generator by showing that if was chosen at random, then the probability that it would be a valid pseudorandom generator is positive. In particular this means that there *exists* a single that is a valid pseudorandom generator. The probabilistic method is just a *proof technique* to demonstrate the existence of such a function. Ultimately, our goal is to show the existence of a *deterministic* function that satisfies the condition.

The above discussion might be rather abstract at this point, but would become clearer after seeing the proof.

Let be as in the lemma’s statement. We need to show that there exists a function that “fools” every line program in the sense of eq:prg. We will show that this follows from the following claim:

**Claim I:** For every fixed NAND-CIRC program , if we pick *at random* then the probability that eq:prg is violated is at most .

Before proving Claim I, let us see why it implies prgexist. We can identify a function with its “truth table” or simply the list of evaluations on all its possible inputs. Since each output is an bit string, we can also think of as a string in . We define to be the set of all functions from to . As discussed above we can identify with and choosing a random function corresponds to choosing a random -long bit string.

For every NAND-CIRC program let be the event that, if we choose at random from then eq:prg is violated with respect to the program . It is important to understand what is the sample space that the event is defined over, namely this event depends on the choice of and so is a subset of . An equivalent way to define the event is that it is the subset of all functions mapping to that violate eq:prg, or in other words:

(We’ve replaced here the probability statements in eq:prg with the equivalent sums so as to reduce confusion as to what is the sample space that is defined over.)

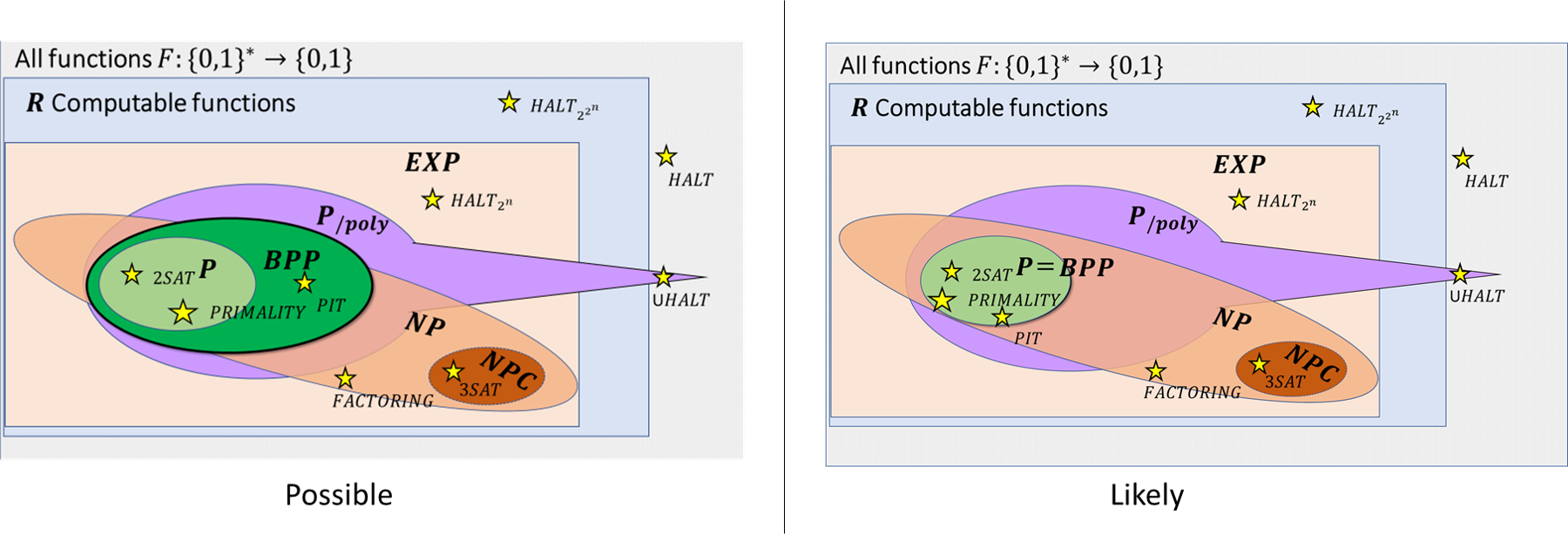
To understand this proof it is crucial that you pause here and see how the definition of above corresponds to eq:eventdefine. This may well take re-reading the above text once or twice, but it is a good exercise at parsing probabilistic statements and learning how to identify the *sample space* that these statements correspond to.

Now, we’ve shown in program-count that up to renaming variables (which makes no difference to program’s functionality) there are NAND-CIRC programs of at most lines. Since for sufficiently large , this means that if Claim I is true, then by the union bound it holds that the probability of the union of over *all* NAND-CIRC programs of at most lines is at most for sufficiently large . What is important for us about the number is that it is smaller than . In particular this means that there *exists* a single such that *does not* violate eq:prg with respect to any NAND-CIRC program of at most lines, but that precisely means that is a pseudorandom generator.

Hence to conclude the proof of prgexist, it suffices to prove Claim I. Choosing a random amounts to choosing random strings and letting (identifying and via the binary representation). This means that proving the claim amounts to showing that for every fixed function , if (which by setting , we can ensure is larger than ) then the probability that

is at most .

{eq:prgchernoff} follows directly from the Chernoff bound. Indeed, if we let for every the random variable denote , then since is chosen independently at random, these are independently and identically distributed random variables with mean and hence the probability that they deviate from their expectation by is at most .



The relation between and the other complexity classes that we have seen. We know that and but we don’t know how compares with and can’t rule out even . Most evidence points out to the possibliity that .

* We can model randomized algorithms by either adding a special “coin toss” operation or assuming an extra randomly chosen input.
* The class contains the set of Boolean functions that can be computed by polynomial time randomized algorithms.
* is a *worst case* class of computation: a randomized algorithm to compute a function must compute it correctly with high probability *on every input*.
* We can *amplify* the success probability of randomized algorithm from any value strictly larger than into a success probability that is *exponentiall close to* .
* We know that .
* We also know that .
* The relation between and is not known, but we do know that if then .
* Pseudorandom generators are objects that take a short random “seed” and expand it to a much longer output that “appears random” for efficient algorithms. We conjecture that exponentially strong pseudorandom generators exist. Under this conjecture, .

## Exercises

## Bibliographical notes

In this chapter we ignore the issue of how we actually get random bits in practice. The output of many physical processes, whether it is thermal heat, network and hard drive latency, user typing pattern and mouse movements, and more can be thought of as a binary string sampled from some distribution that might have significant unpredictability (or *entropy*) but is not necessarily the *uniform* distribution over . Indeed, as [this paper](http://statweb.stanford.edu/~susan/papers/headswithJ.pdf) shows, even (real-world) coin tosses do not have exactly the distribution of a uniformly random string. Therefore, to use the resulting measurements for randomized algorithms, one typically needs to apply a “distillation” or [randomness extraction](https://en.wikipedia.org/wiki/Randomness_extractor) process to the raw measurements to transform them to the uniform distribution. Vadhan’s book [@vadhan2012pseudorandomness] is an excellent source for more discussion on both randomness extractors and pseudorandom generators.

The name stands for “bounded probability polynomial time”. This is an historical accident: this class probably should have been called or but both names were taken by other classes.

The proof of rnandthm actually yields more than its statement. We can use the same “unrolling the loop” arguments we’ve used before to show that the restriction to of every function in is also computable by a polynomial-size RNAND-CIRC program (i.e., NAND-CIRC program with the RAND operation). Like in the vs case, there are also functions outside whose restrictions can be computed by polynomial-size RNAND-CIRC programs. Nevertheless the proof of rnandthm shows that even such functions can be computed by polynomial sized NAND-CIRC programs without using the rand operations. This can be phrased as saying that (where is defined in the natural way using RNAND progams). The stronger version of rnandthm we mentioned can be phrased as saying that .

1. At the time of this writing, the largest “natural” complexity class which we can’t rule out being contained in is the class , which we did not define in this course, but corresponds to non deterministic exponential time. See [this paper](https://people.csail.mit.edu/rrw/nexp-v-bpp.pdf) for a discussion of this question. [↑](#footnote-ref-46)
2. One amusing anecdote is a [recent case](https://www.wired.com/2017/02/russians-engineer-brilliant-slot-machine-cheat-casinos-no-fix/) where scammers managed to predict the imperfect “pseudorandom generator” used by slot machines to cheat casinos. Unfortunately we don’t know the details of how they did it, since the case was [sealed](https://www.plainsite.org/dockets/2j3mlaig6/missouri-eastern-district-court/usa-v-bliev-et-al/). [↑](#footnote-ref-60)
3. A pseudorandom generator of the form we posit, where each output bit can be computed individually in time polynomial in the seed length, is commonly known as a *pseudorandom function generator*. For more on the many interesting results and connections in the study of *pseudorandomness*, see [this monograph of Salil Vadhan](https://people.seas.harvard.edu/~salil/pseudorandomness/). [↑](#footnote-ref-69)
4. The condition of independence here is subtle. It is *not* the case that all of the events are mutually independent. Only for a fixed , the events of the form are mutually independent. [↑](#footnote-ref-78)
5. There is a whole (highly recommended) [book by Alon and Spencer](https://www.amazon.com/Probabilistic-Method-Discrete-Mathematics-Optimization/dp/1119061954/ref=dp_ob_title_bk) devoted to this method. [↑](#footnote-ref-82)