Modeling running time

Boaz Barak

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

# Modeling running time

### 

* Formally modeling running time, and in particular notions such as or time algorithms.
* The classes and modelling polynomial and exponential time respectively.
* The *time hierarchy theorem*, that in particular says that for every there are functions we *can* compute in time but *can not* compute in time.
* The class of *non uniform* computation and the result that

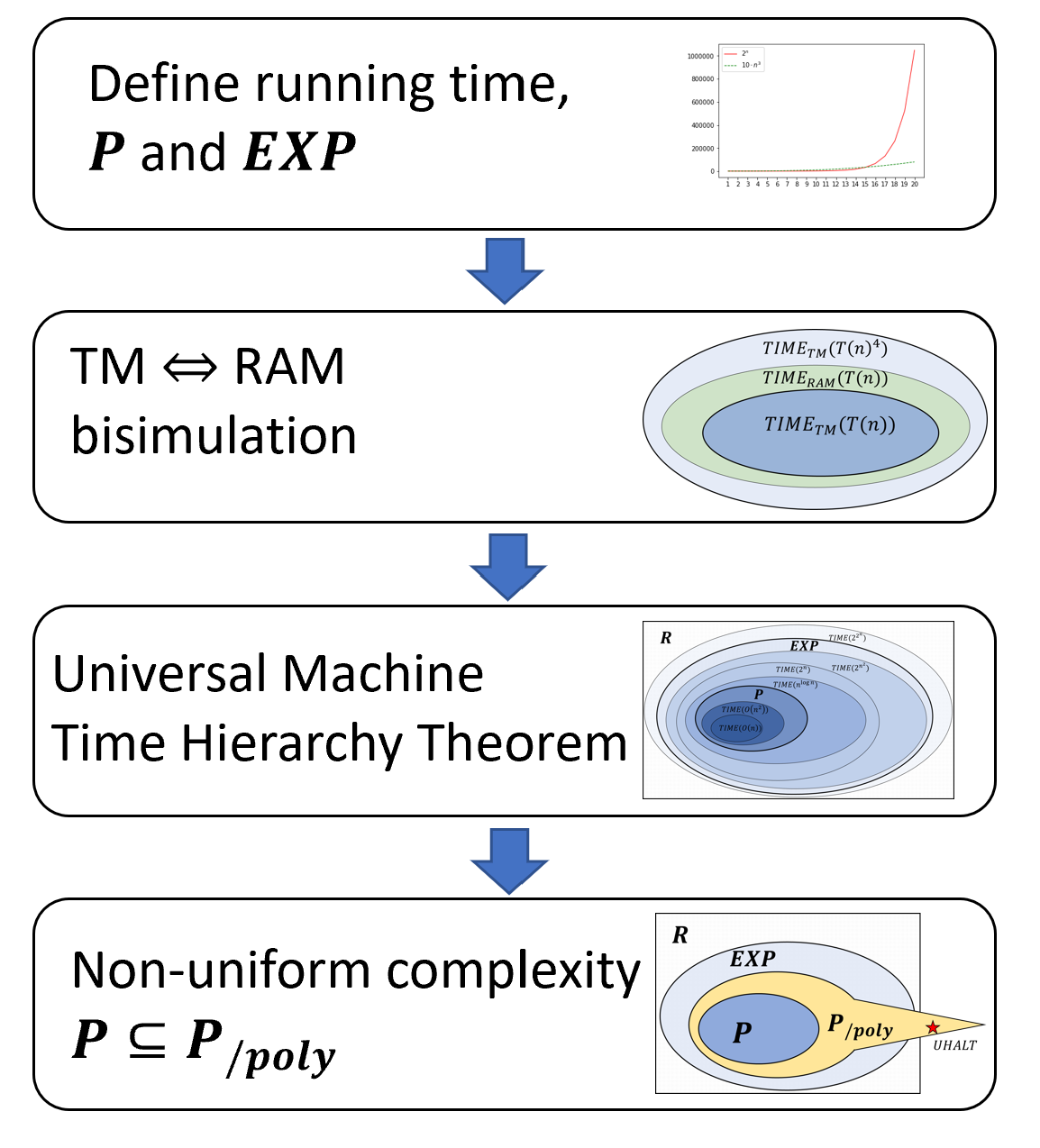
“When the measure of the problem-size is reasonable and when the sizes assume values arbitrarily large, an asymptotic estimate of … the order of difficulty of [an] algorithm .. is theoretically important. It cannot be rigged by making the algorithm artificially difficult for smaller sizes”, Jack Edmonds, “Paths, Trees, and Flowers”, 1963

*Max Newman:* It is all very well to say that a machine could … do this or that, but … what about the time it would take to do it?

*Alan Turing:* To my mind this time factor is the one question which will involve all the real technical difficulty.

BBC radio panel on “Can automatic Calculating Machines Be Said to Think?”, 1952

In chapefficient we saw examples of efficient algorithms, and made some claims about their running time, but did not give a mathematically precise definition for this concept. We do so in this chapter, using the models of Turing machines and RAM machines (or equivalently NAND-TM and NAND-RAM) we have seen before. The running time of an algorithm is not a fixed number since any non-trivial algorithm will take longer to run on longer inputs. Thus, what we want to measure is the *dependence* between the number of steps the algorithms takes and the length of the input. In particular we care about the distinction between algorithms that take at most *polynomial time* (i.e., time for some constant ) and problems for which every algorithm requires at least *exponential time* (i.e., for some ). As mentioned in Edmond’s quote in chapefficient, the difference between these two can sometimes be as important as the difference between being computable and uncomputable.



Overview of the results of this chapter.

In this chapter we formally define the notion of a function being computable in time where is some function mapping the length of the input to a bound on the number of computation steps. We then do the following (see also runtimeoverviewfig):

* Define the class of Boolean functions that can be computed in polynomial time and its superset of functions that can be computed in exponential time.
* Show that the time to compute a function using a Turing Machine and using a RAM machine (or NAND-RAM program) is *polynomially related* which in particular means that the classes and can be equivalently defined using either Turing Machines or RAM machines / NAND-RAM programs.
* Give an *efficient* universal NAND-RAM program and use this to establish the *time hierarchy theorem* that in particular implies that .
* We relate the notions defined here to the *non uniform* models of Boolean circuits and NAND-CIRC programs defined in compchap. We define to be the class of functions computed by a *sequence* of polynomial-sized circuits. We prove that and that contains *uncomputable* functions.

## Formally defining running time

Our models of computation such Turing Machines, NAND-TM and NAND-RAM programs and others all operate by executing a sequence of instructions on an input one step at a time. We can define the *running time* of an algorithm in one of these models by measuring the number of steps takes on input as a *function of the length of the input*. We start by defining running time with respect to Turing Machines:

Let be some function mapping natural numbers to natural numbers. We say that a function is *computable in Single-Tape-Turing-Machine time (TM-time for short)* if there exists a Turing Machine such that for every sufficiently large and every , when given input , the machine halts after executing at most steps and outputs .

We define to be the set of Boolean functions (functions mapping to ) that are computable in TM time.

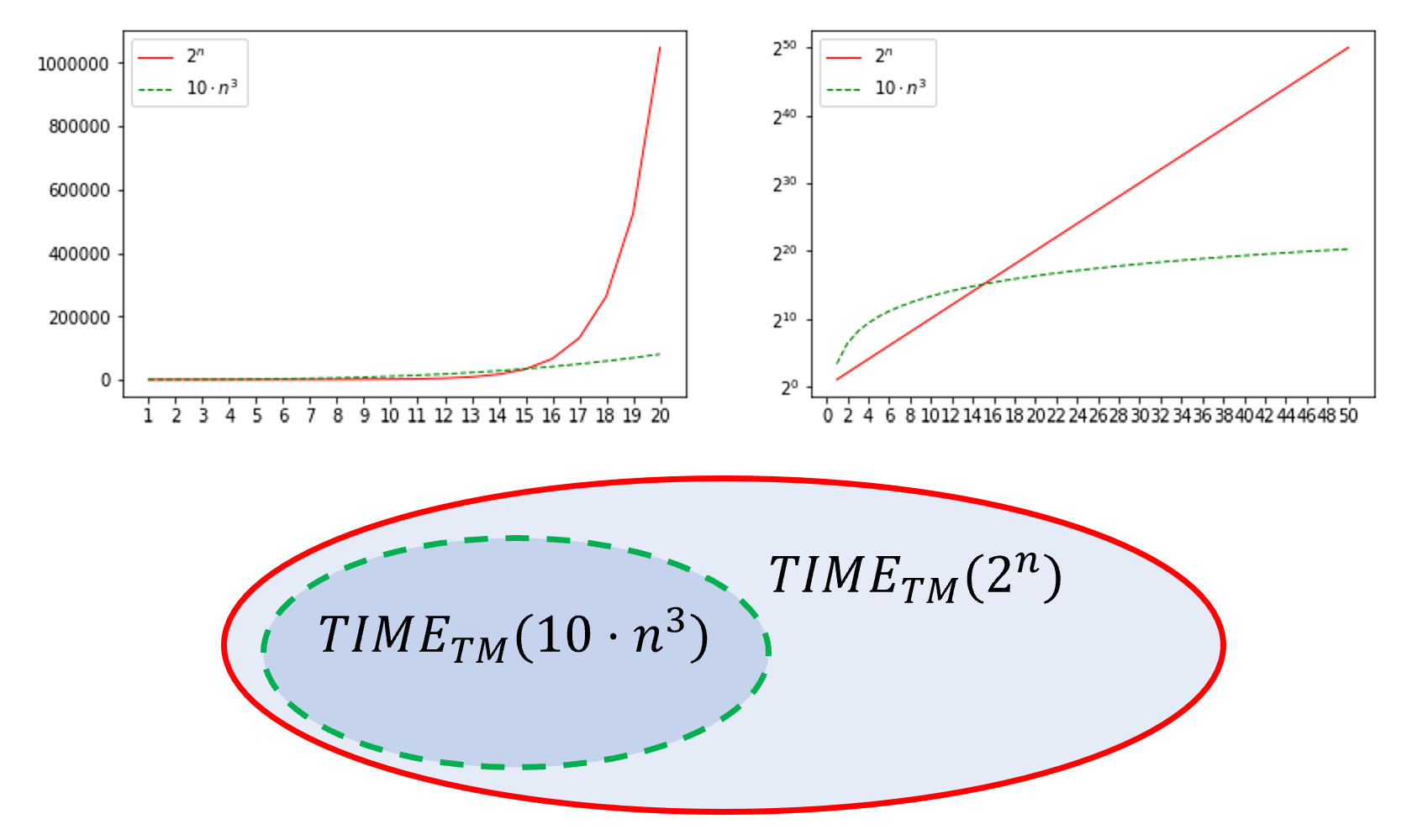
For a function and , we can formally define what it means for to be computable in time at most where is the size of the input.

time-TM-def is not very complicated but is one of the most important definitions of this book. As usual, is a class of *functions*, not of *machines*. If is a Turing Machine then a statement such as “ is a member of ” does not make sense.

The relaxation of considering only “sufficiently large” ’s is not very important but it is convenient since it allows us to avoid dealing explicitly with un-interesting “edge cases”. In most cases we will anyway be interested in determining running time only up to constant and even polynomial factors. Note that we can always compute a function on a finite number of inputs using a lookup table.

While the notion of being computable within a certain running time can be defined for every function, the class is a class of *Boolean functions* that have a single bit of output. This choice is not very important, but is made for simplicity and convenience later on. In fact, every non-Boolean function has a computationally equivalent Boolean variant, see boolex.

Prove that .



Comparing with (on the right figure the Y axis is in log scale). Since for every large enough , , .

The proof is illustrated in examplefimeboundsfig. Suppose that and hence there some number and a machine such that for every , and , outputs within at most steps. Since , there is some number such that for every , . Hence for every , will output within at most steps, just demonstrating that .

### Polynomial and Exponential Time

Unlike the notion of computability, the exact running time can be a function of the model we use. However, it turns out that if we only care about “coarse enough” resolution (as will most often be the case) then the choice of the model, whether Turing Machines, RAM machines, NAND-TM/NAND-RAM programs, or C/Python programs, does not matter. This is known as the *extended* Church-Turing Thesis. Specifically we will mostly care about the difference between *polynomial* and *exponential* time.

The two main time complexity classes we will be interested in are the following:

* **Polynomial time:** A function is *computable in polynomial time* if it is in the class . That is, if there is an algorithm to compute that runs in time at most *polynomial* (i.e., at most for some constant ) in the length of the input.
* **Exponential time:** A function is *computable in exponential time* if it is in the class . That is, if there is an algorithm to compute that runs in time at most *exponential* (i.e., at most for some constant ) in the length of the input.

In other words, these are defined as follows:

Let . We say that if there is a polynomial and a Turing Machine such that for every , when given input , the Turing machine halts within at most steps and outputs .

We say that if there is a polynomial and a Turing Machine such that for every , when given input , halts within at most steps and outputs .

Please take the time to make sure you understand these definitions. In particular, sometimes students think of the class as corresponding to functions that are *not* in . However, this is not the case. If is in then it *can* be computed in exponential time. This does not mean that it cannot be computed in polynomial time as well.

Prove that as defined in PandEXPdef is equal to

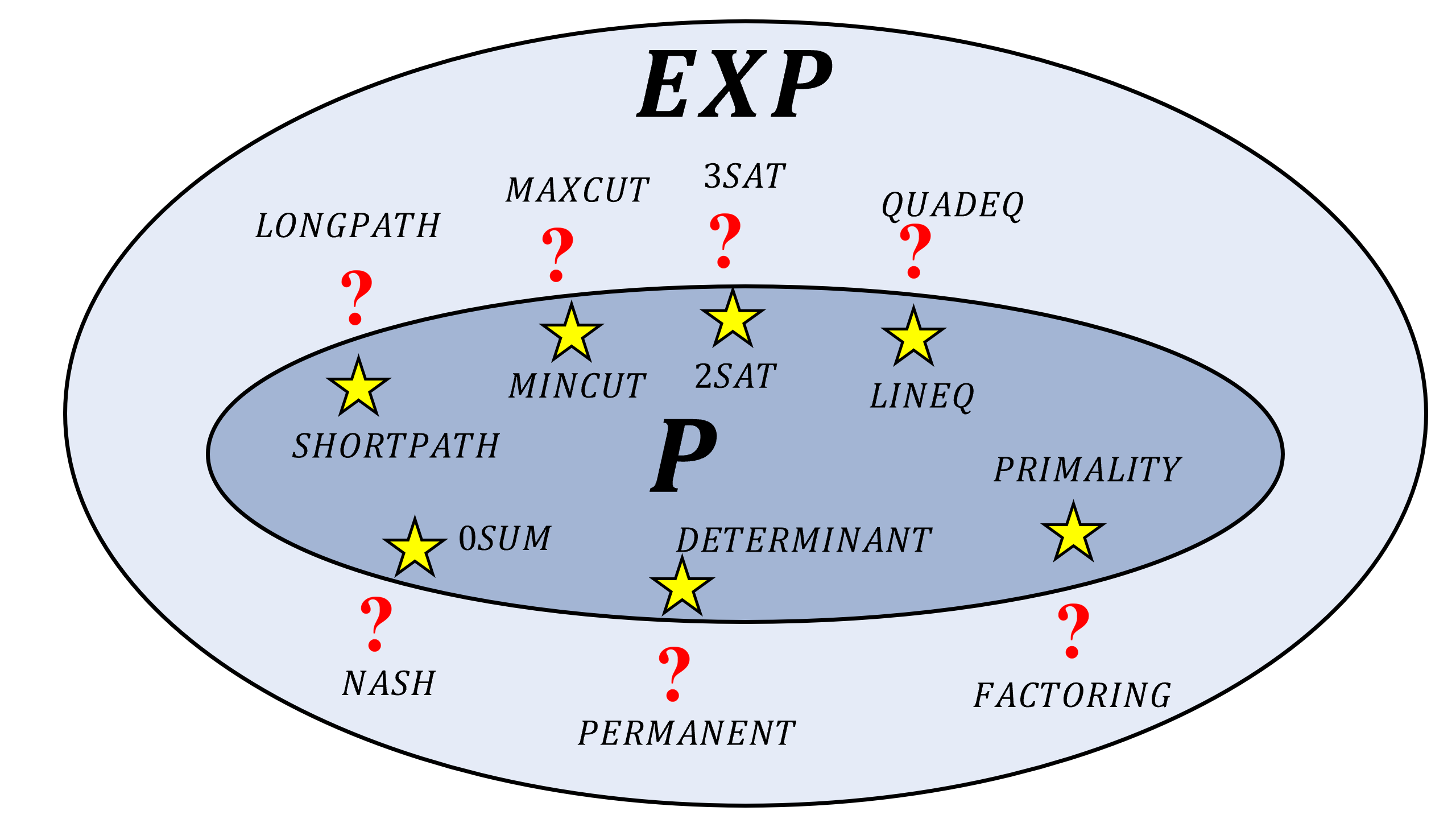
To show these two sets are equal we need to show that and . We start with the former inclusion. Suppose that . Then there is some polynomial and a Turing machine such that computes and halts on every input within at most steps. We can write the polynomial in the form where , and we assume that is nonzero (or otherwise we just let correspond to the largest number such that is nonzero). The *degree* if the number . Since , no matter what is the coefficient , for large enough , which means that the Turing machine will halt on inputs of length within fewer than steps, and hence .

For the second inclusion, suppose that . Then there is some positive such that which means that there is a Turing Machine and some number such that computes and for every , halts on length inputs within at most steps. Let be the maximum number of steps that takes on inputs of length at most . Then if we define the polynomial then we see that halts on every input within at most steps and hence the existence of demonstrates that .

Since exponential time is much larger than polynomial time, . All of the problems we listed in chapefficient are in , but as we’ve seen, for some of them there are much better algorithms that demonstrate that they are in fact in the smaller class .

|  |  |
| --- | --- |
|  | (but not known to be in ) |
| Shortest path | Longest Path |
| Min cut | Max cut |
| 2SAT | 3SAT |
| Linear eqs | Quad. eqs |
| Zerosum | Nash |
| Determinant | Permanent |
| Primality | Factoring |

Table : A table of the examples from chapefficient. All these problems are in but the only the ones on the left column are currently known to be in as well (i.e., they have a polynomial-time algorithm). See also PvsEXPfig.



Some examples of problems that are known to be in and problems that are known to be in but not known whether or not they are in . Since both and are classes of Boolean functions, in this figure we always refer to the *Boolean* (i.e., Yes/No) variant of the problems.

Many of the problems defined in chapefficient correspond to *non Boolean* functions (functions with more than one bit of output) while and are sets of Boolean functions. However, for every non-Boolean function we can always define a computationally-equivalent Boolean function by letting be the -th bit of (see boolex). Hence the table above, as well as PvsEXPfig, refer to the computationally-equivalent Boolean variants of these problems.

## Modeling running time using RAM Machines / NAND-RAM

Turing Machines are a clean theoretical model of computation, but do not closely correspond to real-world computing architectures. The discrepancy between Turing Machines and actual computers does not matter much when we consider the question of which functions are *computable*, but can make a difference in the context of *efficiency*. Even a basic staple of undergraduate algorithms such as “merge sort” cannot be implemented on a Turing Machine in time (see bibnotesrunningtime). *RAM machines* (or equivalently, NAND-RAM programs) match more closely actual computing architecture and what we mean when we say or algorithms in algorithms courses or whiteboard coding interviews. We can define running time with respect to NAND-RAM programs just as we did for Turing Machines.

Let be some function mapping natural numbers to natural numbers. We say that a function is *computable in RAM time (RAM-time for short)* if there exists a NAND-RAM program such that for every sufficiently large and every , when given input , the program halts after executing at most lines and outputs .

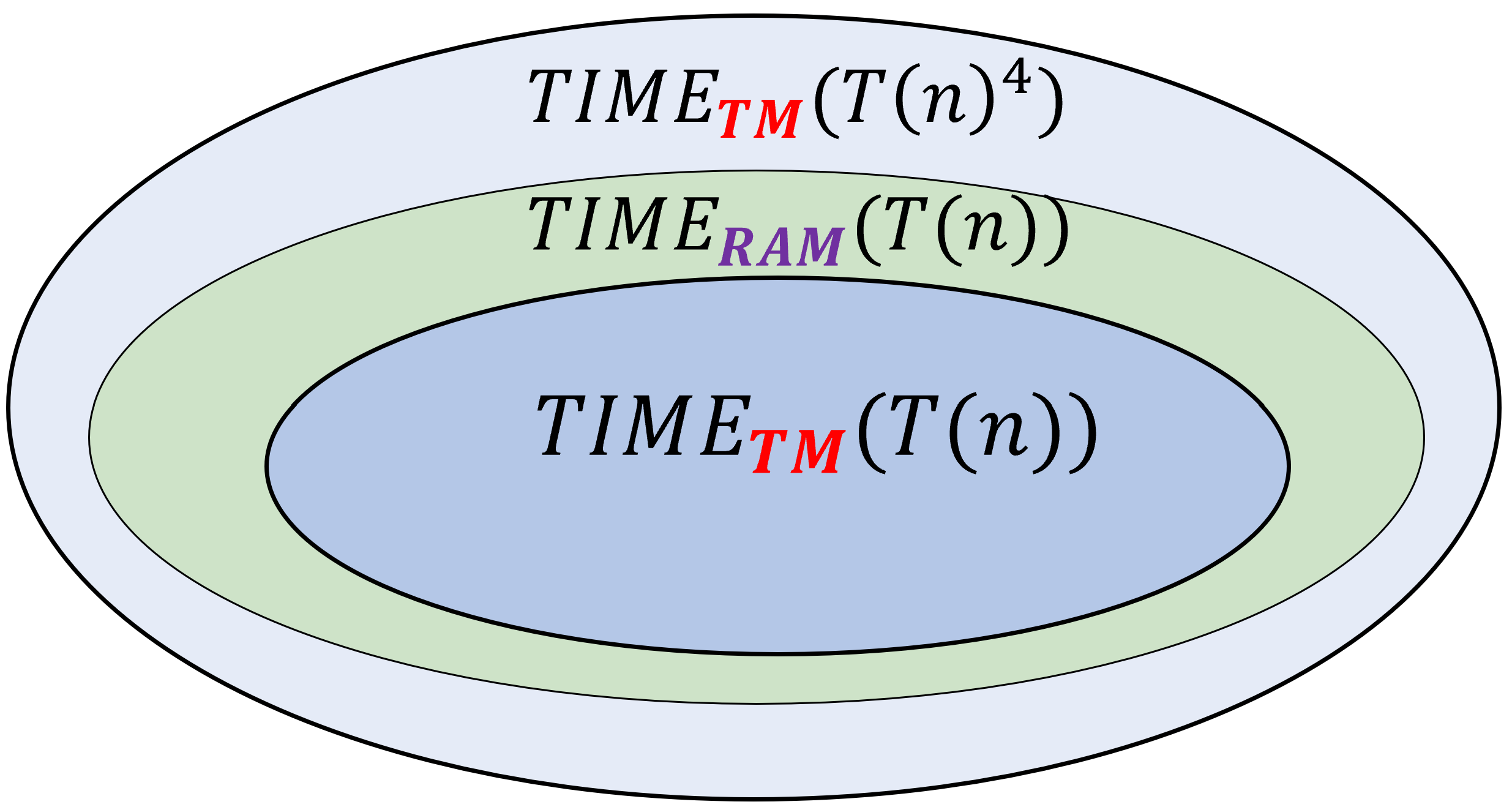
We define to be the set of Boolean functions (functions mapping to ) that are computable in RAM time.

Because NAND-RAM programs correspond more closely to our natural notions of running time, we will use NAND-RAM as our “default” model of running time, and hence use (without any subscript) to denote . However, it turns out that as long as we only care about the difference between exponential and polynomial time, this does not make much difference. The reason is that Turing Machines can simulate NAND-RAM programs with at most a polynomial overhead (see also RAMTMsimulationfig):

Let be a function such that for every and the map can be computed by a Turing machine in time . Then

The technical details of polyRAMTM-thm, such as the condition that is computable in time or the constants and in eqtmrambisimulation (which are not tight and can be improved), are not very important. In particular, all non pathological time bound functions we encounter in practice such as , , etc. will satisfy the conditions of polyRAMTM-thm, see also nicefunctionsrem.

The main message of the theorem is Turing Machines and RAM machines are “roughly equivalent” in the sense that one can simulate the other with polynomial overhead. Similarly, while the proof involves some technical details, it’s not very deep or hard, and merely follows the simulation of RAM machines with Turing Machines we saw in RAMTMequivalencethm with more careful “book keeping”.



The proof of polyRAMTM-thm shows that we can simulate steps of a Turing Machine with steps of a NAND-RAM program, and can simulate steps of a NAND-RAM program with steps of a Turing Machine. Hence .

For example, by instantiating polyRAMTM-thm with and using the fact that , we see that which means that (by diffdefofP)

That is, we could have equally well defined as the class of functions computable by *NAND-RAM programs* (instead of Turing Machines) that run in time polynomial in the length of the input. Similarly, by instantiating polyRAMTM-thm with we see that the class can also be defined as the set of functions computable by NAND-RAM programs in time at most where is some polynomial. Similar equivalence results are known for many models including cellular automata, C/Python/Javascript programs, parallel computers, and a great many other models, which justifies the choice of as capturing a technology-independent notion of tractability. (See #ECTTsec for more discussion of this issue.) This equivalence between Turing machines and NAND-RAM (as well as other models) allows us to pick our favorite model depending on the task at hand (i.e., “have our cake and eat it too”) even when we study questions of efficiency, as long as we only care about the gap between *polynomial* and *exponential* time. When we want to *design* an algorithm, we can use the extra power and convenience afforded by NAND-RAM. When we want to *analyze* a program or prove a *negative result*, we can restrict our attention to Turing machines.

All “reasonable” computational models are equivalent if we only care about the distinction between polynomial and exponential.

The adjective “reasonable” above refers to all scalable computational models that have been implemented, with the possible exception of *quantum computers*, see #ECTTsec and quantumchap.

### 

The direction is not hard to show, since a NAND-RAM program can simulate a Turing Machine with constant overhead by storing the transition table of in an array (as is done in the proof of universaltmthm). Simulating every step of the Turing machine can be done in a constant number of steps of RAM, and it can be shown this constant is smaller than . Thus the heart of the theorem is to prove that . This proof closely follows the proof of RAMTMequivalencethm, where we have shown that every function that is computable by a NAND-RAM program is computable by a Turing Machine (or equivalently a NAND-TM program) . To prove polyRAMTM-thm, we follow the exact same proof but just check that the overhead of the simulation of by is polynomial. The proof has many details, but is not deep. It is therefore much more important that you understand the *statement* of this theorem than its proof.

We only focus on the non-trivial direction . Let . can be computed in time by some NAND-RAM program and we need to show that it can also be computed in time by a Turing Machine . This will follow from showing that can be computed in time by a NAND-TM program, since for every NAND-TM program there is a Turing Machine simulating it such that each iteration of corresponds to a single step of .

As mentioned above, we follow the proof of RAMTMequivalencethm (simulation of NAND-RAM programs using NAND-TM programs) and use the exact same simulation, but with a more careful accounting of the number of steps that the simulation costs. Recall, that the simulation of NAND-RAM works by “peeling off” features of NAND-RAM one by one, until we are left with NAND-TM.

We will not provide the full details but will present the main ideas used in showing that every feature of NAND-RAM can be simulated by NAND-TM with at most a polynomial overhead:

1. Recall that every NAND-RAM variable or array element can contain an integer between and where is the number of lines that have been executed so far. Therefore if is a NAND-RAM program that computes in time, then on inputs of length , all integers used by are of magnitude at most . This means that the largest value i can ever reach is at most and so each one of ’s variables can be thought of as an array of at most indices, each of which holds a natural number of magnitude at most . We let be the number of bits needed to encode such numbers. (We can start off the simulation by computing and .)
2. We can encode a NAND-RAM array of length containing numbers in as an Boolean (i.e., NAND-TM) array of bits, which we can also think of as a *two dimensional array* as we did in the proof of RAMTMequivalencethm. We encode a NAND-RAM scalar containing a number in simply by a shorter NAND-TM array of bits.
3. We can simulate the two dimensional arrays using one-dimensional arrays of length . All the arithmetic operations on integers use the grade-school algorithms, that take time that is polynomial in the number of bits of the integers, which is in our case. Hence we can simulate steps of NAND-RAM with steps of a model that uses random access memory but only *Boolean-valued* one-dimensional arrays.
4. The most expensive step is to translate from random access memory to the sequential memory model of NAND-TM/Turing Machines. As we did in the proof of RAMTMequivalencethm (see nandtmgorydetailssec), we can simulate accessing an array Foo at some location encoded in an array Bar by:
   1. Copying Bar to some temporary array Temp
   2. Having an array Index which is initially all zeros except at the first location.
   3. Repeating the following until Temp encodes the number : *(Number of repetitions is at most .)*
      * Decrease the number encoded temp by . *(Take number of steps polynomial in .)*
      * Decrease i until it is equal to . *(Take steps.)*
      * Scan Index until we reach the point in which it equals and then change this to and go one step further and write in this location. *(Takes steps.)*
   4. When we are done we know that if we scan Index until we reach the point in which Index[i] then i contains the value that was encoded by Bar *(Takes steps.)*

The total cost for each such operation is steps.

In sum, we simulate a single step of NAND-RAM using steps of NAND-TM, and hence the total simulation time is which is smaller than for sufficiently large .

When considering general time bounds such we need to make sure to rule out some “pathological” cases such as functions that don’t give enough time for the algorithm to read the input, or functions where the time bound itself is uncomputable. We say that a function is a *nice time bound function* (or nice function for short) if for every , (i.e., allows enough time to read the input), for every , (i.e., allows more time on longer inputs), and the map (i.e., mapping a string of length to a sequence of ones) can be computed by a NAND-RAM program in time.

All the “normal” time complexity bounds we encounter in applications such as , ,, etc. are “nice”. Hence from now on we will only care about the class when is a “nice” function. The computability condition is in particular typically easily satisfied. For example, for arithmetic functions such as , we can typically compute the binary representation of in time polynomial *in the number of bits* of and hence poly-logarithmic in . Hence the time to write the string in such cases will be .

## Extended Church-Turing Thesis (discussion)

polyRAMTM-thm shows that the computational models of *Turing Machines* and *RAM Machines / NAND-RAM programs* are equivalent up to polynomial factors in the running time. Other examples of polynomially equivalent models include:

* All standard programming languages, including C/Python/JavaScript/Lisp/etc.
* The calculus (see also bibnotesrunningtime).
* Cellular automata
* Parallel computers
* Biological computing devices such as DNA-based computers.

The *Extended Church Turing Thesis* is the statement this is true for all physically realizable computing models. In other words, the extended Church Turing thesis says that for every *scalable computing device* (which has a finite description but can be in principle used to run computation on arbitrarily large inputs), there is some constant such that for every function that can compute on length inputs using an amount of physical resources, is in . This is a strengthening of the (“plain”) Church-Turing Thesis, discussed in churchturingdiscussionsec, which states that the set of computable functions is the same for all physically realizable models, but without requiring the overhead in the simulation between different models to be at most polynomial.

All the current constructions of scalable computational models and programming language conform to the Extended Church-Turing Thesis, in the sense that they can be with polynomial overhead by Turing Machines (and hence also by NAND-TM or NAND-RAM programs). consequently, the classes and are robust to the choice of model, and we can use the programming language of our choice, or high level descriptions of an algorithm, to determine whether or not a problem is in .

Like the Church-Turing thesis itself, the extended Church-Turing thesis is in the asymptotic setting and does not directly yield an experimentally testable prediction. However, it can be instantiated with more concrete bounds on the overhead, yielding experimentally-testable predictions such as the *Physical Extended Church-Turing Thesis* we mentioned in PECTTsec.

In the last hundred+ years of studying and mechanizing computation, no one has yet constructed a scalable computing device that violates the extended Church Turing Thesis. However, *quantum computing*, if realized, will pose a serious challenge to the extended Church-Turing Thesis (see quantumchap). However, even if the promises of quantum computing are fully realized, the extended Church-Turing thesis is “morally” correct, in the sense that, while we do need to adapt the thesis to account for the possibility of quantum computing, its broad outline remains unchanged. We are still able to model computation mathematically, we can still treat programs as strings and have a universal program, we still have time hierarchy and uncomputability results, and there is still no reason to doubt the (“plain”) Church-Turing thesis. Moreover, the prospect of quantum computing does not seem to make a difference for the time complexity of many (though not all!) of the concrete problems that we care about. In particular, as far as we know, out of all the example problems mentioned in chapefficient the complexity of only one— integer factoring— is affected by modifying our model to include quantum computers as well.

## Efficient universal machine: a NAND-RAM interpreter in NAND-RAM

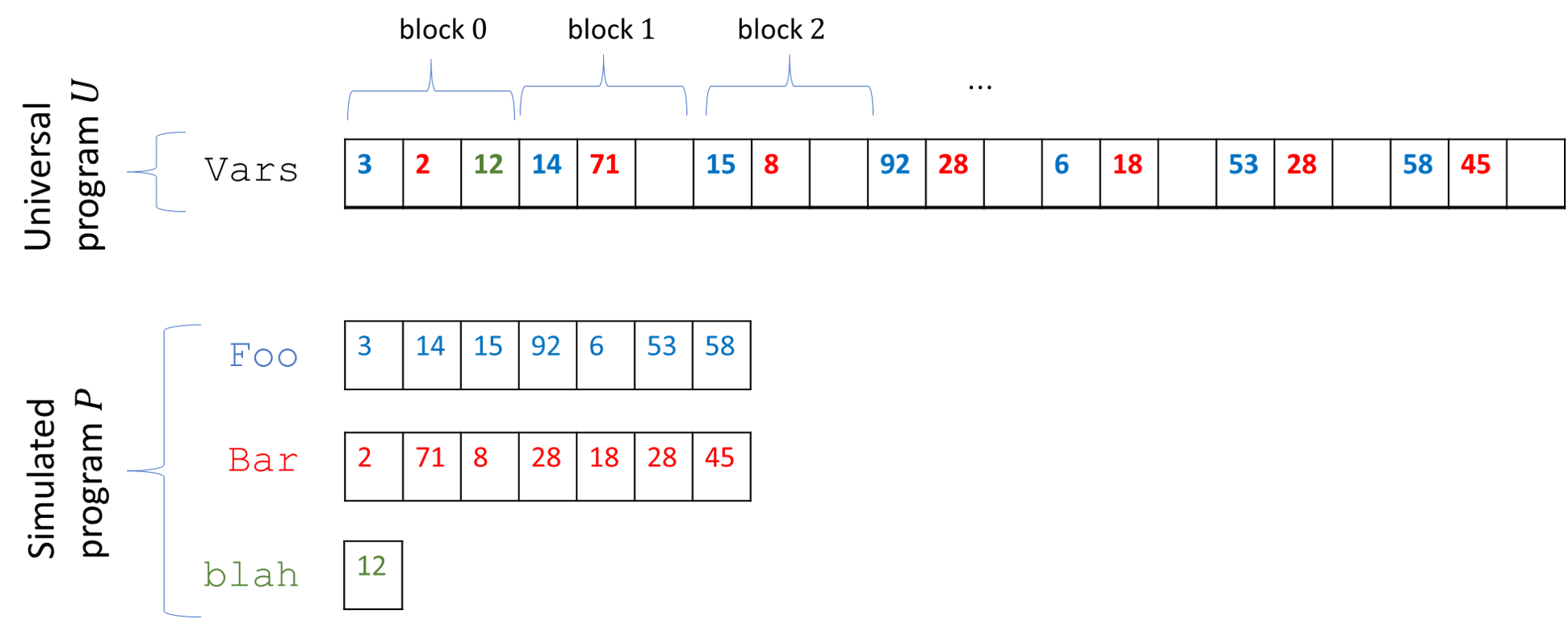
We have seen in universaltmthm the “universal Turing Machine”. Examining that proof, and combining it with polyRAMTM-thm , we can see that the program has a *polynomial* overhead, in the sense that it can simulate steps of a given NAND-TM (or NAND-RAM) program on an input in steps. But in fact, by directly simulating NAND-RAM programs we can do better with only a *constant* multiplicative overhead. That is, there is a *universal NAND-RAM program* such that for every NAND-RAM program , simulates steps of using only steps. (The implicit constant in the notation can depend on the program but does *not* depend on the length of the input.)

There exists a NAND-RAM program satisfying the following:

1. *( is a universal NAND-RAM program.)* For every NAND-RAM program and input , where by we denote the output of on a string encoding the pair .
2. *( is efficient.)* There are some constants such that for every NAND-RAM program , if halts on input after most steps, then halts after at most steps where .

### 

As in the case of polyRAMTM-thm, the proof of univ-nandpp is not very deep and so it is more important to understand its *statement*. Specifically, if you understand how you would go about writing an interpreter for NAND-RAM using a modern programming language such as Python, then you know everything you need to know about the proof of this theorem.



The universal NAND-RAM program simulates an input NAND-RAM program by storing all of ’s variables inside a single array Vars of . If has variables, then the array Vars is divided into blocks of length , where the -th coordinate of the -th block contains the -th element of the -th array of . If the -th variable of is scalar, then we just store its value in the zeroth block of Vars.

To present a universal NAND-RAM program in full we would need to describe a precise representation scheme, as well as the full NAND-RAM instructions for the program. While this can be done, it is more important to focus on the main ideas, and so we just sketch the proof here. A specification of NAND-RAM is given in the [appendix](http://tiny.cc/introtcsappendix), and for the purposes of this simulation, we can simply use the representation of the code NAND-RAM as an ASCII string.

The program gets as input a NAND-RAM program and an input and simulates one step at a time. To do so, does the following:

1. maintains variables program\_counter, and number\_steps for the current line to be executed and the number of steps executed so far.
2. initially scans the code of to find the number of unique variable names that uses. It will translate each variable name into a number between and and use an array Program to store ’s code where for every line , Program[] will store the -th line of where the variable names have been translated to numbers. (More concretely, we will use a constant number of arrays to separately encode the operation used in this line, and the variable names and indices of the operands.)
3. maintains a single array Vars that contains all the values of ’s variables. We divide Vars into blocks of length . If is a number corresponding to an array variable Foo of , then we store Foo[0] in Vars[], we store Foo[1] in Var\_values[], Foo[2] in Vars[] and so on and so forth (see universalrammachinefig). Generally,if the -th variable of is a scalar variable, then its value will be stored in location Vars[]. If it is an array variable then the value of its -th element will be stored in location Vars[].
4. To simulate a single step of , the program recovers from Program the line corresponding to program\_counter and executes it. Since NAND-RAM has a constant number of arithmetic operations, we can implement the logic of which operation to execute using a sequence of a constant number of if-then-else’s. Retrieving from Vars the values of the operands of each instruction can be done using a constant number of arithmetic operations.

The setup stages take only a constant (depending on but not on the input ) number of steps. Once we are done with the setup, to simulate a single step of , we just need to retrieve the corresponding line and do a constant number of “if elses” and accesses to Vars to simulate it. Hence the total running time to simulate steps of the program is at most when suppressing constants that depend on the program .

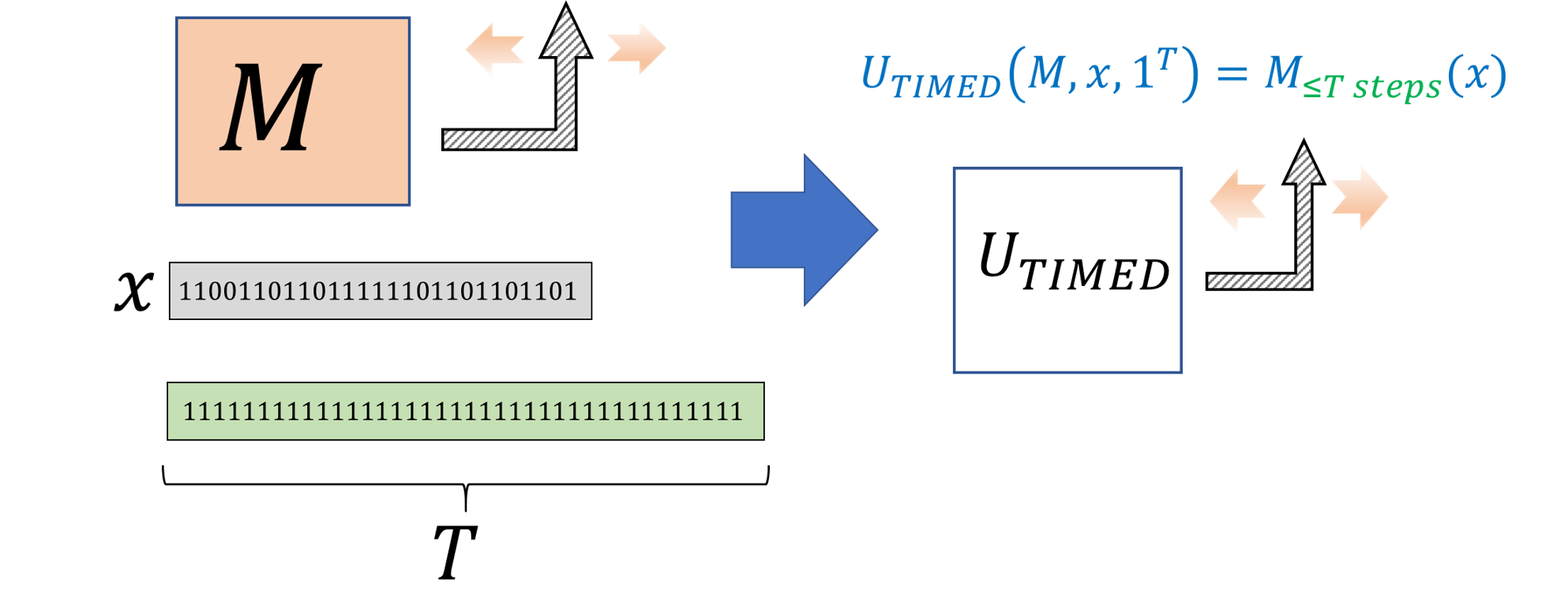
### Timed Universal Turing Machine

One corollary of the efficient universal machine is the following. Given any Turing Machine , input , and “step budget” , we can simulate the execution of for steps in time that is polynomial in . Formally, we define a function that takes the three parameters , , and the time budget, and outputs if halts within at most steps, and outputs otherwise. The timed universal Turing Machine computes in polynomial time (see timeduniversaltmfig). (Since we measure time as a function of the input length, we define as taking the input represented in *unary*: a string of ones.)

### 

Let be the function defined as

Then .



The *timed* universal Turing Machine takes as input a Turing machine , an input , and a time bound , and outputs if halts within at most steps. timeduniversalTMstates that there is such a machine that runs in time polynomial in .

We only sketch the proof since the result follows fairly directly from polyRAMTM-thm and univ-nandpp. By polyRAMTM-thm to show that , it suffices to give a polynomial-time NAND-RAM program to compute .

Such a program can be obtained as follows. Given a Turing Machine , by polyRAMTM-thm we can transform it in time polynomial in its description into a functionally-equivalent NAND-RAM program such that the execution of on steps can be simulated by the execution of on steps. We can then run the universal NAND-RAM machine of univ-nandpp to simulate for steps, using time, and output if the execution did not halt within this budget. This shows that can be computed by a NAND-RAM program in time polynomial in and linear in , which means .

## The time hierarchy theorem

Some functions are *uncomputable*, but are there functions that can be computed, but only at an exorbitant cost? For example, is there a function that *can* be computed in time , but *can not* be computed in time ? It turns out that the answer is **Yes**:

### 

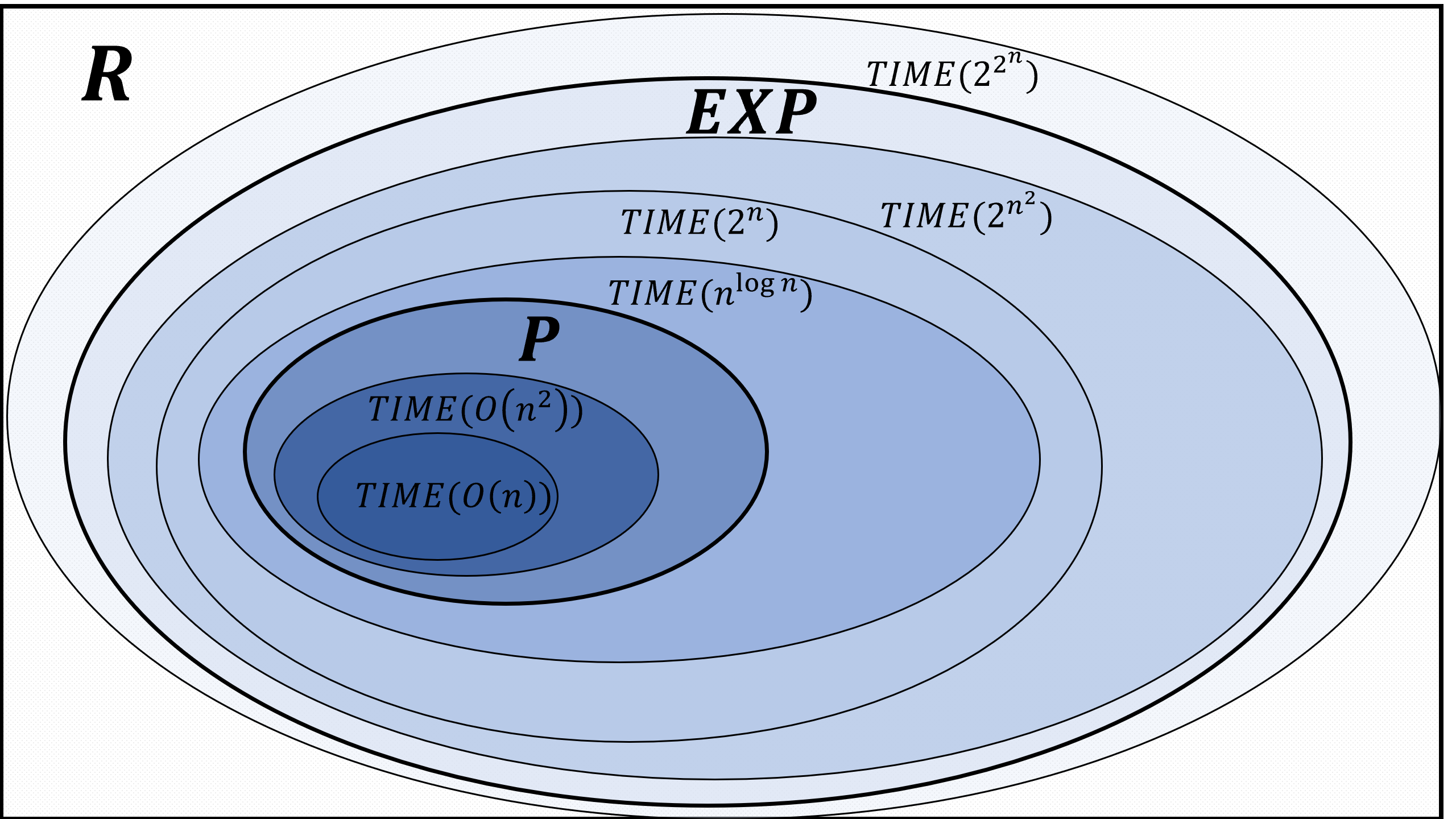
For every nice function , there is a function in .

There is nothing special about , and we could have used any other efficiently computable function that tends to infinity with .

If we have more time, we can compute more functions.

The generality of the time hierarchy theorem can make its proof a little hard to read. It might be easier to follow the proof if you first try to prove by yourself the easier statement .

You can do so by showing that the following function is in : for every Turing Machine and input , if and only if halts on within at most steps. One can show that using the universal Turing machine (or the efficient universal NAND-RAM program of univ-nandpp). On the other harnd, we can use similar ideas to those used to show the uncomputability of in haltalternativesec to prove that .



The *Time Hierarchy Theorem* (time-hierarchy-thm) states that all of these classes are *distinct*.

### 

In the proof of halt-thm (the uncomputability of the Halting problem), we have shown that the function cannot be computed in any finite time. An examination of the proof shows that it gives something stronger. Namely, the proof shows that if we fix our computational budget to be steps, then not only we can’t distinguish between programs that halt and those that do not, but cannot even distinguish between programs that halt within at most steps and those that take more than that (where is some number depending on ). Therefore, the proof of time-hierarchy-thm follows the ideas of the uncomputability of the halting problem, but again with a more careful accounting of the running time.

Our proof is inspired by the proof of the uncomputability of the halting problem. Specifically, for every function as in the theorem’s statement, we define the *Bounded Halting* function as follows. The input to is a pair such that encodes some NAND-RAM program. We define

(The constant and the function are rather arbitrary, and are chosen for convenience in this proof.)

time-hierarchy-thm is an immediate consequence of the following two claims:

**Claim 1:**

and

**Claim 2:** .

Please make sure you understand why indeed the theorem follows directly from the combination of these two claims. We now turn to proving them.

**Proof of claim 1:** We can easily check in linear time whether an input has the form where . Since is a nice function, we can evaluate it in time. Thus, we can compute as follows:

1. Compute in steps.
2. Use the universal NAND-RAM program of univ-nandpp to simulate steps of on the input using at most steps. (Recall that we use to denote a quantity that is bounded by for some constants .)
3. If halts within these steps then output , else output .

The length of the input is . Since and for every , the running time will be and hence the above algorithm demonstrates that , completing the proof of Claim 1.

**Proof of claim 2:** This proof is the heart of time-hierarchy-thm, and is very reminiscent of the proof that is not computable. Assume, for the sake of contradiction, that there is some NAND-RAM program that computes within steps. We are going to show a contradiction by creating a program and showing that under our assumptions, if runs for less than steps when given (a padded version of) its own code as input then it actually runs for more than steps and vice versa. (It is worth re-reading the last sentence twice or thrice to make sure you understand this logic. It is very similar to the direct proof of the uncomputability of the halting problem where we obtained a contradiction by using an assumed “halting solver” to construct a program that, given its own code as input, halts if and only if it does not halt.)

We will define to be the program that on input a string does the following:

1. If does not have the form where represents a NAND-RAM program and then return . (Recall that denotes the string of ones.)
2. Compute (at a cost of at most steps, under our assumptions).
3. If then goes into an infinite loop, otherwise it halts.

Let be the length description of as a string, and let be larger than . We will reach a contradiction by splitting into cases according to whether or not equals or .

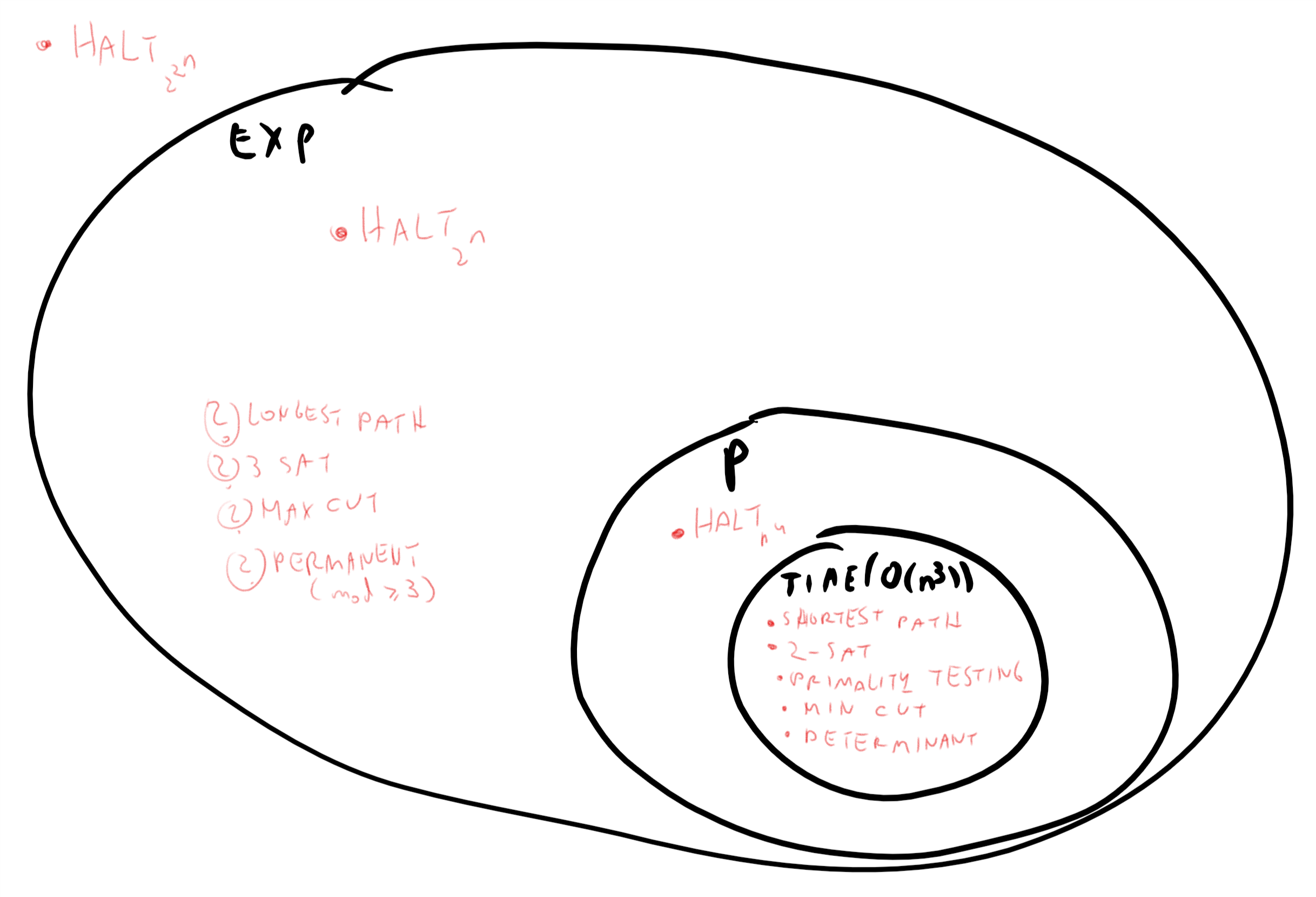
On the one hand, if , then under our assumption that computes , will go into an infinite loop on input , and hence in particular does *not* halt within steps on the input . But this contradicts our assumption that .

This means that it must hold that . But in this case, since we assume computes , does not do anything in phase 3 of its computation, and so the only computation costs come in phases 1 and 2 of the computation. It is not hard to verify that Phase 1 can be done in linear and in fact less than steps. Phase 2 involves executing , which under our assumption requires steps. In total we can perform both phases in less than in steps, which by definition means that , but this is of course a contradiction. This completes the proof of Claim 2 and hence of time-hierarchy-thm.

Prove that .

We show why this statement follows from the time hierarchy theorem, but it can be an instructive exercise to prove it directly, see hierarchytoyrem. We need to show that there exists . Let and . Both are nice functions. Since , by time-hierarchy-thm there exists some in . Since for sufficiently large , , . On the other hand, . Indeed, suppose otherwise that there was a constant and a Turing Machine computing on -length input in at most steps for all sufficiently large . Then since for large enough , it would have followed that contradicting our choice of .

The time hierarchy theorem tells us that there are functions we can compute in time but not , in time, but not , etc.. In particular there are most definitely functions that we can compute in time but not . We have seen that we have no shortage of natural functions for which the best *known* algorithm requires roughly time, and that many people have invested significant effort in trying to improve that. However, unlike in the finite vs. infinite case, for all of the examples above at the moment we do not know how to rule out even an time algorithm. We will however see that there is a single unproven conjecture that would imply such a result for most of these problems.



Some complexity classes and some of the functions we know (or conjecture) to be contained in them.

The time hierarchy theorem relies on the existence of an efficient universal NAND-RAM program, as proven in univ-nandpp. For other models such as Turing Machines we have similar time hierarchy results showing that there are functions computable in time and not in time where corresponds to the overhead in the corresponding universal machine.

## Non uniform computation

We have now seen two measures of “computation cost” for functions. In secdefinesizeclasses we defined the complexity of computing *finite* functions using circuits / straightline programs. Specifically, for a finite function and number , if there is a circuit of at most NAND gates (or equivalently a -line NAND-CIRC program) that computes . To relate this to the classes defined in this chapter we first need to extend the class from finite functions to functions with unbounded input length.

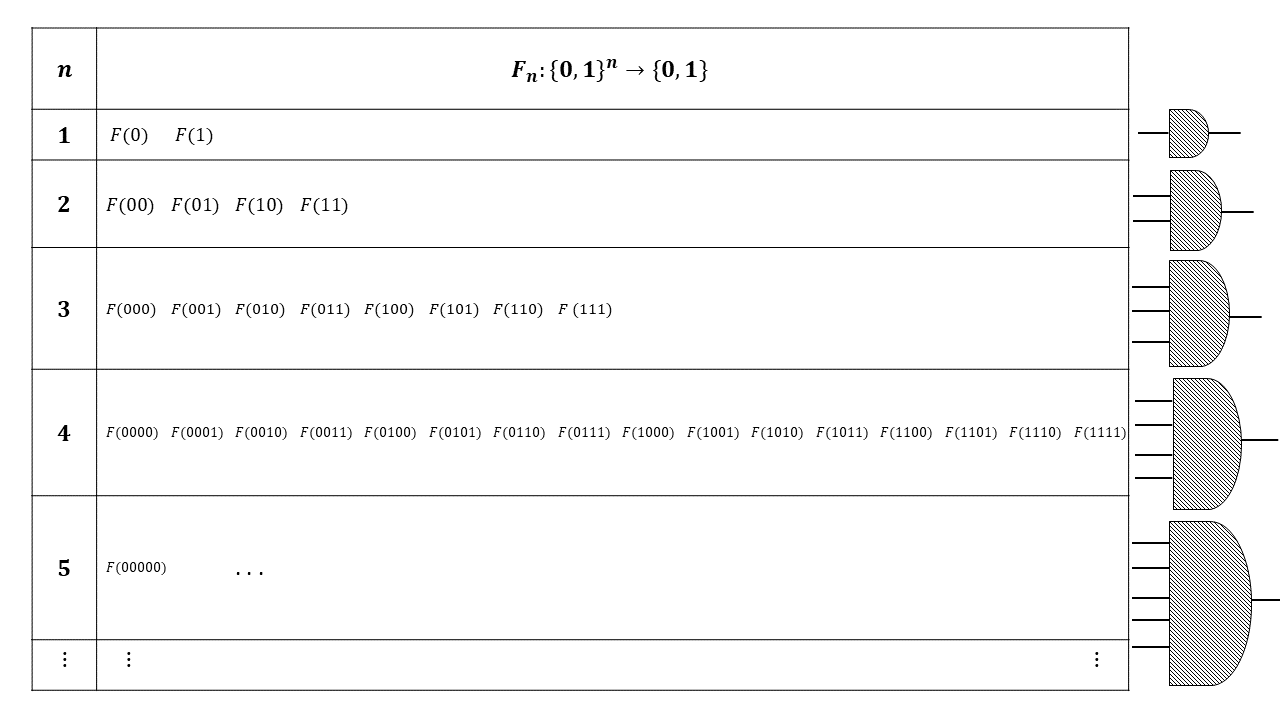
Let and be a nice time bound. For every , define to be the *restriction* of to inputs of size . That is, is the function mapping to such that for every , .

We say that is *non-uniformly computable in at most size*, denoted by if there exists a sequence of NAND circuits such that:

* For every , computes the function
* For every sufficiently large , has at most gates.

The non uniform analog to the class is the class defined as

There is a big difference between non uniform computation and uniform complexity classes such as or . The condition means that there is a *single* Turing machine that computes on all inputs in polynomial time. The condition only means that for every input length there can be a *different* circuit that computes using polynomially many gates on inputs of these lengths. As we will see, does not necessarily imply that . However, the other direction is true:



We can think of an infinite function as a collection of finite functions where is the restriction of to inputs of length . We say is in if for every , the function is computable by a polynomial size NAND-CIRC program, or equivalently, a polynomial sized Boolean circuit.

There is some s.t. for every nice and ,

In particular, non-uniform-thm shows that for every , and hence .

The idea behind the proof is to “unroll the loop”. Specifically, we will use the programming language variants of non-uniform and uniform computation: namely NAND-CIRC and NAND-TM. The main difference between the two is that NAND-TM has *loops*. However, for every fixed , if we know that a NAND-TM program runs in at most steps, then we can replace its loop by simply “copying and pasting” its code times, similar to how in Python we can replace code such as

for i in range(4):  
 print(i)

with the “loop free” code

print(0)  
print(1)  
print(2)  
print(3)

To make this idea into an actual proof we need to tackle one technical difficulty, and this is to ensure that the NAND-TM program is *oblivious* in the sense that the value of the index variable i in the -th iteration of the loop will depend only on and not on the contents of the input. We make a digression to do just that in obliviousnandtm and then complete the proof of non-uniform-thm.

### Oblivious NAND-TM programs

Our approach for proving non-uniform-thm involves “unrolling the loop”. For example, consider the following NAND-TM to compute the function on inputs of arbitrary length:

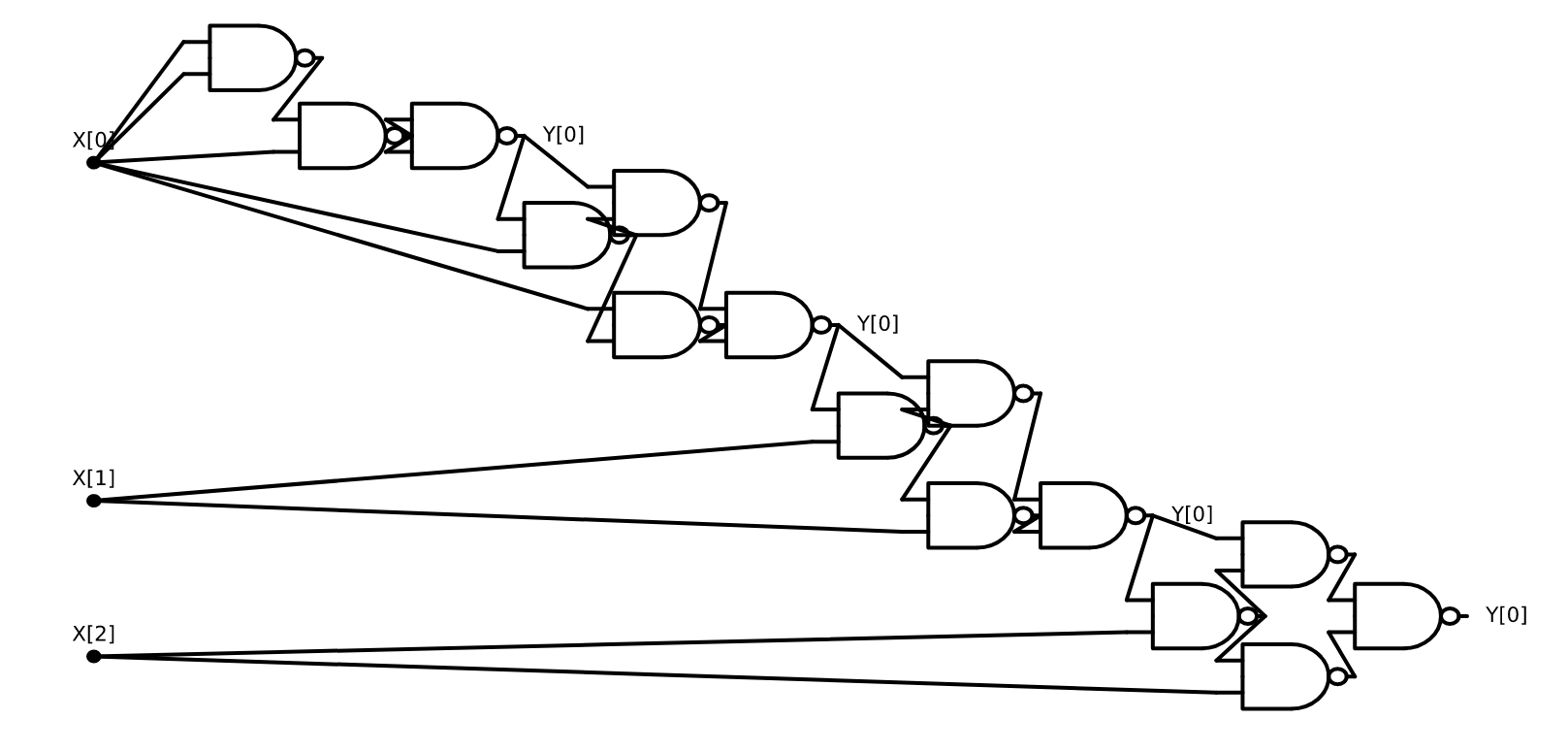
temp\_0 = NAND(X[0],X[0])  
Y\_nonblank[0] = NAND(X[0],temp\_0)  
temp\_2 = NAND(X[i],Y[0])  
temp\_3 = NAND(X[i],temp\_2)  
temp\_4 = NAND(Y[0],temp\_2)  
Y[0] = NAND(temp\_3,temp\_4)  
MODANDJUMP(X\_nonblank[i],X\_nonblank[i])

Setting (as an example) , we can attempt to translate this NAND-TM program into a NAND-CIRC program for computing by simply “copying and pasting” the loop three times (dropping the MODANDJMP line):

temp\_0 = NAND(X[0],X[0])  
Y\_nonblank[0] = NAND(X[0],temp\_0)  
temp\_2 = NAND(X[i],Y[0])  
temp\_3 = NAND(X[i],temp\_2)  
temp\_4 = NAND(Y[0],temp\_2)  
Y[0] = NAND(temp\_3,temp\_4)  
temp\_0 = NAND(X[0],X[0])  
Y\_nonblank[0] = NAND(X[0],temp\_0)  
temp\_2 = NAND(X[i],Y[0])  
temp\_3 = NAND(X[i],temp\_2)  
temp\_4 = NAND(Y[0],temp\_2)  
Y[0] = NAND(temp\_3,temp\_4)  
temp\_0 = NAND(X[0],X[0])  
Y\_nonblank[0] = NAND(X[0],temp\_0)  
temp\_2 = NAND(X[i],Y[0])  
temp\_3 = NAND(X[i],temp\_2)  
temp\_4 = NAND(Y[0],temp\_2)  
Y[0] = NAND(temp\_3,temp\_4)

However, the above is still not a valid NAND-CIRC program since it contains references to the special variable i. To make it into a valid NAND-CIRC program, we replace references to i in the first iteration with , references in the second iteration with , and references in the third iteration with . (We also create a variable zero and use it for the first time any variable is instantiated, as well as remove assignments to non-output variables that are never used later on.) The resulting program is a standard “loop free and index free” NAND-CIRC program that computes (see also unrolledcircuitfig):

temp\_0 = NAND(X[0],X[0])  
one = NAND(X[0],temp\_0)  
zero = NAND(one,one)  
temp\_2 = NAND(X[0],zero)  
temp\_3 = NAND(X[0],temp\_2)  
temp\_4 = NAND(zero,temp\_2)  
Y[0] = NAND(temp\_3,temp\_4)  
temp\_2 = NAND(X[1],Y[0])  
temp\_3 = NAND(X[1],temp\_2)  
temp\_4 = NAND(Y[0],temp\_2)  
Y[0] = NAND(temp\_3,temp\_4)  
temp\_2 = NAND(X[2],Y[0])  
temp\_3 = NAND(X[2],temp\_2)  
temp\_4 = NAND(Y[0],temp\_2)  
Y[0] = NAND(temp\_3,temp\_4)



A NAND circuit for obtained by “unrolling the loop” of the NAND-TM program for computing three times.

Key to this transformation was the fact that in our original NAND-TM program for , regardless of whether the input is , , or any other string, the index variable i is guaranteed to equal in the first iteration, in the second iteration, in the third iteration, and so on and so forth. The particular sequence is immaterial: the crucial property is that the NAND-TM program for is *oblivious* in the sense that the value of the index i in the -th iteration depends only on and does not depend on the particular choice of the input. Luckily, it is possible to transform every NAND-TM program into a functionally equivalent oblivious program with at most quadratic . (Similarly we can transform any Turing machine into a functionally equivalent oblivious Turing machine, see oblivious-ex.)

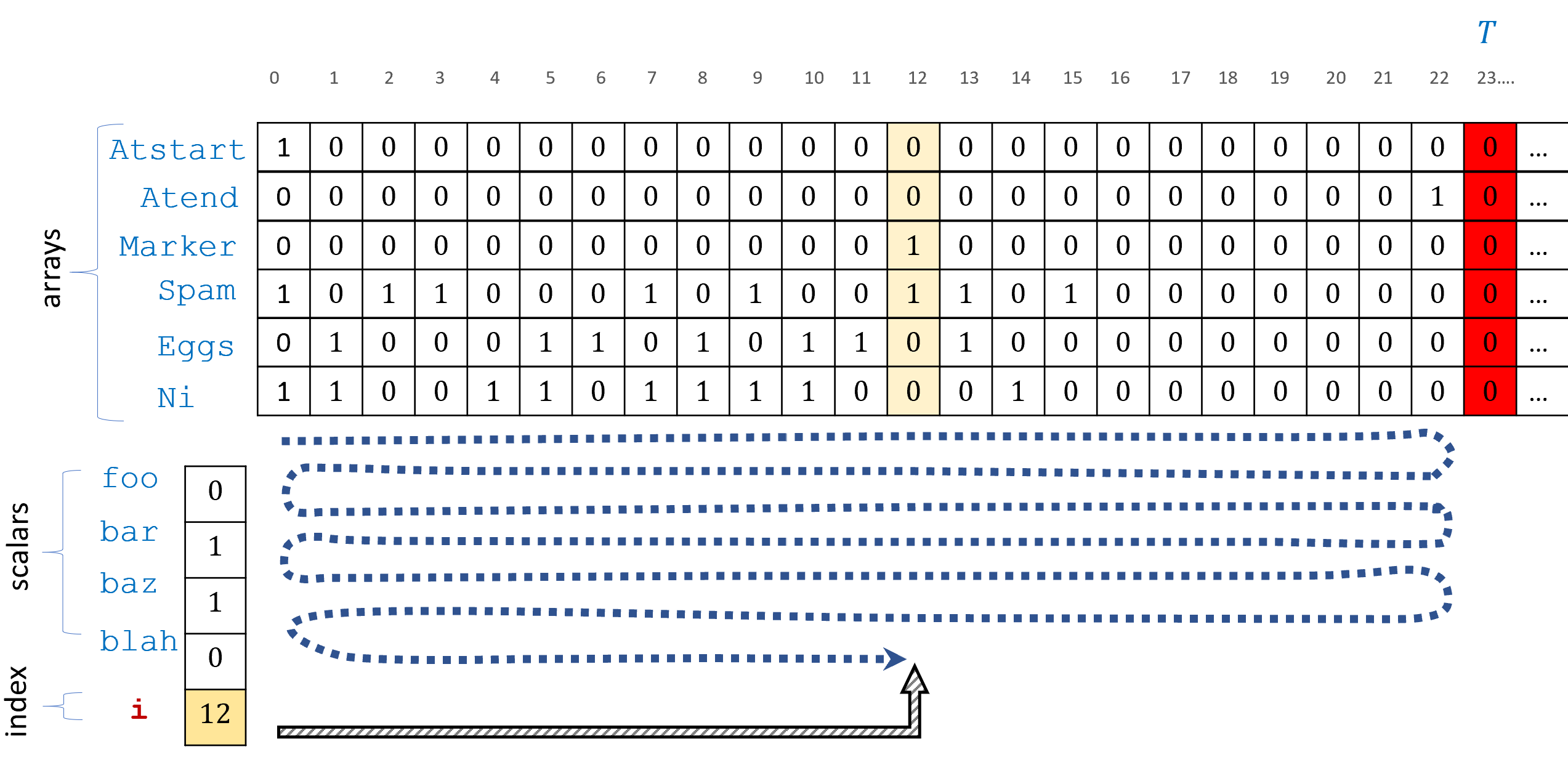
### 

Let be a nice function and let . Then there is a NAND-TM program that computes in steps and satisfying the following. For every there is a sequence such that for every , if is executed on input then in the -th iteration the variable i is equal to .

In other words, obliviousnandtmthm implies that if we can compute in steps, then we can compute it in steps with a program in which the position of i in the -th iteration depends only on and the length of the input, and not on the contents of the input. Such a program can be easily translated into a NAND-CIRC program of lines by “unrolling the loop”.

### 

We can translate any NAND-TM program into an oblivious program by making “sweep” its arrays. That is, the index i in will always move all the way from position to position and back again. We can then simulate the program with at most overhead: if wants to move i left when we are in a rightward sweep then we simply wait the at most steps until the next time we are back in the same position while sweeping to the left.



We simulate a -time NAND-TM program with an *oblivious* NAND-TM program by adding special arrays Atstart and Atend to mark positions and respectively. The program will simply “sweep” its arrays from right to left and back again. If the original program would have moved i in a different direction then we wait steps until we reach the same point back again, and so runs in time.

Let be a NAND-TM program computing in steps. We construct an oblivious NAND-TM program for computing as follows (see also obliviousnandtmfig).

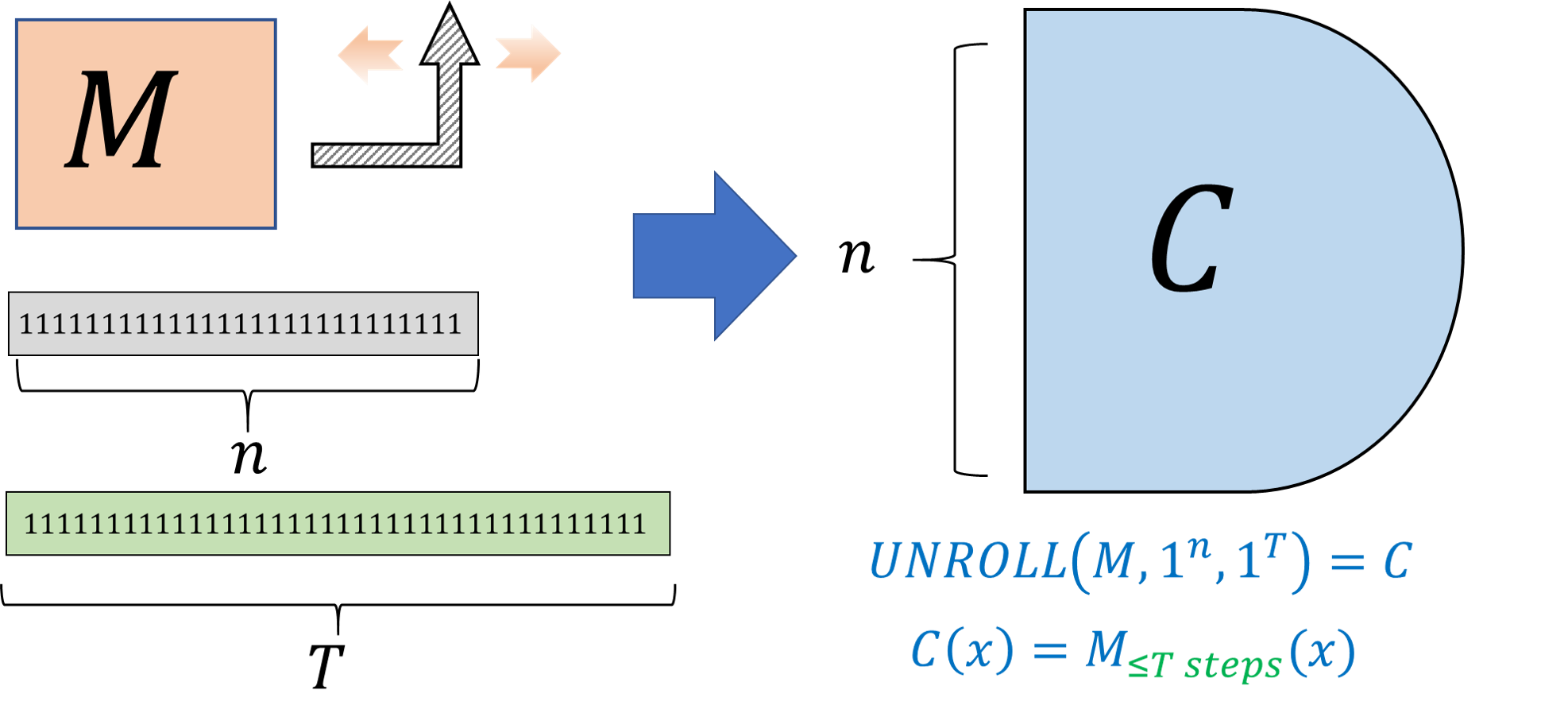
1. On input , will compute and set up arrays Atstart and Atend satisfying Atstart[] and Atstart[] for and Atend[] and Atend[i] for all . We can do this because is a nice function. Note that since this computation does not depend on but only on its length, it is oblivious.
2. will also have a special array Marker initialized to all zeroes.
3. The index variable of will change direction of movement to the right whenever Atstart[i] and to the left whenever Atend[i].
4. The program simulates the execution of . However, if the MODANDJMP instruction in attempts to move to the right when is moving left (or vice versa) then will set Marker[i] to and enter into a special “waiting mode”. In this mode will wait until the next time in which Marker[i] (at the next sweep) at which points zeroes Marker[i] and continues with the simulation. In the worst case this will take steps (if has to go all the way from one end to the other and back again.)
5. We also modify to ensure it ends the computation after simulating exactly steps of , adding “dummy steps” if ends early.

We see that simulates the execution of with an overhead of steps of per one step of , hence completing the proof.

obliviousnandtmthm implies non-uniform-thm. Indeed, if is a -line oblivious NAND-TM program computing in time then for every we can obtain a NAND-CIRC program of lines by simply making copies of (dropping the final MODANDJMP line). In the -th copy we replace all references of the form Foo[i] to foo\_ where is the value of i in the -th iteration.

### “Unrolling the loop”: algorithmic transformation of Turing Machines to circuits

The proof of non-uniform-thm is *algorithmic*, in the sense that the proof yields a polynomial-time algorithm that given a Turing Machine and parameters and , produces a circuit of gates that agrees with on all inputs (as long as runs for less than steps these inputs.) We record this fact in the following theorem, since it will be useful for us later on:

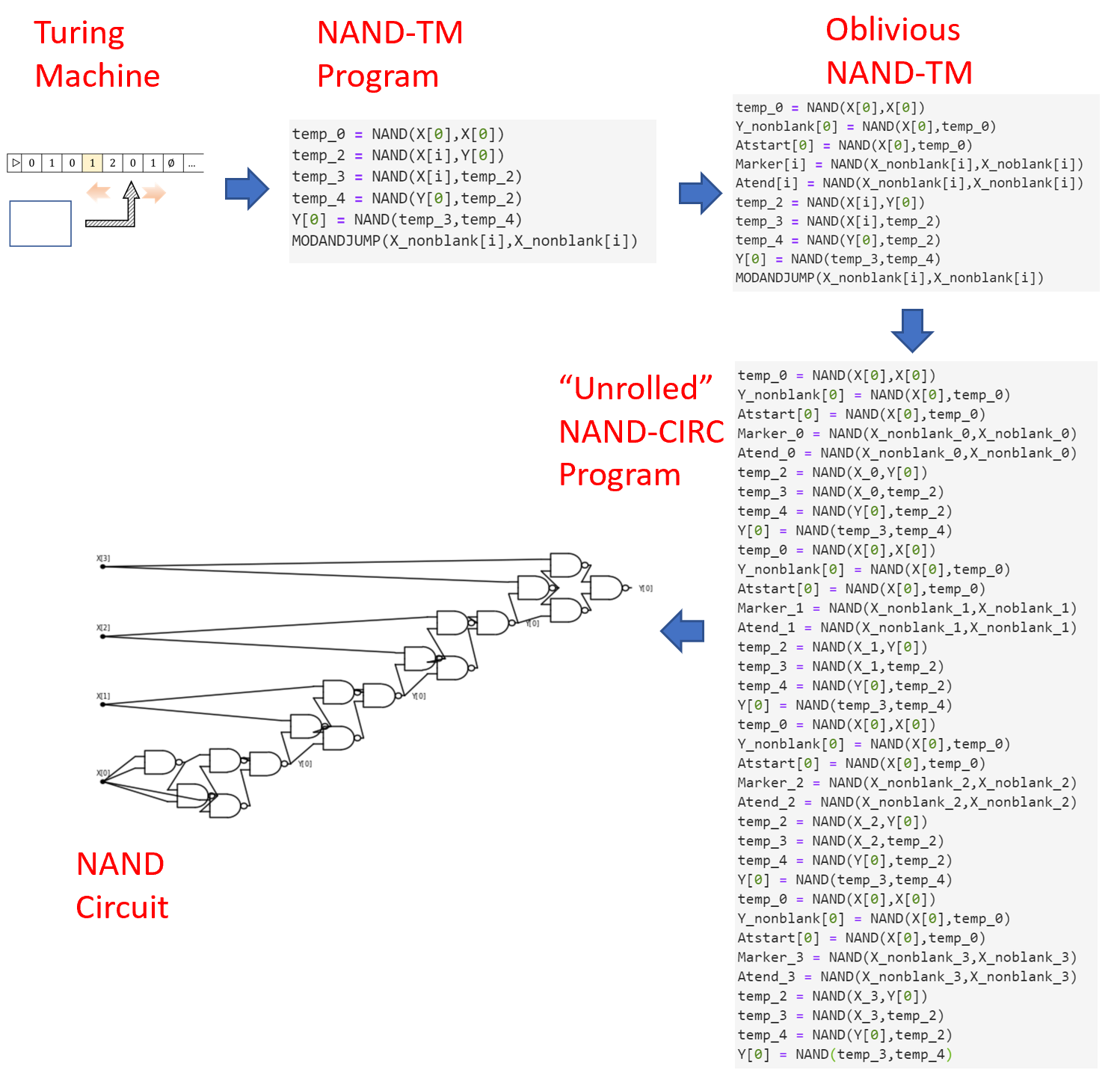


The function takes as input a Turing Machine , an input length parameter , a step budget parameter , and outputs a circuit of size that takes bits of inputs and outputs if halts on within at most steps.

There is algorithm such that for every Turing Machine and numbers , runs for steps and outputs a NAND circuit with inputs, gates, and one output, such that

We only sketch the proof since it follows by directly translating the proof of non-uniform-thm{.ref into an algorithm together with the simulation of Turing machines by NAND-TM programs (see also unrolldescriptionfig). Specifically, does the following:

1. Transform the Turing Machine into an equivalent NAND-TM program .
2. Transform the NAND-TM program into an equivalent oblivious program following the proof of obliviousnandtmthm. The program takes steps to simulate steps of .
3. “Unroll the loop” of by obtaining a NAND-CIRC program of lines (or equivalently a NAND circuit with gates) corresponding to the execution of iterations of .



We can transform a Turing Machine , input length parameter , and time bound into an sized NAND circuit that agrees with on all inputs on which halts in at most steps. The transformation is obtained by first using the equivalence of Turing Machines and NAND-TM programs , then turning into an equivalent *oblivious* NAND-TM program via obliviousnandtmthm, then “unrolling” iterations of the loop of to obtain an line NAND-CIRC program that agrees with on length inputs, and finally translating this program into an equivalent circuit.

By “unrolling the loop” we can transform an algorithm that takes steps to compute into a circuit that uses gates to compute the restriction of to .

Reviewing the transformations described in unrolldescriptionfig, as well as solving the following two exercises is a great way to get more comfort with non-uniform complexity and in particular with and its relation to .

Prove that for every , if and only if there is a polynomial-time Turing Machine such that for every , outputs a description of an input circuit that computes the restriction of to inputs in .

We start with the “if” direction. Suppose that there is a polynomial-time Turing Machine that on input outputs a circuit that computes . Then the following is a polynomial-time Turing Machine to compute . On input , will:

1. Let and compute .
2. Return the evaluation of on .

Since we can evaluate a Boolean circuit on an input in polynomial time, runs in polynomial time and computes on every input .

For the “only if” direction, if is a Turing Machine that computes in polynomial-time, then (applying the equivalence of Turing Machines and NAND-TM as well as obliviousnandtmthm) there is also an oblivious NAND-TM program that computes in time for some polynomial . We can now define to be the Turing Machine that on input outputs the NAND circuit obtained by “unrolling the loop” of for iterations. The resulting NAND circuit computes and has gates. It can also be transformed to a Boolean circuit with AND/OR/NOT gates.

Let . Then if and only if there exists a polynomial , a polynomial-time Turing Machine and a sequence of strings, such that for every :

* For every , .

We only sketch the proof. For the “only if” direction, if then we can use for simply the description of the corresponding circuit and for the program that computes in polynomial time the evaluation of a circuit on its input.

For the “if” direction, we can use the same “unrolling the loop” technique of non-uniform-thm to show that if is a polynomial-time NAND-TM program, then for every , the map can be computed by a polynomial size NAND-CIRC program .

### Can uniform algorithms simulate non uniform ones?

non-uniform-thm shows that every function in is in . One can ask if there is an inverse relation. Suppose that is such that has a “short” NAND-CIRC program for every . Can we say that it must be in for some “small” ? The answer is an emphatic **no**. Not only is not contained in , in fact contains functions that are *uncomputable*!

### 

There exists an *uncomputable* function such that .

### 

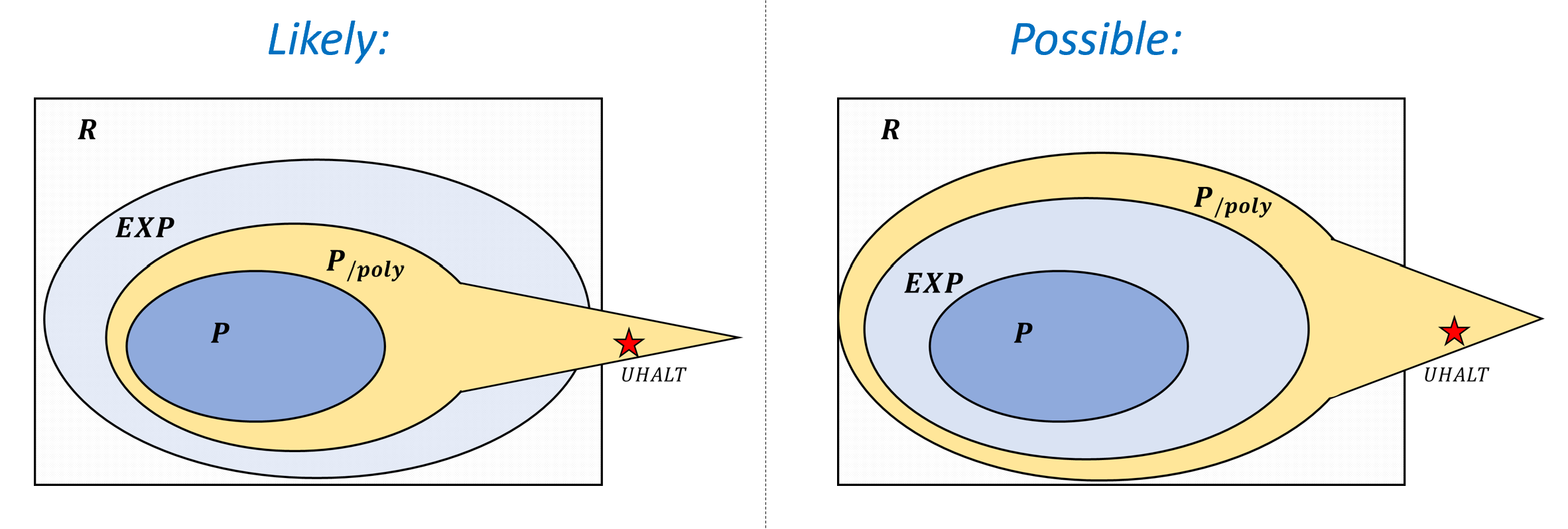
Since corresponds to non uniform computation, a function is in if for every , the restriction to inputs of length has a small circuit/program, even if the circuits for different values of are completely different from one another. In particular, if has the property that for every equal-length inputs and , then this means that is either the constant function zero or the constant function one for every . Since the constant function has a (very!) small circuit, such a function will always be in (indeed even in smaller classes). Yet by a reduction from the Halting problem, we can obtain a function with this property that is uncomputable.

Consider the following “unary halting function” defined as follows. We let be the function that on input , outputs the string that corresponds to the binary representation of the number without the most significant digit. Note that is *onto*. For every , we define . That is, if is the length of , then if and only if the string encodes a NAND-TM program that halts on the input .

is uncomputable, since otherwise we could compute by transforming the input program into the integer such that and then running (i.e., on the string of ones). On the other hand, for every , is either equal to for all inputs or equal to on all inputs , and hence can be computed by a NAND-CIRC program of a *constant* number of lines.

The issue here is of course *uniformity*. For a function , if is in then we have a *single* algorithm that can compute for every . On the other hand, might be in for every using a completely different algorithm for every input length. For this reason we typically use not as a model of *efficient* computation but rather as a way to model *inefficient computation*. For example, in cryptography people often define an encryption scheme to be secure if breaking it for a key of length requires more than a polynomial number of NAND lines. Since , this in particular precludes a polynomial time algorithm for doing so, but there are technical reasons why working in a non uniform model makes more sense in cryptography. It also allows to talk about security in non asymptotic terms such as a scheme having “ bits of security”.

While it can sometimes be a real issue, in many natural settings the difference between uniform and non-uniform computation does not seem to so important. In particular, in all the examples of problems not known to be in we discussed before: longest path, 3SAT, factoring, etc., these problems are also not known to be in either. Thus, for “natural” functions, if you pretend that is roughly the same as , you will be right more often than wrong.



Relations between , , and . It is known that , and that contains uncomputable functions (which in particular are outside of ). It is not known whether or not though it is believed that .

### Uniform vs. Nonuniform computation: A recap

To summarize, the two models of computation we have described so far are:

* **Uniform models:** *Turing machines*, *NAND-TM programs*, *RAM machines*, *NAND-RAM programs*, *C/JavaScript/Python*, etc. These model include loops and unbounded memory hence a single program can compute a function with unbounded input length.
* **Non-uniform models:** *Boolean Circuits* or *straightline programs* have no loops and can only compute finite functions. The time to execute them is exactly the number of lines or gates they contain.

For a function and some nice time bound , we know that:

* If is uniformly computable in time then there is a sequence of circuits where has gates and computes (i.e., restriction of to ) for every .
* The reverse direction is not necessarily true - there are examples of functions such that can be computed by even a constant size circuit but is uncomputable.

This means that non uniform complexity is more useful to establish *hardness* of a function than its *easiness*.

* We can define the time complexity of a function using NAND-TM programs, and similarly to the notion of computability, this appears to capture the inherent complexity of the function.
* There are many natural problems that have polynomial-time algorithms, and other natural problems that we’d love to solve, but for which the best known algorithms are exponential.
* The definition of polynomial time, and hence the class , is robust to the choice of model, whether it is Turing machines, NAND-TM, NAND-RAM, modern programming languages, and many other models.
* The time hierarchy theorem shows that there are *some* problems that can be solved in exponential, but not in polynomial time. However, we do not know if that is the case for the natural examples that we described in this lecture.
* By “unrolling the loop” we can show that every function computable in time can be computed by a sequence of NAND-CIRC programs (one for every input length) each of size at most

## Exercises

Prove that the classes and defined in PandEXPdef are equal to and respectively. (If is a collection of sets then the set is the set of all elements such that there exists some where .)

polyRAMTM-thm shows that the classes and are *robust* with respect to variations in the choice of the computational model. This exercise shows that these classes are also robust with respect to our choice of the representation of the input.

Specifically, let be a function mapping graphs to , and let be the functions defined as follows. For every :

* iff represents a graph via the adjacency matrix representation such that .
* iff represents a graph via the adjacency list representation such that .

Prove that iff .

More generally, for every function , the answer to the question of whether (or whether ) is unchanged by switching representations, as long as transforming one representation to the other can be done in polynomial time (which essentially holds for all reasonable representations).

For every function , define to be the function mapping to such that on input a (string representation of a) triple with , and ,

where is the -th bit of the string .

Prove that if and only if .

### 

Prove that if are in then their *composition* , which is the function s.t. , is also in .

### 

Prove that there is some s.t. but is not in .

We say that a Turing machine is *oblivious* if there is some function such that for every input of length , and it holds that:

* If takes more than steps to halt on the input , then in the -th step ’s head will be in the position . (Note that this position depends only on the *length* of and not its contents.)
* If halts before the -th step then .

Prove that if then there exists an *oblivious* Turing machine that computes in polynomial time. See footnote for hint.[[1]](#footnote-99)

Let be the function such that on input a string representing a triple , where is the adjacency list representation of an vertex graph , and and are numbers in , if the edge is present in the graph. outputs on all other inputs.

1. Prove that .
2. Let be the function that on input an adjacency matrix outputs if and only if the graph represented by is *planar* (that is, can be drawn on the plane without edges crossing one another). For this question, you can use without proof the fact that . Prove that where is the function that on input an adjacency list outputs if and only if represents a planar graph.

Let be the function such that for every string representing a pair where is an -input -output NAND-CIRC (not NAND-TM!) program and , . On all other inputs outputs .

Prove that .

Let be the function such that on input a string representing a pair where

* for some

if there is no NAND-CIRC program of at most lines that computes the function whose truth table is the string . That is, if for every NAND-CIRC program of at most lines, there exists some such that where denote the -the coordinate of , using the binary representation to identify with the numbers .

1. Prove that .
2. (Challenge) Prove that there is an algorithm such that if is sufficiently large, then runs in time and outputs a string that is the truth table of a function that is not contained in . (In other words, if is the string output by then if we let be the function such that outputs the -th coordinate of , then .[[2]](#footnote-103)

Suppose that you are in charge of scheduling courses in computer science in University X. In University X, computer science students wake up late, and have to work on their startups in the afternoon, and take long weekends with their investors. So you only have two possible slots: you can schedule a course either Monday-Wednesday 11am-1pm or Tuesday-Thursday 11am-1pm.

Let be the function that takes as input a list of courses and a list of *conflicts* (i.e., list of pairs of courses that cannot share the same time slot) and outputs if and only if there is a “conflict free” scheduling of the courses in , where no pair in is scheduled in the same time slot.

More precisely, the list is a list of strings and the list is a list of pairs of the form . if and only if there exists a partition of into two parts so that there is no pair such that both and are in the same part.

Prove that . As usual, you do not have to provide the full code to show that this is the case, and can describe operations as a high level, as well as appeal to any data structures or other results mentioned in the book or in lecture. Note that to show that a function is in you need to both **(1)** present an algorithm that computes in polynomial time, **(2)** *prove* that does indeed run in polynomial time, and does indeed compute the correct answer.

Try to think whether or not your algorithm extends to the case where there are *three* possible time slots.

## Bibliographical notes

Because we are interested in the *maximum* number of steps for inputs of a given length, running-time as we defined it is often known as *worst case complexity*. The *minimum* number of steps (or “best case” complexity) to compute a function on length inputs is typically not a meaningful quantity since essentially every natural problem will have some trivially easy instances. However, the *average case complexity* (i.e., complexity on a “typical” or “random” input) is an interesting concept which we’ll return to when we discuss *cryptography*. That said, worst-case complexity is the most standard and basic of the complexity measures, and will be our focus in most of this book.

Some lower bounds for single-tape Turing machines are given in [@maass1985combinatorial].

For defining efficiency in the calculus, one needs to be careful about the order of application of the reduction steps, which can matter for computational efficiency, see for example [this paper](https://lmcs.episciences.org/1627).

The notation is used for historical reasons. It was introduced by Karp and Lipton, who considered this class as corresponding to functions that can be computed by polynomial-time Turing Machines that are given for any input length an *advice string* of length polynomial in .

1. *Hint:* This is the Turing Machine analog of obliviousnandtmthm. We replace one step of the original TM computing with a “sweep” of the obliviouss TM in which it goes steps to the right and then steps to the left. [↑](#footnote-ref-99)
2. **Hint:** Use Item 1, the existence of functions requiring exponentially hard NAND programs, and the fact that there are only finitely many functions mapping to . [↑](#footnote-ref-103)