Learning Objectives:

- Formally modeling running time, and in particular notions such as $O(n)$ or $O(n^3)$ time algorithms.
- The classes **P** and **EXP** modelling polynomial and exponential time respectively.
- The *time hierarchy theorem*, that in particular says that for every $k \geq 1$ there are functions we *can* compute in $O(n^{k+1})$ time but *can not* compute in $O(n^k)$ time.

14

Modeling running time

> "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

> "The computational complexity of a sequence is to be measured by how fast a multitape Turing machine can print out the terms of the sequence. This particular abstract model of a computing device is chosen because much of the work in this area is stimulated by the rapidly growing importance of computation through the use of digital computers, and all digital computers in a slightly idealized form belong to the class of multitape Turing machines.", Juris Hartmanis and Richard Stearns, "On the computational complexity of algorithms", 1963.

In the last lecture 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 lecture, using the NAND++ and NAND« models we have seen before. Since we think of programs that can take as input a string of arbitrary length, their running time is not a fixed number but rather what we are interested in is measuring the *dependence* of the number of steps the program takes on the length of the input. That is, for any program P, we will be interested in the maximum number of steps that P takes on inputs of length n (which we often denote as $T(n)$).[1] For example, if a function F can be computed by a NAND«

[1] Because we are interested in the *maximum* number of steps for inputs of a given length, this concept is often known as *worst case complexity*. The *minimum* number of steps (or "best case" complexity) to compute a function on length n 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 course.

(or NAND++) program that on inputs of length n takes $O(n)$ steps then we will think of F as "efficiently computable", while if any such program requires $2^{\Omega(n)}$ steps to compute F then we consider F "intractable". Formally, we define running time as follows:

Definition 14.1 — Running time. Let $T : \mathbb{N} \to \mathbb{N}$ be some function mapping natural numbers to natural numbers. We say that a function $F : \{0,1\}^* \to \{0,1\}$ is *computable in $T(n)$ NAND« time* if there is a NAND« program P computing F such that for every sufficiently large n and every $x \in \{0,1\}^n$, on input x, P runs for at most $T(n)$ steps. Similarly, we say that F is *computable in $T(n)$ NAND++ time* if there is a NAND++ program P computing F such that on every sufficiently large n and $x \in \{0,1\}^n$, on input x, P runs for at most $T(n)$ steps.

We let $TIME_{<<}(T(n))$ denote the set of Boolean functions that are computable in $T(n)$ NAND« time, and define $TIME_{++}(T(n))$ analogously. [2]

[2] The relaxation of considering only "sufficiently large" n'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.

Definition 14.1 naturally extend to non Boolean and to partial functions as well, and so we will talk about the time complexity of these functions.

Which model to choose? 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 care about "coarse enough" resolution (as we will in this course) then the choice of the model, whether it is NAND«, NAND++, or Turing or RAM machines of various flavors, does not matter. (This is known as the *extended* Church-Turing Thesis). Nevertheless, to be concrete, we will use NAND« programs as our "default" computational model for measuring time, and so if we say that F is computable in $T(n)$ time without any qualifications, or write $TIME(T(n))$ without any subscript, we mean that this holds with respect to NAND« machines.

Nice time bounds. When considering time bounds, we want to restrict attention to "nice" bounds such as $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^{\sqrt{n}})$, $O(2^n)$, etc. and avoid pathological examples such as non-monotone functions (where the time to compute a function on inputs of size n could be smaller than the time to compute it on shorter inputs) or other degenerate cases such as functions that can be computed without reading the input or cases where the running time bound itself is hard to compute. Thus we make the following definition:

Definition 14.2 — Nice functions. A function $T : \mathbb{N} \to \mathbb{N}$ is a *nice time bound function* (or nice function for short) if:

* $T(n) \geq n$

* $T(n) \geq T(n')$ whenever $n \geq n'$

* There is a NAND« program that on input numbers n, i, given in binary, can compute in $O(T(n))$ steps the i-th bit of a prefix-free representation of $T(n)$ (represented as a string in some prefix-free way).

All the functions mentioned above are "nice" per Definition 14.2, and from now on we will only care about the class $TIME(T(n))$ when T is a "nice" function. The last condition simply means that we can compute the binary representation of $T(n)$ in time which itself is roughly $T(n)$. This condition is typically easily satisfied. For example, for arithmetic functions such as $T(n) = n^3$ or $T(n) = \lfloor n \rfloor^{1.2} \log n \rfloor$ we can typically compute the binary representation of $T(n)$ in time which is polynomial in the *number of bits* in this representation. Since the number of bits is $O(\log T(n))$, any quantity that is polynomial in this number will be much smaller than $T(n)$ for large enough n.

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

- **Polynomial time:** We say that a function $F : \{0,1\}^* \to \{0,1\}$ is *computable in polynomial time* if it is in the class $\mathbf{P} = \cup_{c \in \mathbb{N}} TIME(n^c)$.

- **Exponential time:** We say that function $F : \{0,1\}^* \to \{0,1\}$ is *computable in exponential time* if it is in the class $\mathbf{EXP} = \cup_{c \in \mathbb{N}} TIME(2^{n^c})$.

In other words, these are defined as follows:

Definition 14.3 — P and EXP. Let $F : \{0,1\}^* \to \{0,1\}$. We say that $F \in \mathbf{P}$ if there is a polynomial $p : \mathbb{N} \to \mathbb{R}$ and a NAND« program P such that for every $x \in \{0,1\}^*$, $P(x)$ runs in at most $p(|x|)$ steps and outputs $F(x)$.

We say that $F \in \mathbf{EXP}$ if there is a polynomial $p : \mathbb{N} \to \mathbb{R}$ and a NAND« program P such that for every $x \in \{0,1\}^*$, $P(x)$ runs in at most $2^{p(|x|)}$ steps and outputs $F(x)$.

P Please make sure you understand why Definition 14.3 and the bullets above define the same classes.

Since exponential time is much larger than polynomial time, clearly **P** $\subseteq$ **EXP**. All of the problems we listed in the last lecture are in **EXP**,[3] but as we've seen, for some of them there are much better algorithms that demonstrate that they are in fact in **P**.

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

A table of the examples from the previous lecture. All these problems are in **EXP** but the only the ones on the left column are currently known to be in **P** (i.e., have a polynomial-time algorithm).

[3] Strictly speaking, many of these problems correspond to *non Boolean* functions, but we will sometimes "abuse notation" and refer to non Boolean functions as belonging to **P** or **EXP**. We can easily extend the definitions of these classes to non Boolean and partial functions. Also, for every non-Boolean function $F : \{0,1\}^* \to \{0,1\}^*$, we can define a Boolean variant $Bool(F)$ such that F can be computed in polynomial time if and only if $Bool(F)$ is.

14.1 *NAND« vs NAND++*

We have seen that for every NAND« program P there is a NAND++ program P' that computes the same function as P. It turns out that the P' is not much slower than P. That is, we can prove the following theorem:

Theorem 14.4 — Efficient simulation of NAND« with NAND++. There are absolute constants a, b such that for every function F and nice function $T : \mathbb{N} \to \mathbb{N}$, if $F \in TIME_{<<}(T(n))$ then there is a NAND++ program P' that computes F in $T'(n) = a \cdot T(n)^b$. That is, $TIME_{<<}(T(n)) \subseteq TIME_{++}(aT(n)^b)$

The constant b can be easily shown to be at most five, and with more effort can be optimized further. Theorem 14.4 means that the definition of the classes **P** and **EXP** are robust to the choice of model, and will not make a difference whether we use NAND++ or NAND«. In fact, similar results are known for Turing Machines, RAM machines, C programs, and a great many other models, which justifies the choice of **P** as capturing a technology-independent notion

of tractability. As we discussed before, this equivalence between NAND++ and NAND« (as well as other models) allows us to pick our favorite one depending on the task at hand (i.e., "have our cake and eat it too"). When we want to *design* an algorithm, we can use the extra power and convenience afforded by NAND«. When we want to *analyze* a program or prove a *negative result*, we can restrict attention to NAND++ programs.

Proof Idea: We have seen in Theorem 8.1 that every function F that is computable by a NAND« program P is computable by a NAND++ program P'. To prove Theorem 14.4, we follow the exact same proof but just check that the overhead of the simulation of P by P' is polynomial.

Proof of Theorem 14.4. As mentioned above, we follow the proof of Theorem 8.1 (simulation of NAND« programs using NAND++ 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« works by "peeling off" features of NAND« one by one, until we are left with NAND++. We now sketch the main observations we use to show that this "peeling off" costs at most a polynomial overhead:

1. If P is a NAND« program that computes F in $T(n)$ time, then on inputs of length n, all integers used by P are of magnitude at most $T(n)$. This means that the largest value `i` can ever reach is at most $T(n)$ and so each one of P's variables can be thought of as an array of at most $T(n)$ indices, each of which holds a natural number of magnitude at most $T(n)$ (and hence one that can be encoded using $O(\log T(n))$ bits). Such an array can be encoded by a bit array of length $O(T(n) \log T(n))$.

2. All the arithmetic operations on integers use the gradeschool algorithms, that take time that is polynomial in the number of bits of the integers, which is $poly(\log T(n))$ in our case.

3. Using the `i++` and `i-"-` operations we can load an integer (represented in binary) from the variable `foo` into the index `i` using a cost of $O(T(n)^2)$. The idea is that we create an array `marker` that contains a single 1 coordinate and all the rest are zeroes. We will repeat the following for at most $T(n)$ steps: at each step we decrease `foo` by one (at a cost of $O(\log T(n))$) and move the 1 in `marker` one step to the right (at a cost of $O(T(n))$). We stop when `foo` reaches 0, at which point `marker` has 1 in the location encoded

by the number that was in `foo`, and so if we move `i` until `marker_i` equals to 1 then we reach our desired location.

4. Once that is done, all that is left is to simulate `i++` and `i--` in NAND++ using our "breadcrumbs" and "wait for the bus" technique. To simulate T steps of increasing and decreasing the index, we will need at most $O(T^2)$ steps of NAND++ (see Fig. 14.1). In the worst case for every increasing or decreasing step we will need to wait a full round until `i` reaches 0 and gets back to the same location, in which case the total cost will be $O(1+2+3+4+\cdots+T) = O(T^2)$ steps.

Together these observations imply that the simulation of T steps of NAND« can be done in $poly(T)$ step. (In fact the cost is $O(T^4 polylog(T)) = O(T^5)$ steps, and can even be improved further though this does not matter much.) ■

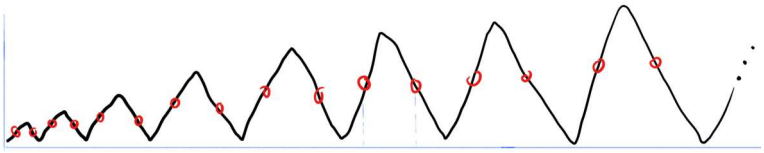

Figure 14.1: The path an index variable takes in a NAND++ program

R **Turing machines and other models** If we follow the equivalence results between NAND++/NAND« and other models, including Turing machines, RAM machines, Game of life, λ calculus, and many others, then we can see that these results also have at most a polynomial overhead in the simulation in each way. [4] It is a good exercise to go through, for example, the proof of Theorem 9.2 and verify that it establishes that Turing machines and NAND++ programs are equivalent up to polynomial overhead.

R **Robustness of representation** Theorem 14.4 shows that the classes **P** and **EXP** are robust with respect to variation in the choice of the computational model. They are also robust with respect to our choice of the representation of the input. For example, whether we decide to represent graphs as adjacency matrices or adjacency lists will not make a difference as to whether a function on graphs is in **P** or **EXP**. The reason is that changing from one representation to another at most squares the size of the input, and a quantity is polynomial in n if and only if it is polynomial in n^2.

[4] One technical point is that for λ calculus, one needs to be careful about the order of application of the reduction steps, which can matter for computational efficiency. Counting running time for λ calculus is somewhat delicate, see this paper.

More generally, for every function $F : \{0,1\}^* \to \{0,1\}$, the answer to the question of whether $F \in \mathbf{P}$ (or whether $F \in \mathbf{EXP}$) 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).

14.2 *Efficient universal machine: a NAND« interpreter in NAND«*

We have seen in Theorem 8.2 the "universal program" or "interpreter" U for NAND++. Examining that proof, and combining it with Theorem 14.4 , we can see that the program U has a *polynomial* overhead, in the sense that it can simulate T steps of a given NAND++ (or NAND«) program P on an input x in $O(T^a)$ steps for some constant a. But in fact, by directly simulating NAND« programs, we can do better with only a *constant* multiplicative overhead:

Theorem 14.5 — Efficient universality of NAND«. There is a NAND« program U that computes the partial function $TIMEDEVAL : \{0,1\}^* \to \{0,1\}^*$ defined as follows:

$$TIMEDEVAL(P,x,1^T) = P(x) \tag{14.1}$$

if P is a valid representation of a NAND« program which produces an output on x within at most T steps. If P does not produce an output within this time then $TIMEDEVAL$ outputs an encoding of a special `fail` symbol. Moreover, for every program P, the running time of U on input $P,x,1^T$ is $O(T)$. (The hidden constant in the Oh notation can depend on the program P but is at most polynomial in the length of P's description as a string.).

P Before reading the proof of Theorem 14.5, try to think how you would compute $TIMEDEVAL$ using your favorite programming language. That is, how you would write a program `TIMEDEVAL(P,x,T)` that gets a NAND« program `P` (represented in some convenient form), a string `x`, and an integer `T`, and simulates `P` for `T` steps. You will likely find that your program requires $O(T)$ steps to perform this simulation.

Proof of Theorem 14.5. To present a universal NAND« program in full we need to describe a precise representation scheme, as well as the full NAND« 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 complete specification for NAND« is given in the Appendix, and for the purposes of this simulation, we can simply use the representation of the code NAND« as an ASCII string.

The program U gets as input a NAND« program P, an input x, and a time bound T (given in the form 1^T) and needs to simulate the execution of P for T steps. To do so, U will do the following:

1.U will maintain variables `icP`, `lcP`, and `iP` for the iteration counter, line counter, and index variable of P.

2.U will maintain an array `varsP` for all other variables of P. If P has s lines then it uses at most $3s$ variable identifiers. U will associate each identifier with a number in $[3s]$. It will encode the contents of the variable with identifier corresponding to a and index j at the location `varsP`$\_\langle 3s \cdot j + a \rangle$.

3. U will maintain an array `LinesP` of $O(s)$ size that will encode the lines of P in some canonical encoding.

4. To simulate a single step of P, the program U will recover the line corresponding to `lcP` from the `LinesP` and execute it. Since NAND« has a constant number of arithmetic operations, we can simulate choosing which operation to execute with a sequence of a constantly many if-then-else's.[5] When executing these operations, U will use the variable `icP` that keeps track of the iteration counter of P.

Simulating a single step of P will take U $O(s)$ steps, , and hence the simulation will be $O(sT)$ which is $O(T)$ when suppressing constants such as s that depend on the program P. ■

[5] While NAND« does not formally have if/then/else, we can easily add this as syntactic sugar.

14.3 *Time hierarchy theorem*

We have seen that there are uncomputable functions, 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 2^n, but *can not* be computed in time $2^{0.9n}$? It turns out that the answer is **Yes**:

Theorem 14.6 — Time Hierarchy Theorem. For every nice function T, there is a function $F : \{0,1\}^* \to \{0,1\}$ in $TIME(T(n)\log n) \setminus TIME(T(n))$. [6]

[6] There is nothing special about $\log n$, and we could have used any other efficiently computable function that tends to infinity with n.

Note that in particular this means that $\mathbf{P} \neq \mathbf{EXP}$.

Proof Idea: In the proof of Theorem 10.2 (the uncomputability of the Halting problem), we have shown that the function $HALT$ 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 T steps, then the proof shows that 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 T' steps and those that take more than that (where T' is some number depending on T). Therefore, the proof of Theorem 14.6 follows the ideas of the uncomputability of the halting problem, but again with a more careful accounting of the running time.

Proof of Theorem 14.6. Recall the Halting function $HALT : \{0,1\}^* \to \{0,1\}$ that was defined as follows: $HALT(P,x)$ equals 1 for every program P and input x s.t. P halts on input x, and is equal to 0 otherwise. We cannot use the Halting function of course, as it is uncomputable and hence not in $TIME(T'(n))$ for any function T'. However, we will use the following variant of it:

We define the *Bounded Halting* function $HALT_T(P,x)$ to equal 1 for every NAND« program P such that $|P| \leq \log\log|x|$, and such that P halts on the input x within $100T(|x|)$ steps. $HALT_T$ equals 0 on all other inputs.[7]

[7] The constant 100 and the function $\log\log n$ are rather arbitrary, and are chosen for convenience in this proof.

Theorem 14.6 is an immediate consequence of the following two claims:

Claim 1: $HALT_T \in TIME(T(n)\ logn)$

and

Claim 2: $HALT_T \notin TIME(T(n))$.

We now turn to proving the two claims.

Proof of claim 1: We can easily check in linear time whether an input has the form P,x where $|P| \leq \log\log|x|$. Since $T(\cdot)$ is a nice function, we can evaluate it in $O(T(n))$ time. Thus, we can perform the check above, compute $T(|P|+|x|)$ and use the universal NAND« program of Theorem 14.5 to evaluate $HALT_T$ in at most $poly(|P|)T(n)$ steps.[8]

[8] Recall that we use $poly(m)$ to denote a quantity that is bounded by am^b for some constants a,b and every sufficiently large m.

Since $(\log\log n)^a = o(\log n)$ for every a, this will be smaller than $T(n)\log n$ for every sufficiently large n.

Proof of claim 2: The proof is very reminiscent of the proof that $HALT$ is not computable. Assume, toward the sake of contradiction, that there is some NAND« program P^* that computes $HALT_T(P,x)$ within $T(|P|+|x|)$ steps. We are going to show a contradiction by creating a program Q and showing that under our assumptions, if Q runs for less than $T(n)$ steps when given (a padded version of) its own code as input then it actually runs for more than $T(n)$ 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 Q to be the program that on input a string z does the following:

1. If z does not have the form $z = P1^m$ where P represents a NAND« program and $|P| < 0.1 \log\log m$ then return 0.

2. Compute $b = P^*(P,z)$ (at a cost of at most $T(|P|+|z|)$ steps, under our assumptions).

3. If $b = 1$ then Q goes into an infinite loop, otherwise it halts.

We chose m sufficiently large so that $|Q| < 0.001 \log\log m$ where $|Q|$ denotes the length of the description of Q as a string. We will reach a contradiction by splitting into cases according to whether or not $HALT_T(Q,Q1^m)$ equals 0 or 1.

On the one hand, if $HALT_T(Q,Q1^m) = 1$, then under our assumption that P^* computes $HALT_T$, Q will go into an infinite loop on input $z = Q1^m$, and hence in particular Q does *not* halt within $100T(|Q|+m)$ steps on the input z. But this contradicts our assumption that $HALT_T(Q,Q1^m) = 1$.

This means that it must hold that $HALT_T(Q,Q1^m) = 0$. But in this case, since we assume P^* computes $HALT_T$, Q 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 $5|z|$ steps. Phase 2 involves executing P^*, which under our assumption requires $T(|Q|+m)$ steps. In total we can perform both phases in less than $10T(|Q|+m)$ in steps, which by definition means that $HALT_T(Q,Q1^m) = 1$, but this is of course a contradiction. ■

The time hierarchy theorem tells us that there are functions we can

compute in $O(n^2)$ time but not $O(n)$, in 2^n time, but not $2^{\sqrt{n}}$, etc.. In particular there are most definitely functions that we can compute in time 2^n but not $O(n)$. We have seen that we have no shortage of natural functions for which the best *known* algorithm requires roughly 2^n 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 $O(n)$ time algorithm. We will however see that there is a single unproven conjecture that would imply such a result for most of these problems.

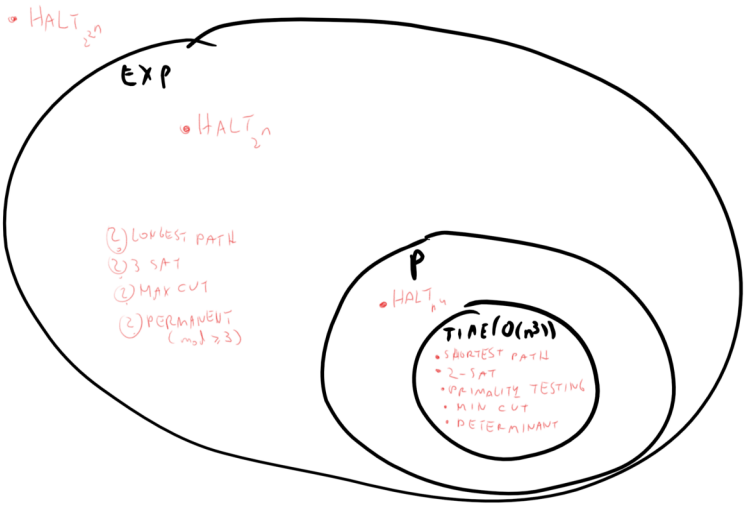

Figure 14.2: Some complexity classes and some of the functions we know (or conjecture) to be contained in them.

14.4 Uniform vs non uniform computation

We have now seen two measures of "computation cost" for functions. For a finite function $G : \{0,1\}^n \rightarrow \{0,1\}^m$, we said that $G \in SIZE(T)$ if there is a T-line NAND program that computes G. We saw that *every* function mapping $\{0,1\}^n$ to $\{0,1\}^m$ can be computed using at most $O(m2^n)$ lines. For infinite functions $F : \{0,1\}^* \rightarrow \{0,1\}^*$, we can define the "complexity" by the smallest T such that $F \in TIME(T(n))$. Is there a relation between the two?

For simplicity, let us restrict attention to Boolean (i.e., single-bit output) functions $F : \{0,1\}^* \rightarrow \{0,1\}$. For every such function, define $F_n : \{0,1\}^n \rightarrow \{0,1\}$ to be the restriction of F to inputs of size n. We have seen two ways to define that F is computable within a roughly $T(n)$ amount of resources:

1. There is a *single algorithm* P that computes F within $T(n)$ steps on all inputs of length n. In such a case we say that F is *uniformly* computable (or more often, simply "computable") within $T(n)$ steps.

2. For every n, there is a $T(n)$ NAND program Q_n that computes F_n. In such a case we say that F has can be computed via a *non uniform* $T(n)$ bounded sequence of algorithms.

Unlike the first condition, where there is a single algorithm or "recipe" to compute F on all possible inputs, in the second condition we allow the restriction F_n to be computed by a completely different program Q_n for every n. One can see that the second condition is much more relaxed, and hence we might expect that every function satisfying the first condition satisfies the second one as well (up to a small overhead in the bound $T(n)$). This indeed turns out to be the case:

Theorem 14.7 — Nonuniform computation contains uniform computation. There is some $c \in \mathbb{N}$ s.t. for every $F : \{0,1\}^* \to \{0,1\}$ in $TIME_{++}(T(n))$ and every sufficiently large $n \in N$, F_n is in $SIZE(cT(n))$.

Proof Idea: To prove Theorem 14.7 we use the technique of "unraveling the loop". That is, we can in general use "copy paste" to replace a program P that uses a loop that iterates for at most T times with a "loop free" program that has about T times as many lines as P. In particular, given $n \in \mathbb{N}$ and a NAND++ program P we can transform a NAND++ program

Proof of Theorem 14.7. The proof follows by the "unraveling" argument that we've already seen in the proof of Theorem 7.4. Given a NAND++ program P and some function $T(n)$, we can transform NAND++ to be "simple" in the sense of Definition 7.2 with a constant factor overhead (in fact the constant is at most 5). Thus we can construct a NAND program on n inputs and with less than $cT(n)$ lines by making it simple and then simply "unraveling the main loop" of P and hence putting $T(n)/L$ copies of P one after the other, where L is the number of lines in P, replacing any instance of `i` with the numerical value of `i` for that iteration. While the original NAND++ program P might have ended on some inputs *before* $T(n)$ iterations have passed, by transforming it to a simple program we ensure that there is no harm in "extra" iterations.[9] By combining Theorem 14.7 with Theorem 14.4, we get that if $F \in TIME(T(n))$

[9] Specifically, Definition 7.2 ensures that there is a variable `halted` that is set to 1 once the program is "supposed" to halt, and all assignments to `loop`, `y_0` and `halted` itself are modified so that if `halted` equals 1 then the value of these variables does not change. Thus continuing for extra iterations does not change the value of these variables.

then there are some constants a, b such that for every large enough n, $F_n \in SIZE(aT(n)^b)$. (In fact, by direct inspection of the proofs we can see that $a = 1$ and $b = 5$ would work.) ■

Algorithmic version: the "NAND++ to NAND compiler": The transformation of the NAND++ program P to the NAND program Q_P is itself algorithmic. Thus we can also phrase this result as follows:

Theorem 14.8 — NAND++ to NAND compiler. There is an $O(n)$-time NAND« program $COMPILE$ such that on input a NAND++ program P, and strings of the form $1^n, 1^m, 1^T$ outputs a NAND program Q_P of at most $O(T)$ lines with n bits of inputs and m bits of output, such that: For every $x \in \{0,1\}^n$, if P halts on input x within fewer than T steps and outputs some string $y \in \{0,1\}^m$, then $Q_P(x) = y$.

The program $COMPILE$ of Theorem 14.8 is fairly easy to implement. In particular this is done by the following very simple python function

```
#Input: Source code of a NAND++ program P,
# time bound T, input length n
#Output: n-input NAND program of T|P| lines
# computing same function as P
#For simplicity we assume that program P
#has a constant m number of outputs,
#and validx only used with the index i.
def expand(P,T,n):
   result = r'''notx_0 := x_0 NAND x_0
one := x_0 NAND notx_0
zero := one NAND one'''

   for t in range(T):
      i=index(t)
      validx = ('one' if i<n else 'zero')
      result += P.replace('validx_i',validx
             ).replace('_i',f'_{i}')
   return result

# Returns value of index variable i in iteration t
def index(t):
   r = math.floor(math.sqrt(t+1/4)-1/2)
   return (t-r*(r+1) if t <= (r+1)*(r+1) else (r+1)*(r+2)-t)
```

Since NAND« programs can be simulated by NAND++ programs with polynomial overhead, we see that we can simulate a $T(n)$ time NAND« program on length n inputs with a $poly(T(n))$ size NAND program.

P To make sure you understand this transformation, it is an excellent exercise to verify the following equivalent characterization of the class **P** (see Exercise 14.4). Prove that for every $F : \{0,1\}^* \to \{0,1\}$, $F \in \mathbf{P}$ if and only if there is a polynomial-time NAND++ (or NAND«, it doesn't matter) program P such that for every $n \in \mathbb{N}$, $P(1^n)$ outputs a description of an n inpute NAND program Q_n that computes the restriction F_n of F to inputs in $\{0,1\}^n$. (Note that since P runs in polynomial time and hence has an output of at most polynomial length, Q_n has at most a polynomial number of lines.)

14.4.1 *The class* $\mathbf{P_{/poly}}$

We can define the "non uniform" analog of the class **P** as follows:

Definition 14.9 — $\mathbf{P_{/poly}}$. For every $F : \{0,1\}^* \to \{0,1\}$, we say that $F \in \mathbf{P_{/poly}}$ if there is some polynomial $p : \mathbb{N} \to \mathbb{R}$ such that for every $n \in \mathbb{N}$, $F_n \in SIZE(p(n))$ where F_n is the restriction of F to inputs in $\{0,1\}^n$.

An immediate corollary of Theorem 14.7 is that $\mathbf{P} \subseteq \mathbf{P_{/poly}}$. Using the equivalence of NAND programs and Boolean circuits, we can also define $P_{/poly}$ as the class of functions $F : \{0,1\}^* \to \{0,1\}$ such that the restriction of F to $\{0,1\}^n$ is computable by a Boolean circuit of $poly(n)$ size (say with gates in the set $\wedge, \vee, \neg$ though any universal gateset will do); see Fig. 14.3.

The notation $\mathbf{P_{/poly}}$ 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 (or equivalently, NAND++ programs) that are given for any input length n a polynomial in n long *advice string*. That this is an equivalent characterization is shown in the following theorem:

Theorem 14.10 — $\mathbf{P_{/poly}}$ characterization by advice. Let $F : \{0,1\}^* \to \{0,1\}$. Then $F \in \mathbf{P_{/poly}}$ if and only if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$, a polynomial-time NAND++ program P and a

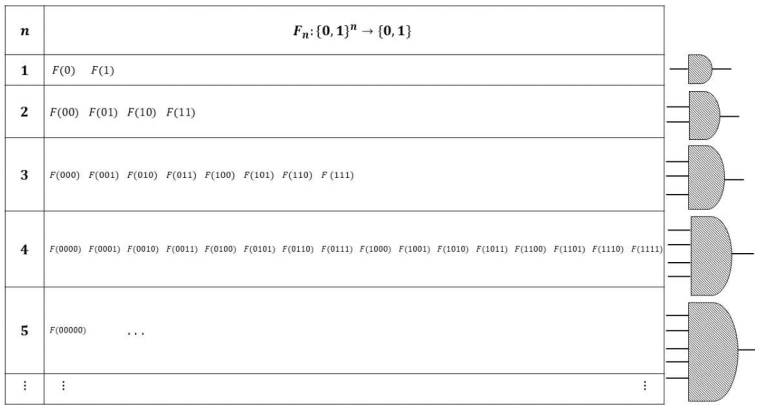

Figure 14.3: We can think of an infintie function $F : \{0,1\}^* \to \{0,1\}$ as a collection of finite functions $F_0, F_1, F_2, \ldots$ where $F_n : \{0,1\}^n \to \{0,1\}$ is the restriction of F to inputs of length n. We say F is in $\mathbf{P_{/poly}}$ if for every n, the function F_n is computable by a polynomial size NAND program, or equivalently, a polynomial sized Boolean circuit. (We drop in this figure the "edge case" of F_0 though as a constant function, it can always be computed by a constant sized NAND program.)

sequence $\{a_n\}_{n\in\mathbb{N}}$ of strings, such that for every $n \in \mathbb{N}$:

* $|a_n| \leq p(n)$

* For every $x \in \{0,1\}^n$, $P(a_n, x) = F(x)$.

Proof. We only sketch the proof. For the "only if" direction, if $F \in \mathbf{P_{/poly}}$ then we can use for a_n simply the description of the corresponding NAND program Q_n, and for P the program that computes in polynomial time the $NANDEVAL$ function that on input an n-input NAND program Q and a string $x \in \{0,1\}^n$, outputs $Q(n)>$

For the "if" direction, we can use the same "unrolling the loop" technique of Theorem 14.7 to show that if P is a polynomial-time NAND++ program, then for every $n \in \mathbb{N}$, the map $x \mapsto P(a_n, x)$ can be computed by a polynomial size NAND program Q_n. ■

P To make sure you understand the definition of $\mathbf{P_{/poly}}$, I highly encourage you to work out fully the details of the proof of Theorem 14.10.

14.4.2 *Simulating NAND with NAND++?*

Theorem 14.7 shows that every function in $TIME(T(n))$ is in $SIZE(poly(T(n)))$. One can ask if there is an inverse relation. Suppose that F is such that F_n has a "short" NAND program for

every n. Can we say that it must be in $TIME(T(n))$ for some "small" T?

The answer is **no**. Indeed, consider the following "unary halting function" $UH : \{0,1\}^* \to \{0,1\}$ defined as follows: $UH(x) = 1$ if and only if the binary representation of $|x|$ corresponds to a program P such that P halts on input P. UH is uncomputable, since otherwise we could compute the halting function by transforming the input program P into the integer n whose representation is the string P, and then running $UH(1^n)$ (i.e., UH on the string of n ones). On the other hand, for every n, $UH_n(x)$ is either equal to 0 for all inputs x or equal to 1 on all inputs x, and hence can be computed by a NAND program of a *constant* number of lines.

The issue here is of course *uniformity*. For a function $F : \{0,1\}^* \to \{0,1\}$, if F is in $TIME(T(n))$ then we have a *single* algorithm that can compute F_n for every n. On the other hand, F_n might be in $SIZE(T(n))$ for every n using a completely different algorithm for every input length. For this reason we typically use $\mathbf{P_{/poly}}$ 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 n requires more then a polynomial number of NAND lines. Since $\mathbf{P} \subseteq \mathbf{P_{/poly}}$, 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 "128 bits of security".

> R **Non uniformity in practice** While it can sometimes be a real issue, in many natural settings the difference between uniformity and non-uniform-thmity does not seem to arise. In particular, in all the examples of problems not known to be in $\mathbf{P}$ we discussed before: longest path, 3SAT, factoring, etc., these problems are also not known to be in $\mathbf{P_{/poly}}$ either. Thus, for "natural" functions, if you pretend that $TIME(T(n))$ is roughly the same as $SIZE(T(n))$, you will be right more often than wrong.

14.4.3 Uniform vs. Nonuniform computation: A recap

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

- NAND programs, which have no loops, can only compute finite functions, and the time to execute them is exactly the number of lines they contain. These are also known as *straightline programs* or *Boolean circuits*.
- NAND++ programs, which include loops, and hence a single program can compute a function with unbounded input length. These are equivalent (up to polynomial factors) to *Turing Machines* or (up to polylogarithmic factors) to *RAM machines*.

For a function $F : \{0,1\}^* \to \{0,1\}$ and some nice time bound $T : \mathbb{N} \to \mathbb{N}$, we know that:

- If F is computable in time $T(n)$ then there is a sequence $\{P_n\}$ of NAND programs with $|P_n| = poly(T(n))$ such that P_n computes F_n (i.e., restriction of F to $\{0,1\}^n$) for every n.
- The reverse direction is not necessarily true - there are examples of functions $F : \{0,1\}^n \to \{0,1\}$ such that F_n can be computed by even a constant size NAND program but F is uncomputable.

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

14.5 Extended Church-Turing Thesis

We have mentioned the Church-Turing thesis, that posits that the definition of computable functions using NAND++ programs captures the definition that would be obtained by all physically realizable computing devices. The *extended* Church Turing is the statement that the same holds for *efficiently computable* functions, which is typically interpreted as saying that NAND++ programs can simulate every physically realizable computing device with polynomial overhead.

In other words, the extended Church Turing thesis says that for every *scalable computing device* C (which has a finite description but can be in principle used to run computation on arbitrarily large inputs), there are some constants a, b such that for every function $F : \{0,1\}^* \to \{0,1\}$ that C can compute on n length inputs using an $S(n)$ amount of physical resources, F is in $TIME(aS(n)^b)$.

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++ or NAND« programs). consequently, the classes **P** and $\mathbb{EXP}$ are robust to the choice of

model, and we can the programming language of our choice, or high level descriptions of an algorithm, to determine whether or not a problem is in **P**.

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, which would yield predictions such as the *Physical Extended Church-Turing Thesis* we mentioned before, which would be experimentally testable.

In the last hundred+ years of studying and mechanizing computation, no one has yet constructed a scalable computing device (or even gave a convincing blueprint) that violates the extended Church Turing Thesis. That said, as we mentioned before *quantum computing*, if realized, does pose a serious challenge to this thesis. However, even if the promises of quantum computing are fully realized, it still seems that the extended Church-Turing thesis is fundamentally or "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, and we still have hierarchy and uncomputability results.[10] Moreover, for most (though not all!) concrete problems we care about, the prospect of quantum computing does not seem to change their time complexity. In particular, out of all the example problems mentioned in the previous lecture, as far as we know, the complexity of only one— integer factoring— is affected by modifying our model to include quantum computers as well.

[10] Note that indeed, quantum computing is *not* a challenge to the Church Turing itself, as a function is computable by a quantum computer if and only if it is computable by a "classical" computer or a NAND++ program. It is only the running time of computing the function that can be affected by moving to the quantum model.

14.6 Lecture summary

- We can define the time complexity of a function using NAND++ 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 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 $T(n)$ can be computed by a sequence of NAND programs (one for every input length) each of size at most $poly(T(n))$

14.7 Exercises

For these exercises, a class $\overline{C}$ is the multi-bit output analog of the class C, where we consider programs that output more than one bit.

Exercise 14.1 — Composition of polynomial time. Prove that if $F, G : \{0,1\}^* \to \{0,1\}^*$ are in $\overline{\mathbf{P}}$ then their *composition* $F \circ G$, which is the function H s.t. $H(x) = F(G(x))$, is also in $\overline{\mathbf{P}}$. ■

Exercise 14.2 — Non composition of exponential time. Prove that there is some $F, G : \{0,1\}^* \to \{0,1\}^*$ s.t. $F, G \in \overline{\mathbf{EXP}}$ but $F \circ G$ is not in $\mathbf{EXP}$.[11] ■

Exercise 14.3 — Oblivious program. We say that a NAND++ program P is oblivious if there is some functions $T : \mathbb{N} \to \mathbb{N}$ and $i : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that for every input x of length n, it holds that:

* P halts when given input x after exactly $T(n)$ steps.

* For $t \in \{1, \ldots, T(n)\}$, after P executes the t^{th} step of the execution the value of the index `i` is equal to $t(n, i)$. In particular this value does *not* depend on x but only on its length.[12] Let $F : \{0,1\}^* \to \{0,1\}^*$ be such that there is some function $m : \mathbb{N} \to \mathbb{N}$ satisfying $|F(x)| = m(|x|)$ for every x, and let P be a NAND++ program that computes F in $T(n)$ time for some nice T. Then there is an *oblivious* NAND++ program P' that computes F in time $O(T^2(n) \log T(n))$. ■

Exercise 14.4 — Alternative characterization of P. Prove that for every $F : \{0,1\}^* \to \{0,1\}$, $F \in \mathbf{P}$ if and only if there exists a polynomial time NAND++ program P such that $P(1^n)$ outputs a NAND program Q_n that computes the restriction of F to $\{0,1\}^n$. ■

14.8 Bibliographical notes

[11] TODO: check that this works, idea is that we can do bounded halting.

[12] An oblivious program P cannot compute functions whose output length is not a function of the input length, though this is not a real restriction, as we can always embed variable output functions in fixed length ones using some special "end of output" marker.

[13] TODO: add reference to best algorithm for longest path - probably the Bjorklund algorithm

14.9 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

14.10 Acknowledgements