The aim of computational complexity theory is to study the amount of resources necessary – such as time, memory, randomness, "quantumness", interaction – to perform given computations. The origins of this theory can be traced back to the 1960s when it was realized that certain simple problems would take an inordinate amount of time to solve on any computer, even though these problems are in principle solvable. Moreover, the inherent difficulty of these problems does not have anything to do with the computing technology that was available in the 1960s. With today's technology we can do no better, and complexity theory indicates that even 100 years from now we will be stuck at the same place. It appears that nature imposes intrinsic obstacles at performing certain computations, and a central question in complexity theory is to understand why and how these obstacles arise.

To initiate a study of computation we must agree on some essentials. First, what are the computational problems and processes that we are interested in studying? And second, what is a good model that will allow us to reason about computation in a way that is general enough to encompass all computing technology of the day?

1 Computational problems

The answer to the first question depends somewhat on your personal taste. However unlike computability theory, which mostly studies problems *about* computations, much of the focus of modern complexity theory has been on "natural" problems that are inspired by other scientific disciplines and areas of life – be it mathematics, economics, physics, or sociology – and is somewhat unique in its ability to yield interesting insights on problems arising from such a wide range of sources. Of course, complexity theory also studies problems arising from computations, but these are often used just as a stepping stone to say something about problems that scientists and engineers from other disciplines might be interested in.

Here are some examples of problems that complexity theory is interested in.

Decision problems A decision problem (or a language) is one that can be answered by "yes" or "no":

MATCHING (PERFECT MATCHING) Given an undirected graph, does it contain a perfect matching, i.e. a pairing of its vertices such that each pair is connected by an edge?

SAT (BOOLEAN FORMULA SATISFIABILITY) Given a boolean formula in conjunctive normal form (CNF), for instance

$$(x_1 \vee \overline{x_2}) \wedge (x_1 \vee x_3 \vee \overline{x_4}) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_4})$$

is there an assignment that satisfies the formula (i.e. it satisfies all the clauses simultaneously)? For instance the assignment

$$x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0$$

satisfies the above formula.

As a standard convention, we represent inputs to decision problems as binary strings; for instance a graph of n vertices can be represented as a string of length $\binom{n}{2}$, where each position in the string indicates a potential edge in the graph, and the value at this position indicates the presence or absence of this edge. Sometimes several representations are possible – for instance a graph can be represented as an adjacency matrix or an adjacency list – and usually it will not matter which representation we choose.

We will denote decision problems either as subsets L of the set $\{0,1\}^*$ of all possible strings – for instance, for the matching problem, L is the set of all strings representing graphs that have a perfect matching – or as functions $L : \{0,1\}^* \to \{0,1\}$ where $L(x) = 1$ if the string x is in L and 0 otherwise.

Search problems In a search problem we not only want to know if a solution exists, but find the actual solution as well:

FIND-MATCHING Given a graph, find a perfect matching, if one exists.

FIND-SAT Given a boolean formula, find an assignment that satisfies it, if one exists.

We will represent search problems as relations R over pairs of strings $(x, y) \in \{0,1\}^* \times \{0,1\}^*$. The input x represents the problem (the graph) while y represents potential solutions (the perfect matching). The pair (x, y) satisfies R if y is a solution of x (graph x contains the matching y).

Optimization problems Optimization problems ask for the best possible solution to a problem. A decision or search problem can have several optimization variants.

MAX-MATCHING Given a graph, find a maximum matching, i.e. the largest perfect matching present in one of its subgraphs.

MAX-SAT Given a boolean formula, find an assignment that satisfies as many of its clauses as possible.

MIN-EQUIV-SAT Given a boolean formula, find the smallest formula that is equivalent to it, i.e., one that shares the same set of satisfying assignments.

Sometimes we may be willing to settle for *approximate* optimization – namely instead of looking for the best solution, we may be happy with one that is close enough:

APPROX-MAX-SAT Given a boolean formula, find an assignment that satisfies at least 90% of the maximum possible number of clauses that can be satisfied.

Counting problems Counting problems ask for the number of solutions of a given instance:

#MATCHING Given a graph, count the number of perfect matchings it contains.

#SAT Given a boolean formula, count how many satisfying assignments it has.

Now let us jump ahead a little bit and see what kind of insights complexity theory can give us about these problems.

At a very general level, we are looking at two problems – MATCHING and SAT – of the following kind: We are given some object (a graph, a formula) and want to check, or count, or find if the object contains structures of a certain type (a matching, a satisfying assignment). In principle, we can solve both of these problems by trying out all possible structures of interest. However, even for moderately sized objects (a 100 node graph, or a 100 clause formula) this would take an inordinate amount of time even on very fast computers.

In 1965 Edmonds found an ingenious way of finding perfect matchings in graphs that takes much less effort than looking at all possible matchings in the graph. For an n vertex graph, Edmonds' algorithm performs a sequence of about n^3 operations which consist of marking certain edges in the graph and looking at their neighbors, at the end of which the maximum matching is highlighted. With today's technology we can run this algorithm on graphs with tens or hundreds of thousands of nodes.

In contrast to the MATCHING problem, for the SAT problem we know of no better algorithm than one that essentially performs an exhaustive search of all possible assignments. Most complexity theorists (and computer scientists in general) believe that no amount of ingenuity will help us here; rather it is a fundamental fact of nature that SAT cannot be solved by any algorithm substantially better than exhaustive search.

This is the famous "P versus NP" question – the most famous problem in complexity theory, and one to which we don't know the answer. So if complexity theory fails to explain why solving SAT requires exhaustive search, what is it good for? What complexity theory *can* do – and is very good at – is present evidence for the hardness of SAT in relation to other questions we might be interested in. Here are some examples.

- Suppose our intuition is incorrect and the SAT problem is in fact tractable. What can we then say about FIND-SAT, which is apparently harder than SAT? It turns out that if we can solve the decision problem SAT, then we can also solve the search problem FIND-SAT. How about MAX-SAT? It turns out that we can solve this one as well.
- The problem MIN-EQUIV-SAT also looks harder than SAT. Solving it seems to require two levels of exhaustive search: One that loops over all possible instances of SAT and another one that checks that the two instances are equivalent. However, it again turns out that if SAT is tractable, then so is MIN-EQUIV-SAT.
- How about APPROX-SAT? This appears easier than MAX-SAT. However, it turns out that these two problems are equivalent in hardness: This is a consequence of the "PCP theorem", one of the crowning achievements of complexity theory.
- What about counting solutions, namely #SAT? Here complexity theory gives us evidence to believe that #SAT is in fact harder than SAT. So even if we were wrong all along and one day found a clever algorithm for SAT, #SAT might be still standing.
- More surprisingly, complexity theory shows that #MATCHING is *just as hard as* #SAT; so

for the matching problem deciding, finding, and optimizing solutions is easy, but counting solutions seems much, much harder.

2 Models of computation

To reason rigourously about the complexity of computations we need a model that faithfully captures the resources that we have at our disposal. This appears somewhat difficult as the resources we have available change dramatically over time. In the early 1800s the mathematician Gauss was admired for his prowess at performing difficult calculations. In the early 1900 mechanical calculating devices put the best humans to shame, and by the 1950s electronic computers were already doing calculations that noone thought possible before. Today an ordinary cell phone can do much much more than a 1950s computer. Given such huge disparities among the devices at our disposal, how can we hope to make any sort of general statement about computation?

Complexity theory takes the perspective that the difference in resources between various computational devices is insignificant compared to the vast gap between the tractable and the intractable. For problems like MATCHING, it might matter what kind of computational device we use; perhaps Gauss could solve graphs of size 10, the 1950 machines could do size 100 while today's computers can solve problems of size 10,000. However, for instances of SAT the differences are much less dramatic; maybe Gauss could handle formulas in 10 variables, while today's computers can only do up to 40 or so variables.

When trying to separate the tractable from the intractable it does not really matter what model of computation we use, as long as it is a reasonable one – in the sense that it can both simulate and be simulated by a realistic computational device in a relatively efficient way. A standard model which is convenient for most purposes in complexity theory is the Turing Machine. This is a device with a finite control and an infinite tape which is initialized with the input and terminates the computation with the output. The tape contains a moving head, and at each step of computation the machine decides what to do – move to a different control state, move the head left or right, modify the current cell of the work tape – only as a function of its current control state and the contents of the tapes where the heads point to. The most important property of this definition is that *computation is local*: What happens next is determined by applying some local rule to what we have computed up to now. At some point the machine goes into a special halting state and the computation is over.

For decision problems, we require that at the end of the computation the output tape contains either the symbol 0 (for "no", or "reject") or the symbol 1 (for "yes", or "accept"). We say that machine M *decides* decision problem $L : \{0,1\}^n \to \{0,1\}$ if for every input x, $M(x) = L(x)$, namely the machine always halts with output $L(x)$.

The most important resource we will be interested in is the *computation time* of the machine, namely number of steps it takes for a machine on a given input to reach the halting state. Later we will also talk about *computation space*, the number of cells that the machine inspects on a given input.

In complexity theory we are interested in proving lower bounds on the amount of resources required

for a Turing Machine to produce its output. For instance we might aim to prove something like this:

> "For every Turing Machine with 1000 states there exists a SAT instance of size 50 on which the machine fails to find a satisfying assignment in time 10^6."

However attempts to prove concrete statements of this type have not met with much success, nor have they yielded particularly interesting insights on the nature of computational hardness. It is much more common to reason about computations *asymptotically*; we see what happens to the running time of computations not for particular inputs, but as the size of the input grows. So we typically look at statements like this:

> "For every Turing Machine there exists a sufficiently large number n and a SAT instance of size n on which the machine fails to find a satisfying assignment in time $O(n^5)$."

I find this to be a somewhat unsatisfactory aspect of complexity theory; perhaps the size n of the instance for which the statement might hold is much too large to be of any practical significance. However at the present stage of development these are the only kinds of questions that we have a hope at answering. The asymptotic nature of the questions and theorems of complexity theory is not something that reflects our objectives and desires, but merely our current level of understanding.

3 P and NP

The next thing we need is a notion of efficiency. The convention in complexity theory is to consider a computation efficient if the running time of the computation is bounded by some polynomial of the input size. Such Turing Machines are said to run in polynonmial time.

There are several reasons for choosing polynomial time as a notion of efficiency. First, we want to do better than exhaustively running through all possible solutions, a procedure that typically takes time exponential in the size of the inputs. Polynomials grow slower than exponentials, so an algorithm that runs in polynomial time must be using something about the structure of the problem in question that allows it to sidestep the "brute force" procedure of going through all possible solutions.

Of course, we could also consider more restrictive or more liberal notions of efficiency. However if the notion is too restrictive – for instance linear time – then it becomes too dependent on the details of our model of computation. Polynomial time does not seem to suffer from this problem. The *Cobham-Edmonds thesis* (also known as the *extended Church-Turing thesis*) states that any reasonable model of computation can be simulated on any other with a slowdown that is at most polynomial in the size of the input. On the other hand more liberal notions – for instance time $n^{O(\log n)}$ – do not appear to capture too many "natural" computations outside polynomial time; we don't seem to gain much by considering these as efficient.

There is another reason why polynomial-time appears to be a good notion of efficiency. In practice we sometimes compose algorithms by feeding the output of one algorithm as an input to another.

Polynomial-time computation behaves well under this kind of operation. If both algorithms run in polynomial time, so will their composition.

It is important to keep in mind that not all computations that can be done in polynomial time are considered efficient in practice. Rather, the choice of polynomial time as a notion of efficiency is an operational choice that allows us to reason about tractability versus hardness without thinking too much about details of the computational model we are using.

In the setting of decision problems, this notion of efficient computation is captured by the class P. The class P consists of all decision problems that can be decided by a Turing Machine that runs in polynomial time.

Many of the problems we are interested in solving are described by NP-relations. We say a relation R is an NP-relation if it is efficiently computable and it describes a search problem whose solutions are short (if they exist). In other words,

1. There is a polynomial-time algorithm that, on input (x, y), decides if $(x, y) \in R$, and
2. There is a polynomial p such that if $(x, y) \in R$, then $|y| \leq p(|x|)$.

If $(x, y) \in R$, then y is called a *witness* or *certificate* for x. For instance in the SAT problem, x is a boolean formula and y is a satisfying assignment for x; thus y witnesses the fact that x is satisfiable.

A decision problem L is an NP decision problem if L is the decision version of some NP-relation R. The class NP consists of all NP decision problems. Formally:

Definition 1. The class NP consists of all decision problems L for which there exists a polynomial-time Turing Machine V and a polynomial p such that

$$x \in L \iff \exists y, |y| \leq p(|x|) \text{ s.t. } V(x, y) \text{ accepts.}$$

Sometimes we call V *the verifier*, as its task is to verify that the NP-relation holds.

There is an alternative description of NP in terms of an imaginary computational device called a nondeterministic Turing Machine (NTM). This machine contains a special tape called a nondeterminism tape which is one-way read-only and is initialized at the beginning of the computation. We say that NTM N accepts its input x if there exists a setting of the nondeterminism tape that makes $N(x)$ accept, and we say N rejects x otherwise. The running time of N on input x is the maximum running time of N over all possible settings of the nondeterminism tape. Sometimes we will call the contents of the nondeterminism tape a *computation path*. Intuitively, the nondeterminism tape specifies which one of several possible "paths" the computation should follow.

It is not hard to see that $L \in$ NP iff L is decided by some polynomial-time NTM. The nondeterminism tape of N corresponds to the witness y for x, and N computes the NP-relation $R(x, y)$.

The nondeterministic Turing Machine is not a realistic model of computation. However it is a useful tool for describing and reasoning about problems in the class NP. We will also see other settings where introducing imaginary, unrealistic devices will help us gain insight about computations.

4 NP completeness

Many problems in the class NP appear to be difficult in the same way: There is no better way of finding a solution than doing an exhaustive search over all possible witnesses. NP-completeness gives a partial explanation of this phenomenon. We will focus on decision problems; a similar notion can also be defined for search problems.

For two decision problems A and B, we say A (polynomial-time) reduces to B, denoted by $A \leq B$, if there is a polynomial-time computable function f (a reduction) such that $x \in A$ if and only if $f(x) \in B$.[1] So if $A \leq B$ and $B \in \mathrm{P}$, then $A \in \mathrm{P}$. We say a decision problem L is NP-complete if $L \in \mathrm{NP}$ and $A \leq L$ for all $A \in \mathrm{NP}$. So an NP-complete problem is in P if and only if $\mathrm{P} = \mathrm{NP}$.

NP-complete problems are abundant, and this is a remarkable phenomenon. However their mere existence is interesting already. Here is an example:

BH (BOUNDED HALTING): Given a triple $(\langle N \rangle, x, 1^t)$, where N is an NTM, does N accept x in time at most t?

Claim 2. BH *is* NP*-complete.*

Proof. Showing that $\mathrm{BH} \in \mathrm{NP}$ requires a bit of technical work. We have to design a NTM for BH that simulates t steps of Turing Machine N (which is provided to us explicitly as an input) in time polynomial in $|\langle N \rangle| + |x| + t$. Any reasonable way of doing the simulation will run within this time bound. We won't worry about the details.

We now show that every NP decision problem A reduces to BH. Let A be an NP decision problem and let N be a non-deterministic Turing Machine that decides A. Suppose A runs in time at most $p(n)$ on inputs of length n, where p is some polynomial. The reduction f will then map instance x of A to instance $(\langle N \rangle, x, 1^{p(|x|)})$ of BH. It is easy to see that $f(x)$ can be computed in time $O(|\langle N \rangle| + x + p(|x|))$, which is polynomial in the input length $|x|$.

The correctness of the reduction is straightforward: By definition of A, $x \in A$ if and only if N accepts x in time $p(|x|)$, and by definition of BH this is the case if and only if $(\langle N \rangle, x, 1^{p(|x|)}) \in \mathrm{BH}$. □

The NP-completeness of BH is interesting, though for us it will be much more convenient to work with other NP-complete problems. In a few lectures we will show that $\mathrm{BH} \leq \mathrm{SAT}$. Since reductions are transitive, we obtain the following important consequence.

Theorem 3 (Cook and Levin)**.** SAT *is* NP*-complete.*

5 The class coNP

The class NP represents decision problems whose solutions, if they exist, can be checked efficiently. How about those instances of NP problems that do *not* have solutions? We saw two examples of decision problems in NP, SAT and MATCHING. Now let us look at the "opposite" problems:

[1] Clearly, $|f(x)|$ is polynomially bounded in $|x|$.

UNSAT: given boolean formula ϕ, does ϕ have no satisfiable assignment?

NO-MATCHING: given a graph G, does it have no perfect matching?

These problems are obtained by flipping the "yes" and "no" instances of problems in NP: UNSAT describes those formulas that are not in SAT, and NO-MATCHING describes those graphs that are not in MATCHING. The class coNP consists of all such problems:

Definition 4. The class coNP consists of those languages L such that $\overline{L} \in$ NP.

Let us compare the decision problems SAT and UNSAT. Since SAT is in NP, given any boolean formula, we can always provide a (efficiently verifiable) certificate that the formula is satisfiable: The certificate is simply the satisfying assignment. But what if the formula is not satisfiable? Do we still expect have a "certificate" that this is the case?

Consider, for instance, the formula:

$$(x_1 \vee \overline{x_2}) \wedge (x_1 \vee x_3 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (x_2).$$

This formula is not satisfiable for the following reason: The clauses $(x_1 \vee \overline{x_2})$ and $(\overline{x_1} \vee \overline{x_2})$ can only be simultaneously satisfied if x_2 is false, while the clause (x_2) requires x_2 to be true. So no matter which assignment we choose, the formula will not be satisfied.

We can think of this reasoning as providing a certificate that the above formula is not satisfiable. Is it possible to provide such a certificate for *every* unsatisfiable formula? If we allow the certificates to have length exponential in the size of the formula, the answer is yes. However, it is not known whether we can do so with polynomial length certificates that are verifiable in polynomial time. Most computer scientists believe that in general such certificates do not exist.

Now let's look at UNMATCHING: Can we get a certificate that a graph does not have a perfect matching? Here the answer is yes: Tutte's theorem tells us that a graph has no perfect matching *if and only if* there exists a subset of vertices S such that after removing S and all its incident edges, the rest of the graph has more than $|S|$ connected components with an odd number of vertices. So the set of vertices S is a certificate that the graph has no perfect matching: This set is certainly of polynomial size, and once we have S the conclusion of Tutte's theorem can be verified in polynomial time.

Actually, there is a more brutal way to certify that a graph has no perfect matching: Run Edmonds' perfect matching algorithm on the graph. If the algorithm does not find a perfect matching, we can take this as a certificate that the perfect matching does not exist – if it did exist, the algorithm would have found it.

These examples illustrate the relationship between the classes P, NP, and coNP. To state the conclusions in general, we need an alternative description of coNP. Unwinding the definition, we have that L is a coNP language if

$$x \in L \iff x \notin \overline{L} \iff \forall y, |y| \leq p(|x|) \colon V(x, y) \text{ rejects.}$$

for some polynomial-time TM V and some polynomial p. Now let U be the TM that runs V and flips its answer – if V accepts then U rejects, and vice versa. We get the following description of coNP:

Claim 5. *The class* coNP *consists of all decision problems L for which there exists a polynomial-time Turing Machine U and a polynomial p such that*

$$x \in L \iff \forall y, |y| \leq p(|x|) \colon U(x, y) \text{ accepts.}$$

It is now clear that P $\subseteq$ coNP: If L is in P, it has a polynomial-time TM M, which can in particular play the role of U by simply ignoring the second part of its input y. Therefore, MATCHING is in coNP simply because it is in P.

Is SAT in coNP? We said that, in general, we do not know how to efficiently verify that formulas are unsatisfiable. However, we can provide some evidence why we do not expect this to be the case: If it is, then we can get certificates for every problem in coNP, i.e., NP = coNP:

Theorem 6. *If* SAT $\in$ coNP*, then* NP = coNP*.*

Proof. Assume SAT $\in$ coNP. So there is a polynomial-time TM U and a polynomial p such that

$$\phi \in \text{SAT} \iff \forall w, |w| = p(|\phi|) \colon U(\phi, w) \text{ accepts.}$$

Let L be any problem in NP. We show that $L \in$ coNP. Since SAT is NP-complete, L reduces to SAT. Now consider the TM U' that on input (x, w) (where x is a potential instance of L), first runs the reduction on x to produce ϕ, then outputs $U(\phi, w)$. Clearly U' runs in polynomial time, and $|\phi| \leq q(|x|)$, where q is some polynomial. By the correctness of the reduction

$$\begin{aligned} x \in L &\iff \phi \in \text{SAT} \\ &\iff \forall w, |w| \leq p(|\phi|) \colon U(\phi, w) \text{ accepts} \\ &\iff \forall w, |w| \leq p(q(|x|)) \colon U'(x, w) \text{ accepts} \end{aligned}$$

so $L \in$ coNP. It follows that NP $\subseteq$ coNP. Automatically coNP $\subseteq$ NP because for every L:

$$L \in \text{coNP} \implies \overline{L} \in \text{NP} \implies \overline{L} \in \text{coNP} \implies L \in \text{NP}$$

so NP = coNP. □

6 EXP and NEXP

One way to interpret the question "does P equal NP?" is to ask if every nondeterministic polynomial time computation can be simulated in deterministic polynomial time. As we do not know the answer to this, we can ask an easier question: What is the most efficient deterministic simulation of NP that we can come up with? One way to simulate NP deterministically is to go over all possible witnesses y, and accept if any of them makes the verifier V accept. On an input of length n, the length of each witness can be as large as $p(n)$, so going over all possible witnesses requires time $2^{p(n)}$. For each of these witnesses, the verification $V(x, y)$ will take some polynomial time $t(n)$, so the total running time of this simulation is $t(n)2^{p(n)}$, which is exponential in n.

The class EXP is an exponential time analogue of P: Formally, EXP is the class of all decision problems that are solved by some Turing Machine whose running time is at most $2^{p(n)}$ on inputs of length n for some polynomial p. We just argued that NP $\subseteq$ EXP.

Therefore, P $\subseteq$ NP $\subseteq$ EXP. We believe that P $\neq$ NP. Is it possible that NP = EXP? While we cannot rule out this possibility, this is not believed to be true. One reason is that if NP = EXP, then coNP = EXP, so NP = coNP, which is unlikely.

Despite this, we cannot rule out with certainty either the possibility that P = NP, or the possibility that NP = EXP. However, we will see next that P $\neq$ EXP, so at least one of the statements P $\neq$ NP and NP $\neq$ EXP must be true.

The nondeterministic version of EXP is NEXP. This is the class of all decision problems that can be solved by some nondeterministic Turing Machine whose running time is at most $2^{p(n)}$ on inputs of length n for some polynomial p.

7 Time Hierarchy Theorems

Time hierarchy theorems formalize the intuition that given more time, a Turing Machine can do more things. These theorems can be made quite general, but the following case illustrates the main ideas.

Theorem 7 (Deterministic time hierarchy)**.** P $\neq$ EXP.

How do we go about proving this theorem? Intuitively, we need to construct a TM that runs in exponential time, and achieves some task that no polynomial-time TM can do. To do this, we will make the exponential-time TM simulate all polynomial-time TMs and output answers that are different from all of them.

One apparent obstacle is that we have only one exponential-time TM, but (possibly) infinitely many polynomial-time TMs to simulate. Wouldn't simulating infinitely many TMs take infinite (much more than exponential) time? The trick is to simulate the polynomial-time TMs one at a time, on different inputs.

Another issue is the running time: We want to simulate Turing Machines running in polynomial time, but we do not know how large their running time is – it could be n, n^3, or n^{100}. To be safe that we handle all these cases, we will run in time 2^n – which is large enough to encompass (asymptotically) all polynomial-time Turing Machines, yet allows us to carry out the simulation in exponential time.

Proof. We construct a TM D that runs in exponential time, but decides a different language from every polynomial time TM:

D: On input $(\langle M\rangle, x)$, where M is a TM,
Run M on input $(\langle M\rangle, x)$ for at most $2^{|x|}$ steps.
If M accepts, reject;
Otherwise (if M rejects or hasn't halted), accept.

On input $(\langle M\rangle, x)$, D will run in $O((|\langle M\rangle| + |x|)2^{|x|})$ steps, so its running time is exponential. Now we will argue that for every M that runs in polynomial time, the outputs of D and M differ on at least one input. Fix M and consider all inputs of the form $(|M|, x)$. Since M runs in polynomial time, for some (sufficiently long) x, the running time of M on input $(\langle M\rangle, x)$ will be bounded by $2^{|x|}$. For this x, the simulation of M on input $(\langle M\rangle, x)$ will terminate, and by definition of D, $D(\langle M\rangle, x) \neq M(\langle M\rangle, x)$. Therefore D and M differ on this input.

Since for every polynomial-time M, D and M differ on some input, it follows that D cannot decide any language in P. □

We have an analogous hierarchy theorem for nondeterministic time:

Theorem 8 (Nondeterministic time hierarchy)**.** NP $\neq$ NEXP.

To prove this theorem, we want to do something similar: Define a NTM that runs in exponential time and design it so it does something different from every polynomial time NTM. Let's attempt to define it like this:

S: On input $(\langle N\rangle, x)$, where N is a NTM,
Run N on input $(\langle N\rangle, x)$ for at most $2^{|x|}$ steps.
If N accepts, reject;
Otherwise (if N rejects or hasn't halted), accept.

A second look at it reveals that this simulation does not quite do what we intended. Suppose N is a NTM that accepts input x. However, there could still be rejecting paths in the computation of N on input x. When S follows one of these paths, it will accept – so the goal that S and N behave differently won't be achieved.

In order to ensure that S does the opposite of N, we have to design S in such a way that S rejects its input whenever at least one computation path of S accepts. To make sure this is the case, S will need to simulate N on all possible computation paths. How many such paths are there? If the input of S has length n, then N has 2^{n^c} computation paths for some constant c. Simulating N on all these paths takes time roughly 2^{n^c}. But remember that c can be an arbitrarily large constant: The running time of S is not bounded by any fixed exponential in n.

To resolve this issue we do the following trick. Let's fix an input y to N of length n. To simplify the discussion, assume y is the string 1^n. Suppose that, somehow, we could guarantee that $N(1^n) = N(1^{\log n})$. Then to make $S(1^n) \neq N(1^n)$, it is sufficient for S to simulate N on input $1^{\log n}$ and output the opposite answer. But now the number of computation paths of N on input $1^{\log n}$ is at most $2^{(\log n)^c}$, which is much smaller than 2^n (for large n); so S can carry out the simulation in time 2^n.

But how can we guarantee that $N(1^n) = N(1^{\log n})$? In fact we won't guarantee this, but we will design S in such a way that either $N(1^n) = N(1^{\log n})$, or $S(1^k) \neq N(1^k)$ for some k between $\log n$ and $n-1$. In either case, S and N will behave differently on some input.

Proof. We design a NTM S that runs in exponential time, but differs from every polynomial-time NTM N on at least one input:

S: On input $(\langle N \rangle, 1^n)$, where N is a NTM
  If n a *special number*
    Simulate $N(\langle N \rangle, 1^{\log n})$ for n steps deterministically on all computation paths.
    If N accepts on some path, reject; otherwise, accept.
  Otherwise,
    Simulate $N(\langle N \rangle, 1^{n+1})$ non-deterministically for 2^n steps.
    If N accepts (on this path), accept; otherwise, reject.

The "special numbers" are defined by the sequence $n_0 = 1$, $n_i = 2^{n_{i-1}+1}$. First, let us convince ourselves that N runs in exponential time. Testing if n is special can be done in time exponential (in fact linear) in n. If n is special, the deterministic simulation takes time $O(|\langle N \rangle| 2^n)$, as there are at most 2^n computation paths to cover. If n is not special, the simulation takes time $O(|\langle N \rangle| 2^n)$. The running time of N is exponential in either case.

Now let N be an arbitrary polynomial-time NTM. We will show that $S(\langle N \rangle, 1^k) \neq N(\langle N \rangle, 1^k)$ for some k. Let n be a sufficiently large special number. Notice that n is the only special number in the range $[\log n, n]$. If n is sufficiently large, then $N(\langle N \rangle, 1^{\log n})$ will halt (on all computation paths) within at most n steps, so because n is special we have

$$S(\langle N \rangle, 1^n) \neq N(\langle N \rangle, 1^{\log n}). \tag{1}$$

We consider two cases. In the first case, for some k in the range $[\log n, n-1]$, $S(\langle N \rangle, 1^k) \neq N(\langle N \rangle, 1^k)$, and we are done. In the second case, $S(\langle N \rangle, 1^k) = N(\langle N \rangle, 1^k)$ for every k in the range $[\log n, n-1]$. When n is sufficiently large, for every k in this range the nondeterministic simulations of $N(\langle N \rangle, 1^{k+1})$ for 2^k steps all terminate, so

$$\begin{aligned} N(\langle N \rangle, 1^{\log n}) &= S(\langle N \rangle, 1^{\log n}) = N(\langle N \rangle, 1^{\log n+1}) \\ &= S(\langle N \rangle, 1^{\log n+1}) = N(\langle N \rangle, 1^{\log n+2}) \\ &= \dots \\ &= S(\langle N \rangle, 1^{n-1}) = N(\langle N \rangle, 1^n) \end{aligned}$$

so by (1), $S(\langle N \rangle, 1^n) \neq N(\langle N \rangle, 1^n)$. □