SE 3310b

Theoretical Foundations of Software Engineering

Introduction to Computational Complexity. The Classes **P** and **NP**

Aleksander Essex

Western Engineering

Intro to Computational Complexity

In the previous lectures we concerned ourselves with issues of decidability and recognizability, i.e., what is computable in *theory*? We now turn our attention to a related subject: what is computable in *practice*?

Computational complexity is an area of study that asks a simple question: *how the computational requirements of a problem grow relative to input size*? From this point forward, we'll be focusing on problems that are decidable. As we'll see, however, solvable in theory does not imply solvable in practice.

Western Engineering

Computational Requirements

What do we mean by *computation requirements*? There are a number of ways you could quantify the requirements of computing:

- Time
- Memory
- Circuit complexity
- Cost, etc

Our main focus for the rest of the course will be ***time***, i.e., how does the running time of an algorithm grow relative to the size of an input argument?

Western Engineering

Running Time

Running Time

How do we measure running time? We define it in terms of how many *steps* a TM takes to decide the problem:

Definition 1 (Running time)**.**

The running time (or time complexity) of a deterministic Turing machine (DTM) M is the function $t : \mathcal{N} \to \mathcal{N}$, where $t(n)$ is the maximum number of steps M takes on an n-character string.

Notice that our definition of running time is framed in terms of the *worst case*.

Western Engineering

Big-Oh Notation

Let's take a moment to review Big-Oh notation.

Definition 2 (Big-Oh Notation)**.**

Let f, g be functions. We say $f(n) = O(g(n))$ if there exists a constant c and threshold n_0 such that for every $n \geqslant n_0$ we have:

$$f(n) \leqslant cg(n)$$

In other words $f(n) = O(g(n))$ if g grows as fast or faster than f.

Western Engineering

Consequences of Big-Oh

Suppose we have an algorithm A that solves a problem in $O(n)$ time. That is a linear increase in input size results in a linear increase in run time. Now consider another algorithm B that solves the same problem in $O(2^n)$ time. Here a linear increase in input results in an exponential increase in run time.

Question: which algorithm is faster? Well a linear function clearly grows more slowly than an exponential function, right? But from definition, all we are guaranteeing is that there is some crossover point n_0 such that for $n \geqslant n_0$. But what if all the real-world instances of the problem exist in a region where $n < n_0$?

Western Engineering

Consequences of Big-Oh

Consequence: Just because the complexity of one algorithm (or problem) is less than another, doesn't mean its faster to solve in all cases. There are situations (though not many) where an exponentially growing algorithm is faster than a linear growing algorithm for real-world instances.

So, while there are counter examples where a "hard" problem might be easier to solve than an "easy" problem, ultimately we're focused on the hardness of the problem ***in general***.

Time Complexity

Definition 3 (DTIME)**.**

Let $t : \mathcal{N} \to \mathcal{R}^+$ be a function. We define the time complexity class **DTIME**$(t(n))$ to be the collection of all languages that are decidable by a DTM in $O(t(n))$ steps.

Western Engineering

Some Common Running Times

Here are some common forms of $t()$ (i.e., running times):

Name	Running time	Example
Constant	$O(1)$	Integer is even?
Logarithmic	$O(\log(n))$	Binary search
Linear	$O(n)$	Smallest item in array
Linearithmic	$O(n\log(n))$	Merge sort
Quadratic	$O(n)$	Bubble sort
Polynomial	$O(\text{poly}(n))$	Greatest common divisor
Sub-exponential	$O(2^{\log(n)})$	Factoring
Exponential	$O(2^{\text{poly}(n)})$	Brute-force search

Western Engineering

Basic Complexity Classes

Definition 4 (The class **P**)**.**

The complexity class **P** is defined as all problems solvable by a DTM in polynomial time, i.e.,

$$\mathbf{P} = \bigcup_{k \geqslant 0} \mathbf{DTIME}(n^k)$$

Western Engineering

Basic Complexity Classes

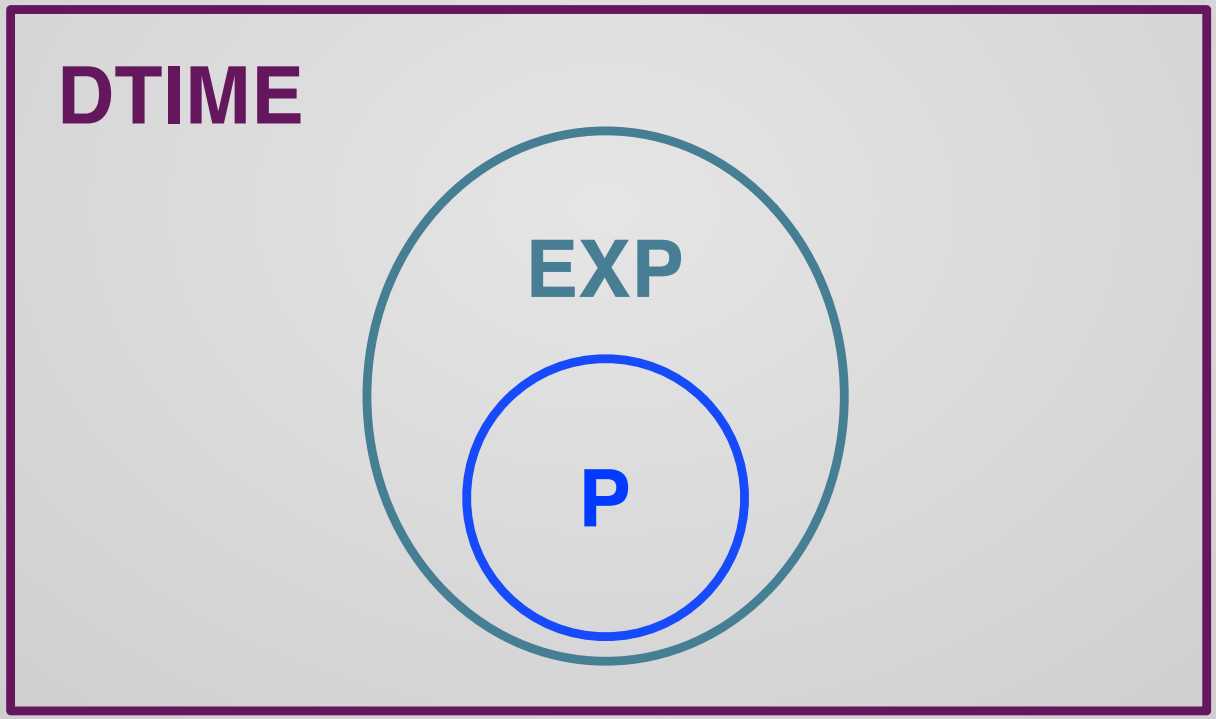
Definition 5 (The class **Exp**)**.**

The complexity class **Exp** is defined as all problems solvable by a DTM in exponential time, i.e.,

$$\mathbf{Exp} = \bigcup_{k \geqslant 0} \mathbf{DTIME}(2^{n^k})$$

Western Engineering

Relationship Between Basic Classes



Western Engineering

Non-deterministic Time

So far we've considered the running time of deterministic Turing machines. What about non-deterministic ones? Recall a non-deterministic Turing machine (NTM) M decides a language L if for all $w \in \Sigma^*$:

1. Every path M takes on w halts,
2. $w \in L$ iff *at least* one path accepts,
3. $w \notin L$ iff every path rejects.

Western Engineering

Non-deterministic Time

M decides L in non-deterministic time t iff for all w, M takes *at most*

Definition 6 (Running time)**.**

The running time (or time complexity) of a non-deterministic Turing machine (NTM) M is the function $t : \mathcal{N} \to \mathcal{N}$, where $t(n)$ is the maximum number of steps M takes in every computational path on any n-character string.

Western Engineering

Non-deterministic Time

Definition 7 (NTIME)**.**

Let $t : \mathcal{N} \to \mathcal{R}^+$ be a function. We define the time complexity class **NTIME**$(t(n))$ to be the collection of all languages that are decidable by an NTM in $O(t(n))$ steps.

Western Engineering

Basic Complexity Classes

Definition 8 (The class **NP**)**.**

The complexity class **NP** is defined as all problems solvable by an NTM in polynomial time, i.e.,

$$\mathbf{NP} = \bigcup_{k \geqslant 0} \mathbf{NTIME}(n^k)$$

Western Engineering

Basic Complexity Classes

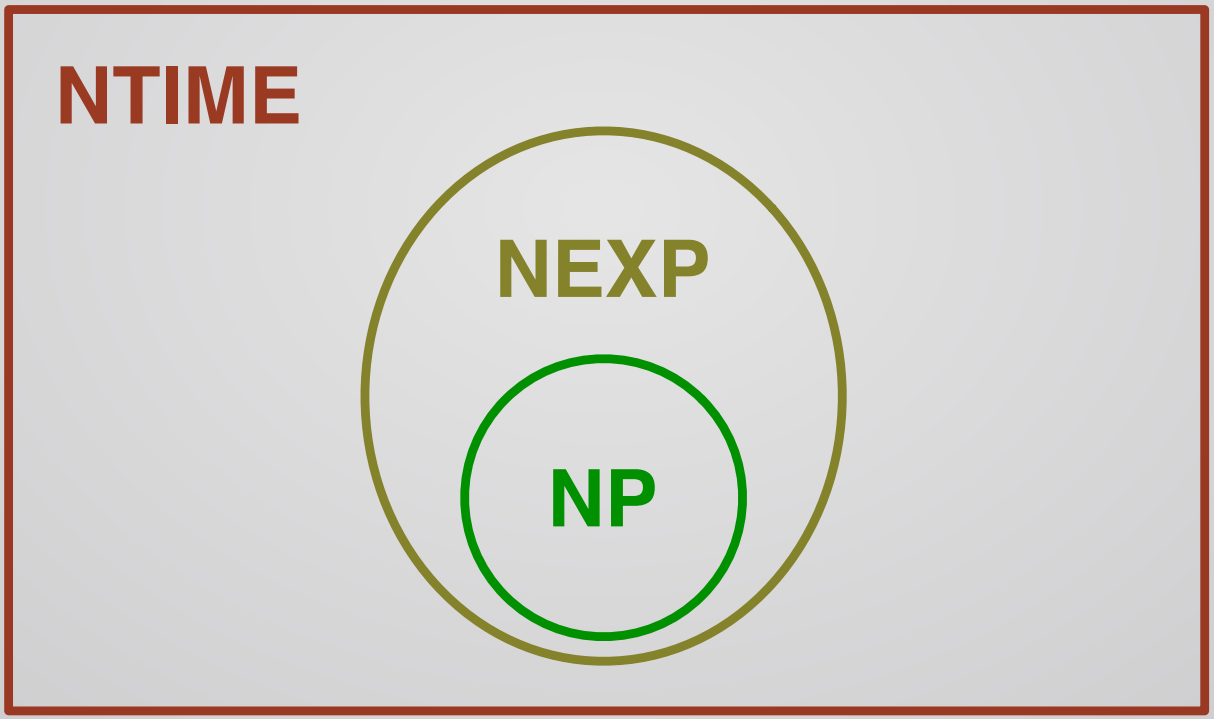
Definition 9 (The class **NEXP**)**.**

The complexity class **NEXP** is defined as all problems solvable by a NTM in exponential time, i.e.,

$$\mathbf{NEXP} = \bigcup_{k \geqslant 0} \mathbf{NTIME}(2^{n^k})$$

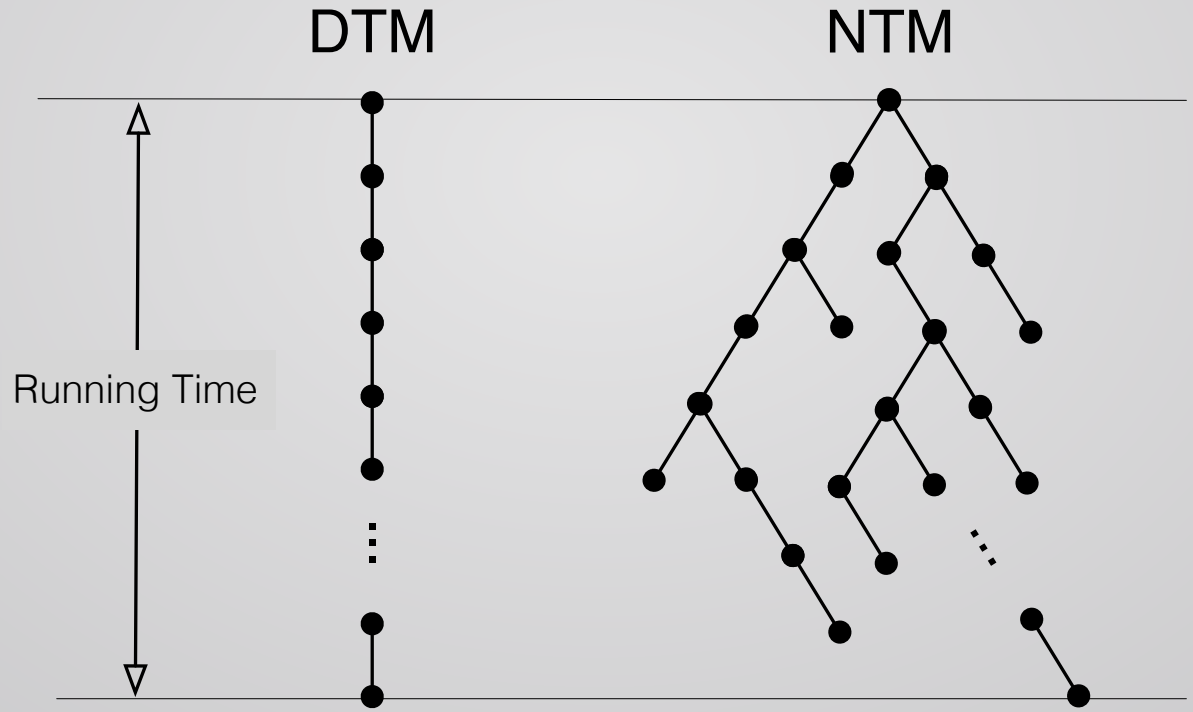
Western Engineering

Relationship Between Basic Classes



Western Engineering

Running Time: DTMs vs NTMs



Western Engineering

Language Verifiers

A ***verifier*** for a language L is an algorithm V where

$$L = \{w : V \text{ accepts } < w, c > \text{ for string c}\}$$

The verifier uses this extra information c, called a "***certificate***" (or proof of membership) to decide if $w \in L$.

NP is the class of languages that have polynomial-time verifiers.

Western Engineering

Language Verifiers

Consider the following example of a language verifier:

$$FACTOR = \{< n, f > : n \text{ has a factor less than or equal to } f\}$$

Here the certificate c would be the factors p, q, and the verifier algorithm V would be one that (a) checks $pq \cdots = n$ and that $0 < p < n$.

Clearly V runs in polynomial time, so checking a solution to $< n, f > \in FACTOR$ is efficient.

Western Engineering

P and **NP** Explained

In other words, the classes **P** and **NP** can be characterized as follows:

- **P** is the class of problems that can be solved in polynomial time (i.e., **efficiently solvable**)
- **NP** is the class of problems that can be verified in polynomial time (i.e., **efficiently checkable**)

Western Engineering