In a previous lecture, we introduced the class P but we didn’t say much about it.

In particular, we didn’t say why it was an interesting class to study.

Let’s look at some of the reasons.
Why P is interesting

Almost every problem that people think of as efficiently solvable lies in P, for example:

- sorting
- graph reachability
- minimum spanning tree
- shortest paths in graphs
- addition, subtraction, multiplication, and division of numbers
- greatest common divisor
- primality testing
- matrix multiplication
- determinant of a matrix
- linear programming (but the simplex algorithm doesn’t always run in polynomial time)
On the other hand, virtually no problem that we think of as “intractable” is known to be polynomial-time computable, for example:

- existence of Hamiltonian cycle in a graph
- travelling salesperson problem
- prime factorization of integers
- permanent of a matrix
- discrete logarithm
We do *not* want our notion of polynomial time to depend on the particular model of Turing machine we pick.

Luckily, something running in polynomial time using one model of Turing machine runs in polynomial time on the others, with one possible exception.

This includes changing the number of tracks, number of tapes, whether tape is one-sided or two-sided, etc.

Thus the definition of polynomial time is *robust*, and is another reason to consider P as a “natural” class.

The one possible exception is nondeterminism. We do not know if $P = NP$, although most people believe they are different.
Encoding and traps to avoid

There is one sticky point. We associate decision problems with the language of encodings of their “yes” instances.

But which encodings? There are infinitely many possibilities.

The issue is that if we choose a ridiculous encoding, we can turn a hard problem into an easy one.
For example, consider integer factorization of a number $n$. Here the obvious way to encode $n$ is in binary, so the number $n$ can be encoded in $\lceil \log_2 n \rceil$ bits.

So when we ask for a polynomial-time algorithm for integer factorization, we \textit{really} mean an algorithm that runs in polynomial time in $\log_2 n$.

If we choose a less suitable encoding (for example, encoding the integer $n$ in unary by $a^n$) then even simple algorithms like trial division by every integer up to $n$ can factor integers in “polynomial time”, because now we have $O(n^k)$ time to factor $n$.

This is generally not an issue because it is usually clear what kind of encoding is appropriate for a given problem.
We can extend our notion of polynomial time to functions.

We say that a function $f : \Sigma^* \rightarrow \Delta^*$ is *polynomial-time computable* if there is a DTM running in polynomial time such that

$$(q_0, \omega) \xrightarrow{\star} (q_{\text{acc}}, \omega) f(x).$$

**Theorem.** If $f, g$ are both polynomial-time computable functions, then so is their composition $g \circ f$. 
Proof. Let $M_1$ compute $f$ in polynomial time, say bounded by $O(n^k)$ time, and let $M_2$ compute $g$ in polynomial time, say bounded by $O(n^\ell)$ time.

To compute $g \circ f$ on input $x$ of length $n$, first run $M_1$ on $x$, getting $y = f(x)$ on the tape at the end. This takes $O(n^k)$ time.

Now run $M_2$ on the result, getting $g(y) = g(f(x)) = (g \circ f)(x)$ on the tape at the end. This takes $O(m^\ell)$ time, where $m = |f(x)|$.

Now we need to know how long $f(x)$ can be! But since the DTM $M_1$ ran in $O(n^k)$ time, it can write out at most $O(n^k)$ symbols. So $m = O(n^k)$.

So the total time is $O(n^k) + O((n^k)^\ell)$, which is still polynomial time in $n$. 
If $f$ is polynomial-time computable, and we use it in a loop that has polynomially-many iterations, then the result runs in polynomial time (provided we don’t expand the input size).

Example of what can go wrong: if we repeatedly set $x := xx$, $n$ times, then the result is $x^{2^n}$, which can’t even be written down in polynomial time.