In CS, we're concerned w/ solving problems with a computer.

A problem for a computer must be defined precisely + unambiguously by its input and its desired output.

ex sort an array
  input: array + way to compare elements
  output: sorted array

  compute the factorial of a positive int
  input: $n \in \mathbb{Z}^{>0}$
  output: $n!$

Note that we need the tools of discrete math to define these inputs/outputs precisely!

A solution is some method of taking in an arbitrary input and computing an an output w/ desired properties defined by the problem.

Typically this method is an algorithm, a sequence of steps you can perform to get from input to output.

In practice, this is a mix of precise + unambiguous notation and some words for intuition. We call this mix pseudocode.

ex for the factorial problem above:

```
fact(n):
   if n=1 then
      return 1
   else
      return n· fact(n-1)
```

For any algorithm, you should ask yourself:

1. Does the alg. actually work? i.e., does it give the correct output for every valid input?

Proving this is a focus of later courses... but we certainly need discrete math to do it! And we can do it now...

<u>ex</u> The recursive algorithm fact computes $n!$ $\forall n \geq 1$.

<u>pf</u> For pos. integer n, let $P(n)$ denote the property that $fact(n) = n!$. We prove by mathematical induction that $\forall n \geq 1 : P(n)$.

base case $(n=1)$: $fact(1)$ returns 1 and $1! = 1$

inductive case: we WTS $\forall n \geq 2 : P(n-1) \Rightarrow P(n)$.

Assume $P(n-1)$. That is, $fact(n-1)$ returns $(n-1)!$. WTS $fact(n) = n!$. Note that:

$$
\begin{aligned}
fact(n) &= n \cdot fact(n-1) && \text{by def. of fact} \\
&= n \cdot (n-1)! && \text{by IH} \\
&= n! && \text{by def. of !}
\end{aligned}
$$

$\square$

2. Does the alg. work efficiently?

ex for the array sorting problem:

```
Sort (A):
  let S = the set of all orderings of A's elts
  for x in S:
    if x is sorted:
      return x
```

Is this alg efficient? No — when elts of A are distinct, $|S| = \text{length}(A)!$ ← factorial

we focus on runtime in this class.

How to measure runtime?

idea #1: implement alg., run it, time it.
- depends on software, hardware, os, ...
- implementation takes time + is error prone
- what input do we run it on?

idea #2: ① find a function that expresses alg's runtime as a function of input size

→ # of primitive operations: arithmetic ops, logical ops, variable retrieval + assignment, etc

② use big O to represent the function, so that we can get a bigger-picture idea of the runtime and compare it to other algs

let's see some examples of ①

ex How many primitive ops does the following pseudocode snippet do?

```
for i=1 to i=n do
  for j=1 to j=n do
    sum = sum + i·j
```

6? 3? doesn't matter. O(1)

$$\sum_{j=1}^{n} 1 = n \qquad \sum_{i=1}^{n} n = n^2$$

$O(n^2)$

ex

```
for r=1 to n=1 do
  for c=1 to m do
    p[r][c] = r + c
```

O(1)

$$\sum_{c=1}^{m} 1 = m \qquad \sum_{r=1}^{n} m = mn$$

$O(mn)$

ex

```
for x=1 to x=n do
  for y=1 to y=n do
    foobar()
```

O(runtime of fobar)

$O(n^2 \cdot$ RT of fobar)

ex

```
for i=1 to i=n do
  for j=i to j=n do
    sum = sum + i·j
```

O(1)

$$\sum_{j=i}^{n} 1 = n-i+1$$

$O(n^2)$

$$\sum_{i=1}^{n} (n-i+1)$$

$$= \sum_{i=1}^{n} n - \sum_{i=1}^{n} i + \sum_{i=1}^{n} 1$$

$$= n^2 - \frac{n(n+1)}{2} + n$$

ex for i=1 to i=n do
for j=1 to j=n do
for k=1 to k=n do
— something O(1) —

$O(n^3)$

ex while n>1 do
n=n-1] O(1)] O(n) times do O(1) ops

$O(n)$

ex while n>1 do
n=n/2] O(1)] O(log n) times do O(1) ops

$O(\log n)$