Quick Review

\( n \in O(n^{100}) \)

\( g \in O(f) \) \quad \text{“} g \leq f \text{”} \quad g(n) \leq c f(n)

\( g \in \Omega(f) \) \quad \text{“} g \geq f \text{”} \quad g(n) \geq c f(n)

\( g \in \Theta(f) \) \quad \text{“} g = f \text{”} \quad c_1 f(n) \leq g(n) \leq c_2 f(n)

\( e.g. \ f(n) = n \quad g(n) = \Theta(n) \)

Two properties (for more, see posted handout)

1. \( \forall f, g, h : \mathbb{N} \to \mathbb{R}^{\geq 0}, \)

   \( f \in \Theta(g) \land g \in \Theta(h) \Rightarrow f \in \Theta(h) \)

2. \( \forall f : \mathbb{N} \to \mathbb{R}^{\geq 0}, \) \( (\forall n \in \mathbb{N}, f(n) \geq 1) \Rightarrow \)
Back to analysing code

Goal: find an approximate number of steps that a program takes, as a function of input size, in the long-term.

Input size: number of bits (0 or 1) required to represent the input.

Often in this course, we'll approximate this in different ways, depending on type of input and context.

"Step" or "basic operation": Any block of code whose runtime
doesn't depend on input size, e.g. “constant time”, $\Theta(1)$ time.

E.g. 
1. comparison $\text{==, <, >}$
2. arithmetic $\text{+, -, *, /, \%}$
3. variable assignment $x = 5$
   and lookup $x + x$
4. print
5. return

```
Ex

def f(n):
    x = n
    print(x * 2)
    return x + 3
```

What parts of the code depend on $n$?

Now, we can treat the entire

have a runtime that

which
None. We can treat the entire function body as a single step, so the runtime is 1. This is \( \Theta(1) \).

```python
def f2(n):
    for i in range(10):
        print(n)
```

⚠️ When you have a loop, be careful!

Count the # of iterations and cost per iteration.

- There are 10 iterations in total \( (i = 0, 1, \ldots, 9) \)
- The cost for a single iteration is \( n^2 \) step. (print’s runtime is 1) (and updating i)

So the total cost is \( 10 \times 1 = 10 \) steps,
\[
10 \times 2 = 20
\]
So the total cost is \(10 \cdot 1 - 10\) step, which is \(\Theta(1)\).

```python
def f3(n):
    for i in range(n):
        print(n)
```

- There are \(n\) iterations in total
- Each iteration takes \(1\) step.

So total cost is \(n \times 1 = n\) steps, which is \(\Theta(n)\).

\[ n \times 2 = 2n \]

Nested loops

```python
def nested1(n):
    i = 0
    while i < n:
        for j in range(n):
            print(n)
for $j \in \text{range}(n)$:
    $i = i + 2$
    print($i + j$)

Start by analysing the cost of the inner loop for one iteration of the outer loop.

For one iteration of the outer loop, the inner loop takes $n$ steps.

So the total cost of one iteration of the outer loop is $n$ (we ignore the cost of $i = i + 2$ here).

The number of iterations of the outer loop is $\lceil \frac{n}{2} \rceil$.

So the total cost is $n \times \lceil \frac{n}{2} \rceil \in \Theta(n^2)$. 
```python
def nested2(n):
    i = 0, 1, ..., n-1
    for i in range(n):
        for j in range(i):
            print(i+j)
```

Same as before, work from inside-out

For a fixed iteration of the outer loop, the inner loop takes \(i\) steps.

\[\text{inner loop has } i \text{ iterations, and } 1 \text{ step per iteration}\]

So the cost for an iteration of the outer loop is \(i\) steps. The cost depends on the iteration!

The total cost is \(\sum_{i=0}^{n-1} \frac{1}{(n-1)^i}\)
\[
\sum_{i=1}^{n-1} \left( \frac{(n-1)i}{2} \right)
\]

\[
0 + 1 + 2 + 3 + 4 + \cdots + (n-1)
\]

So the total \# of steps is \( \frac{(n-1)n}{2} \), which is \( \Theta(n^2) \)

def factor(n):
    d = 2
    while d < n:
        if n % d == 0:
            return d
        d = d + 1
    return -1

* Loop might stop early!
We don’t have a “nice” formula for # of iterations.

New goal: find bounds on the # of iterations.

There are at most ? iterations and at least $\frac{1}{\text{}}$ iterations (there are infinitely many cases where there’s only 1 iteration).