Algorithm time complexity

- **Goal:** Given algorithm/program, want approximate measure of # steps executed, as a function of input size.

- **Input size:** Formally, "standard size" = total # bits to represent input fully in binary.
  - Informally, we simplify through assumptions.
    1) Integers require a fixed # bits
    2) Size of list = len(list)  
      —careful with nesting!
    3) For algorithms whose input is a natural number — "size" = value of number (exception)

- **Steps:** Any block of code whose running time does not depend on input size

**Ex:**
- Arithmetic ops: +, -, *, /, %
- Comparisons: <, <=, >, >=, =, !=
- Assignment: \( x = \text{expr} \)  
  \( \frac{1 \text{ step}}{1 \text{ step}} \)
- Print

* Function call, return
  \( \Rightarrow \) only the call itself, not function execution
def f(n: int) -> int:
    x = n + 1
    print(x % 2)
    return x * n

What makes runtime grow?

- Loops
- Function calls — time to execute the function
- Recursive functions
- Complex compound data structures (linked lists, trees, Python dictionaries, etc.)

Loops!

* In general:

for i in lst:
    body
while cond:
    body

need to work out:
- How long the body takes — runtime
- How many iterations
- Multiply — based on assumption:
  runtime of body remains the same for each iteration; if not... more work!
EX: For $i$ in range($n$):
  for $j$ in range($n$):
    print($i+j$)

Iterations? $\Theta(n)$

Time? $\Theta(n)$

Overall $\Theta(n^2)$

EX: For $i$ in range($n/2$):
  print($i$)

$x = 3*n$

For $i$ in range($n$):
  for $j$ in range($n$):
    print($i+j$)

$\Theta(n^2)$

$\Theta(n+1+n^2) = \Theta(n^2)$

EX: For $i$ in range($n$):
  for $j$ in range($i$):
    print($i+j$)

Note: Inner loop takes time that depends on value of $i$ — cannot simply multiply fixed runtime for body by $n$ iterations

(*) We’ll see how to work around this later.
Approach: count more carefully! "unroll" the outer loop —
turn n body into \( \sum_{i=0}^{n-1} \) body

outer loop becomes:

```python
for j in range(n):
    i = 0
    print(i+j)
for j in range(n):
    i = 1
    print(i+j)
    ...
for j in range(n):
    i = n-1
    print(i+j)
```

\[ \text{\# iterations overall:} \]
\[ \frac{n-1}{\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}} \]

body is \( \Theta(1) \)
overall \( \Theta(\frac{n^2}{2} - \frac{n}{2}) = \Theta(n^2) \)
Ex 1: `def is_prime(n: int): # pre: n ≥ 2`

```python
    for d in range(2, n):
        if n % d == 0:
            return False
    return True
```

Q: exact # iterations depends on n

**Upper bound (Θ)**:

for every n, # iterations ≤ n-2

so running time ∈ O(n)

**Lower bound (Ω)**:

for every n≥3, # iterations ≥ 1 (when n even)

so running time ∈ Ω(1)

---

Notation: `RT_is_prime(n)` denotes the running time function for algo. is_prime on inputs of “size” n.
Ex 2: def print_primes(n: int):
    for k in range(2, n+1):
        if is_prime(k):
            print(k)
        # iterations = n-1

Overestimating & underestimating

- upper bound:
  - time for loop body \( \leq c \cdot k \leq c \cdot n \)
  - total \( O(n \cdot n) = O(n^2) \)

  * danger is too much overestimating might lead to a "loose" bound

- lower bound:
  - time for loop body \( \geq c \)
  - total is \( \Omega(c \cdot n) = \Omega(n) \)

More careful analysis

- upper bound:
  \[
  RT_{pp}(n) = \sum_{k=2}^{n} RT_{ip}(k)
  \leq \sum_{k=2}^{n} c \cdot k
  \leq c \cdot \frac{n(n+1)}{2}
  \]
  - \( RT_{ip}(k) \in O(k) \)

  So \( RT_{pp}(n) \in O(n^2) \)
lower bound:

\[ RT_{pp}(n) = \sum_{k=2}^{n} RT_{ip}(k) \]

\[ \geq \sum_{2 \leq k \leq n \atop k \text{ is prime}} RT_{ip}(k) \geq \sum_{2 \leq k \leq n \atop k \text{ is prime}} c \cdot k \]

We need external facts from number theory:

- there are approx. \( \frac{n}{\log n} \) many primes \( \leq n \)

\[ \sum_{k \text{ prime} \atop k \leq n} k \approx \frac{n^2}{\log n} \]

so \( RT_{pp}(n) \in \Omega \left( \frac{n^2}{\log n} \right) \) ... 

Q: actual \( RT_{pp}(n) \)? \( \text{open} \)