Recall our goal:
Given an algorithm, find an approximate # of steps it takes, as a function of input size, as the input size gets large.

More precisely:
Given an algorithm, find a simple function $f: \mathbb{N} \rightarrow \mathbb{R}^+$ where the # of steps is $\Theta(f)$. 

A step is any block of code that takes constant time — $\Theta(1)$.

E.g.
1. Arithmetic $\;+,-,\times$ etc.
2. Comparisons $\;==,\;>,$ etc.
3. Variable lookup and assignment $\;y = x + 5$
4. Return statements $\;\text{return } x+5$

Sources of non-constant running time
1. Loops
2. Helper functions

> CSC165
② Helper functions
③ Operations on data structures (e.g. Python list)
④ Recursion

Today’s example

Analyse the running time of the following function (in terms of its input n).

```python
def f(n: int) -> int:  # Assume n ≥ 0.
    r = 0
    for i in range(10):  # Loop 1
        for j in range(n * n):  # Loop 2
            r = r + j
    for i in range(n // 2):  # Loop 3
        for j in range(i * i):  # Loop 4
            r = r + j
    return r
```

Analysis
First, lines 2 and 9 take constant time,
First, lines 2 and 9 take constant time, so we'll count them as 1 step.

**Analysis of lines 3-5 (Loops 1 and 2)**

For Loop 2:
- Loop 2 has \( n^2 \) iterations 
  \( (j=0,1,2,\ldots,n^2-1) \)
- Each iteration takes 1 step.

So the total number of steps is:

\[
\underbrace{1+1+\ldots+1}_{n^2 \text{ times}} = n^2
\]

for one iteration of Loop 1.

For Loop 1:
- It has 10 iterations
  \( (i=0,1,2,\ldots,9) \)
- Each iteration takes \( n^2 \) steps.

So the total number of steps for Loop 1 is:

\[
\frac{n^2+n^2+\ldots+n^2}{10 \text{ times}} = 10n^2
\]
10 times

Analysis for lines 6-8 (Loops 3 and 4)

For Loop 4:

1. It has \(i^2\) iterations 
   \((j=0,1,\ldots,i^2-1)\), 
   where \(i\) is the loop variable 
   for Loop 3.

2. It takes 1 step per iteration.

So the total # of steps is 
\[
1 + 1 + \ldots + 1 = i^2, \\
\frac{i^2}{i^2 \text{ times}}
\]
for one iteration of Loop 3.

For Loop 3:

1. It takes \(\lfloor\frac{n}{2}\rfloor\) iterations 
   \((i=0,1,\ldots,\lfloor\frac{n}{2}\rfloor-1)\).

2. Each iteration takes \(i^2\) steps.

So the total # of steps is 
\[
0^2 + 1^2 + 2^2 + \ldots + \left(\lfloor\frac{n}{2}\rfloor-1\right)^2 \\
\frac{\lfloor\frac{n}{2}\rfloor-1}{x}
\]
\[
\sum_{i=0}^{\frac{n^2-1}{2}} i^2 = \frac{\left(\frac{n^2-1}{2}\right)\left(\frac{n^2}{2}\right)\left(2\left(\frac{n^2}{2}\right) - 1\right)}{6}
\]

Putting it all together

So the total \# of steps \# takes is:

\[
1 + 10n^2 + \frac{\left(\frac{n^2-1}{2}\right)\left(\frac{n^2}{2}\right)\left(2\left(\frac{n^2}{2}\right) - 1\right)}{6}
\]

\[\Theta(1) \quad \Theta(n^2) \quad \Theta(n^3)\]

\[\subseteq \Theta(n^3)\]

Recap Basic strategy for analysing runtime:

1. Determine an "exact" step count.
2. Conclude with a Theta expression.

Example Analyse the running time of:

```
1  def is_prime(n: int) -> bool:
2    # Precondition: n ≥ 2
```
```
# Precondition: n = x.

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

return True
```

**Analysis (assuming n > 2)**

For the loop:

1. Each iteration takes 1 step
2. The # of iterations is
   - at most \( n-2 \) \((d=2, 3, \ldots, n-1)\)
   - at least 1 \((d=2)\)

So in total the loop takes
   - at most \( \frac{1+1+\ldots+1}{n-2} \) = \( \frac{n-2}{n-2} \) steps.
   - at least \( \frac{1}{1} = 1 \) step.
Line 6 takes 1 step (constant time) but may or may not execute, so we say it is at most 1 step and at least 0 steps.

So the total # of steps is:
- at most \( n-2+1 = n-1 \in \Theta(n) \)
- at least \( 1+0 = 1 \in \Omega(1) \)

Alternate analysis strategy
1. Find an exact upper bound on runtime
2. Conclude a simple Big-Oh, say \( O(f) \)
3. Find an “exact” lower bound on runtime
4. Conclude a simple Omega, say \( \Omega(f) \)
Two possibilities:
- If $f_1 = f_2$, you can conclude $\Theta(f_1)$. We call $O(f_1)$ and $\Omega(f_2)$ tight bounds here.
- If $f_1 \neq f_2$ ....
  • It might be impossible to find a simple Theta expression
  • Or, might need to do a "better" analysis...

**Example**

Analyse the runtime of:

def print_primes(n: int) -> None:
    for i in range(2, n):
        if is_prime(i):
            print(i)

**Analysis** (assume $n \geq 2$)

For the loop:
  1. The # of iterations is $n-2$
     ($i = 2, 3, \ldots, n-1$)
② Each iteration takes
- at most \( i-1 \) steps
- at least 1 step
(from our analysis of is-prime)

So the total cost of the loop is
- at most \( \sum_{i=2}^{n-1} i-1 \)

... calculation ...
\( O(n^2) \)

- at least \( 1+1+\ldots+1 = n-2 \) steps,
which is \( \Omega(n) \).

Better lower bound

The actual total runtime is
\[
\sum_{i=2}^{n-1} R_{\text{is-prime}}(i) \leq \text{the real or exact runtime of is-prime}
\]
\[
\geq \sum R_{\text{is-prime}}(i)
\]
\[ \sum_{2 \leq i \leq n-1} (i-1) \]

(special case for prime numbers)

\[ \Omega \left( \frac{n^2}{\log n} \right) \]

BIGGEST EXTERNAL FACT