

Lab 3

# Big-O Notation & Object-Oriented Programming

# Part 1: Asymptotic Analysis

## Big-O $O(f(n))$

We say  $g(n) = O(f(n))$  if there exist constants  $c > 0$  and  $n_0 \geq 0$  such that

$g(n) \leq c \cdot f(n)$  for all  $n \geq n_0$ .

Interpretation:  $f(n)$  is an asymptotic upper bound on  $g(n)$ .

## Big- $\Theta$ $\Theta(f(n))$

$g(n) = \Theta(f(n))$  means  $g$  is both  $O(f(n))$

and  $\Omega(f(n))$ . Tight bound.

## Small-o $o(f(n))$

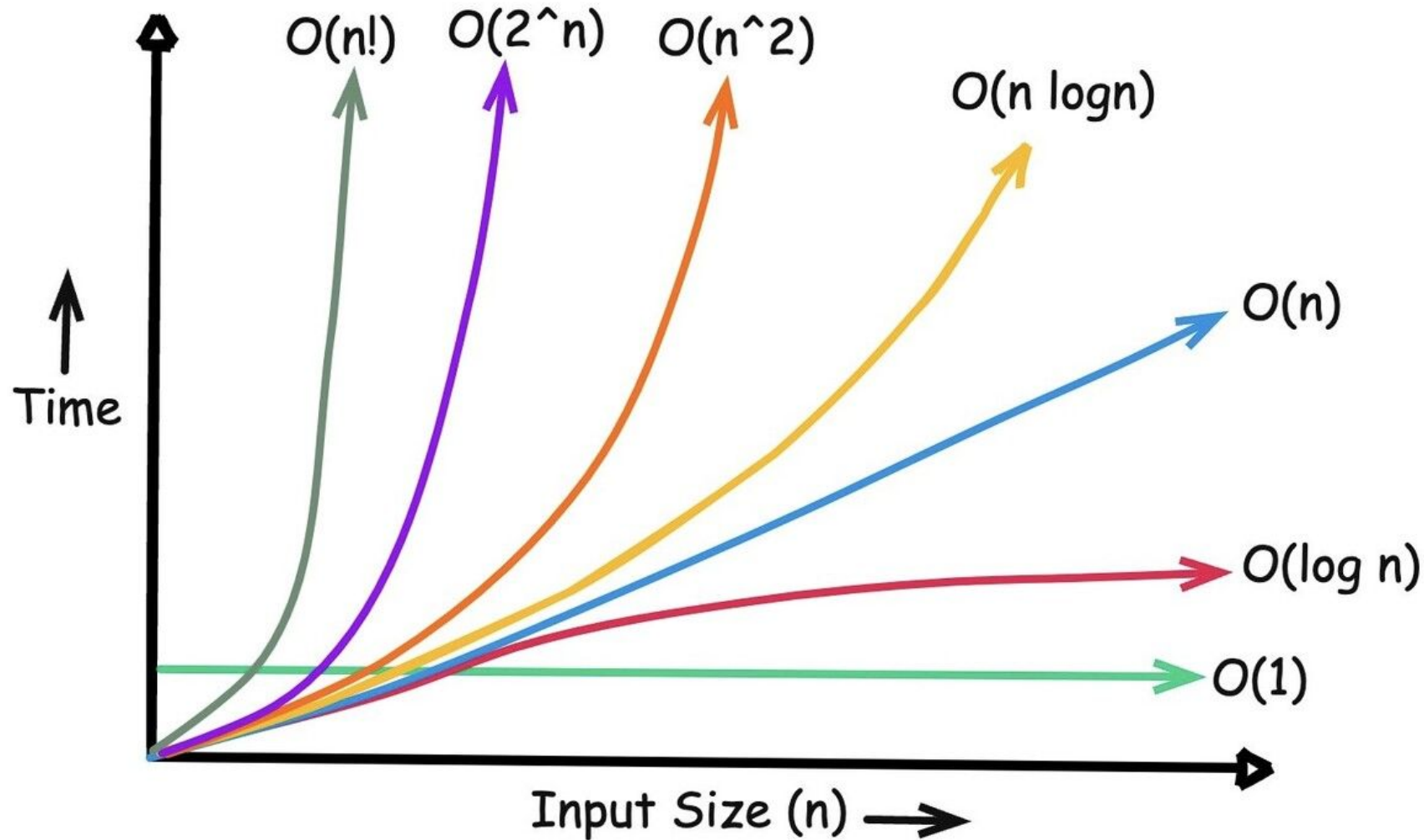
$g(n) = o(f(n))$  means for all  $c > 0$ ,

$g(n) < c \cdot f(n)$  eventually. Strict upper bound.

*Big-O = upper bound ( $\leq$ ),*

*Big- $\Theta$  = tight bound ( $=$ )*

*Small-o = strict upper bound ( $<$ )*



## Example 1

Prove the following:

$$5n^2 + 3n \log n + 2n + 5 = O(n^2)$$

## Example 1: Prove that $5n^2 + 3n \log n + 2n + 5 = O(n^2)$

Strategy: Show that each term is  $O(n^2)$ , then sum them up.

For  $n \geq 1$ :

- $5n^2 \leq 5n^2 \rightarrow O(n^2) \checkmark$
- $3n \log n \leq 3n^2 \rightarrow O(n^2) \checkmark$  (since  $\log n \leq n$ )
- $2n \leq 2n^2 \rightarrow O(n^2) \checkmark$  (since  $n \leq n^2$ )
- $5 \leq 5n^2 \rightarrow O(n^2) \checkmark$  (since  $1 \leq n^2$ )

Sum:  $5n^2 + 3n^2 + 2n^2 + 5n^2 = 15n^2$

$\therefore 5n^2 + 3n \log n + 2n + 5 \leq 15n^2$  for all  $n \geq 1$ .

## Example 2: Show that $2^{n+2} = O(2^n)$

**Key Insight:** Use exponent rules to factor out the constant.

$$2^{n+2} = 2^n \cdot 2^2 = 4 \cdot 2^n$$

$$\text{So: } 2^{n+2} \leq 4 \cdot 2^n \quad \text{for all } n \geq 0$$

Takeaway: Constant factors in the exponent just become a multiplicative constant — they don't change the Big-O class.

# Analyzing a Program: Primality Testing

## First approach

```
def is_prime(n):  
    if n <= 1:  
        return False  
    if n == 2:  
        return True  
    if n % 2 == 0:  
        return False  
  
    i = 3  
    while i < n:  
        if n % i == 0:  
            return False  
        i += 1  
  
    return True
```

## What's the Big-O?

The while loop runs from  $i = 3$  up to  $n - 1$ .  
Worst case:  $n$  is prime, loop runs  $n$  times.

**Running time:  $O(n)$**



## Can we do better?

*Think: Do we really need to check all the way up to  $n$ ?*

# Improved Primality Test

## Improved approach

```
def is_prime(n):  
    if n <= 1:  
        return False  
    if n == 2:  
        return True  
    if n % 2 == 0:  
        return False  
  
    i = 3  
    while i * i < n:          # ← key change!  
        if n % i == 0:  
            return False  
        i += 1  
  
    return True
```

## Why $\sqrt{n}$ is enough

If  $n = a \times b$  and both  $a, b > \sqrt{n}$ , then  $a \times b > n$  — contradiction!

So at least one factor must be  $\leq \sqrt{n}$ . We only need to check up to  $\sqrt{n}$ .

**Running time:  $O(\sqrt{n}) = O(n^{1/2})$**

**Q: What if we want to find all primes from 3 to  $n$ ? (i.e., call `is_prime`  $n$  times)**



# Finding All Primes from 3 to n

We call `is_prime(k)` for  $k = 3, 4, 5, \dots, n$

Number of calls:  $O(n)$

Cost per call:  $O(\sqrt{n})$  (worst case)

**Total:**  $O(n) \times O(\sqrt{n}) = O(n \cdot n^{1/2}) = O(n^{3/2})$

*"But for smaller values, you don't need to check all  $\sqrt{n}$  options!"*

That's true, but Big-O is an upper bound. We're bounding the worst case for each call, so  $O(n^{3/2})$  is correct.

Bonus: Better algorithms exist! The Sieve of Eratosthenes finds all primes up to  $n$  in  $O(N \log \log N)$ .

## Part 2:

# Object-Oriented Programming

Instead of built-in types like `int` or `float`, we'll look at how to build our own data type using classes, using rational numbers as the example.

# Review: OOP Concepts

## Encapsulation

Data and functionality can be stored within objects

## Polymorphism

one operation works on multiple data types in different ways

## Inheritance

Child classes extend parent class functionality

Simple example:

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        raise NotImplementedError("Subclass must implement abstract method")

class Dog(Animal):
    def speak(self):
        return f"{self.name} says Woof!"

class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow!"

dog = Dog("Buddy")
cat = Cat("Whiskers")

print(dog.speak())
print(cat.speak())
```

**Encapsulation:** *name* stored in class *Animal*, can be accessed when you call *speak()*

**Polymorphism:** *speak()* works on both Dog and Cat classes

**Inheritance:** Parent class *Animal* extending to children (Dog/Cat) classes

# Fraction Class (Rational Number)

## Why build our own data type?

Python already has built-in types like int or float.

But sometimes we want:

- Exact arithmetic (no floating point error)
- Custom behavior
- Control over representation

## What is a rational number?

A rational number consists of:

- A numerator
- A denominator
- A rule: denominator  $\neq 0$

Example:  $\frac{1}{2}$ ,  $\frac{3}{4}$ ,  $-\frac{5}{6}$

# Helper functions (gcd)

**gcd(a,b)**: Reduce fractions to simplest form

**lcm(a,b)**: Smallest number that is a multiple of two or more given numbers

```
def gcd(a,b):  
    if b > a:  
        a,b = b,a  
    while b > 0:  
        remainder = a % b  
        a = b  
        b = remainder  
    return a
```

```
def gcd_rec(a,b):  
    if b == 0:  
        return a  
    return gcd_rec(b, a % b)  
  
def gcd(a,b):  
    if b > a:  
        a,b = b,a  
    return gcd_rec(a,b)
```

```
def lcm(a,b):  
    return a * b // gcd(a,b)
```

Example with two numbers **24 & 18**:

- GCD: 6 → largest number that can evenly divide both of the numbers
- LCM: 72 → smallest number that both 12 and 18 can divide evenly

# Encapsulation: The Fraction Class

```
class Fraction:
    def __init__(self, numer, denom):
        self.numerator = numer
        self.denominator = denom
```

- A **class** defines a new data type
- Each object represents one rational number

`__init__` (constructor) responsibilities:

- Validate input
- Store instance variables

Instance variables (i.e. `self.var`):

- Each object has its own copy
- Data is stored inside the object

```
def __repr__(self):
    return f"{self.numerator}/{self.denominator}"
```

`__repr__` (display) responsibilities:

- Controls how the object is represented

```
def simplify(self):
    my_gcd = gcd(self.numerator, self.denominator)
    new_numerator = self.numerator // my_gcd
    new_denominator = self.denominator // my_gcd
    #self.denominator = new_denominator
    #self.numerator = new_numerator
    return Fraction(new_numerator, new_denominator)
```

# Polymorphism: Operator Overloading (Arithmetic)

Python dunder (magic) methods let custom types work with Python's built-in syntax: `+`, `*`, `<`, `==`, `print()`, for loops. By defining these methods, you can customize how your classes behave for common operations like object initialization, arithmetic, and string representation, a concept known as operator overloading.

`__mul__` → `Fraction * Fraction`

```
def __mul__(self, other):  
    new_number = self.numerator * other.numerator  
    new_denom = self.denominator * other.denominator  
    return Fraction(new_number, new_denom).simplify()
```

```
# Fraction(1,2) * Fraction(2,3)  
# → 2/6 → simplify → 1/3
```

`__add__` → `Fraction + Fraction`

```
def __add__(self, other):  
    my_lcm = lcm(self.denominator, other.denominator)  
    new_self_number = self.numerator * my_lcm // self.denominator  
    new_other_number = other.numerator * my_lcm // other.denominator  
    return Fraction(new_self_number + new_other_number, my_lcm)
```

```
# 1/4 + 1/3 → lcm=12 → 3+4 = 7/12
```

`f1 + f2` → Python calls `f1.__add__(f2)` → your method runs → returns new Fraction

*We're redefining what `+` means for Rational objects. Instead of modifying existing objects, we create and return a new one.*

# Polymorphism: Comparisons & Sorting

binary\_search only needs <, >, ==. Python sort algorithm is designed to only need less-than comparisons.

`__lt__` → Fraction < Fraction

```
def __lt__(self, other):
    my_lcm = lcm(self.denominator, other.denominator)
    new_self_number = self.numerator * my_lcm // self.denominator
    new_other_number = other.numerator * my_lcm // other.denominator
    return new_self_number < new_other_number
```

`__eq__` (==)

```
def __eq__(self, other):
    # same LCM pattern
    return new_self == new_other
```

`__le__` (<=)

```
def __le__(self, other):
    return (self < other or self == other)
```

**Payoff:** Python sort() works because `__lt__` is defined!

```
li = [Fraction(1,4), Fraction(1,3), Fraction(2,5), Fraction(7,10)]
li.sort()           # Python uses __lt__ to compare elements
print(li)           # [1/4, 1/3, 2/5, 7/10] ← __repr__ for display
```