Lab 3

Big-O Notation & Object-Oriented Programming

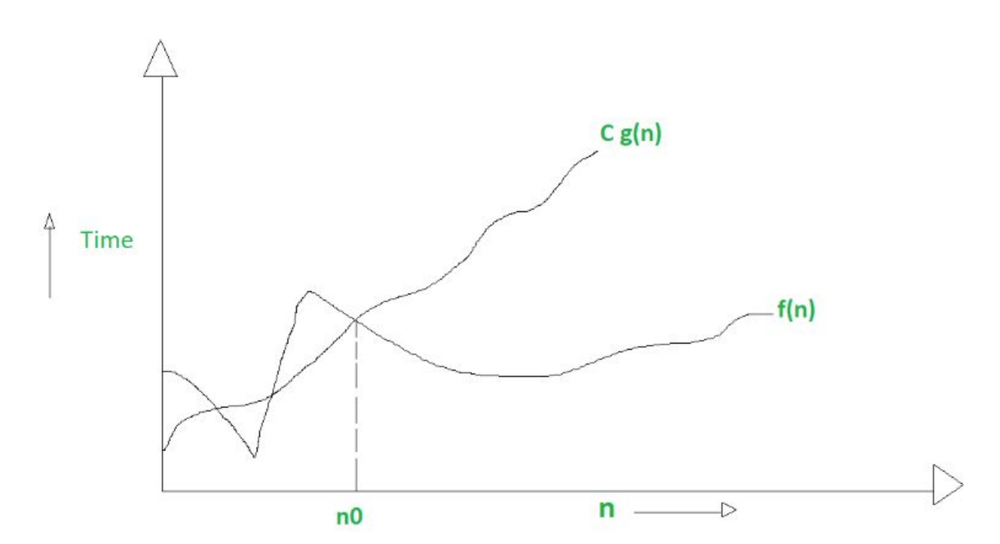
Part 1: Asymptotic Analysis

Big-O O(f(n))

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

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

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



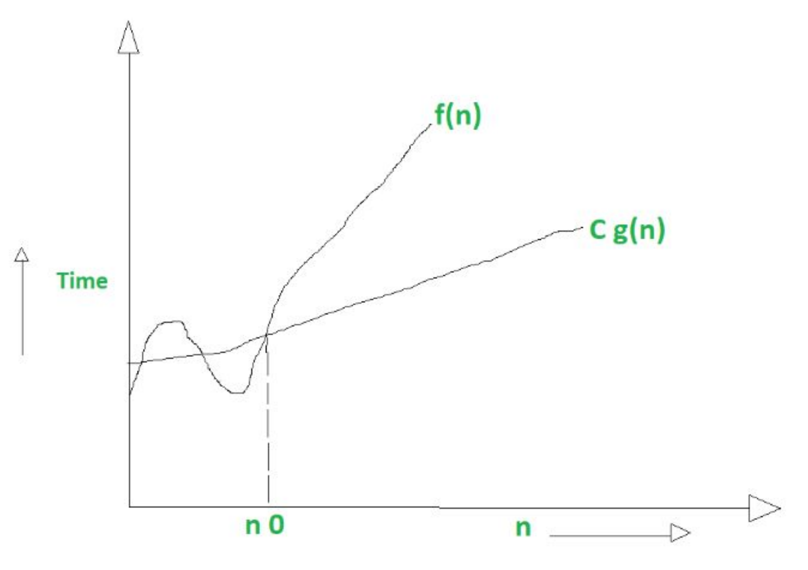
Part 1: Asymptotic Analysis

Big-Ω Ω(f(n))

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

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

Interpretation: **Lower bound** on order of growth of time taken by an algorithm or code with input size



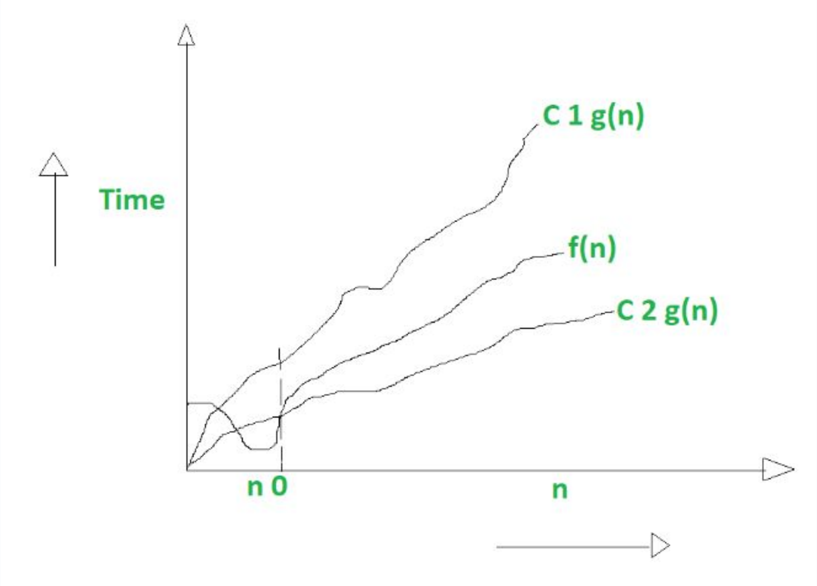
Part 1: Asymptotic Analysis

Big-Θ Θ(f(n))

f(n) = Θ(g(n)) means f is both O(g(n)) and Ω(g(n)). There exist positive constants C1 and C2 such that

f(n) is sandwich between C2g(n) and C1g(n)

Interpretation: defines **exact order of growth** of time taken by an algorithm or code with input size



Important Properties of Big O Notation

3. Constant Factor

For any constant c > 0 and functions f(n) and g(n), if f(n) = O(g(n)), then cf(n) = O(g(n)).

Example:

$f(n) = n$, $g(n) = n^2$. Then $f(n) = O(g(n))$. Therefore, $2f(n) = O(g(n))$.

4. Sum Rule

If f(n) = O(g(n)) and h(n) = O(k(n)), then f(n) + h(n) = O(max(g(n), k(n)) When combining complexities, only the largest term dominates.

Example:

$f(n) = n^2$, $h(n) = n^3$. Then , $f(n) + h(n) = O(max(n^2 + n^3) = O (n^3)$

5. Product Rule

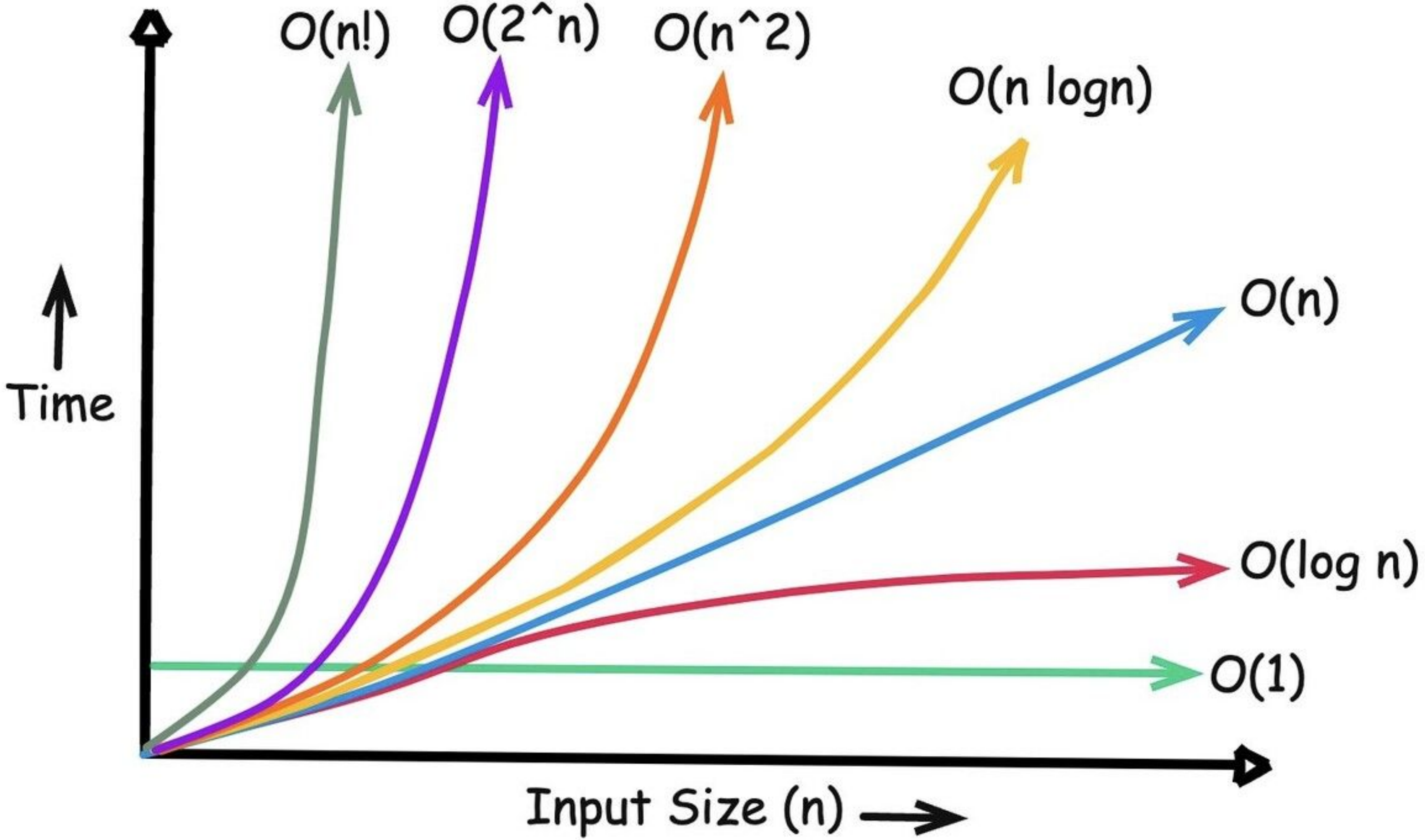
If f(n) = O(g(n)) and h(n) = O(k(n)), then f(n) * h(n) = O(g(n) * k(n)).

Example:

$f(n) = n$, $g(n) = n^2$, $h(n) = n^3$, $k(n) = n^4$. Then $f(n) = O(g(n))$ and $h(n) = O(k(n))$. Therefore, $f(n) * h(n) = O(g(n) * k(n)) = O(n^6)$.

Read more at:

https://www.geeksforgeeks.org/dsa/analysis-algorithms-big-o-analysis/



O(n!)
O(2^n)
O(n^2)
O(n logn)
O(n)
O(log n)
O(1)
Time
Input Size (n)

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 ≥ 1:

  • 5n²           ≤  5n²      →  O(n²)  ✓
  • 3n log n   ≤  3n²      →  O(n²)  ✓    (since n log n ≤ n)
  • 2n            ≤  2n²      →  O(n²)  ✓    (since n ≤ n²)
  • 5             ≤  5n²      →  O(n²)  ✓    (since 1 ≤ n²)

Sum:  5n² + 3n² + 2n² + 5n²  =  15n²
```

∴ $5n^2 + 3n \log n + 2n + 5 \le 15n^2$ for all $n \ge 1$.

Example 2

Prove the following:

$$2^{n+2} = O(2^n)$$

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

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

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

So: $2^{n+2} \leq 4 \cdot 2^n$ 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 √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, ..., n

Number of calls:        O(n)
Cost per call:          O(√n)    (worst case)

Total:  O(n)  ×  O(√n)  =  O(n · n¹/²)  =  O(n³/²)
```

"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:

```python
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: ½, ¾, -⅚

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

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

```python
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)
```

```python
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

```python
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

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

__repr__ (display) responsibilities:

- Controls how the object is represented

```python
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

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

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

__add__ → Fraction + Fraction

```python
def __add__(self, other):
    my_lcm = lcm(self.denominator, other.denominator)
    new_self_numer = self.numerator * my_lcm // self.denominator
    new_other_numer = other.numerator * my_lcm // other.denominator
    return Fraction(new_self_numer + new_other_numer, 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_numer = self.numerator * my_lcm // self.denominator
    new_other_numer = other.numerator * my_lcm // other.denominator
    return new_self_numer < new_other_numer
```

__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
```