INTRODUCTION TO ANALYSIS OF ALGORITHM

Data Structures and Algorithms

Waheed Iqbal

Department of Data Science, FCIT
University of the Punjab, Lahore, Pakistan

Let's solve a simple puzzle!

Apples and oranges boxes labeling

Three boxes are labeled Apples, Oranges, and Apples & Oranges. All of them are labeled incorrect. You need to correct the labels given that you are allowed to open only one box.

Algorithm

You know, what is the definition of Algorithm!

"In mathematics and computer science, an algorithm is a step-by-step procedure for calculations, data processing, and automated reasoning."

A Simple Algorithm

Pseudo-code of finding __________ of x[n]:

```
M = x[0];
for i = 1 to n-1 do
    if (x[i] > M)
        M = x[i];
    endif
end for
return M;
```

Let's try to think! how we may identify the performance of this algorithm?

Algorithm Performance Analysis

- Determining an estimate of the time and memory requirement of the algorithm.
- Time estimation is called time complexity analysis.
- Memory size estimation is called space complexity analysis.
- Because memory is cheap and abundant, we rarely do space complexity analysis.
- Since time is expensive, analysis now defaults to time complexity analysis.

Why Algorithm Analysis?

- As problem sizes get bigger, analysis is becoming *more* important.

- The difference between good and bad algorithms is getting bigger.

- Being able to analyze algorithms will help us identify good ones without having to program them and test them first.

Measuring Performance of Algorithm

Two main techniques to measure performance of algorithms:

- Empirical
- Analytical

Measuring Performance: Empirical Approach

Empirical Approach: Implement the code, run it, and time it (averaging trials)

- **Advantages**
  - No math!
- **Disadvantages**
  - Need to implement code
  - When comparing two algorithms, all other factors need to be held constant (e.g., same computer, OS, processor, load)
  - A really bad algorithm could take a really long time to execute and measure performance may take a lot of time
  - Specific measurements based on specific OS and Hardware

Measuring Performance: Analytical Approach

Analytical Approach: Analyze steps of algorithm, estimating amount of work each step takes and express it in mathematical form.

- **Advantages**
  - Independent of system-specific configuration
  - Good for estimating
  - Don't need to implement code
- **Disadvantages**
  - Won't give you info exact runtimes optimizations made by the computer architecture
  - Only gives useful information for large problem sizes
  - In real life, not all operations take exactly the same time (multiplication takes longer than addition) and have memory limitations

Analyzing Performance

General "rules" to help measure how long it takes to do things:

Basic operations	Constant time
Consecutive statements	Sum of number of statements
Conditionals	Test, plus larger branch cost
Loops	Sum of iterations
Function calls	Cost of function body
Recursive functions	Solve "recurrence relation" You will learn more about it in Analysis of Algorithm course in the next semester!

Statements Execution Count

```
statement1;
statement2;
statement3;
```

3

```
for (int i = 1; i <= N; i++) {
      statement4;
}
```

N

```
for (int i = 1; i <= N; i++) {
      statement5;
      statement6;
      statement7;
}
```

3N

4N + 3

Statement Execution Count (Cont.)

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
        statement1;
    }
}
```

N^2

```
for (int i = 1; i <= N; i++) {
    statement2;
    statement3;
    statement4;
    statement5;
}
```

$4N$

$N^2 + 4N$

•

Relative Rates of Growth

- Most algorithms' runtime can be expressed as a function of the input size *N*

- **Rate of growth**: measure of how quickly the graph of a function rises

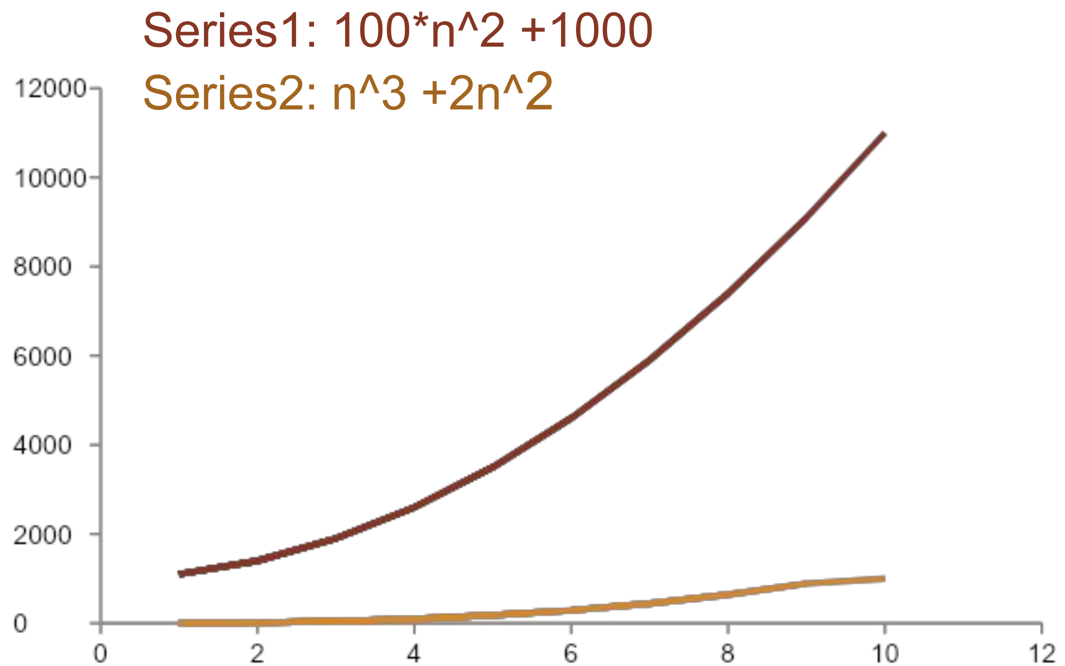
- Goal: distinguish between fast- and slow-growing functions
  - We only care about very large input sizes (for small sizes, almost any algorithm is fast enough)
  - This helps us discover which algorithms will run more quickly or slowly, for large input sizes

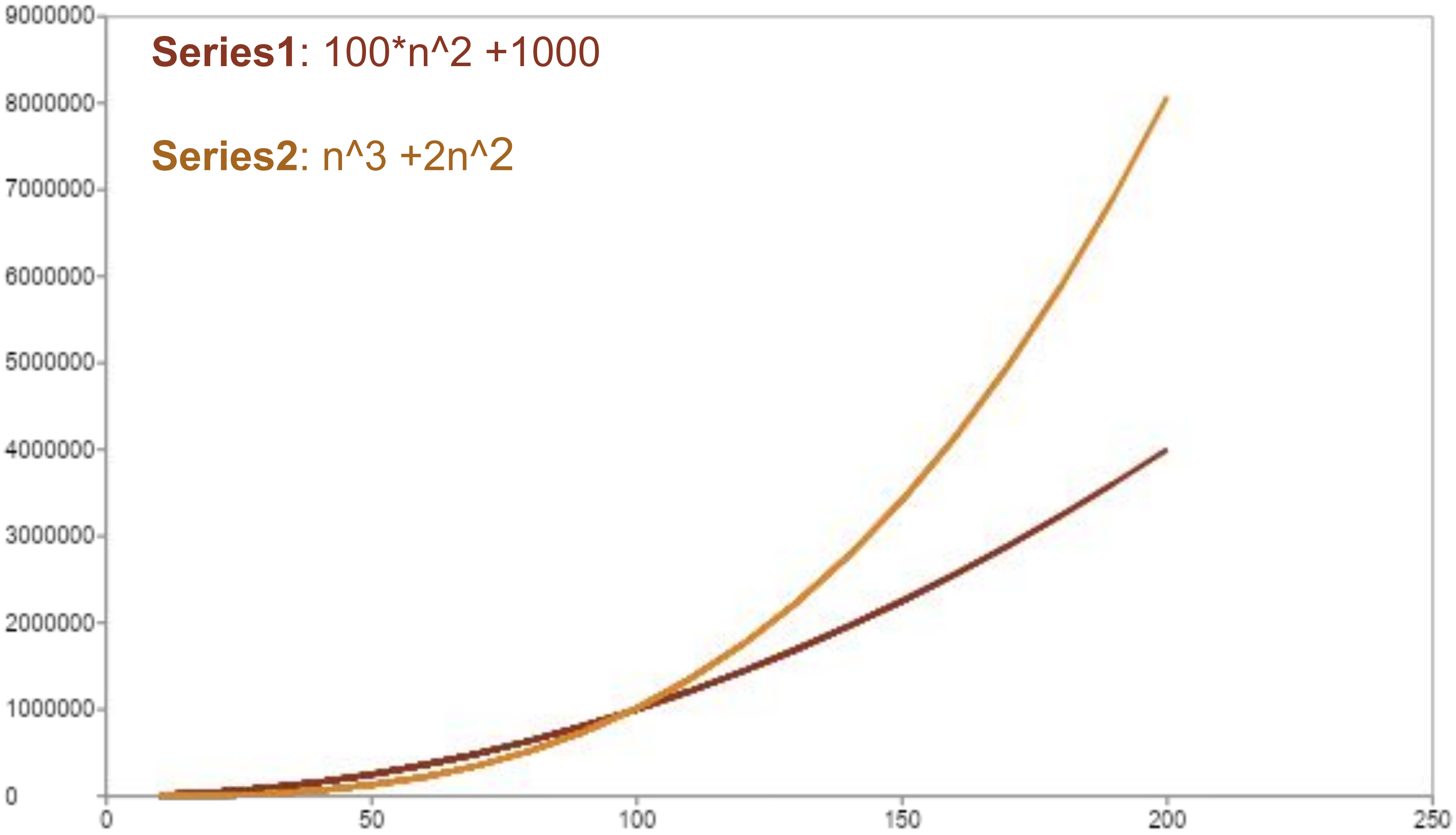
- Most of the time interested in **worst case performance**; sometimes look at best or average performance

Growth rate example

- Consider these graphs of functions.
  Perhaps each one represents an algorithm:
  $n^3 + 2n^2$
  $100n^2 + 1000$

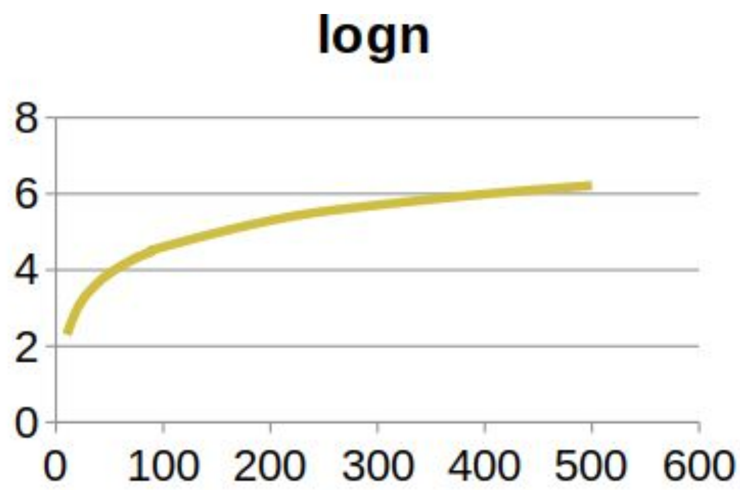
- Which grows faster?

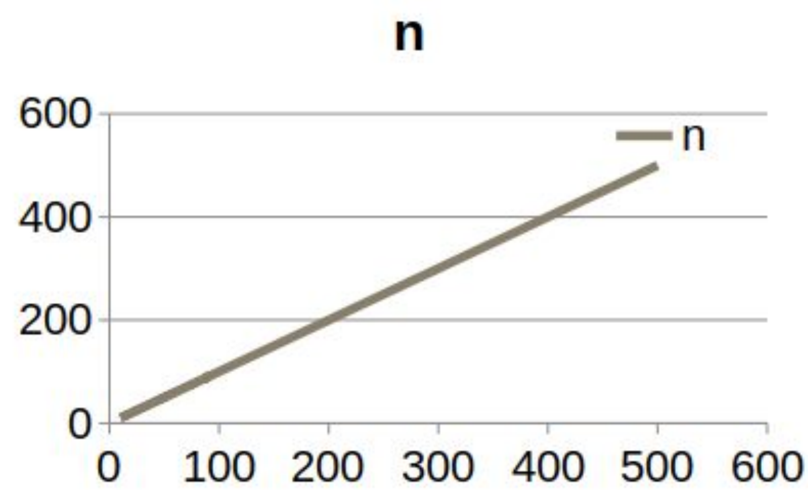


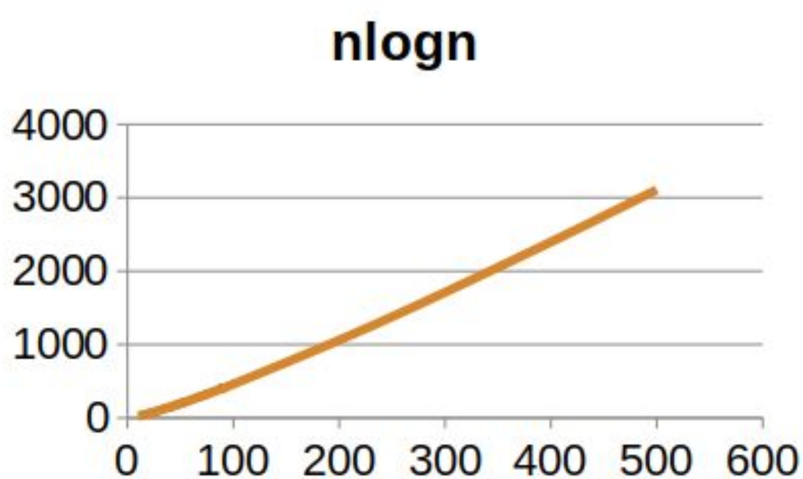


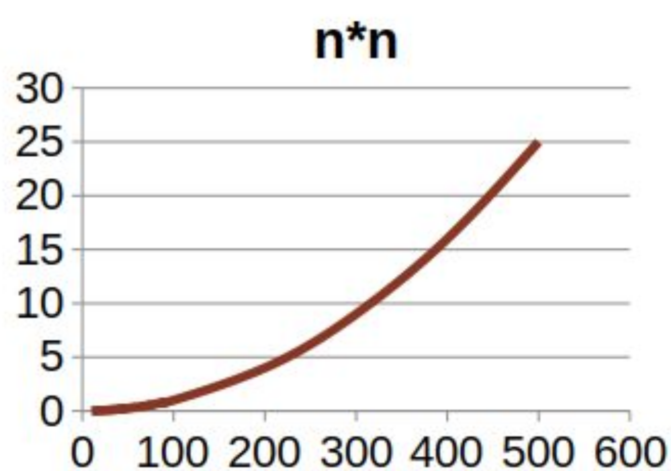
Series1: 100*n^2 +1000
Series2: n^3 +2n^2
9000000
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
0
50
100
150
200
250

Growth of Function









Cost of Basic Operations

operation	example	nanoseconds †
variable declaration	`int a`	c_1
assignment statement	`a = b`	c_2
integer compare	`a < b`	c_3
array element access	`a[i]`	c_4
array length	`a.length`	c_5
1D array allocation	`new int[N]`	$c_6 N$
2D array allocation	`new int[N][N]`	$c_7 N^2$
string length	`s.length()`	c_8
substring extraction	`s.substring(N/2, N)`	c_9
string concatenation	`s + t`	$c_{10} N$

Common Order of Growth Classification

order of growth	name	typical code framework	description	example
1	constant	`a = b + c;`	statement	add two numbers
log N	logarithmic	`while (N > 1)` `{ N = N / 2; ... }`	divide in half	binary search
N	linear	`for (int i = 0; i < N; i++)` `{ ... }`	loop	find the maximum
N log N	linearithmic	[see mergesort lecture]	divide and conquer	mergesort

N^2	quadratic	`for (int i = 0; i < N; i++)`
		`for (int j = 0; j < N; j++)`
		`{ ... }`
			double loop	check all pairs

N^3	cubic	`for (int i = 0; i < N; i++)`
		`for (int j = 0; j < N; j++)`
		`for (int k = 0; k < N; k++)`
		`{ ... }`
			triple loop	check all triples

2^N	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets

Asymptotic Analysis

High Level Idea

- Suppress constant factors and lower-order terms
  - Constant factors; too system dependent
  - Lower-order terms: irrelevant for large inputs

- Example: $\mathbf{an^2+bn+c}$ is just O($\mathbf{n^2}$)

Asymptotic Analysis (Cont.)

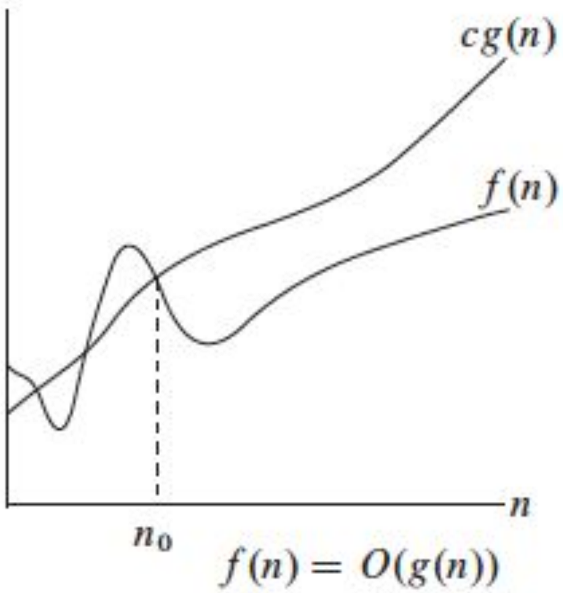
- Let's assume an algorithm can be represented as f(n); where **n** is the input size. We need to calculate the running time of **f(n)**.

- We define another function lets call it g(n) which represents the running time of the algorithm.

- Now three inequalities are possible
  1. f(n) < g(n)
  2. f(n) > g(n)
  3. f(n) = g(n)

Big-Oh (O)

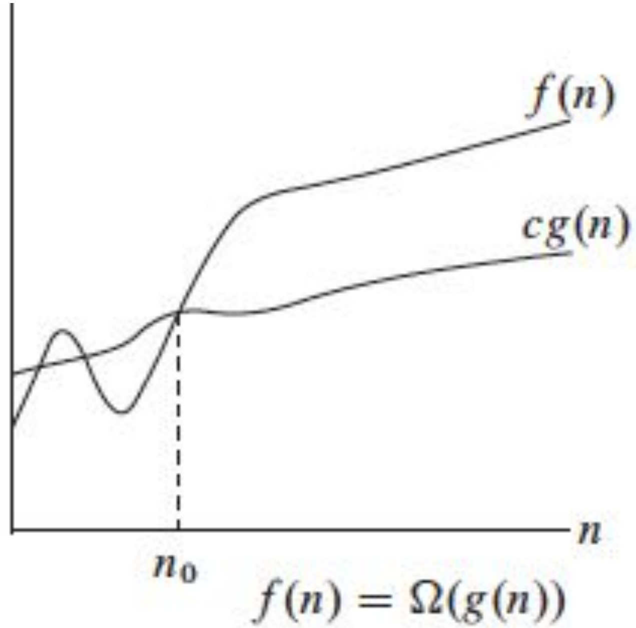
- We use Big-Oh to represent the worst case running time of an algorithm.
- Upper bound of f(n)
- We define Big-Oh as:



$$O(g(n)) = \{ f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \} .$$

Big-Omega (Ω)

- We use Big-Omega to represent the best case running time of an algorithm
- Lower bound of f(n)
- We define Big-Omega as:



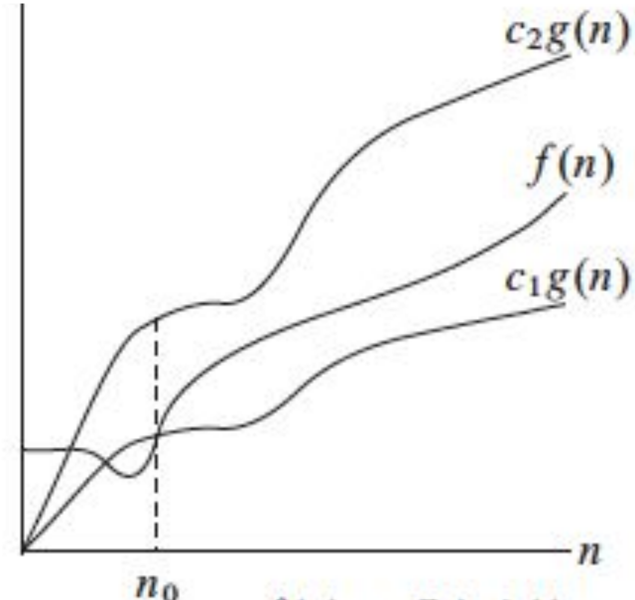
$$\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \} .$$

Big-Theta (Θ)

- Big-Theta represents the range; upper and lower

- Tight bound of f(n)

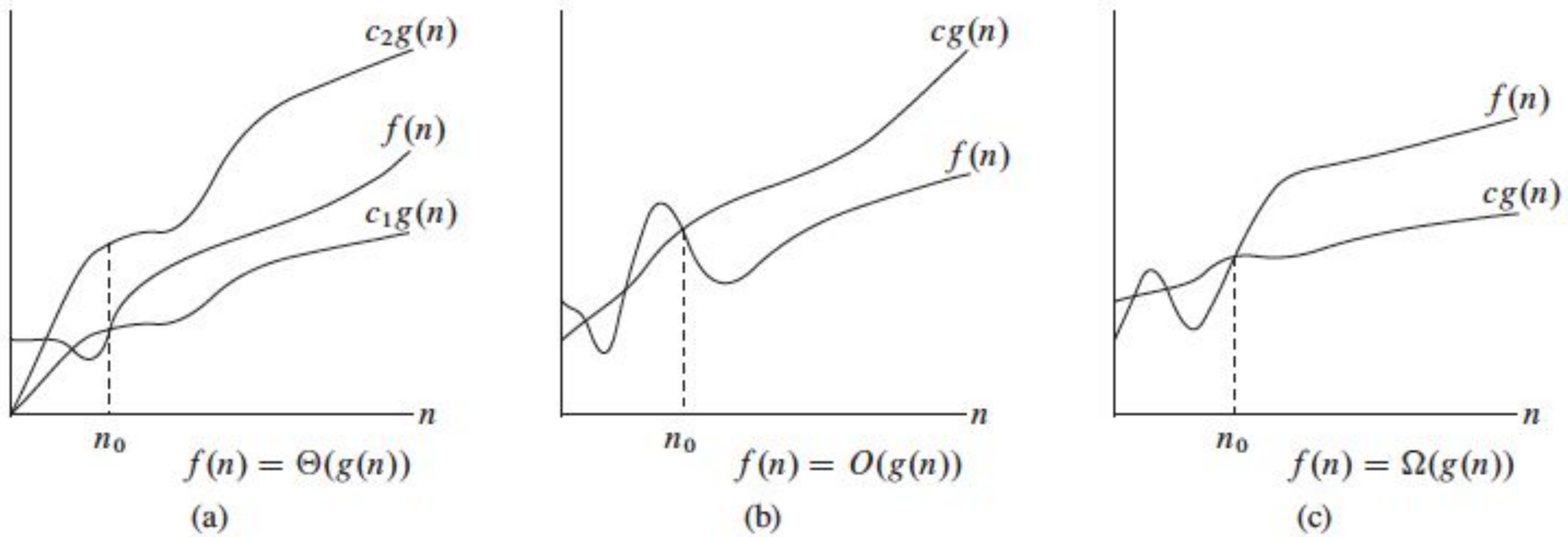
- We define Big-Theta as:



$f(n) = \Theta(g(n))$

$$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \}$$.[1]

Θ, O, and Ω Comparison



a. $\Theta(g(n)) = \{ f(n) :$ there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0 \}$.[1]

c. $\Omega(g(n)) = \{ f(n) :$ there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0 \}$.

b. $O(g(n)) = \{ f(n) :$ there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0 \}$.

Example 1

```python
def print_first_element(lst):
    print(lst[0])
```

Example 2

```python
def print_all_elements(lst):
    for item in lst:
        print(item)
```

Example 3

```python
def print_all_pairs(lst):
    for i in lst:
        for j in lst:
            print(i, j)
```

Example 4

```python
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1
```

Lets try to understand log (n)

Dry-run for f1(8) call:

```
def f1(n):
    if n <= 1:
        return
    print(n)
    f1(n // 2)
```

Credits

This lecture notes contains some of the material from the following resources:

- Chapter 3 of Cormen, Leiserson, Rivest, and Stein (3rd Edition).

- Lecture notes of Jessica Miller in Data Structures and Algorithms course taught at University of Washington.