CBS Department

Algorithm Analysis & Big-O-Notation

Soma Soleiman Zadeh

Data Structures & Algorithms (CBS 216)

Spring 2025 - 2026

Week 2

February 08, 2026

Outline

- **Algorithm Analysis**
- **Rate of Growth**
- **Big-O-Notation**

How do we Analyze Algorithms?

- We need to define a number of objective measures.

1. Compare **running times**?
   - **Not good:** times are specific to a particular computer.
2. Count the **number of statements (operations)** executed?
   - **Not good:** number of statements is dependent on programming language (Java, Python, etc.) and the style of the individual programmer.

Example (Number of Statements)

Algorithm 1

```
lst[0] = 0
lst[1] = 0
lst[2] = 0
...
lst[N-1] = 0
```

Number of Statements → **N statements**

Algorithm 2

```
for i in range(N):
    lst[i] = 0
```

Number of Statements → **2 statements**

How do we Analyze Algorithms?

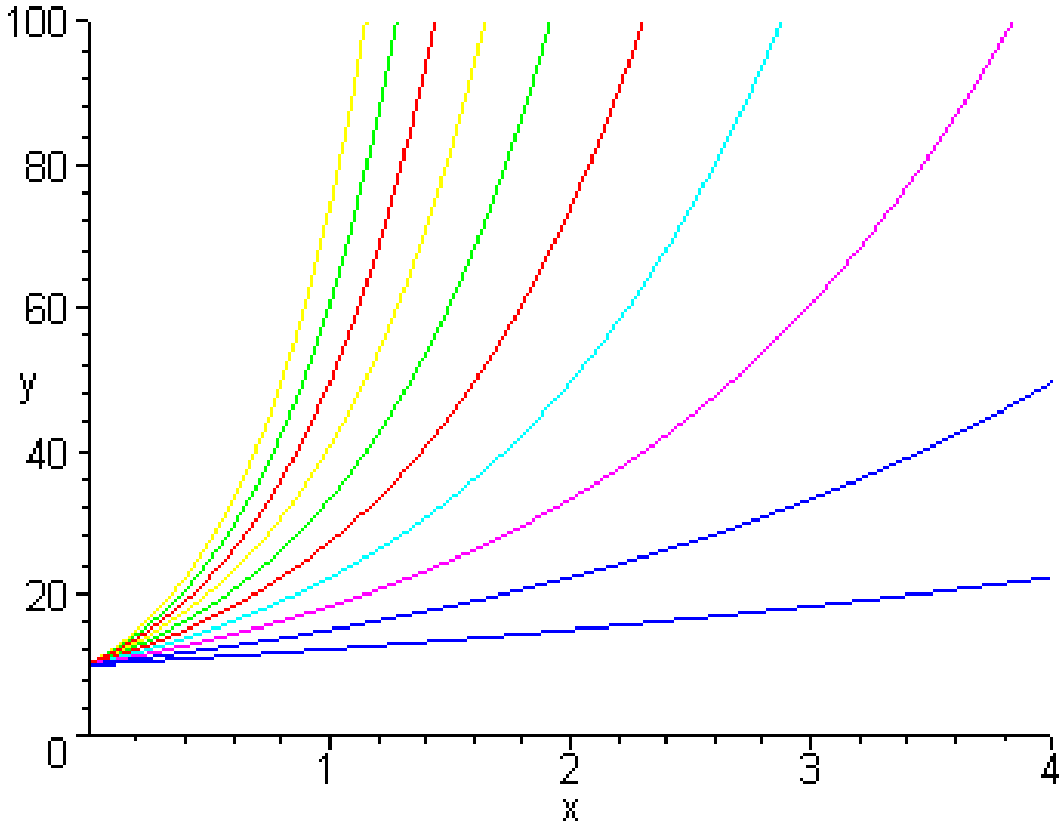
3. Express running time as a **function of the input size *n***

   (For example, f(n)).

- To compare two algorithms with running times f(n) and g(n), we need a measure of **how fast a function grows (rate of growth)**.
- This way of algorithm analysis is independent of computer type and programming style.

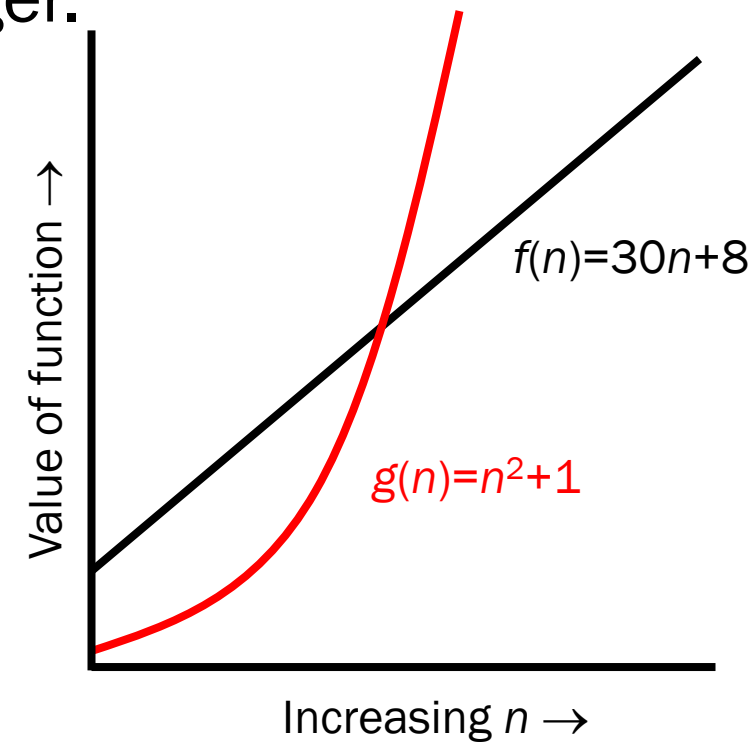
What is Rate of Growth?

- The rate at which the running time of an algorithm increases as the input size increases, is called rate of growth.



Rate of Growth

- On a graph, as you go to the right(n becomes larger), the faster growing function, g(n), eventually becomes larger.
- Rate of growth of **g(n)**> Rate of growth of **f(n)**



Asymptotic Analysis

- Using **rate of growth** as a measure to compare different functions implies comparing them asymptotically.
- If **g(n)** is faster growing than **f(n)**, then **g(n)** always eventually becomes larger than **f(n)** for large enough values of **n**.

Running Time?

	Cost
sum = 0	c1 * 1
for i **in** range(N):	
for j **in** range(N):	
sum += i*j	c2 * (N^2)

c1 and **c2** are two constant numbers, which are representing cost of one statement.

- sum=0 has a c1 cost and it executes only once, that's why its total cost is **c1*1.**
- Sum+=i*j has a c2 cost and it executes N*N times, that's why its total cost is **c2*N*N** or **c2*** N^2.

The total cost of this code is :

T(n) = c1 + (c2* N^2)

Comparing Algorithms Using Rate of Growth

- Consider an algorithms and its cost:

Algorithm's Cost: $n^4 + 100n^2 + 10n + 50$

- The low order terms are relatively insignificant for large values of **n.**
  - **50** → constant number, not dependent to ***n,*** so we can ignore it.
  - $\mathbf{100}\boldsymbol{n}^{\mathbf{2}}$ and $\mathbf{10}\boldsymbol{n}$ → They are dependent to ***n***, but for large enough values of ***n***, they can be ignored.

- So we can have an approximate cost:

Algorithm's Cost ~ n^4

Big-O Notation

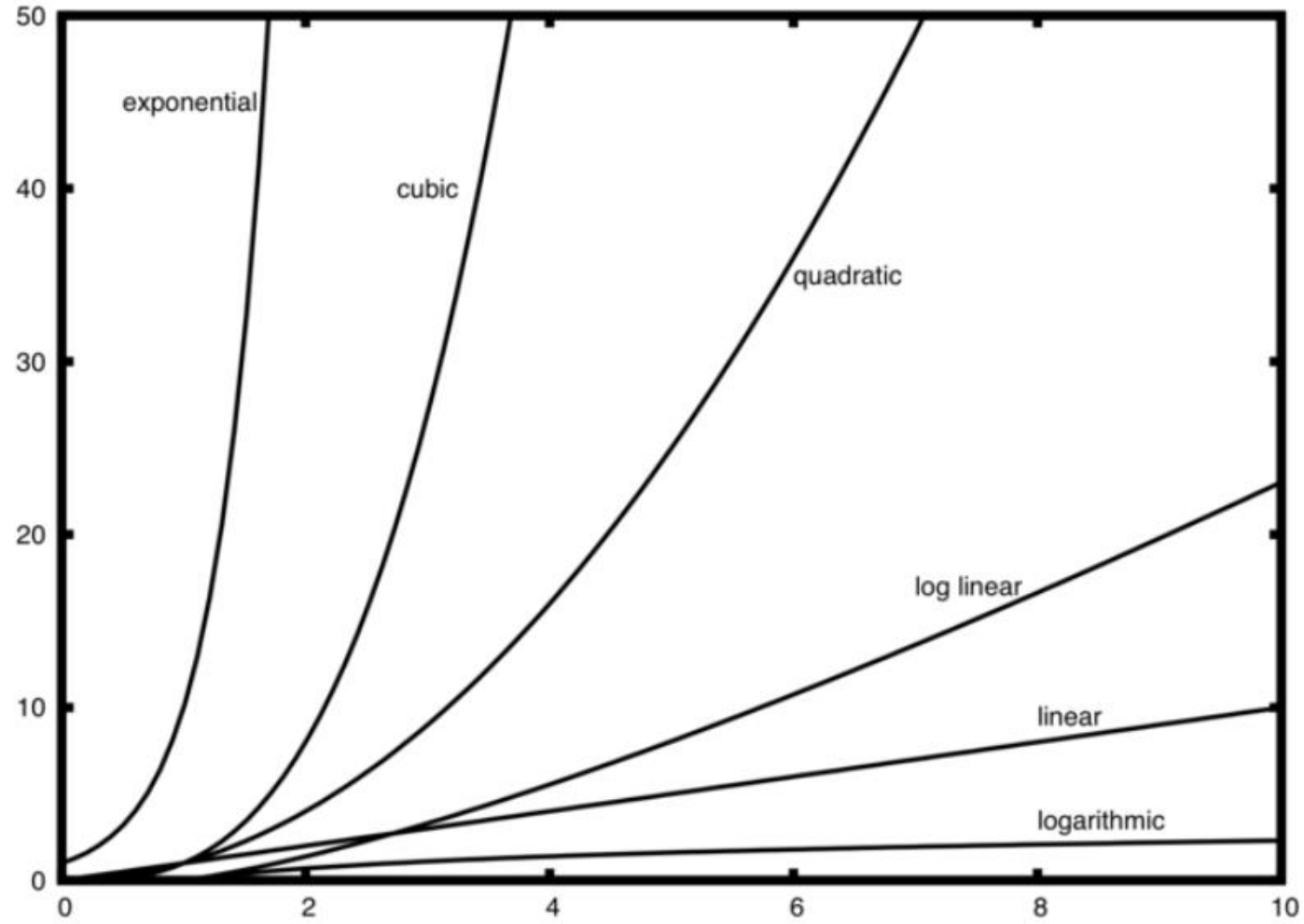
- For comparing algorithms based on their rate of growth, we use a method called **Big-O Notation**.
- **Big-O Notation** is the most widely used method which describes algorithm complexity in terms of:
  - the execution time required or
  - the space used in memory or in disk by an algorithm.
- **Big-O notation** is a mathematical technique that allows us to take a function and, typically, simplify it.
  - Big-O is an upper bounds that hides a lot of unimportant details.

Big-O Notation

- Consider the exact rate of growth of two algorithms as follows:
  - $T(n)=30n+8$ → is *order n*, or $O(n)$.
  - $T(n)=n^2+1$ → is *order* n^2, or $O(n^2)$.

- In general, an $O(n^2)$ algorithm will be slower than $O(n)$ algorithm.
- an $O(n^2)$ function will grow faster than an $O(n)$ function.

Common Big-O Functions



Common Big-O Functions

f(n)	O(f(n))	Name
1	O(1)	Constant
Log n	O(log n)	Logarithmic
n	O(n)	Linear
n log n	O(n log n)	Log Linear
n^2	O(n^2)	Quadratic
n^3	O(n^3)	Cubic
2^n	O(2^n)	Exponential

Time Complexity Using Big-O Notation

	Cost	Time Complexity
sum = 0	c1 * 1	O(1)
for i **in** range(N):		
for j **in** range(N):		
sum += i*j	c2 * (N^2)	O(N^2)

- The total time complexity of this code is O(1) + O(N^2).
- We can ignore O(1),
- So, the time complexity of code is **O(N^2).**

Examples of O(1) – Constant Time Algorithms

```
a = 0
b = 5
c = a + b
print(a)
print(b)
print(c)
```

```
a = 100
for i in range(1000):
    print(a)
```

- Both codes above are **O(1).**
- We call these algorithms **constant time algorithms.**
- They do not depend on the input size *n*,
- So, O(2), O(6) or even O(1000) are the same as **O(1).**

Examples of O(n) – Linear Time Algorithms

```
for i in range(n):
    print(i)
```

- The above code is **O(n).**
- We call these algorithms **linear time algorithms.**
- They depend on the input size *n*.

Examples of $O(n^2)$ – Quadratic Time Algorithm

```
for  i  in  range(n):
        for  j  in  range(n):
                num = i * j
```

```
while i < n:
        while  j< n:
                num = i * j
```

- The above codes are **$O(n^2)$.**
- We call these algorithms **Quadratic time algorithms.**
- They depend on the input size *n*.
- The loops are nested. The outer loop executes *n* times and the inner loop executes *n* times, so the statements inside inner loop execute n^2 times.