

# Phân tích độ phức tạp thuật toán không đệ quy



Nhóm 2


Hoàng Long, Cẩm Nguyên



# Mục tiêu

- Biết các khái niệm trong phân tích thuật toán
- Biết cách tính độ phức tạp bằng lý thuyết
- Thành thạo cách ước lượng độ phức tạp bằng cách sử dụng 4 quy tắc
- Cách ước tính độ phức tạp của thuật toán không đệ quy





## Các khái niệm cơ bản trong phân tích thuật toán


- Giới thiệu về khái niệm “Độ phức tạp thuật toán”
- Ý nghĩa của việc phân tích độ phức tạp thuật toán
- Giới thiệu về các loại efficiency

## Lý thuyết cách tính độ phức tạp

- Đơn vị đo thời gian thực thi thuật toán
- Khái niệm về O-Notation
- Khái niệm về Omega-Notation
- Khái niệm về Theta-Notation



## Ước lượng độ phức tạp bằng cách sử dụng 4 quy tắc

- Quy tắc bỏ hằng số
  - Quy tắc cộng
  - Quy tắc lấy max
  - Quy tắc nhân
- 



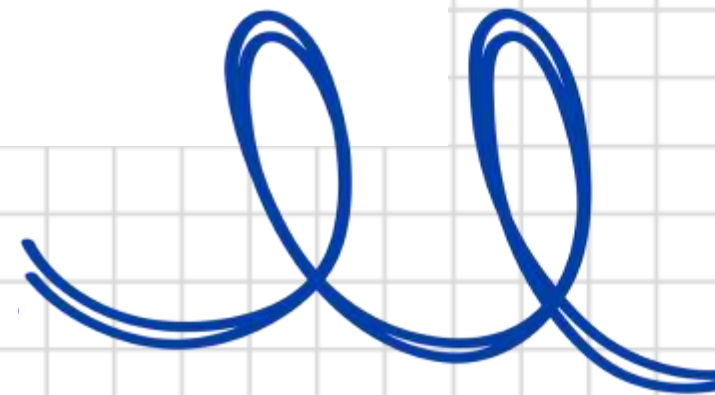
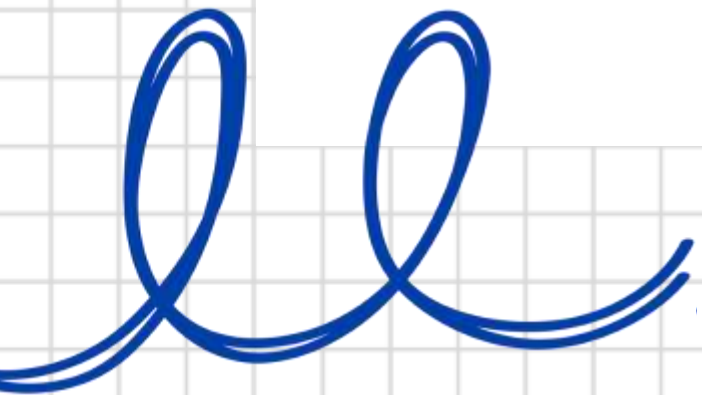




# Các khái niệm cơ bản trong phân tích thuật toán




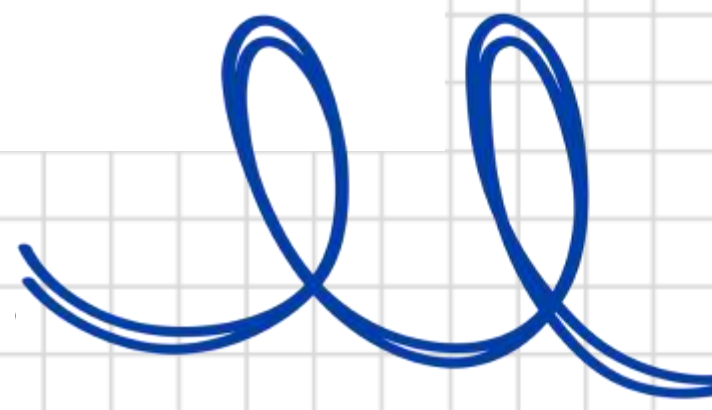
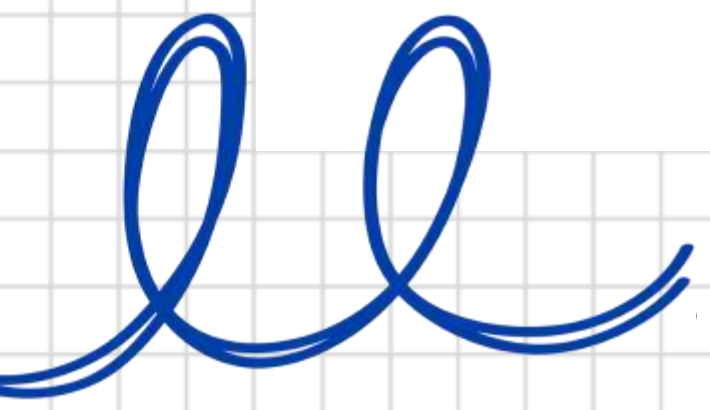
**Độ phức tạp thuật toán là gì?**



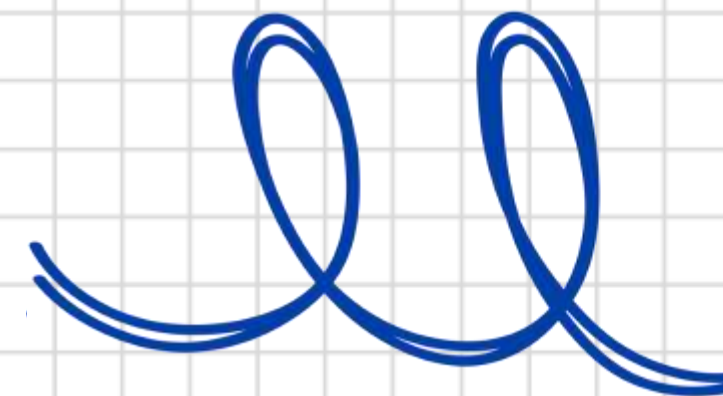
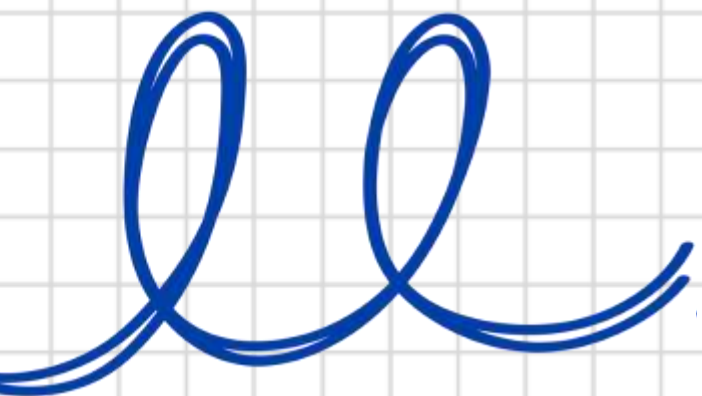


# Độ phức tạp thuật toán là gì?

Là 1 khái niệm thể hiện hiệu năng của giải thuật  
theo kích thước của đầu vào.



\* **Tại sao phải tính độ phức tạp thuật toán?**  
**Cho ví dụ về tình huống trong cuộc sống mà có áp dụng  
phân tích độ phức tạp thuật toán**





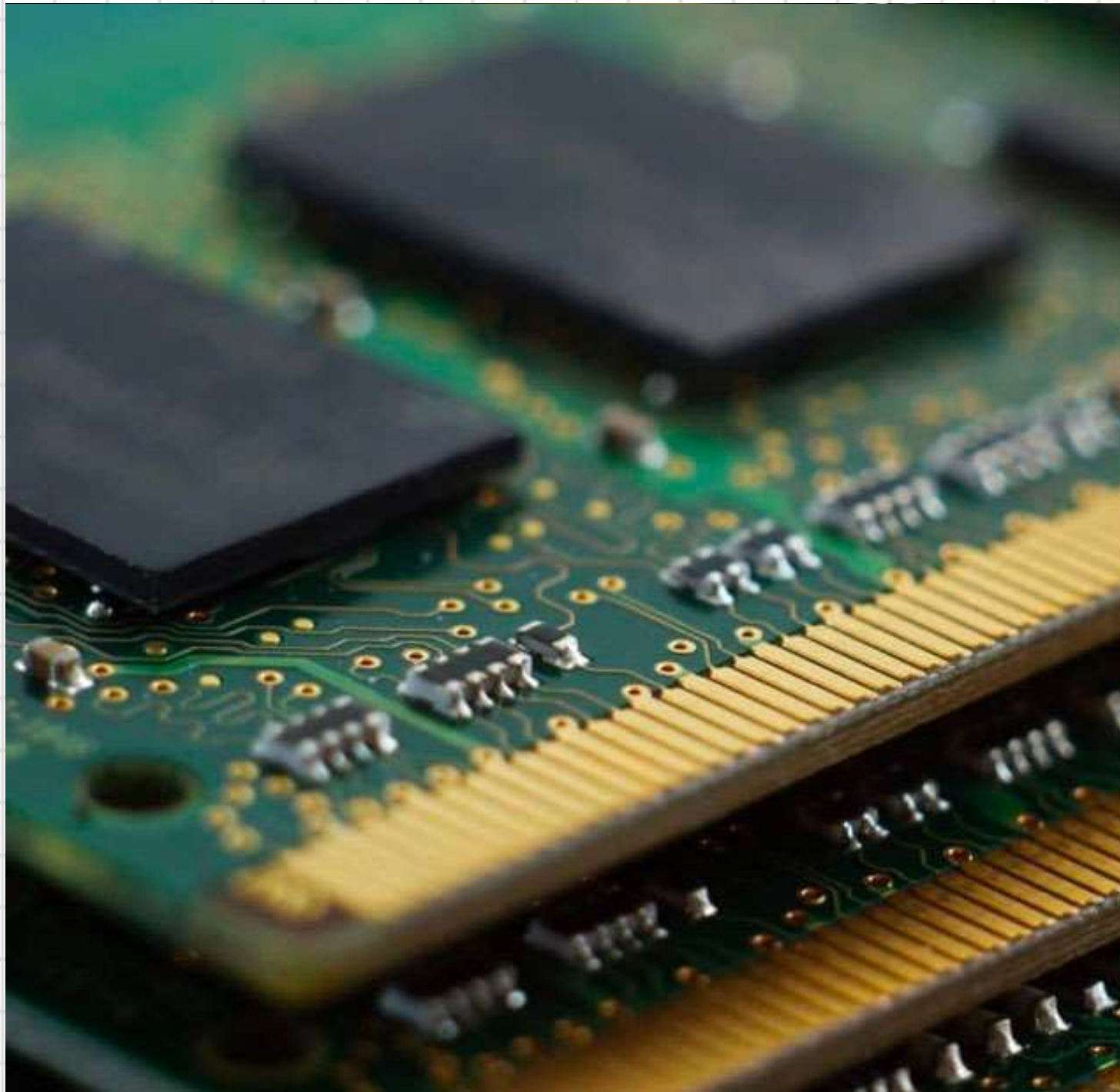
# \* Tại sao phải tính độ phức tạp thuật toán?

Cho ví dụ về tình huống trong cuộc sống mà có áp dụng phân tích độ phức tạp thuật toán

- **Dự đoán tài nguyên cần thiết:** Hạn chế lãng phí hoặc thiếu tài nguyên (thời gian, bộ nhớ,...)
- **Tối ưu hóa:** Biết cách làm cho nó hoạt động nhanh hơn hoặc tiêu tốn ít tài nguyên hơn.
- **Đánh giá hiệu suất và so sánh thuật toán:** Biết cách thuật toán hoạt động trong trường hợp tốt và xấu nhất. Đảm bảo chạy tốt trong mọi tình huống. Chọn thuật toán hiệu quả nhất trong trường hợp cụ thể



# Dự đoán tài nguyên cần thiết



- Xác định độ phức tạp thuật toán
- Xác định đầu vào dự kiến
- **Tính toán tài nguyên cần thiết:**
  - Thời gian: Tính thời gian thực thi ước tính cho các bước của thuật toán.
  - Không gian: Tính không gian bộ nhớ cần thiết cho các biến và cấu trúc dữ liệu

# Tối ưu hóa



- **Xác định độ phức tạp thuật toán hiện tại**
- **Xác định điểm yếu:**
  - Tìm thành phần của thuật toán gây ra hiệu suất kém
  - Có thể là vòng lặp lặp đi lặp lại quá nhiều, cấu trúc dữ liệu không hiệu quả, thao tác dư,...
- **Áp dụng các phương pháp tối ưu hóa:**
  - Giảm thiểu số lần lặp
  - Dùng cấu trúc dữ liệu tối ưu hơn
  - Nghiên cứu các thuật toán có độ phức tạp tối ưu hơn
  - ...

# Đánh giá hiệu suất và so sánh thuật toán



## Độ phức tạp thời gian

**Time efficiency**, hay còn gọi là **time complexity**: Thể hiện cho việc thực hiện một thuật toán nhanh như thế nào.

Thường được chú trọng hơn vì việc tăng tốc độ của vi xử lý khó hơn rất nhiều so với tăng bộ nhớ



## Độ phức tạp không gian

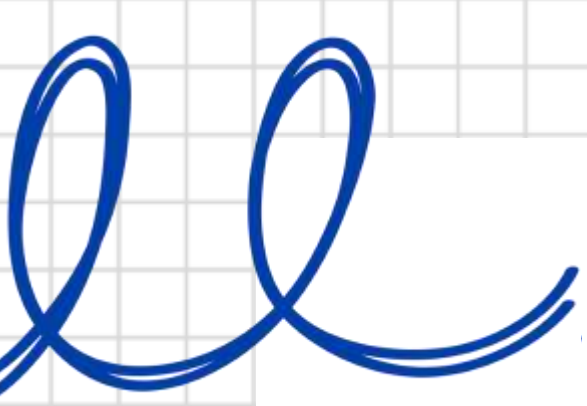
**Space efficiency**, hay còn gọi là **space complexity**: Thể hiện cho số đơn vị bộ nhớ dùng cho thuật toán và input, output của nó







# Lý thuyết về cách tính độ phức tạp

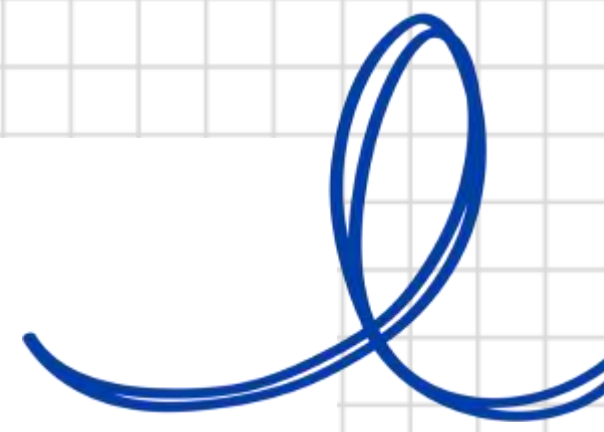


# Thời gian thực thi thuật toán

**Chúng ta có nên sử dụng đơn vị tiêu chuẩn như giây hoặc mili giây để đo thời gian chạy của một thuật toán?**



# Thời gian thực thi thuật toán



**Chúng ta có nên sử dụng đơn vị tiêu chuẩn như giây hoặc mili giây để đo thời gian chạy của một thuật toán?**

**Việc đo lường như trên dễ làm và trực quan, nhưng phụ thuộc vào:**

- Tốc độ của một máy tính cụ thể.
  - Chất lượng chương trình thực thi và compiler dùng để biên dịch sang mã máy.
- > Tính chủ quan lớn, khó khăn trong việc clocking trong thời gian thật

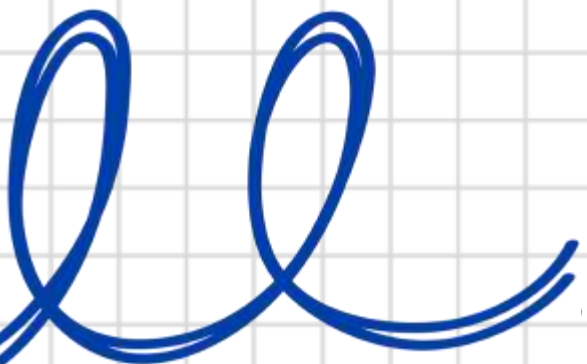
**Cần tìm một cách đo lường khác, độc lập với thiết bị và trình biên dịch**  
**=> Dùng các kí hiệu tiệm cận**

# Thời gian thực thi thuật toán

- **Phép tính cơ bản:** là phép tính chiếm hầu hết tổng thời gian chạy của thuật toán, thường thuộc vòng lặp sâu nhất

=> **Xác định phép tính cơ bản của thuật toán, tính số lần phép tính đó được thực thi.**

Ví dụ: Đây là phép tính trọng tâm trong các thuật toán sort?



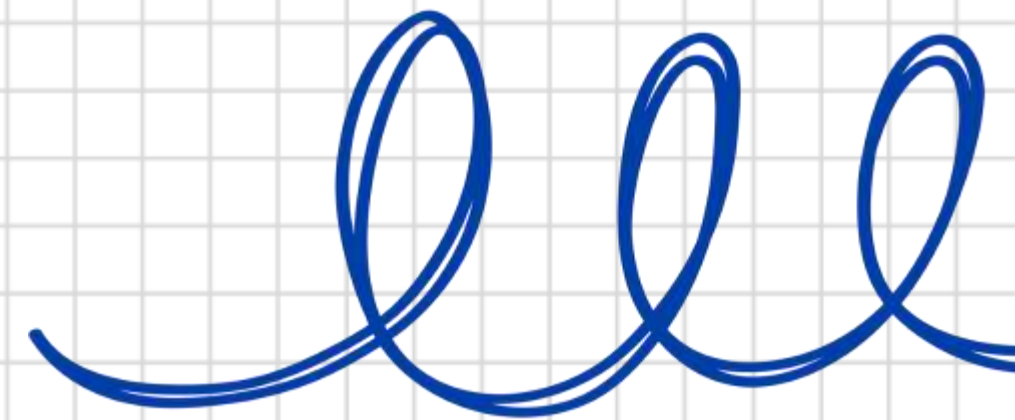
# Thời gian thực thi thuật toán

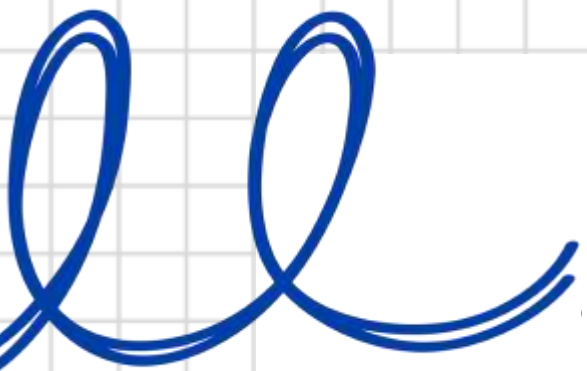
- **Phép tính cơ bản:** là phép tính chiếm hầu hết tổng thời gian chạy của thuật toán, thường thuộc vòng lặp sâu nhất

=> **Xác định phép tính cơ bản của thuật toán, tính số lần phép tính đó được thực thi.**

Ví dụ: Đây là phép tính trọng tâm trong các thuật toán sort?

**Phép so sánh và đổi chỗ là các phép tính cơ bản của thuật toán sort**





# Thời gian thực thi thuật toán

- **Phép tính cơ bản:** là phép tính chiếm hầu hết tổng thời gian chạy của thuật toán, thường thuộc vòng lặp sâu nhất

=> **Xác định phép tính cơ bản của thuật toán, tính số lần phép tính đó được thực thi.**

Ví dụ: Đây là phép tính trọng tâm trong các thuật toán sort?

**Phép so sánh và đổi chỗ là các phép tính cơ bản của thuật toán sort**

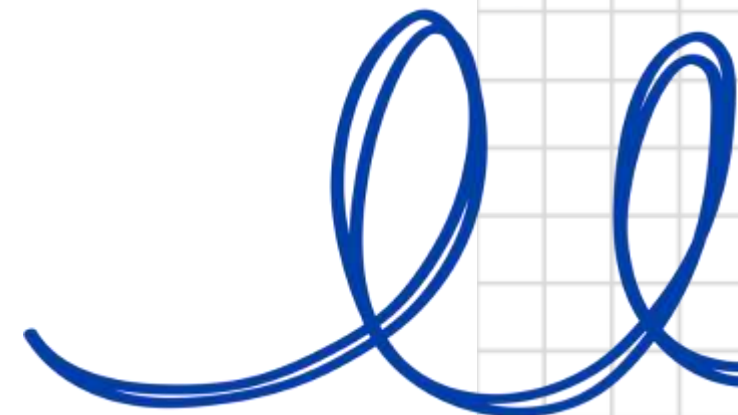
**Trong cách tiếp cận này, độ phức tạp thuật toán được tính bằng cách đếm số lần phép tính cơ bản được thực hiện với input có size là  $N$**

# Thời gian thực thi thuật toán

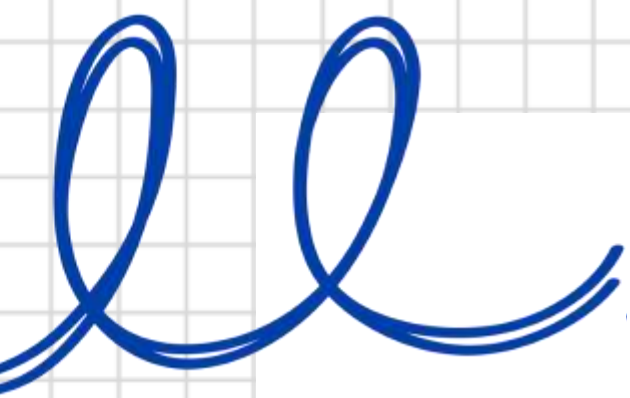
- **n**: Input size
- **$c_{op}$** : Thời gian thực thi của một phép tính cơ bản
- **$C(n)$** : Là số lần phép tính cơ bản được thực thi trong thuật toán
- **$T(n)$** : Thời gian ước lượng để thực thi thuật toán trên máy tính đó

Công thức:

$$T(n) = c_{op}C(n)$$







# Thời gian thực thi thuật toán

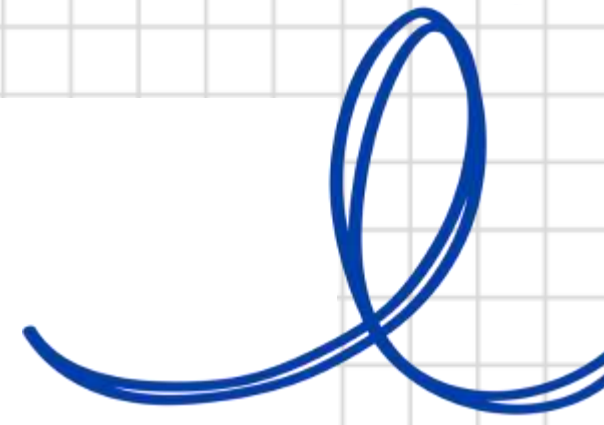
- **n**: Input size
- **c<sub>op</sub>**: Thời gian thực thi của một phép tính cơ bản
- **C(n)**: Là số lần phép tính cơ bản được thực thi trong thuật toán
- **T(n)**: Thời gian ước lượng để thực thi thuật toán trên máy tính đó

Công thức:  $T(n) = c_{op}C(n)$

Thuật toán sẽ chạy chậm hơn bao nhiêu lần nếu ta gấp đôi input size? Biết số lần phép tính cơ bản được thực thi là:

$$C(n) = \frac{1}{2}n(n - 1)$$

# Thời gian thực thi thuật toán



Thuật toán sẽ chạy chậm hơn bao nhiêu lần nếu ta gấp đôi input size? Biết số lần phép tính cơ bản của thuật toán được thực thi là:

$$C(n) = \frac{1}{2}n(n-1)$$

Chú ý rằng ta có thể trả lời câu hỏi trên mà không cần biết giá trị của  $c_{op}$ . Khung phân tích hiệu suất thường bỏ qua các hằng số nhân, không chỉ quan tâm đến tốc độ chạy thực tế của thuật toán, mà quan tâm đến cách tốc độ đó thay đổi khi kích thước đầu vào tăng lên

$$\lim_{n \rightarrow \infty} \frac{T(2n)}{T(n)} = \frac{c_{op}C(2n)}{c_{op}C(n)}$$

Gấp đôi input size  $\rightarrow$  chậm hơn 4 lần

- 1 : Không có sự khác biệt đáng kể
- 0 :  $T(2n)$  tăng chậm hơn  $T(n)$  đáng kể
- $+\infty$  :  $T(2n)$  tăng nhanh hơn  $T(n)$  đáng kể

# Kí hiệu tiệm cận (Notation)

Khung phân tích hiệu suất tập trung vào bậc tăng trưởng số lượng tính toán cơ bản của một thuật toán thay đổi khi kích thước đầu vào tăng lên như một thước đo thể hiện hiệu suất của thuật toán

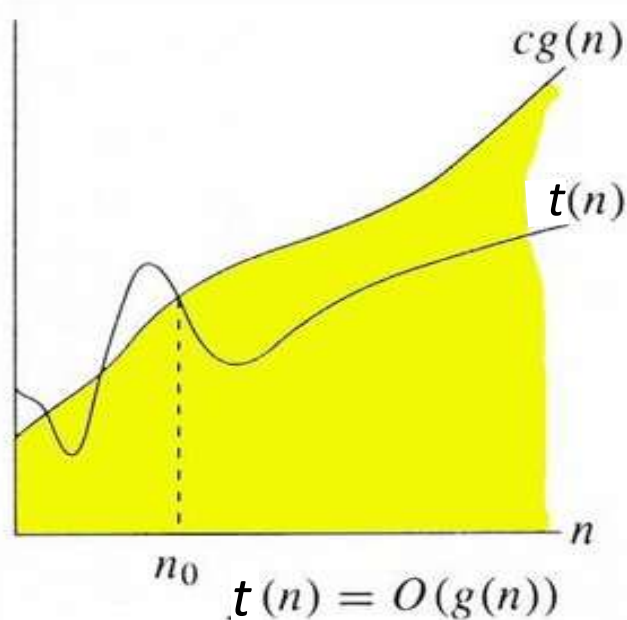
Kí hiệu để so sánh và xếp hạng các bậc tăng trưởng:

- Big O: Xác định một tiệm cận trên

Thường được sử dụng

**Định nghĩa:** Một hàm  $t(n)$  được cho là thuộc  $O(g(n))$ , nếu tồn tại một hằng số  $c$  không âm và một số nguyên không âm  $n_0$  với mọi  $n > n_0$  sao cho:

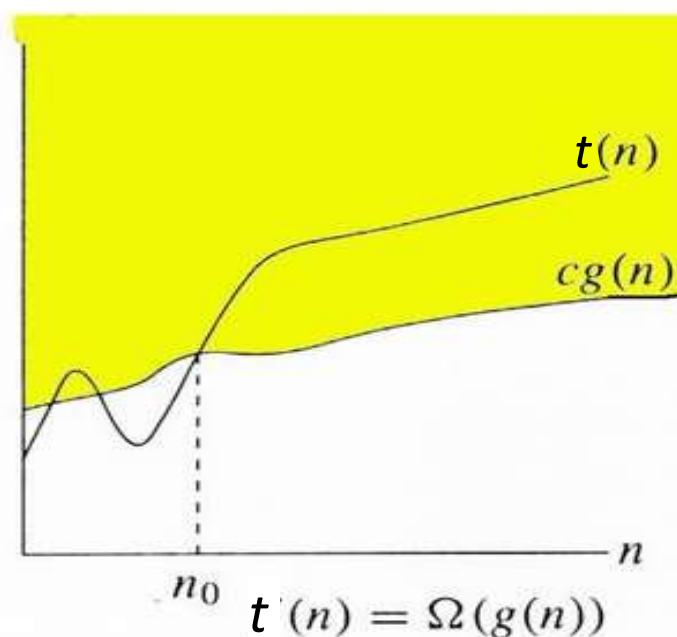
$$t(n) < c.g(n)$$



- Omega: Xác định một tiệm cận dưới

**Định nghĩa:** Một hàm  $t(n)$  được cho là thuộc  $\Omega(g(n))$ , nếu tồn tại một hằng số  $c$  không âm và một số nguyên không âm  $n_0$  với mọi  $n > n_0$ , sao cho:

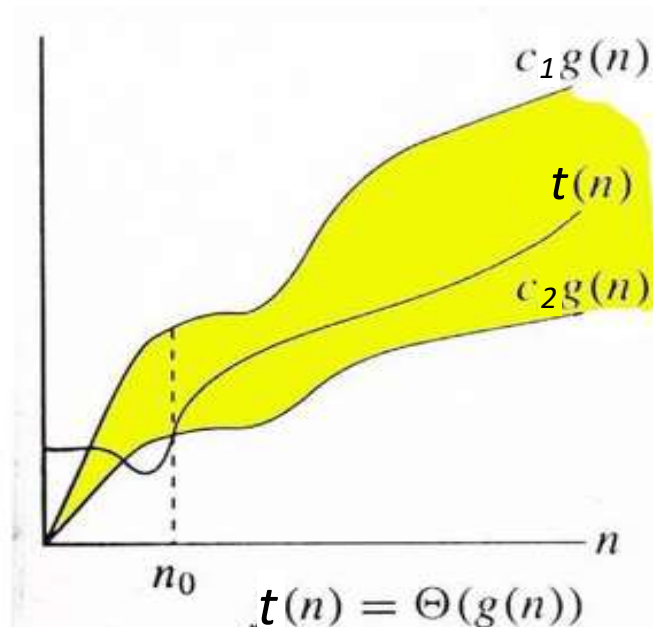
$$t(n) > c.g(n)$$



- Theta: xác định bậc tăng trưởng thông qua các tiệm cận trên và dưới

**Định nghĩa:** Một hàm  $t(n)$  được cho là thuộc  $\Theta(g(n))$ , nếu tồn tại hai hằng số  $c$  và  $c$  không âm và một số nguyên không âm  $n_0$  với mọi  $n > n_0$ , sao cho

$$c_2g(n) < t(n) < c_1g(n)$$

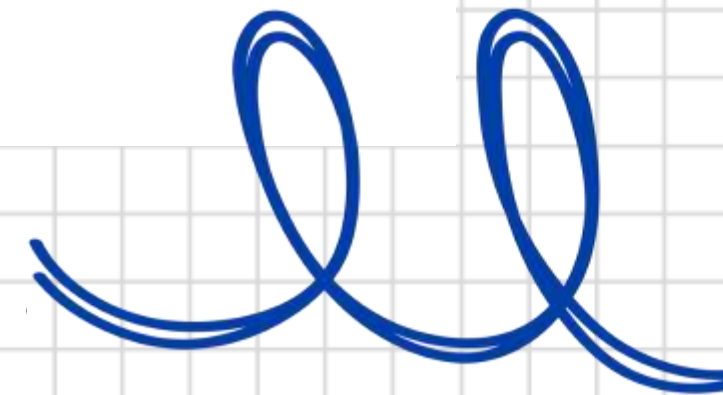
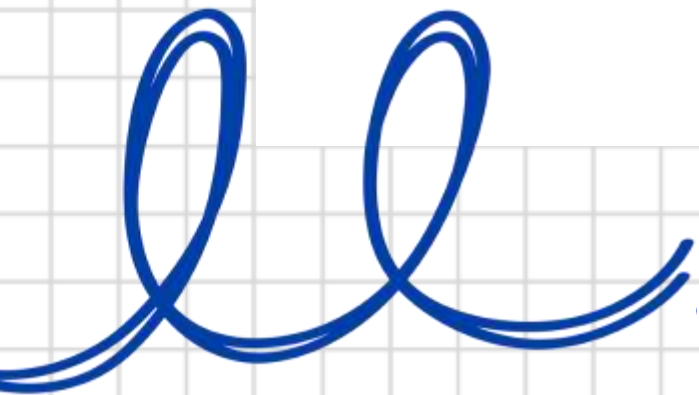




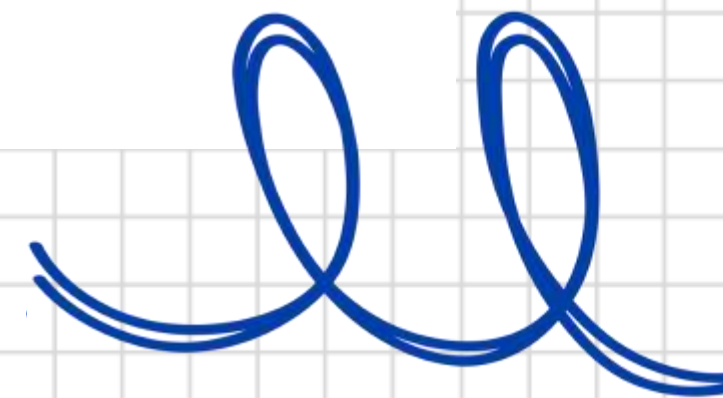
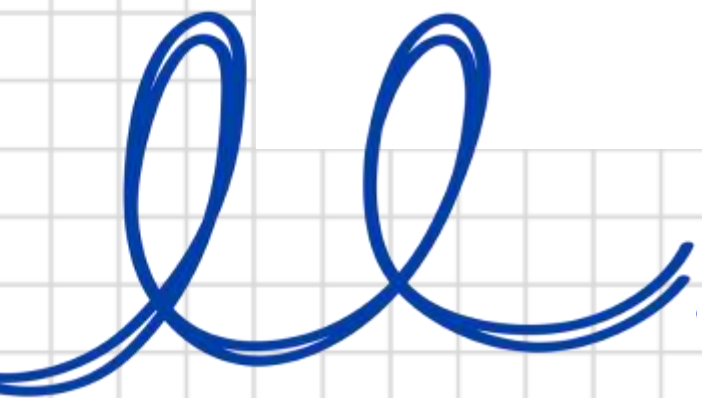
# Tại sao Big O thường được dùng ?



- Big O tập trung vào *giới hạn trên* của time complexity. Người dùng chỉ quan tâm thuật toán có đáp ứng các giới hạn về tài nguyên của hệ thống
- Dễ tính toán hơn Omega và Theta



\* Big O có thể hiện trường hợp xấu nhất  
của một thuật toán không ? \*



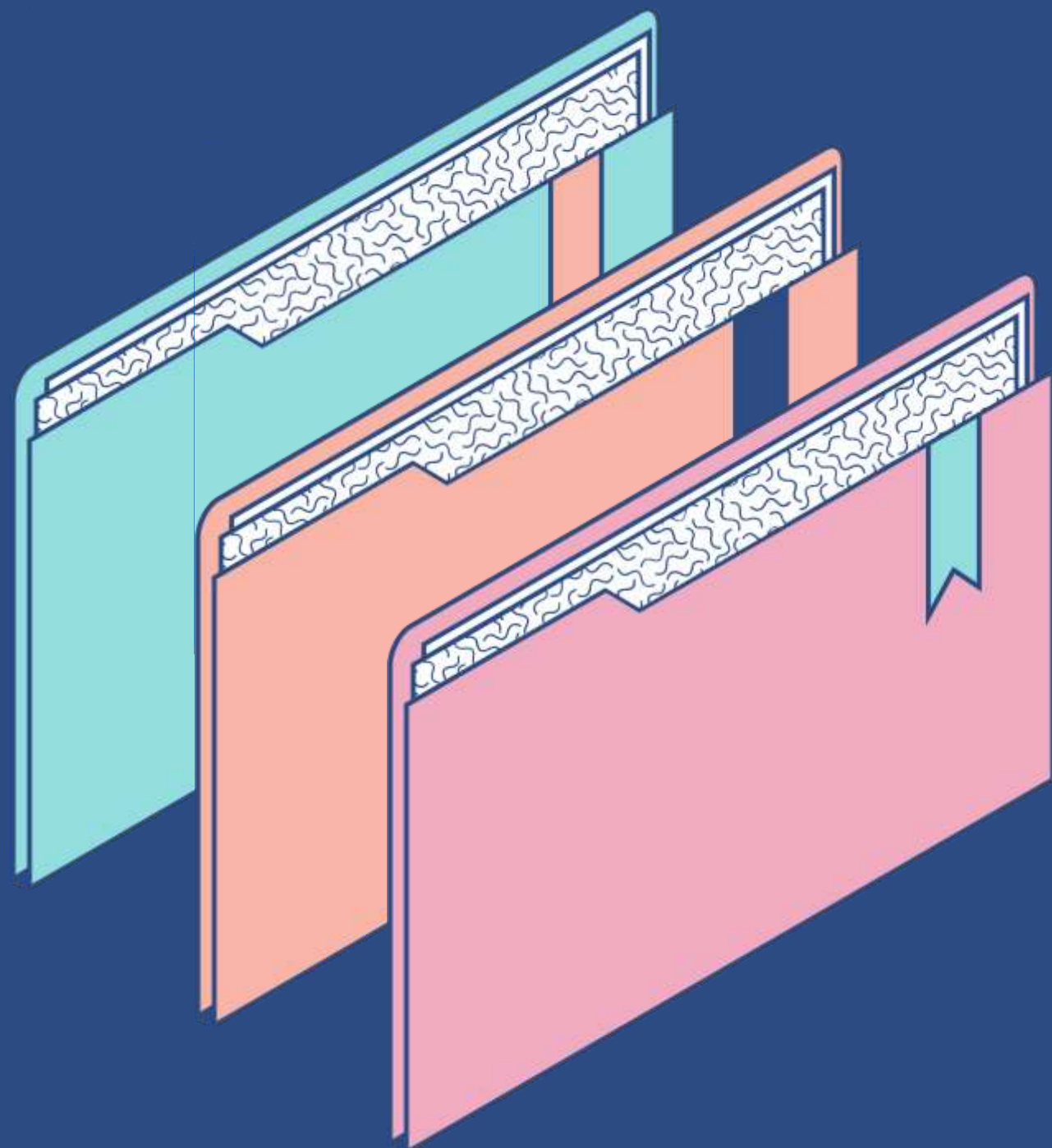


# Big O có thể hiện trường hợp xấu nhất của một thuật toán không ?

**Không.** Big O không nhất thiết thể hiện trường hợp xấu nhất mà chỉ thể hiện một giới hạn trên về tốc độ tăng trưởng trong điều kiện xấu nhất, tức là tốc độ tăng trưởng lớn nhất mà thuật toán có thể đạt được trong mọi tình huống

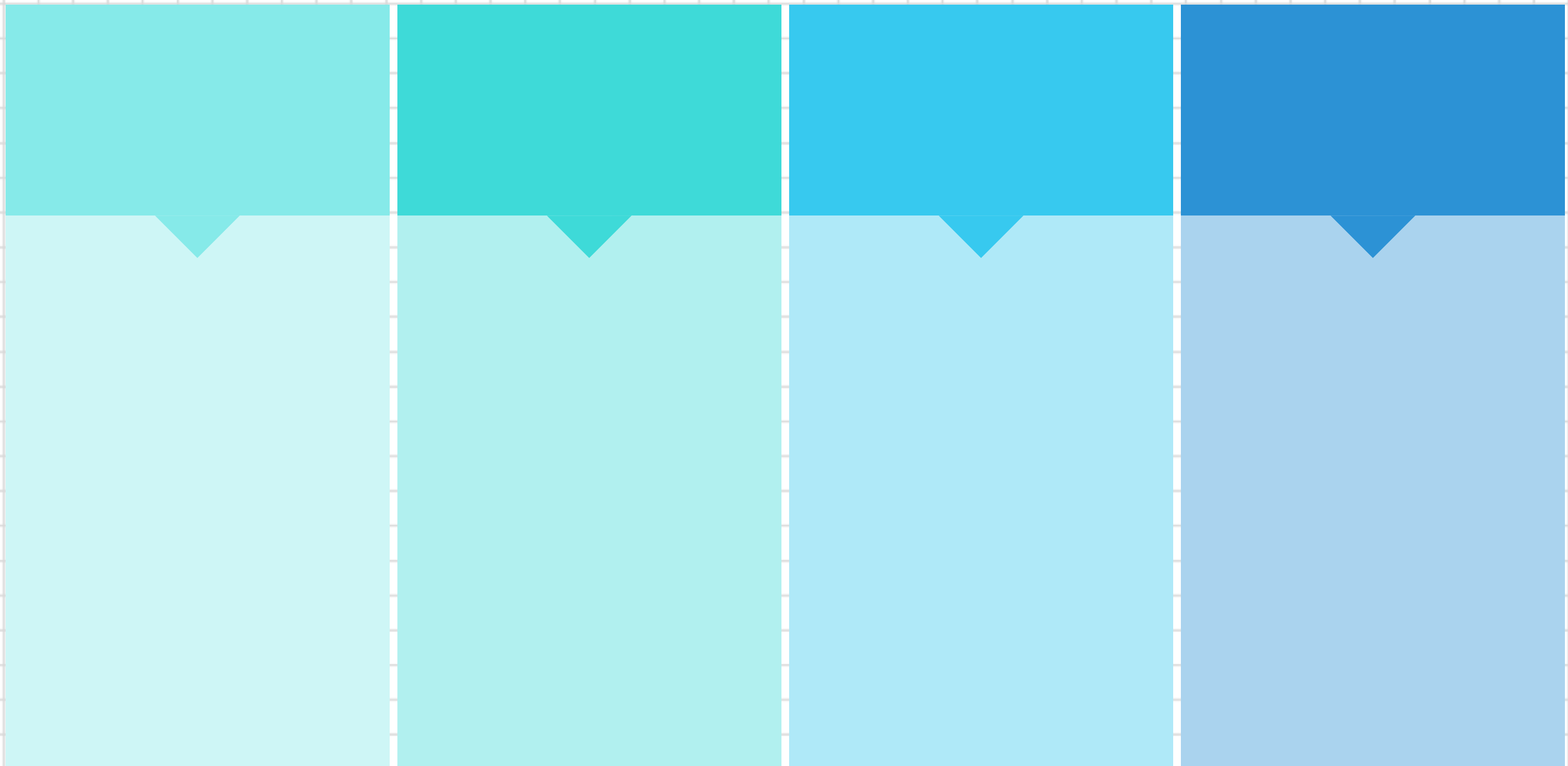
Ví dụ: Thuật toán sắp xếp QuickSort

- Trường hợp trung bình: Thường có độ phức tạp là  $O(n \log n)$
- Trường hợp xấu nhất: Khi chia mảng không cân bằng, độ phức tạp là  $O(n^2)$



Các qui tắc chính  
ước lượng độ phức  
tạp thuật toán

# 4 quy tắc chính để tính độ phức tạp thuật toán



# 4 quy tắc chính để tính độ phức tạp thuật toán

## QUY TẮC BỎ HẰNG SỐ

- Loại bỏ các hằng số  $c$  và chỉ quan tâm đến tốc độ tăng tương đối của độ phức tạp
- $T(n) = O(c * f(n))$   
 $= O(f(n))$

## QUY TẮC CỘNG

- Thuật toán có thể chia thành phần 1 và phần 2 riêng biệt, ta có thể cộng độ phức tạp của từng phần để tính độ phức tạp của thuật toán.
- $T1(n) = O(f(n))$   
 $T2(n) = O(g(n))$   
 $T(n) = T1(n) + T2(n)$   
 $= O(f(n) + g(n))$

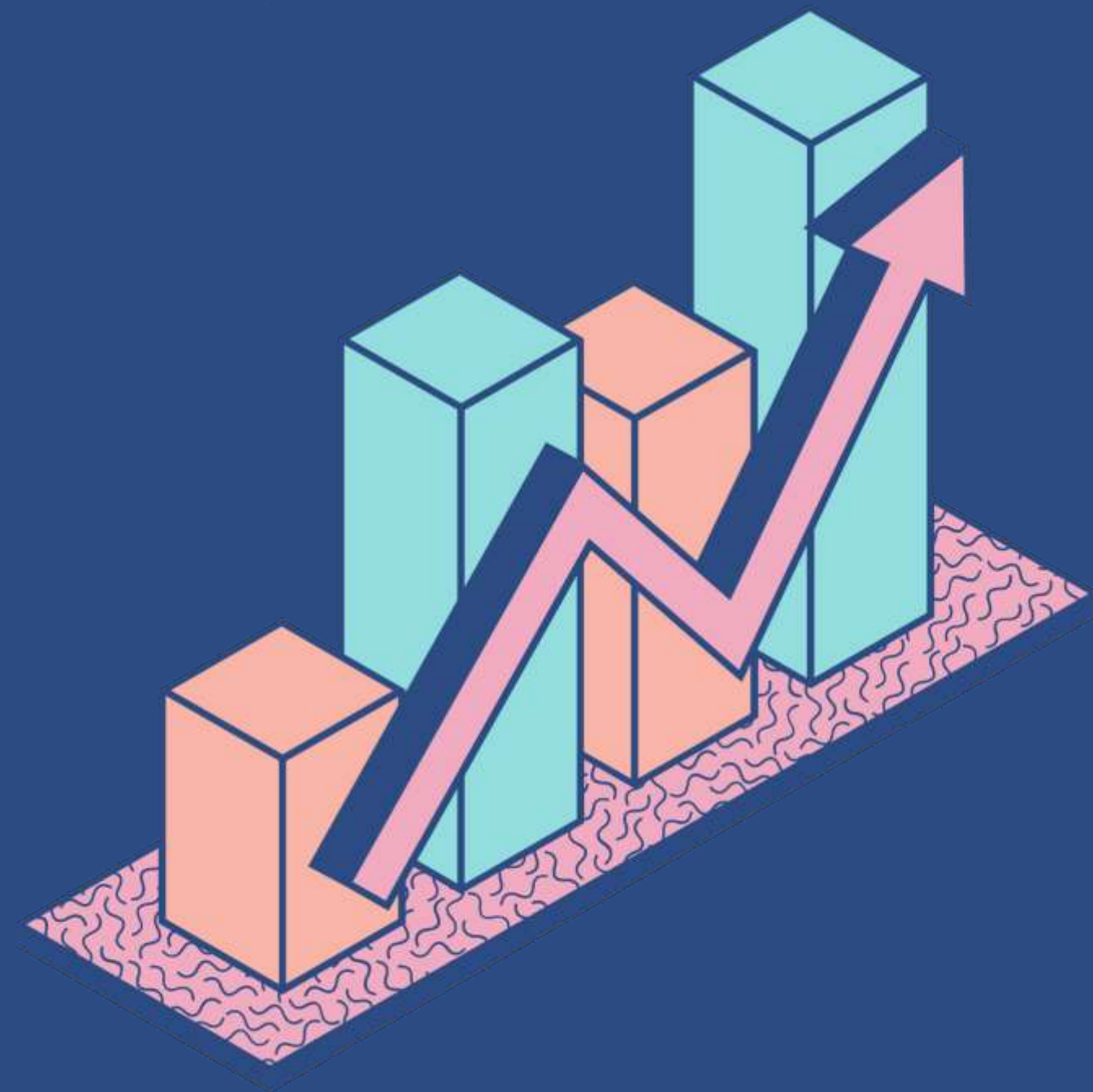
## QUY TẮC LẤY MAX

- Thuật toán có nhiều phần chạy nối tiếp nhau, ta tính độ phức tạp thuật toán bằng phần có độ phức tạp lớn nhất
- $T(n) = O(f(n) + g(n))$   
 $= O(\max(f(n), g(n)))$

## QUY TẮC NHÂN

- Thuật toán có phần 1 và phần 2 lồng nhau, ta nhân các độ phức tạp của từng phần để tính độ phức tạp thuật toán
- $T1(n) = O(f(n))$   
 $T2(n) = O(g(n))$   
 $T(n) = T1(n) * T2(n)$   
 $= O(f(n) * g(n))$

# Phân tích đối với các thuật toán không đệ quy





# Các bước cơ bản phân tích time efficiency



Ví dụ 1: Tìm giá trị lớn nhất của dãy có n số. Dựa theo pseudocode dưới đây để thực hiện phân tích độ phức tạp thuật toán

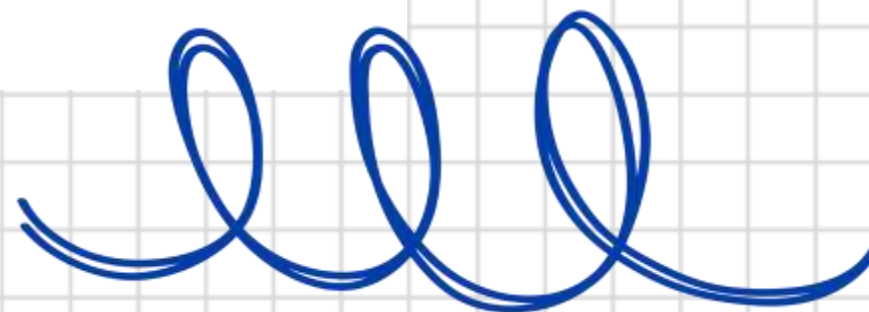


Thực hiện các yêu cầu sau:

- Xác định input size của thuật toán trên
- Phép tính trọng tâm của thuật toán trên là gì?
- Tính số lần phép tính trọng tâm được thực thi
- Tính độ phức tạp theo quy tắc Big O



```
MaxElement(A[0..n-1])  
// Determines the value of the largest element in a given array  
// Input: An array A[0..n-1] of real numbers  
// Output: The value of the largest element in A  
maxval <- A[0]  
for i <- 1 to n - 1 do  
    if A[i] > maxval  
        maxval <- A[i]  
return maxval
```



Ví dụ 2: Kiểm tra xem một dãy số có chứa các phần tử khác nhau. Dựa theo pseudocode dưới đây để thực hiện phân tích độ phức tạp thuật toán

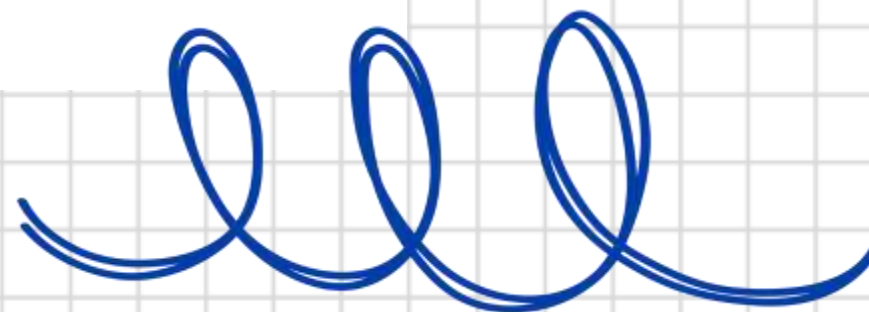


Thực hiện các yêu cầu sau:

- Xác định input size của thuật toán trên
- Phép tính trọng tâm của thuật toán trên là gì?
- Tính số lần phép tính trọng tâm được thực thi
- Tính độ phức tạp theo quy tắc Omega



```
UniqueElements(A[0..n - 1])  
//Determines whether all the elements in a given array are distinct  
//Input: An array A[0..n - 1]  
//Output: Returns "true" if all the elements in A are distinct  
// and "false" otherwise  
for i ← 0 to n - 2 do  
    for j ← i + 1 to n - 1 do  
        if A[i] = A[j] return false  
return true
```



### Ví dụ 3: Thực hiện nhân hai ma trận A và B. Thực hiện phân tích độ phức tạp thuật toán

#### Thực hiện các yêu cầu sau:

- Xác định input size của thuật toán trên
- Phép tính trọng tâm của thuật toán trên là gì?
- Tính số lần phép tính trọng tâm được thực thi
- Tính độ phức tạp theo quy tắc Theta

```
MatrixMultiplication(A[0..n - 1, 0..n - 1], B[0..n - 1, 0..n - 1])
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two n x n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n - 1 do
    for j ← 0 to n - 1 do
        C[i, j] ← 0.0
        for k ← 0 to n - 1 do
            C[i, j] ← C[i, j] + A[i, k] * B[k, j]
return C
```



**Bạn có câu hỏi nào  
cho Hoàng Long  
và Cẩm Nguyên  
không?**

