

# BÀI GIẢNG CẤU TRÚC DỮ LIỆU & GIẢI THUẬT CHƯƠNG 2-PHÂN TÍCH THUẬT TOÁN

TS. NGUYỄN ĐÌNH DƯƠNG  
BỘ MÔN KHMT - KHOA CNTT

Email: duongnd@utc.edu.vn

Ngày 15/02/2021

# Nội dung

## Độ phức tạp thuật toán

- 1.1 Một số khái niệm cơ bản
- 1.2 Một số bài toán cơ bản
- 1.3 Bài tập

## Trao đổi

# Nội dung

## Độ phức tạp thuật toán

- 1.1 Một số khái niệm cơ bản
- 1.2 Một số bài toán cơ bản
- 1.3 Bài tập

## Trao đổi

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Ví dụ 1.1 (Mở đầu)

Assume that someone in your class has stolen your favourite chocolate. You need to find that chocolate. Below are some ways to find it.

- **By asking each and every person in the class**, which means if  $n$  number of students are there, then you need to ask  $n$  persons. Hence, the complexity is  $O(n)$ .
- **By asking only one of the students in the class** and if he/she doesn't have that chocolate then ask the same person about the remaining students. So, to each and every person you are asking about the same person and also about the remaining persons. Hence, the complexity is twice when compared to the previous case. i.e.  $O(n^2)$ . The above two ways seem to be difficult.
- So, say if you are divide the entire class into two halves and ask whether the chocolate is on the right side or the left side. If it is on the right side, then divide the number of people on the right side

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Ví dụ 1.2 (Kiểm tra số nguyên tố)

```
bool is_prime(int N){  
    if (N < 2) // Nếu N nhỏ hơn 2 thì chắc chắn không phải số  
    nguyên tố.  
        return false;  
    // Nếu N có ước trong đoạn [2, N - 1] thì N không phải số  
    nguyên tố.  
    for (int i = 2; i < N; ++i)  
        if (N % i == 0)  
            return false;  
    return true; }
```

Thuật toán này cần  $N - 2$  bước kiểm tra trong vòng lặp. Giả sử cần kiểm tra một số có khoảng 25 chữ số và ta sở hữu một siêu máy tính có thể tính toán được một trăm nghìn tỉ ( $10^{14}$ ) phép tính trên giây, thì tổng thời gian cần để kiểm tra là:

$$10^{25}$$

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

Ví dụ 1.3 (Kiểm tra số nguyên tố)

```
bool is_prime(int N){  
    if (N < 2) // Nếu N nhỏ hơn 2 thì chắc chắn không phải số  
    nguyên tố.  
        return false;  
    // Nếu N có ước trong đoạn [2, N - 1] thì N không phải số  
    nguyên tố.  
    for (int i = 2; i * i <= N; ++i)  
        if (N % i == 0)  
            return false;  
    return true; }
```

Thời gian kiểm tra sẽ giảm xuống còn:

$$\frac{\sqrt{10^{25}}}{10^{14}} \approx 0.03\text{giây}$$

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Sự cần thiết phải phân tích thuật toán ?

Một bài toán có thể có một số giải thuật khác nhau, cần phải đánh giá các giải thuật đó để lựa chọn một giải thuật tốt (nhất).

- Tính đúng đắn
- Tính đơn giản
- Tính tối ưu (hiệu quả)

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Sự cần thiết phải phân tích thuật toán ?

Một bài toán có thể có một số giải thuật khác nhau, cần phải đánh giá các giải thuật đó để lựa chọn một giải thuật tốt (nhất).

- Tính đúng đắn
- Tính đơn giản
- Tính tối ưu (hiệu quả)

### Độ phức tạp thuật toán

- Độ phức tạp của một thuật toán là 1 hàm phụ thuộc vào kích thước của dữ liệu đầu vào: Tìm xem 1 đối tượng có trong danh sách  $n$  phần tử hay không? sắp xếp tăng dần dãy số gồm  $n$  số, nhân hai ma trận vuông cấp  $n$ , ...
- Khi phân tích độ phức tạp thuật toán: thời gian tính toán (độ phức tạp về thời gian) và dung lượng bộ nhớ (độ phức tạp về không gian)  $\rightarrow T(n)$ .

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

- Trong một số trường hợp, kích thước dữ liệu đầu vào là như nhau, nhưng thời gian tính lại rất khác nhau

### Ví dụ 1.4

Để tìm số nguyên tố đầu tiên có trong dãy: ta duyệt dãy từ trái sang phải

Dãy 1: 3 9 8 12 15 20 (thuật toán dừng ngay khi xét phần tử đầu tiên)

Dãy 2: 9 8 3 12 15 20 (thuật toán dừng khi xét phần tử thứ ba)

Dãy 3: 9 8 12 15 20 3 (thuật toán dừng khi xét phần tử cuối cùng)

⇒ 3 loại thời gian tính

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

- **Thời gian tính tốt nhất (Best-case):**  $T(n)$  là thời gian tối thiểu cần thiết để thực hiện thuật toán với mọi bộ dữ liệu đầu vào kích thước  $n$ .
- **Thời gian tính trung bình (Average-case):**
  - $T(n)$ : Thời gian trung bình cần thiết để thực hiện thuật toán trên tập hữu hạn các đầu vào kích thước  $n$ .
  - Rất hữu ích, nhưng khó xác định.
- **Thời gian tính tồi nhất (Worst-case):**
  - $T(n)$ : Thời gian nhiều nhất cần thiết để thực hiện thuật toán với mọi bộ dữ liệu đầu vào kích thước  $n$ .
  - Dễ xác định

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Đánh giá bằng cách nào?

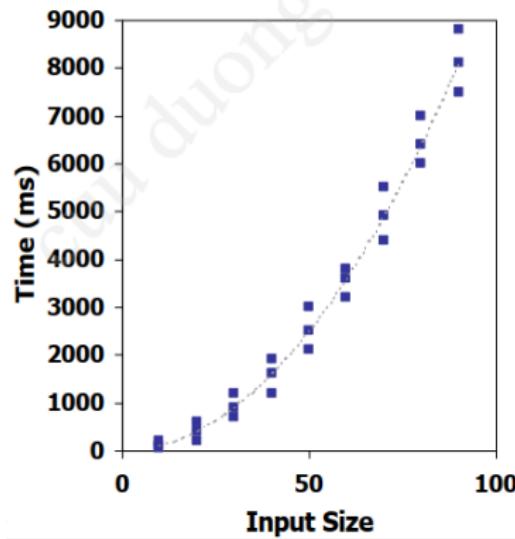
- Hai cách đánh giá độ phức tạp thuật toán:
  - Từ thời gian chạy thực nghiệm
  - Lý thuyết: khái niệm xấp xỉ tiệm cận

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Phân tích thời gian thuật toán bằng thực nghiệm

- Ta có thể đánh giá thuật toán bằng phương pháp thực nghiệm thông qua việc cài đặt thuật toán, rồi chọn các bộ dữ liệu đầu vào thử nghiệm:
  - Viết chương trình thực hiện thuật toán
  - Chạy chương trình với các dữ liệu đầu vào kích thước khác nhau
  - Sử dụng hàm `clock()` để đo thời gian chạy chương trình
  - Vẽ đồ thị kết quả



# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Phân tích thời gian thuật toán bằng thực nghiệm

- Bắt buộc phải cài đặt chương trình thì mới có thể đánh giá được thời gian tính của thuật toán → thuật toán sẽ chịu sự hạn chế của ngữ lập trình này và hiệu quả của thuật toán sẽ bị ảnh hưởng bởi trình độ của người cài đặt.
- Kết quả thu được sẽ không bao gồm thời gian chạy của những dữ liệu đầu vào không được chạy thực nghiệm → cần chọn được các bộ dữ liệu thử đặc trưng cho tất cả tập các dữ liệu vào của thuật toán → rất khó khăn và tốn nhiều chi phí.
- Để so sánh thời gian tính của hai thuật toán, cần phải chạy thực nghiệm hai thuật toán trên cùng một máy, và sử dụng cùng một phần mềm.

⇒ tìm kiếm những phương pháp đánh giá thuật toán hình thức hơn, ít phụ thuộc môi trường cũng như phần cứng hơn → phương pháp đánh giá thuật toán theo hướng xấp xỉ tiệm cận.

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Phân tích thời gian thuật toán bằng lý thuyết

- Sử dụng giả ngôn ngữ (pseudo-code) để mô tả thuật toán, thay vì tiến hành cài đặt thực sự
- Phân tích thời gian tính của thuật toán như là hàm của kích thước dữ liệu đầu vào  $\rightarrow T(n)$
- Phân tích tất cả các trường hợp có thể có của dữ liệu đầu vào
- Cho phép đánh giá thời gian tính của thuật toán không bị phụ thuộc vào phần cứng/phần mềm chạy chương trình (Thay đổi phần cứng/phần mềm chỉ làm thay đổi thời gian chạy một lượng hằng số, chứ không làm thay đổi tốc độ tăng của thời gian tính)

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Kí hiệu tiệm cận (Asymptotic notation):

$O, \Theta, \Omega$

- Được sử dụng để mô tả thời gian tính của thuật toán, mô tả tốc độ tăng của thời gian chạy phụ thuộc vào kích thước dữ liệu đầu vào.
- Được xác định đối với các hàm nhận giá trị nguyên không âm
- Dùng để so sánh tốc độ tăng của 2 hàm  
Ví dụ:  $f(n) = \Theta(n^2)$ : mô tả tốc độ tăng của hàm  $f(n)$  và  $n^2$

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Các phép toán cơ bản thường dùng

- Gán giá trị cho biến số
- Gọi hàm hay thủ tục
- Thực hiện các phép toán số học
- Tham chiếu vào mảng
- Trả kết quả
- Thực hiện các phép so sánh

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

Ví dụ 1.5 (Phân tích thời gian tính toán)

SumOfArray ( $A[1, 2, \dots, n]$ ):

$s \leftarrow 0$

for  $i \leftarrow 1$  to  $n$

$s \leftarrow s + A[i]$

return  $s$

### ⊕ Nhận xét:

- SumOfArray sử dụng khoảng  $5n$  phép toán cơ bản.
- Không phải thuật toán nào cũng tìm được chính xác số phép toán cơ bản  $\rightarrow O(\cdot)$  (big-O) được sử dụng.
- $O(\cdot)$  cho phép đếm **tương đối** số phép toán cơ bản.
- Trong Ví dụ (1.5), số phép toán cơ bản đều không quá  $C.n \rightarrow$  SumOfArray có độ phức tạp thời gian  $O(n)$ .

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Định nghĩa 1.1

$T(n) = O(f(n))$  nếu tồn tại hằng số  $c$ ,  $n_0 > 0$  sao cho  $T(n) \leq cf(n)$  với mọi  $n \geq n_0$ .

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Định nghĩa 1.1

$T(n) = O(f(n))$  nếu tồn tại hằng số  $c$ ,  $n_0 > 0$  sao cho  $T(n) \leq cf(n)$  với mọi  $n \geq n_0$ .

### Ví dụ 1.6

- a)  $8n - 2 = O(n)$  vì  $8n - 2 \leq 8n$  với mọi  $n \geq 1$ .
- b)  $T(n) = 3n^2 + 2n + 4 \Rightarrow T(n) = O(n^2)$  vì  $3n^2 + 2n + 4 < 4n^2$  với mọi  $n \geq 1000$ .
- c)  $T(n) = 3n^2 + 2n + 4 \Rightarrow T(n) = O(n^3)$  vì  $3n^2 + 2n + 4 < n^3$  với mọi  $n \geq 1000$ .

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Định nghĩa 1.1

$T(n) = O(f(n))$  nếu tồn tại hằng số  $c, n_0 > 0$  sao cho  $T(n) \leq cf(n)$  với mọi  $n \geq n_0$ .

### Ví dụ 1.6

- $8n - 2 = O(n)$  vì  $8n - 2 \leq 8n$  với mọi  $n \geq 1$ .
- $T(n) = 3n^2 + 2n + 4 \Rightarrow T(n) = O(n^2)$  vì  $3n^2 + 2n + 4 < 4n^2$  với mọi  $n \geq 1000$ .
- $T(n) = 3n^2 + 2n + 4 \Rightarrow T(n) = O(n^3)$  vì  $3n^2 + 2n + 4 < n^3$  với mọi  $n \geq 1000$ .

⊕ **Nhận xét:**  $T(n) = O(f(n)) \rightarrow$  thuật toán có thời gian chạy **nhiều nhất** là  $f(n)$  (upper bound).

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

- Các lệnh đơn (lệnh khai báo, gán, nhập xuất dữ liệu, phép toán số học,...): Thời gian  $O(1)$ .
- Các khối lệnh: Giả sử một khối lệnh gồm các câu lệnh  $S_1, S_2, \dots, S_m$  có thời gian thực hiện lần lượt là  $O(f_1(n)), O(f_2(n)), \dots, O(f_m(n))$  thì thời gian thực hiện của cả khối lệnh là:  
 $O(\max(f_1(n), f_2(n), \dots, f_m(n)))$ .
- Câu lệnh rẽ nhánh:

```
if <Điều kiện>
    <Câu lệnh 1>
    else
        <Câu lệnh 2>
```

Giả sử thời gian thực hiện của câu lệnh 1 và câu lệnh 2 lần lượt là  $O(f_1(n))$  và  $O(f_2(n))$  thì thời gian thực hiện lệnh rẽ nhánh là:  
 $O(\max(f_1(n), f_2(n)))$ .

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

- Câu lệnh lặp: Giả sử thời gian thực hiện phần thân của lệnh lặp là  $O(f(n))$  và số lần lặp tối đa của vòng lặp là  $g(n)$  thì thời gian thực hiện của cả vòng lặp là  $O(f(n).g(n))$ . Điều này áp dụng cho tất cả các vòng lặp `for`, `while` và `do...while`.
- Sau khi đánh giá được thời gian thực hiện của tất cả các câu lệnh trong chương trình, thời gian thực hiện của toàn bộ chương trình sẽ là thời gian thực hiện của câu lệnh có thời gian thực hiện lớn nhất.

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Ví dụ 1.7

```
#include <iostream>
using namespace std;
int main(){
    int n; // (1)
    cin >> n; // (2)
    int s1 = 0; // (3)
    for (int i = 1; i <= n; ++i) // (4)
        s1 += i; // (5)
    int s2 = 0; // (6)
    for (int i = 1; i <= n; ++i) // (7)
        s2 += i * i; // (8)
    cout << s1 << endl << s2; // (9)
}
```

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

- Các lệnh (1),(2),(3),(5),(6),(8),(9) đều có thời gian thực hiện là  $O(1)$ .
- Vòng for số 4 có số lần lặp là  $n$  và câu lệnh ở phần thân (là câu lệnh (5)) có thời gian thực hiện  $O(1)$ . Vậy cả vòng lặp có thời gian thực hiện là  $O(n)$ . Tương tự với vòng lặp số (7).
- Thời gian thực hiện của cả thuật toán là:  $\max(O(1), O(n)) = O(n)$ .

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Ví dụ 1.8

```
int main(){
    int sum = 0; // (1)
    for (int i = 1; i <= n; ++i) // (2)
        for (int j = 1; j <= i; ++j) // (3)
            sum = sum + 1; // (4)
    cout << sum; // (5)
}
```

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Ví dụ 1.8

```
int main(){  
    int sum = 0; // (1)  
    for (int i = 1; i <= n; ++i) // (2)  
        for (int j = 1; j <= i; ++j) // (3)  
            sum = sum + 1; // (4)  
    cout << sum; // (5)  
}
```

- Câu lệnh (1) và (5) có thời gian thực hiện  $O(1)$ .
- Câu lệnh (4) có thời gian thực hiện  $O(1)$ .
- Ứng với mỗi giá trị  $i$  của câu lệnh lặp (2) thì:
  - Khi  $i = 1$  lệnh lặp (3) thực

- Khi  $i = n$  lệnh lặp (3) thực hiện  $n$  lần.  
→ lệnh lặp (3) có số lần thực hiện là  $\frac{n(n+1)}{2} \Rightarrow$  lệnh lặp (2) có thời gian thực hiện là  $\approx O(n^2)$ .

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Định nghĩa 1.2

$T(n) = \Omega(f(n))$  nếu tồn tại hằng số  $c, n_0 > 0$  sao cho  $T(n) \geq cf(n)$  với mọi  $n \geq n_0$ .

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Định nghĩa 1.2

$T(n) = \Omega(f(n))$  nếu tồn tại hằng số  $c, n_0 > 0$  sao cho  $T(n) \geq cf(n)$  với mọi  $n \geq n_0$ .

### Ví dụ 1.9

- a)  $T(n) = 3n^2 + 2n + 4 \Rightarrow T(n) = \Omega(n^2)$  vì  $3n^2 + 2n + 4 > 3n^2$  với mọi  $n \geq 1$ .
- b)  $T(n) = 3n^2 + 2n + 4 \Rightarrow T(n) = \Omega(n)$  vì  $3n^2 + 2n + 4 > n$  với mọi  $n \geq 1$ .

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Định nghĩa 1.2

$T(n) = \Omega(f(n))$  nếu tồn tại hằng số  $c, n_0 > 0$  sao cho  $T(n) \geq cf(n)$  với mọi  $n \geq n_0$ .

### Ví dụ 1.9

- a)  $T(n) = 3n^2 + 2n + 4 \Rightarrow T(n) = \Omega(n^2)$  vì  $3n^2 + 2n + 4 > 3n^2$  với mọi  $n \geq 1$ .
- b)  $T(n) = 3n^2 + 2n + 4 \Rightarrow T(n) = \Omega(n)$  vì  $3n^2 + 2n + 4 > n$  với mọi  $n \geq 1$ .

⊕ **Nhận xét:**  $T(n) = \Omega(f(n)) \rightarrow$  thuật toán có thời gian chạy **ít nhất** là  $f(n)$  (lower bound).

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Định nghĩa 1.3

$T(n) = \Theta(f(n))$  nếu tồn tại hằng số  $c_1, c_2, n_0 > 0$  sao cho

$$c_1 f(n) \leq T(n) \leq c_2 f(n), \forall n \geq n_0 \text{ hoặc } \lim_{n \rightarrow +\infty} \frac{T(n)}{f(n)} = c, \quad 0 < c < +\infty.$$

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Định nghĩa 1.3

$T(n) = \Theta(f(n))$  nếu tồn tại hằng số  $c_1, c_2, n_0 > 0$  sao cho

$$c_1 f(n) \leq T(n) \leq c_2 f(n), \forall n \geq n_0 \text{ hoặc } \lim_{n \rightarrow +\infty} \frac{T(n)}{f(n)} = c, \quad 0 < c < +\infty.$$

### Ví dụ 1.10

$T(n) = 3n^2 + 2n + 4 \Rightarrow T(n) = \Theta(n^2)$  vì

$$\lim \frac{T(n)}{n^2} = \lim \frac{3n^2 + 2n + 4}{n^2} = 3.$$

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### Định nghĩa 1.3

$T(n) = \Theta(f(n))$  nếu tồn tại hằng số  $c_1, c_2, n_0 > 0$  sao cho

$$c_1 f(n) \leq T(n) \leq c_2 f(n), \forall n \geq n_0 \text{ hoặc } \lim_{n \rightarrow +\infty} \frac{T(n)}{f(n)} = c, \quad 0 < c < +\infty.$$

### Ví dụ 1.10

$T(n) = 3n^2 + 2n + 4 \Rightarrow T(n) = \Theta(n^2)$  vì

$$\lim \frac{T(n)}{n^2} = \lim \frac{3n^2 + 2n + 4}{n^2} = 3.$$

### ⊕ Nhận xét:

- $T(n) = \Theta(f(n)) \rightarrow$  t.toán có thời gian chạy **gần đúng** là  $f(n)$ .
- $\lim_{n \rightarrow +\infty} \frac{T(n)}{f(n)} = 0 \rightarrow T(n) \ll f(n).$

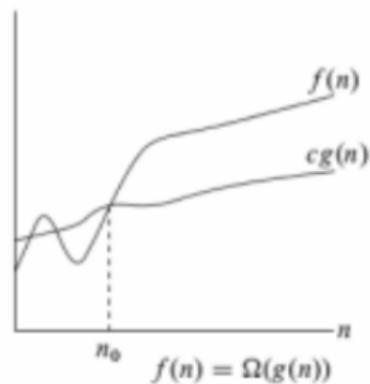
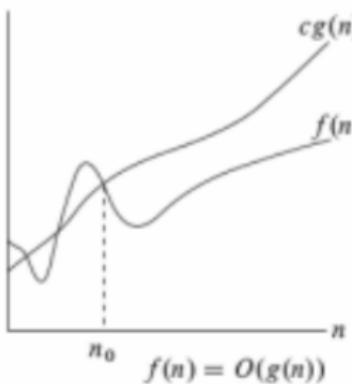
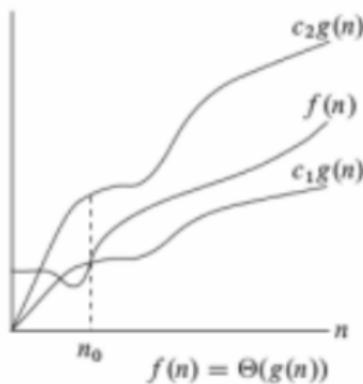
$$\log^\alpha n \ll n^\beta (\beta > 0) \ll a^n (a > 1) \ll n! \ll n^n$$



# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

⊕ **Nhận xét:**  $T(n) = \Theta(f(n)) \Leftrightarrow T(n) = O(f(n))$  và  $T(n) = \Omega(f(n))$ .



# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

Một số quy tắc tính độ phức tạp

- ① **(Quy tắc bỏ hằng số)**  $T(n) = O(c.f(n)) = O(f(n))$ .

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

Một số quy tắc tính độ phức tạp

- ① **(Quy tắc bỏ hằng số)**  $T(n) = O(c.f(n)) = O(f(n))$ .
- ② **(Quy tắc bắc cầu)** Nếu  $T(n) = O(f(n))$  và  $f(n) = O(g(n))$  thì  $T(n) = O(g(n))$ .

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

Một số quy tắc tính độ phức tạp

- ① **(Quy tắc bỏ hằng số)**  $T(n) = O(c.f(n)) = O(f(n))$ .
- ② **(Quy tắc bắc cầu)** Nếu  $T(n) = O(f(n))$  và  $f(n) = O(g(n))$  thì  $T(n) = O(g(n))$ .
- ③ **(Quy tắc lấy max)** Nếu  $T(n) = O(f(n) + g(n))$  và  $g(n) \ll f(n)$  thì  $T(n) = O(f(n))$ .

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

Một số quy tắc tính độ phức tạp

- ① **(Quy tắc bỏ hằng số)**  $T(n) = O(c.f(n)) = O(f(n))$ .
- ② **(Quy tắc bắc cầu)** Nếu  $T(n) = O(f(n))$  và  $f(n) = O(g(n))$  thì  $T(n) = O(g(n))$ .
- ③ **(Quy tắc lấy max)** Nếu  $T(n) = O(f(n) + g(n))$  và  $g(n) \ll f(n)$  thì  $T(n) = O(f(n))$ .
- ④ **(Quy tắc cộng)** Nếu thuật toán thực hiện 2 đoạn chương trình  $P_1$ ,  $P_2$  rời nhau và có độ phức tạp lần lượt là  $T_1(n) = O(f(n))$ ,  $T_2(n) = O(g(n))$  thì độ phức tạp của thuật toán  $T(n) = O(f(n) + g(n))$ .

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

Một số quy tắc tính độ phức tạp

- ① **(Quy tắc bỏ hằng số)**  $T(n) = O(c.f(n)) = O(f(n))$ .
- ② **(Quy tắc bắc cầu)** Nếu  $T(n) = O(f(n))$  và  $f(n) = O(g(n))$  thì  $T(n) = O(g(n))$ .
- ③ **(Quy tắc lấy max)** Nếu  $T(n) = O(f(n) + g(n))$  và  $g(n) \ll f(n)$  thì  $T(n) = O(f(n))$ .
- ④ **(Quy tắc cộng)** Nếu thuật toán thực hiện 2 đoạn chương trình  $P_1$ ,  $P_2$  rời nhau và có độ phức tạp lần lượt là  $T_1(n) = O(f(n))$ ,  $T_2(n) = O(g(n))$  thì độ phức tạp của thuật toán  $T(n) = O(f(n) + g(n))$ .
- ⑤ **(Quy tắc nhân)** Nếu thuật toán thực hiện 2 đoạn chương trình  $P_1$ ,  $P_2$ , trong đó  $P_2$  lồng trong  $P_1$  và có độ phức tạp lần lượt là  $T_1(n) = O(f(n))$ ,  $T_2(n) = O(g(n))$  thì độ phức tạp của thuật toán  $T(n) = O(f(n).g(n))$ .

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

### ⑥ (Quy tắc đánh giá hàm đệ quy):

- Giả sử độ phức tạp của hàm đệ quy là  $T(n) \rightarrow$  độ phức tạp của các lời gọi đệ quy thủ tục là  $T(m)$  với  $m < n$ .
- Đánh giá độ phức tạp  $T(n_0)$  với  $n_0$  là cỡ dữ liệu nhỏ nhất.
- Đánh giá thân hàm đệ nhện được quan hệ sau:

$$T(n) = F(T(n_1), T(n_2), \dots, T(n_k)) \quad \text{với } n_1, \dots, n_k < n$$

- Giải phương trình đệ quy này ta nhận được  $T(n)$ .

### Ví dụ 1.11

Đánh giá độ phức tạp của hàm tính  $n!$  sau đây:

Fact(n)

```
if (n==1)
    Fact ← 1
else
    Fact ← n*Fact(n-1)
```

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

- $T(n) = T(n - 1) + C = T(n - 2) + 2C = \dots = T(1) + (n - 1)C = C' + (n - 1)C$
- Độ phức tạp  $T(n) = O(n)$

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

⊕ **Nhận xét:** Khi biểu diễn cấp của độ phức tạp thuật toán bởi  $f(n)$  ta thường chọn  $f(n)$  có dạng đơn giản nhất sao cho  $T(n) = O(f(n))$ .

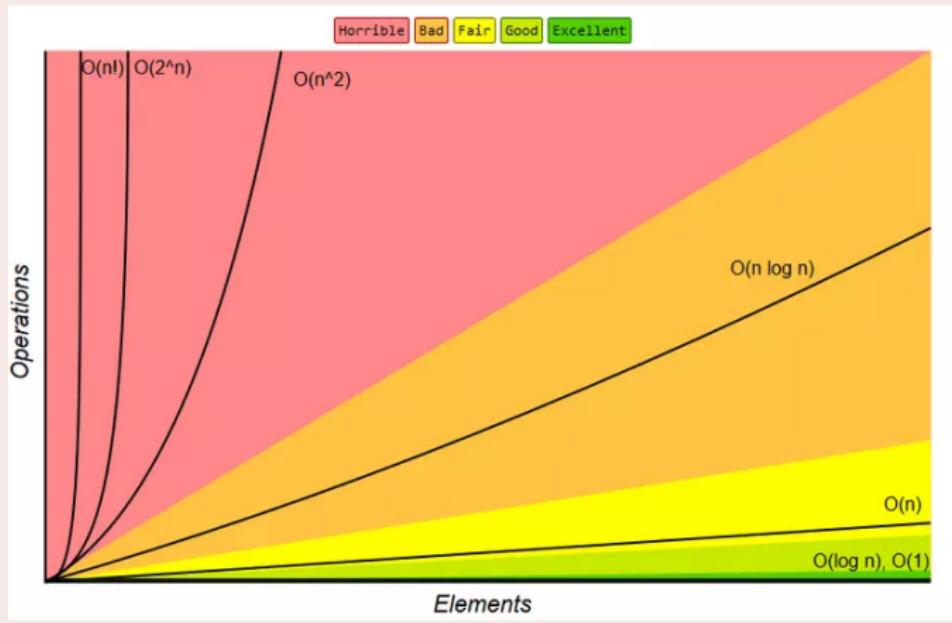
Kí hiệu $O(f(n))$	Tên gọi độ phức tạp
$O(1)$	Hằng số
$O(\log n)$	Logarit
$O(n)$	Tuyến tính
$O(n \log n)$	$n \log n$
$O(n^2)$	Bình phương
$O(n^k)$	Đa thức
$O(2^n)$	Mũ
$O(n!)$	Giai thừa

**Table:** Bảng cấp độ phức tạp thường gặp

# 1. Độ phức tạp thuật toán

## 1. 1. Một số khái niệm cơ bản

$O(1)$ ,  $O(\log n)$ ,  $O(n)$ ,  $O(n \log n)$ ,  $O(n^2)$ ,  $O(n^3)$ ,  $O(2^n)$ ,  $O(n!)$



# 1. Độ phức tạp thuật toán

## 1. 2. Một số bài toán cơ bản

### Ví dụ 1.12

MinOfArray ( $A[1, 2, \dots, n]$ ):

```
Min ← A[1]
for i ← 2 to n
    if (A[i] < Min)
        Min ← A[i]
return Min
```

$P_1$  có độ phức tạp  $O(1)$

$P_2$  có độ phức tạp  $O(n)$

# 1. Độ phức tạp thuật toán

## 1. 2. Một số bài toán cơ bản

### Ví dụ 1.12

MinOfArray ( $A[1, 2, \dots, n]$ ):

```
    Min ← A[1] } ← P1 có độ phức tạp O(1)
    for i ← 2 to n
        if (A[i] < Min) } ← P2 có độ phức tạp O(n)
            Min ← A[i]
    return Min
```

Độ phức tạp của thuật toán là  $T(n) = O(n)$ .

# 1. Độ phức tạp thuật toán

## 1. 2. Một số bài toán cơ bản

### Ví dụ 1.13

Tính trung bình  $i$  phần tử đầu của dãy  $X[1, 2, \dots, n]$ .

PrefixAverage1( $X[1, 2, \dots, n]$ )

```
    Initializing A[1, 2, ..., n]
    for i ← 1 to n
        s ← 0
        for j ← 1 to i
            s ← s + X[j]
        A[i] ← s/i
    return A[1, 2, ..., n]
```

# 1. Độ phức tạp thuật toán

## 1. 2. Một số bài toán cơ bản

### Ví dụ 1.13

Tính trung bình  $i$  phần tử đầu của dãy  $X[1, 2, \dots, n]$ .

PrefixAverage1( $X[1, 2, \dots, n]$ )

    Initializing  $A[1, 2, \dots, n]$

    for  $i \leftarrow 1$  to  $n$

$s \leftarrow 0$

        for  $j \leftarrow 1$  to  $i$

$s \leftarrow s + X[j]$

$A[i] \leftarrow s/i$

    return  $A[1, 2, \dots, n]$

Độ phức tạp của thuật toán là  $O(n^2)$ .

# 1. Độ phức tạp thuật toán

## 1. 2. Một số bài toán cơ bản

Ví dụ (1.13) ...

```
PrefixAverage2(X[1,2,...,n])
    Initializing A[1,2,...,n]
    s ← 0
    for i ← 1 to n
        s ← s + X[i]
        A[i] ← s/i
    return A[1,2,...,n]
```

# 1. Độ phức tạp thuật toán

## 1. 2. Một số bài toán cơ bản

Ví dụ (1.13) ...

```
PrefixAverage2(X[1,2,...,n])
    Initializing A[1,2,...,n]
    s ← 0
    for i ← 1 to n
        s ← s + X[i]
        A[i] ← s/i
    return A[1,2,...,n]
```

Độ phức tạp của thuật toán là  $O(n)$ .

# 1. Độ phức tạp thuật toán

## 1. 2. Một số bài toán cơ bản

### Ví dụ 1.14 (Thuật toán BubbleSort)

BubbleSort( $A[1, 2, \dots, n]$ ):

```
for i ← 1 to n - 1
    for j ← i + 1 to n
        if (A[i] > A[j])
            tmp ← A[i]
            A[i] ← A[j]
            A[j] ← tmp
return A[1, 2, ..., n]
```

# 1. Độ phức tạp thuật toán

## 1. 2. Một số bài toán cơ bản

### Ví dụ 1.14 (Thuật toán BubbleSort)

BubbleSort( $A[1, 2, \dots, n]$ ):

```
for  $i \leftarrow 1$  to  $n - 1$ 
    for  $j \leftarrow i + 1$  to  $n$ 
        if ( $A[i] > A[j]$ )
            tmp  $\leftarrow A[i]$ 
             $A[i] \leftarrow A[j]$ 
             $A[j] \leftarrow \text{tmp}$ 
return  $A[1, 2, \dots, n]$ 
```

- **Cách 1:** với mỗi  $i$ , vòng lặp trong có  $n - (i + 1) + 1 = n - i$  lần lặp  $\rightarrow$  số lần lặp là
$$\sum_{i=1}^{n-1} (n - i) = 1 + 2 + \dots + n - 1 = \frac{n(n - 1)}{2} = O(n^2)$$

# 1. Độ phức tạp thuật toán

## 1. 2. Một số bài toán cơ bản

### Ví dụ 1.14 (Thuật toán BubbleSort)

BubbleSort( $A[1, 2, \dots, n]$ ):

```
for  $i \leftarrow 1$  to  $n - 1$ 
    for  $j \leftarrow i + 1$  to  $n$ 
        if ( $A[i] > A[j]$ )
            tmp  $\leftarrow A[i]$ 
             $A[i] \leftarrow A[j]$ 
             $A[j] \leftarrow \text{tmp}$ 
return  $A[1, 2, \dots, n]$ 
```

- **Cách 1:** với mỗi  $i$ , vòng lặp trong có  $n - (i + 1) + 1 = n - i$  lần lặp  $\rightarrow$  số lần lặp là
$$\sum_{i=1}^{n-1} (n - i) = 1 + 2 + \dots + n - 1 = \frac{n(n - 1)}{2} = O(n^2)$$
- **Cách 2:** sử dụng quy tắc nhân

# 1. Độ phức tạp thuật toán

## 1. 2. Một số bài toán cơ bản

### Ví dụ 1.14 (Thuật toán BubbleSort)

BubbleSort( $A[1, 2, \dots, n]$ ):

```
for  $i \leftarrow 1$  to  $n - 1$ 
    for  $j \leftarrow i + 1$  to  $n$ 
        if ( $A[i] > A[j]$ )
            tmp  $\leftarrow A[i]$ 
             $A[i] \leftarrow A[j]$ 
             $A[j] \leftarrow \text{tmp}$ 
return  $A[1, 2, \dots, n]$ 
```

$P_1$  có độ phức tạp  $O(n)$

- **Cách 1:** với mỗi  $i$ , vòng lặp trong có  $n - (i + 1) + 1 = n - i$  lần lặp  $\rightarrow$  số lần lặp là
$$\sum_{i=1}^{n-1} (n - i) = 1 + 2 + \dots + n - 1 = \frac{n(n - 1)}{2} = O(n^2)$$
- **Cách 2:** sử dụng quy tắc nhân



# 1. Độ phức tạp thuật toán

## 1. 2. Một số bài toán cơ bản

### Ví dụ 1.14 (Thuật toán BubbleSort)

BubbleSort( $A[1, 2, \dots, n]$ ):

```
for i ← 1 to n – 1
    for j ← i + 1 to n
        if (A[i] > A[j])
            tmp ← A[i]
            A[i] ← A[j]
            A[j] ← tmp
return A[1, 2, ..., n]
```

$\left. \begin{array}{l} P_1 \text{ có độ phức tạp } O(n) \\ P_2 \text{ có độ phức tạp } O(n) \end{array} \right\}$

- **Cách 1:** với mỗi  $i$ , vòng lặp trong có  $n - (i + 1) + 1 = n - i$  lần lặp  $\rightarrow$  số lần lặp là
$$\sum_{i=1}^{n-1} (n - i) = 1 + 2 + \dots + n - 1 = \frac{n(n - 1)}{2} = O(n^2)$$
- **Cách 2:** sử dụng quy tắc nhân

# 1. Độ phức tạp thuật toán

## 1. 2. Một số bài toán cơ bản

### Ví dụ 1.15 (Thuật toán MergeSort)

MergeSort( $A[1, 2, \dots, n]$ ):

    if  $n > 1$

$m \leftarrow \lfloor n/2 \rfloor$

        MergeSort( $A[1, 2, \dots, m]$ )

        MergeSort( $A[m+1, 2, \dots, n]$ )

        Merge( $A[1, 2, \dots, n], m$ )

Merge( $A[1, 2, \dots, n]$ ):

$i \leftarrow 1; j \leftarrow m + 1$

    for  $k \leftarrow 1$  to  $n$

        if  $j > n$

$B[k] \leftarrow A[i]; i \leftarrow i + 1$

        else if  $i > m$

$B[k] \leftarrow A[j]; j \leftarrow j + 1$

        else if  $A[i] > A[j]$

$B[k] \leftarrow A[i]; i \leftarrow i + 1$

        else

$B[k] \leftarrow A[j]; j \leftarrow j + 1$

    for  $k \leftarrow 1$  to  $n$ :  $A[k] \leftarrow B[k]$

# 1. Độ phức tạp thuật toán

## 1. 2. Một số bài toán cơ bản

- Gọi  $T(n)$  là độ phức tạp của thuật toán MergeSort  $\rightarrow$  độ phức tạp của hai thủ tục gọi đệ quy lần lượt là  $T(\lfloor n/2 \rfloor)$  và  $T(\lceil n/2 \rceil)$ .

# 1. Độ phức tạp thuật toán

## 1. 2. Một số bài toán cơ bản

- Gọi  $T(n)$  là độ phức tạp của thuật toán MergeSort  $\rightarrow$  độ phức tạp của hai thủ tục gọi đệ quy lần lượt là  $T(\lfloor n/2 \rfloor)$  và  $T(\lceil n/2 \rceil)$ .
- Độ phức tạp của thủ tục Merge là  $O(n)$ , do đó

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) = 2T(n/2) + O(n).$$



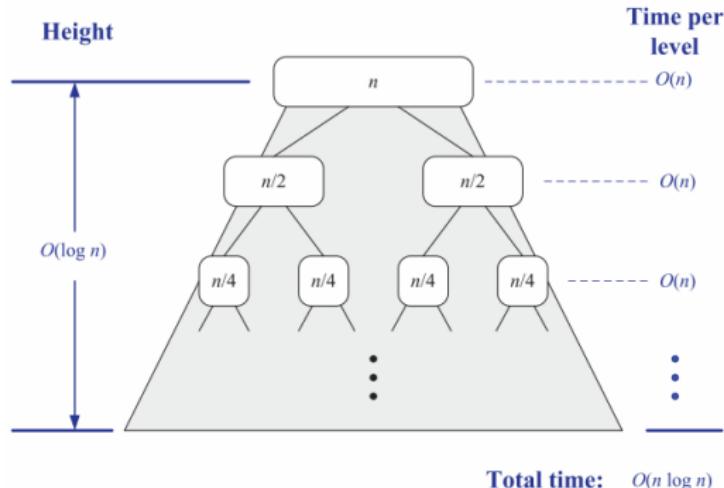
# 1. Độ phức tạp thuật toán

## 1. 2. Một số bài toán cơ bản

- Gọi  $T(n)$  là độ phức tạp của thuật toán MergeSort  $\rightarrow$  độ phức tạp của hai thủ tục gọi đệ quy lần lượt là  $T(\lfloor n/2 \rfloor)$  và  $T(\lceil n/2 \rceil)$ .
- Độ phức tạp của thủ tục Merge là  $O(n)$ , do đó

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) = 2T(n/2) + O(n).$$

- Sử dụng cây đệ quy ta có  $T(n) = O(n \log n)$ .



# 1. Độ phức tạp thuật toán

## 1. 3. Bài tập

### Bài 1

Tìm đánh giá cho hàm

a)  $T(n) = n \log(n!) + (3n^2 + 2n) \log n$

b)  $T(n) = (n + 3) \log(n^2 + 4) + 5n^2$

# 1. Độ phức tạp thuật toán

## 1. 3. Bài tập

### Bài 1

Tìm đánh giá cho hàm

a)  $T(n) = n \log(n!) + (3n^2 + 2n) \log n$

b)  $T(n) = (n + 3) \log(n^2 + 4) + 5n^2$

Giải

**Đáp số:** a)  $T(n) = O(n^2 \log n)$ , b)  $T(n) = O(n^2)$

### Bài 2

Sắp xếp các hàm sau theo thứ tự tiệm cận tốc độ tăng dần

$4n \log n + 2n$	$2^{10}$	$2^{\log n}$
$3n + 100 \log n$	$4n$	$2^n$
$n^2 + 10n$	$n^3$	$n \log n$

# 1. Độ phức tạp thuật toán

## 1. 3. Bài tập

### Bài 1

Tìm đánh giá cho hàm

a)  $T(n) = n \log(n!) + (3n^2 + 2n) \log n$

b)  $T(n) = (n + 3) \log(n^2 + 4) + 5n^2$

Giải

**Đáp số:** a)  $T(n) = O(n^2 \log n)$ , b)  $T(n) = O(n^2)$

### Bài 2

Sắp xếp các hàm sau theo thứ tự tiệm cận tốc độ tăng dần

$4n \log n + 2n$	$2^{10}$	$2^{\log n}$
$3n + 100 \log n$	$4n$	$2^n$
$n^2 + 10n$	$n^3$	$n \log n$

Giải

# 1. Độ phức tạp thuật toán

## 1. 3. Bài tập

### Bài 3

Đánh giá độ phức tạp của thuật toán tìm số Fibonacci thứ  $n$  theo công thức truy hồi sau:

$$F(n) = \begin{cases} 0 & \text{khi } n = 0 \\ 1 & \text{khi } n = 1 \\ F(n - 1) + F(n - 2) & \text{khi } n > 1 \end{cases}$$

# 1. Độ phức tạp thuật toán

## 1. 3. Bài tập

### Bài 3

Đánh giá độ phức tạp của thuật toán tìm số Fibonacci thứ  $n$  theo công thức truy hồi sau:

$$F(n) = \begin{cases} 0 & \text{khi } n = 0 \\ 1 & \text{khi } n = 1 \\ F(n - 1) + F(n - 2) & \text{khi } n > 1 \end{cases}$$

Giải

**Đáp số:**  $O(2^n)$

# 1. Độ phức tạp thuật toán

## 1. 3. Bài tập

### Bài 3

Đánh giá độ phức tạp của thuật toán tìm số Fibonacci thứ  $n$  theo công thức truy hồi sau:

$$F(n) = \begin{cases} 0 & \text{khi } n = 0 \\ 1 & \text{khi } n = 1 \\ F(n - 1) + F(n - 2) & \text{khi } n > 1 \end{cases}$$

Giải

**Đáp số:**  $O(2^n)$

Cải tiến???

# 1. Độ phức tạp thuật toán

## 1. 3. Bài tập

### Bài 4

So sánh độ phức tạp của 2 thuật toán tính giá trị đa thức dưới đây:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \quad \text{tại } x = x_0$$

- Thuật toán 1:
  - Tính giá trị của từng hạng tử: với  $i = 1$  đến  $n$ , tính  $a_i x_0^i$
  - Tính  $P(x_0) = a_0 + \sum_{i=1}^n a_i x_0^i$
- Thuật toán 2:  $P(x) = ((a_n x + a_{n-1}) x + \cdots + a_1) x + a_0$ 
  - $P = a_n$
  - Với  $i = 1$  đến  $n$ :  $P = Px_0 + a_{n-i}$

# 1. Độ phức tạp thuật toán

## 1. 3. Bài tập

### Bài 5

Hãy xây dựng và đánh giá độ phức tạp của các thuật toán sau:

- ① Thuật toán tìm phần tử lớn nhất của một dãy hữu hạn số thực
- ② Thuật toán tìm phần tử bé nhất của một tập con của  $N$
- ③ Thuật toán sắp xếp lại một dãy theo thứ tự tăng dần
- ④ Thuật toán tìm một dãy số liên tiếp nhau (dài nhất có thể) có tổng dương trong một dãy số thực cho trước

# Nội dung

## Độ phức tạp thuật toán

- 1.1 Một số khái niệm cơ bản
- 1.2 Một số bài toán cơ bản
- 1.3 Bài tập

## Trao đổi

# TRAO ĐỔI