

Phân tích độ phức tạp của thuật toán

Trước khi chúng ta đi sâu vào chủ đề Phân tích độ phức tạp của thuật toán, chúng ta hãy hiểu rõ về thuật toán thực sự là gì.

I. Tìm hiểu về thuật toán:

What is the algorithm?

Thuật toán là một bộ quy tắc / hướng dẫn cụ thể mà máy tính sẽ tuân theo để giải quyết một vấn đề cụ thể. Nói cách khác, chúng ta cần nói cho máy tính biết cách xử lý dữ liệu, để chúng ta có thể hiểu ý nghĩa của nó. Thuật toán là phương pháp để giải quyết một bài toán, công việc cụ thể

What is Algorithmic Analysis?

Chúng ta muốn xử lý số lượng lớn dữ liệu và muốn thực hiện nhanh nhất có thể. Đây là nơi phân tích thuật toán xuất hiện.

Khi thực hiện phân tích thuật toán, chúng ta muốn đánh giá hiệu suất của thuật toán theo kích thước đầu vào của nó.

Chúng ta muốn xử lý số lượng lớn dữ liệu và thực hiện nhanh nhất có thể.

Yếu tố đầu tiên được quan tâm là số lượng đầu vào phải được xử lý. Do đó, nhiều dữ liệu bằng thời gian hơn. Một yếu tố khác mà bạn có thể muốn xem xét quan trọng, là tốc độ của máy chủ.

1 bài toán có rất nhiều thuật toán khác nhau

- Vì vậy phải phân tích thuật toán để lựa chọn 1 thuật toán tốt nhất - cải tiến thuật toán hiện có để nó tốt hơn

When is an Algorithm considered best?

- Thực hiện đúng
- Tốn ít bộ nhớ
- Thực hiện nhanh

Biểu diễn thuật toán:

- Biểu diễn bằng cách liệt kê theo bước
- Biểu diễn bằng lưu đồ
- Biểu diễn bằng ngôn ngữ cấu trúc

Ví dụ:

Ví dụ: mô tả thuật toán tìm ước số chung lớn nhất của hai số nguyên.

Input: Hai số nguyên a, b .

Output: Ước số chung lớn nhất của a, b .

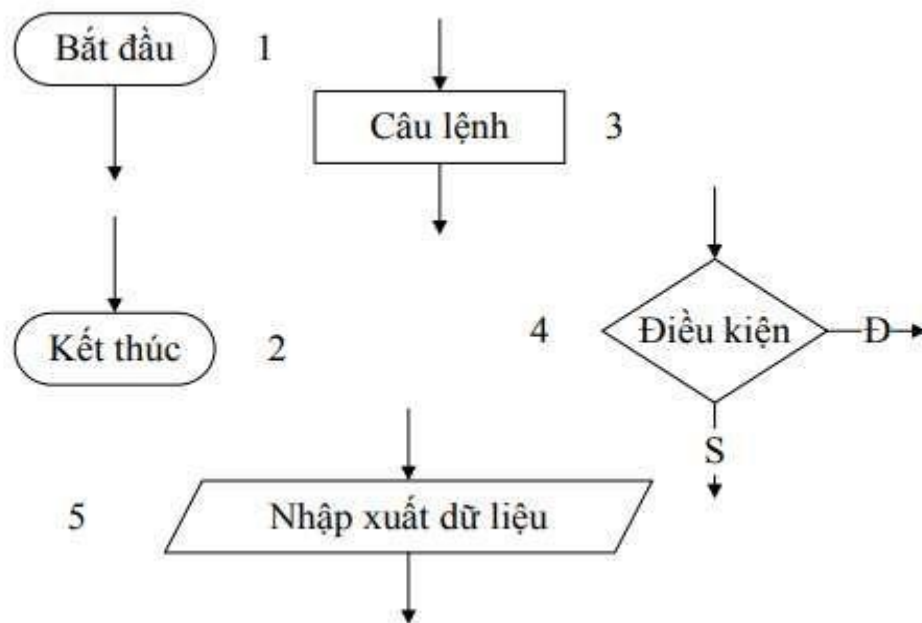
Thuật toán:

Bước 1: Nếu $a=b$ thì $USCLN(a, b)=a$.

Bước 2: Nếu $a > b$ thì tìm USCLN của $a-b$ và b , quay lại bước 1;

Bước 3: Nếu $a < b$ thì tìm USCLN của a và $b-a$, quay lại bước 1;

-Sử dụng sơ đồ giải thuật:



Các tiêu chí để đánh giá thuật toán là gì:

+Thuật toán đơn giản dễ hiểu, dễ cài đặt

+Dựa vào thời gian thực hiện và tài nguyên mà thuật toán sử dụng để thực hiện trên các bộ dữ liệu

Các Đặc trưng của thuật toán

+Tính đúng đắn: Thuật toán cần đảm bảo cho một kết quả đúng sau khi thực hiện đối với các bộ dữ liệu đầu vào.

→Đặc trưng quan trọng nhất đối với mỗi thuật toán

+Tính dừng: Thuật toán cần đảm bảo sẽ dừng một số hữu hạn bước

+Tính xác định:Các bước của thuật toán được phát biểu rõ ràng cụ thể, tránh gây nhập nhằng hoặc nhầm lẫn

+Tính hiệu quả: Khả năng giải quyết bài toán đặt ra trong thời gian hoặc các điều kiện cho phép

+Tính phổ quát: Có thể giải quyết một lớp các bài toán tương tự

II. Độ phức tạp của thuật toán

Phân tích giải thuật có thể đánh giá được giải thuật

- Tính đúng đắn
- Tính đơn giản
- Tính nhanh chóng (thời gian thực thi)

Thời gian thực hiện chương trình

. Là một hàm của kích thước dữ liệu vào, ký hiệu $T(n)$ trong đó n là kích thước (độ lớn) của dữ liệu vào

. Thời gian thực hiện chương trình là một hàm không âm, tức là $T(n) \geq 0 \forall n \geq 0$

. Dùng để đo thời gian thực hiện chương trình

Có thể được xác định thời gian thực hiện chương trình từ mấy yếu tố?

Hai yếu tố chính:

- Thời gian thực hiện từng lệnh. Trong đó, những phép toán thường được đánh giá như phép so sánh và phép gán.
- Tần suất thực hiện của các lệnh đó. Thường được đánh giá phụ thuộc độ lớn của dữ liệu.

→ Tính độ phức tạp thời gian thực hiện của giải thuật = độ đo sự thực thi của giải thuật

Tỷ suất tăng (growth rate):

$T(n)$ có tỷ suất tăng $f(n)$ nếu tồn tại hằng $C > 0$ và n_0 sao cho

$$T(n) \leq C f(n) \forall n \geq n_0$$

Cho một hàm không âm $T(n)$, luôn tồn tại tỷ suất tăng $f(n)$ của nó

Ví dụ: $T(0) = 1, T(1) = 4, T(n) = (n+1)^2$, ta có:

$$f(n) = n^2 \text{ (với } C = 4, n_0 = 1 \text{)}$$

Ta có thể chứng minh được:

$\text{Tỷ suất tăng của } T(n) = 3n^3 + 2n \text{ là } n^3$

Ví dụ: cho một hàm $T(n)$, $T(n)$: độ phức tạp của $f(n)$

→ $T(n)$ có tỷ suất gia tăng là $f(n)$ và ký hiệu $O(f(n))$

$$O(c \cdot f(n)) = O(f(n)) \text{ với } c \text{ là một hằng số dương}$$

Có thể nói độ phức tạp của giải thuật là một hàm chặn trên của hàm thời gian. Vì hàm nhân tử C trong hàm chặn không có ý nghĩa nên ta có thể bỏ qua. Khi nói đến độ phức tạp của giải thuật là ta muốn nói đến hiệu quả của thời gian thực hiện của chương trình nên ta có thể xem việc xác định thời gian thực hiện của chương trình là xác định độ phức tạp của thuật toán.

Phân loại độ phức tạp của thuật toán

Độ phức tạp của thuật toán chia làm hai loại:

- Độ phức tạp không gian (M): Số ô nhớ cần thiết để có thể triển khai thực hiện thuật toán.
- Độ phức tạp thời gian (T): Thời gian cần thiết để thực hiện xong các bước của thuật toán, cho ra

*Khi xét đến độ phức tạp thuật toán, chúng ta chú ý các điều sau đây:

Trong trường hợp giả thiết sử dụng hết RAM để thực hiện thuật toán, chúng ta chỉ cần quan tâm đến độ phức tạp thời gian (T) để đánh giá thuật toán.

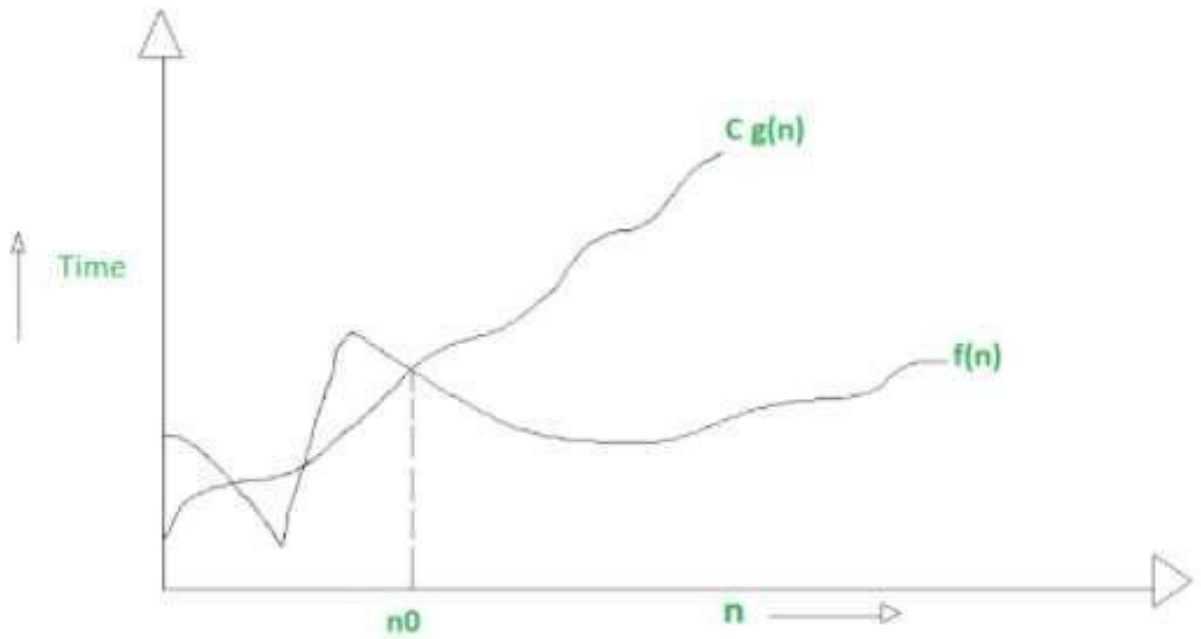
-**Big-O** được gọi là tình huống xấu nhất của một chương trình. Với một đầu vào rất lớn (nghĩ là vô hạn), Big-O sẽ cho chúng ta ý tưởng về việc thuật toán của chúng ta có thể chạy chậm như thế nào.

- $f(n)$ mô tả thời gian chạy của một thuật toán; $f(n)$ là $O(g(n))$ nếu tồn tại hằng số dương C và n_0 sao cho

$$0 \leq f(n) \leq Cg(n) \quad \forall n \geq n_0$$

n = được sử dụng để cung cấp giới hạn trên của một hàm

ví dụ:



-Big-omega

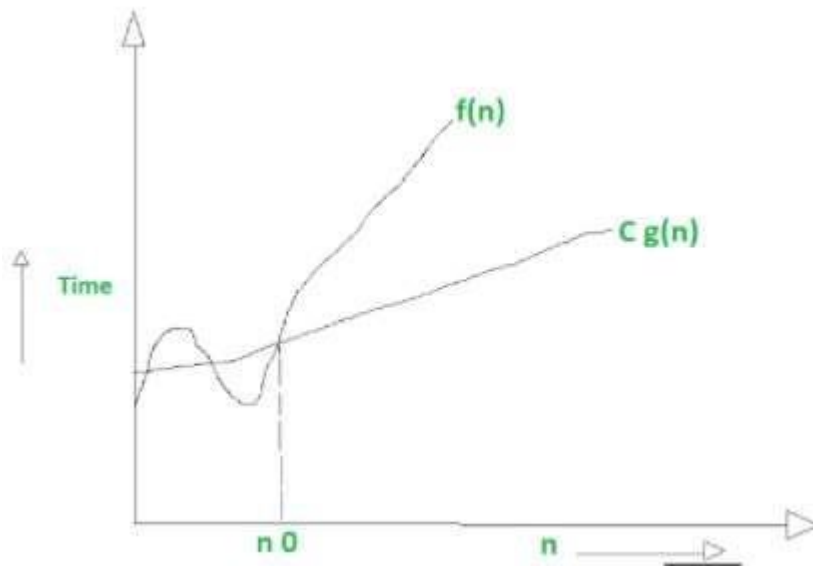
-Nếu Big O đề cập đến tình huống xấu nhất trong chương trình của chúng ta và BigOmega đề cập đến tình huống tốt nhất

Cho $f(n)$ xác định thời gian chạy của một thuật toán;

$f(n)$ được cho là $\Omega(g(n))$ nếu tồn tại hằng số dương C và (n_0) sao cho

$$0 \leq C g(n) \leq f(n) \quad \forall n \geq n_0$$

Ví dụ:



-Big-Theta

là thời gian chạy trường hợp trung bình của chương trình

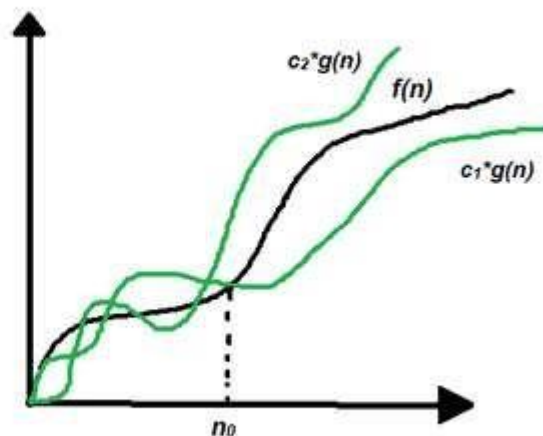
$\Theta(g(n)) = \{f(n) : \text{tồn tại các hằng số dương } c_1, c_2 \text{ và } n_0 \text{ sao cho } 0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n) \text{ với mọi } n \geq n_0\}$

Lưu ý: $\Theta(g)$ là một tập hợp

Định nghĩa trên có nghĩa là, nếu $f(n)$ là giá trị của $g(n)$, thì giá trị $f(n)$ luôn nằm giữa $c_1 * g(n)$ và $c_2 * g(n)$ đối với các giá trị lớn của n ($n \geq n_0$). Định nghĩa của theta cũng yêu cầu $f(n)$ phải không âm đối với các giá trị của n lớn hơn n_0 .

Ví dụ:

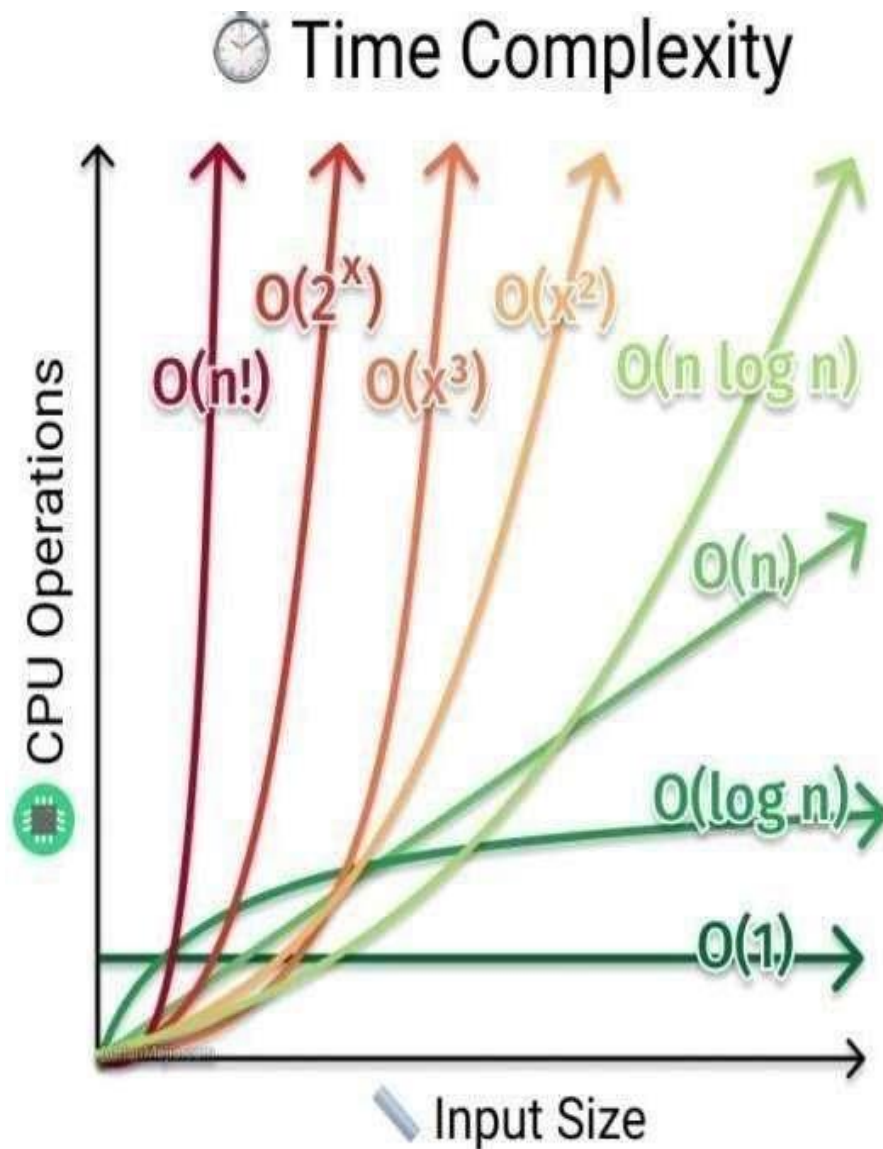
Biểu diễn đồ họa:



Biểu diễn đồ họa

Khác	Big O	Big omega	Big Theta
1	Trường hợp tệ nhất	Trường hợp tốt nhất	Trường hợp j trung bình
2	Big-O là thước đo khoảng thời gian dài nhất có thể mất để thuật toán hoàn thành	Big- Ω mất một khoảng thời gian nhỏ so với Big-O có thể mất một khoảng thời gian để thuật toán hoàn thành.	Big- Θ mất một khoảng thời gian rất ngắn so với Big-O và Big - Ω nó có thể mất để thuật toán hoàn thành.
3	tốc độ của một thuật toán nhỏ hơn hoặc bằng một giá trị cụ thể	tốc độ của thuật toán lớn hơn hoặc bằng một giá trị cụ thể	tốc độ thuật toán bằng một giá trị cụ thể

Các loại độ phức tạp



2.1. Constant $\{O(1)\}$

Một chương trình có độ phức tạp hằng số $O(1)$ nếu nó chỉ chạy đúng một số lượng thao tác, cho dù có tăng kích thước đầu vào.

□ Vì vậy, nó không phụ thuộc vào số lượng phần tử đầu vào NN.

Những thao tác gán hay phép tính cộng trừ nhân chia đều có độ phức tạp $O(1)$.
Vd: bảng băm, mỗi thao tác lấy dữ liệu đều tốn một số lượng thao tác như nhau, cho dù có tăng số lượng key trong bảng băm

2.2. Logarithmic

Chương trình có độ phức tạp logarit $O(\log N)$ sẽ chậm hơn độ phức tạp hằng số khi tăng kích thước đầu vào.

Trong vòng lặp nếu sau mỗi bước lặp thì thao tác còn lại giảm đi thì các bạn có thể ngầm dự đoán nó có độ phức tạp logarit.

Vd: Binary Search là một thuật toán điển hình, sau mỗi bước lặp, kích thước mảng để xét giảm đi một nửa nên có độ phức tạp $O(\log_2 N)$.

Nếu thuật toán giảm đi gấp 3 sau mỗi bước lặp thì có thể xem nó có độ phức tạp $O(\log_3 N) \rightarrow O(\log N)$.

2.3. Linear

Độ phức tạp tuyến tính $O(N)$ ám chỉ rằng nếu tăng gấp đôi kích thước đầu vào thì thời gian chạy cũng tăng gấp đôi, nghĩa là tăng tuyến tính theo kích thước của N .

Dấu hiệu: một vòng for duyệt hết N phần tử.

2.4. Linearithmic

Độ phức tạp $O(N \log N)$ biểu thị thời gian chạy cho kích thước N là $N \log N$. Nghĩa là nhiều hơn $O(N)$ nhưng nhỏ hơn $O(N^2)$.

Chẳng hạn, một chương trình bạn đang viết có 2 vòng for lồng nhau.

Vòng for ngoài duyệt đến N , vòng for bên trong sau mỗi lần lặp thì giảm đi, thì có thể xem là $O(N \log N)$.

Vd: Merge sort hay Quick sort.

2.5. Quadratic

$O(N^2)$ biểu thị thời gian chạy là N^2 cho dữ liệu kích thước N . Thông thường chúng ta viết 2 vòng for lồng nhau mà đều duyệt đến N thì có thể coi là $O(N^2)$

vd: Bubble sort, Selection sort, ...

2.6. Cubic

Tương tự $O(N^2)$, thuật toán có độ phức tạp $O(N^3)$ thường được viết như 3 vòng for lồng nhau và kiểm tra tất cả các cặp 3 phần tử.

*Khi đánh giá thuật toán ta thường đánh giá thuật toán trong trường hợp xấu nhất

TRƯỜNG HỢP TỐT NHẤT, XẤU NHẤT VÀ TRUNG BÌNH

Tìm kiếm tuần tự

Search(x, a, n)

0	1	2	3	4	5	6	7
5	6	3	10	1	4	7	9

Search(5, a, 8): Trường hợp tốt nhất

Search(10, a, 8): Trường hợp TB

Search(15, a, 8): Trường hợp xấu nhất

LỰA CHỌN THUẬT TOÁN NHƯ THẾ NÀO?

Cùng một bài toán, có 2 thuật toán P1 và P2 với thời gian thực hiện là $T1(n) = 100n^2$ và $T2(n) = 5n^3$.

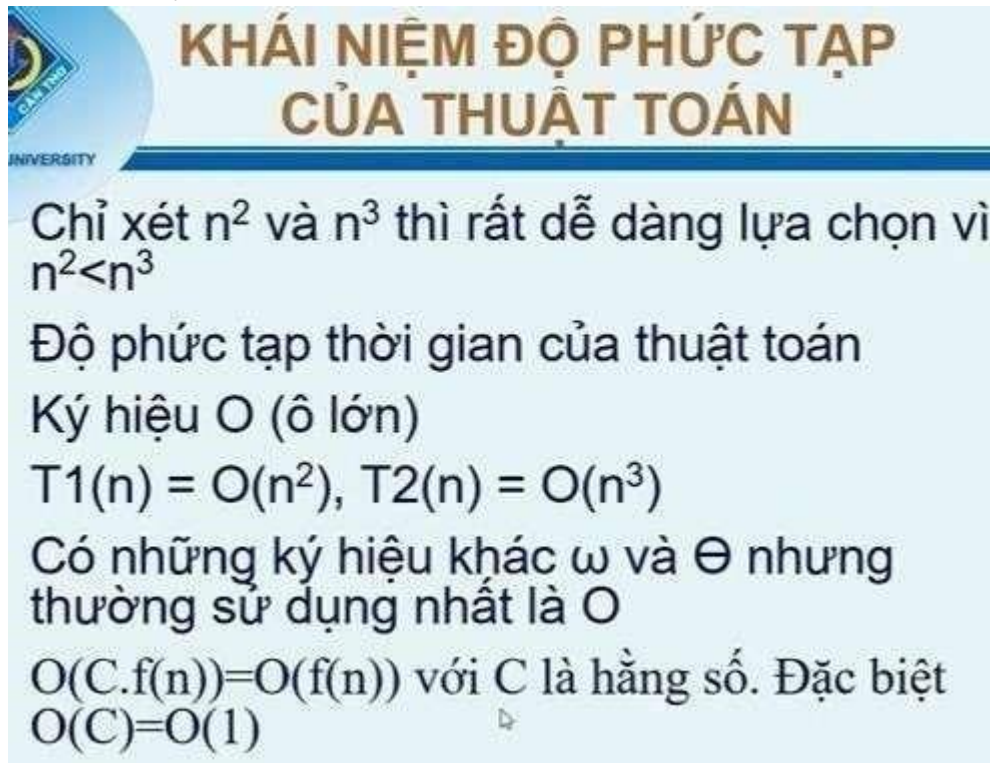
Chọn thuật toán nào?

Khi $n < 20$ thì $T2 < T1$. Khi $n > 20$ thì $T1 < T2$

Chọn P1 vì hầu hết các trường hợp $T1 < T2$

Dùng thời gian để lựa chọn rất khó khăn vì phải so sánh 2 đa thức.

Ta thường xét như này:



KHÁI NIỆM ĐỘ PHỨC TẠP CỦA THUẬT TOÁN

Chỉ xét n^2 và n^3 thì rất dễ dàng lựa chọn vì $n^2 < n^3$

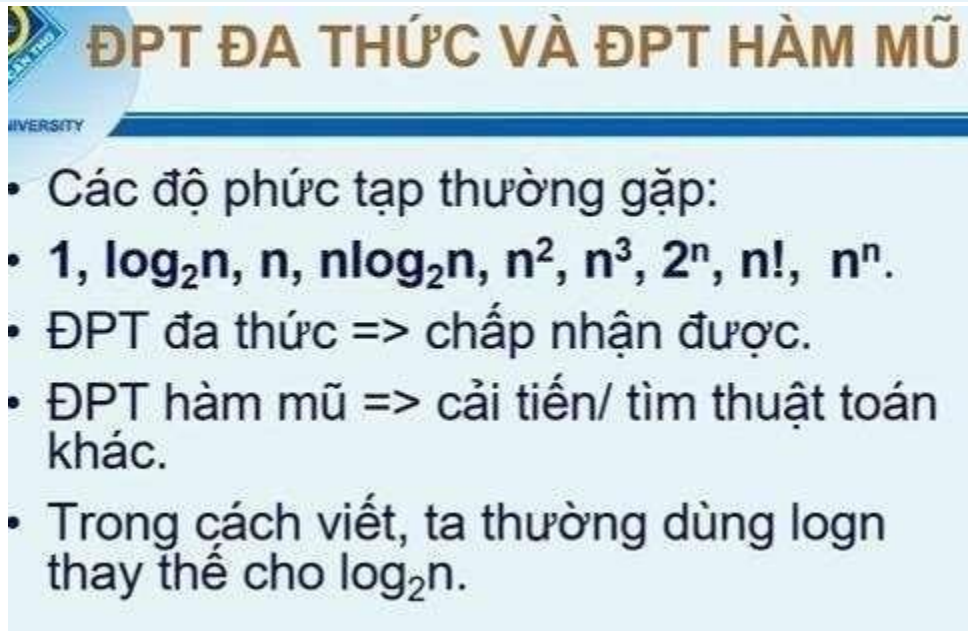
Độ phức tạp thời gian của thuật toán

Ký hiệu O (ô lớn)

$T1(n) = O(n^2)$, $T2(n) = O(n^3)$

Có những ký hiệu khác ω và Θ nhưng thường sử dụng nhất là O

$O(C \cdot f(n)) = O(f(n))$ với C là hằng số. Đặc biệt $O(C) = O(1)$



ĐPT ĐA THỨC VÀ ĐPT HÀM MŨ

- Các độ phức tạp thường gặp:
- **1, $\log_2 n$, n , $n \log_2 n$, n^2 , n^3 , 2^n , $n!$, n^n .**
- ĐPT đa thức => chấp nhận được.
- ĐPT hàm mũ => cải tiến/ tìm thuật toán khác.
- Trong cách viết, ta thường dùng $\log n$ thay thế cho $\log_2 n$.

Nên chọn đa thức->

Nếu là hàm mũ thì nên tìm thuật toán khác tốt hơn

-

QUY TẮC XÁC ĐỊNH ĐỘ PHỨC TẠP

- Độ phức tạp tính toán của giải thuật: $O(f(n))$
- Việc xác định độ phức tạp tính toán của giải thuật trong thực tế có thể tính bằng một số quy tắc đơn giản sau:

– Quy tắc bỏ hằng số:

$$T(n) = O(c \cdot f(n)) = O(f(n)) \text{ với } c \text{ là một hằng số dương}$$

– Quy tắc lấy max:

$$T(n) = O(f(n) + g(n)) = O(\max(f(n), g(n)))$$

– Quy tắc cộng:

$$T1(n) = O(f(n))$$

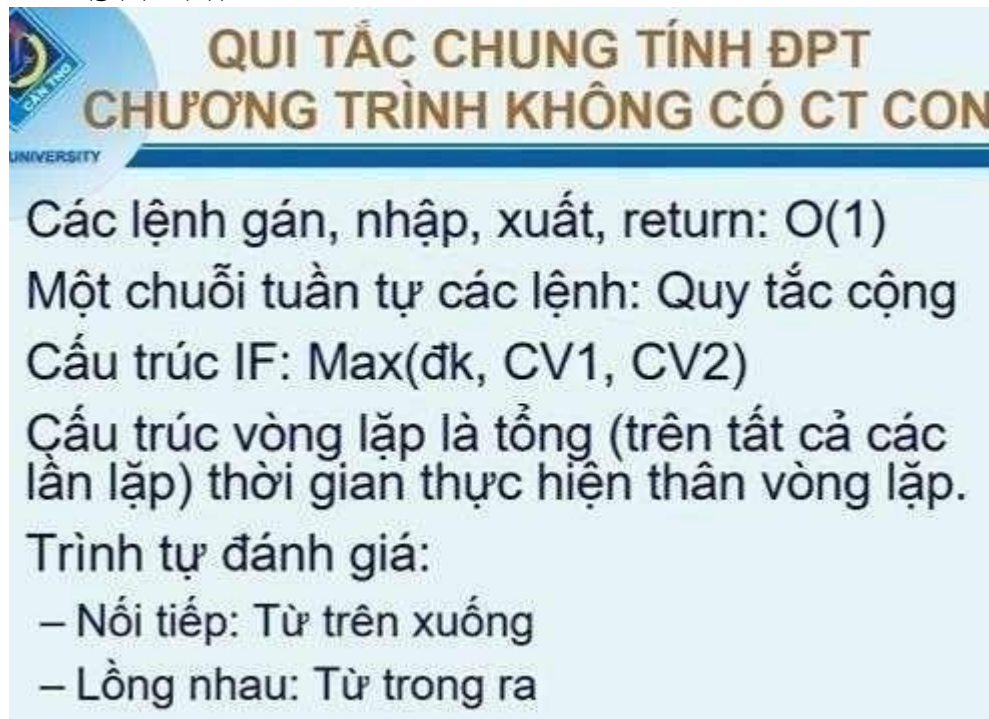
$$T2(n) = O(g(n))$$

$$T1(n) + T2(n) = O(f(n) + g(n))$$

– Quy tắc nhân:

Đoạn chương trình có thời gian thực hiện $T(n) = O(f(n))$

Nếu thực hiện $k(n)$ lần đoạn chương trình với $k(n) = O(g(n))$ thì độ phức tạp sẽ là $O(g(n) \cdot f(n))$



**QUI TẮC CHUNG TÍNH ĐPT
CHƯƠNG TRÌNH KHÔNG CÓ CT CON**

Các lệnh gán, nhập, xuất, return: $O(1)$
Một chuỗi tuần tự các lệnh: Quy tắc cộng
Cấu trúc IF: $\max(\text{đk}, CV1, CV2)$
Cấu trúc vòng lặp là tổng (trên tất cả các lần lặp) thời gian thực hiện thân vòng lặp.
Trình tự đánh giá:
– Nối tiếp: Từ trên xuống
– lồng nhau: Từ trong ra

Xác định được số lần lặp

- `for(i=a; i<=b; i++)`: số lần lặp = $b-a+1$
- `for(i=1; i<=n; i= 2*i)`
- Sau lần lặp thứ 1: $i= 2$
- Sau lần lặp thứ 2: $i= 4$
- ...
- Sau lần lặp thứ k : $i= 2^k$
- Lặp kết thúc khi $i= 2^k= n$
- $k= \log_2 n \Rightarrow$ số lần lặp = $\log_2 n$

Ví dụ:

```
/*1*/ Sum=0;
/*2*/ for (i=1; i<=n; i=i*2) {
/*3*/     scanf ("%d", &x);
/*4*/     Sum=Sum+x; }
```

- 1: $O(1)$
- 3 và 4: $O(1)$
- 2 thực hiện $\log_2 n$ lần \Rightarrow 2: $O(\log n)$
- $T(n) = O(\log n)$

Dùng quy tắc + (max...)

-3,4 nối tiếp nhau nên dùng quy tắc $\max(3,4)O(1)$

$\rightarrow 2,3,4 O(1) \rightarrow O(\log n) * O(1) = O(\log n)$

-1,2 dùng quy tắc + $\max(1,2) \rightarrow O(\log n)$

3-4 $\rightarrow O(1)$ QUY TẮC NHÂN $2(O(1) * O(n-i-1))$

1 lồng 2 nên tính tổng

Số lần lặp không xác định

Quy tắc đánh giá từ trên xuống và từ trong vòng lặp ra

1-2-3 nối tiếp nhau dùng quy tắc $\max(1,2,3) \rightarrow 3$

```
s = 1, p = 1;  
for (i=1; i<=n; i++) {  
    p = p * x / i;  
    s = s + p;  
}
```

=> Vậy độ phức tạp của thuật toán là $O(n)$