Độ phức tạp thuật toán

Duc-Minh Vu @ Phenikaa - ORLab

Mục tiêu

- Hiểu được khái niệm độ phức tạp thuật toán
  - Thời gian / Bộ nhớ
- Nắm được các kí hiệu đánh giá độ phức tạp thuật toán cơ bản (big-O)
- Nắm được cách chứng minh độ phức tạp thuật toán và có khả năng chứng minh độ phức tạp thuật toán
- Nắm được cách tính độ phức tạp thuật toán và có khả năng tính độ phức tạp thuật toán của các thuật toán cụ thể.
- Nắm được định lý định lý thợ và áp dụng.

So sánh thuật toán

- Làm sao để so sánh hai thuật toán:
  - So sánh thời gian chạy / bộ nhớ
  - Bộ nhớ

- Đo thời gian chạy:
  - Đo thời gian chạy thực tế
    - Phụ thuộc input / phụ thuộc ngôn ngữ lập trình / phụ thuộc máy tính cụ thể
  - Ước lượng thời gian chạy thông qua số phép toán “cơ bản”
    - “Bất biến” - không phụ thuộc input cụ thể.

Độ phức tạp tính toán và độ phức tạp tiệm cận

- Một bài toán có thể giải bằng nhiều thuật toán với hiệu quả khác nhau.

- Để đo độ hiệu quả của thuật toán - sử dụng khái niệm độ phức tạp tính toán (computational complexity) được phát triển bởi Juris Hartmanis và Richard E. Stearns.
    - Đo độ phức tạp về thời gian và bộ nhớ.

- Không phụ thuộc ngôn ngữ lập trình/máy tính cụ thể.

Độ phức tạp tính toán và độ phức tạp tiệm cận

- Sử dụng các khái niệm độc lập kích thước dữ liệu (n)
- Độ phức tạp xấp xỉ (asymptotic complexity):
  - Hàm của n
- Sử dụng các hàm đơn giản để đánh giá độ phức tạp của thuật toán

Làm sao để tính độ phức tạp thuật toán

- Các lệnh cơ bản như gán, đọc, ghi, so sánh, etc. có độ phức tạp hằng số.
- Độ phức tạp của một đoạn lệnh là tổng độ phức tạp của từng lệnh.
- Độ phức tạp của cấu trúc rẽ nhánh là độ phức tạp lớn nhất của lệnh
- Thời gian thực hiện vòng lặp là tổng thời gian tất cả các lần lặp của thân vòng lặp.
- **=> Đếm/Ước lượng số lệnh cơ bản và so sánh.**
- **=> Làm thế nào để tránh không phụ thuộc vào cách code/cách viết mã giả?**

Ký hiệu tiệm cận O, Ω, Θ

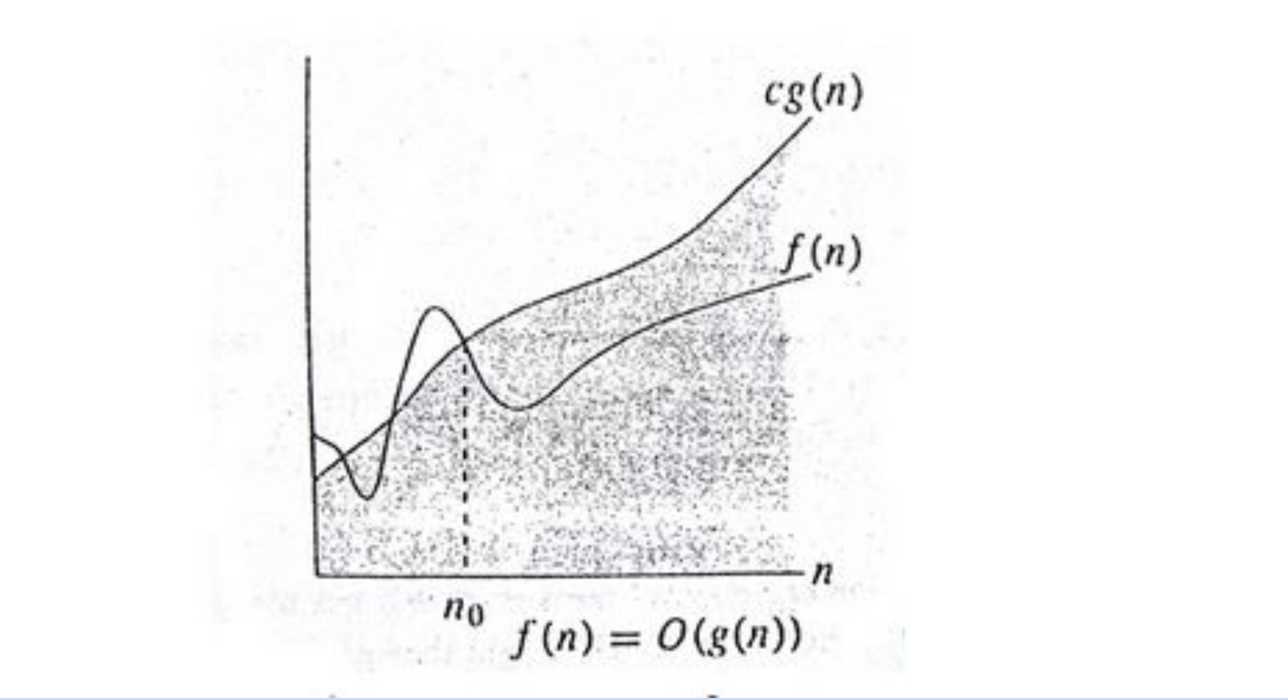
Ký hiệu tiệm cận O, Ω, Θ

- Dùng để đánh giá cận trên/dưới và cận chặt của thuật toán theo mức độ tăng trưởng của hàm
- O, Ω, Θ dùng đại diện cho 1 tập các hàm số.
  - Tập hàm số có cùng cận trên/cận dưới/cận chặt
- Mục đích là dùng để đánh giá độ phức tạp thuật toán của thuật toán thông qua các hàm đơn giản/quen thuộc.
  - e.g. hàm của n, nlogn, n^2, etc.
  - Dễ dàng so sánh, ước lượng độ phức tạp.

Kí hiệu O lớn (Big-O)

- Kí hiệu O lớn được giới thiệu năm 1894 bởi Paul Bachmann.
- Đánh giá cận trên của số phép toán.
- f(n) = O(g(n)) nếu tồn tại hằng số c và n0 sao cho f(n) <= cg(n) với mọi n >= n0.
- O(g(n)) = {f(n) | f(n) <= c*g(n) với một hằng số c và n0 nào đó và n >= n0}.

hay f(n) là O(g(n)) nếu lim f(n)/g(n) khi n -> vô cùng bị chặn bởi một hằng số dương nào đó.

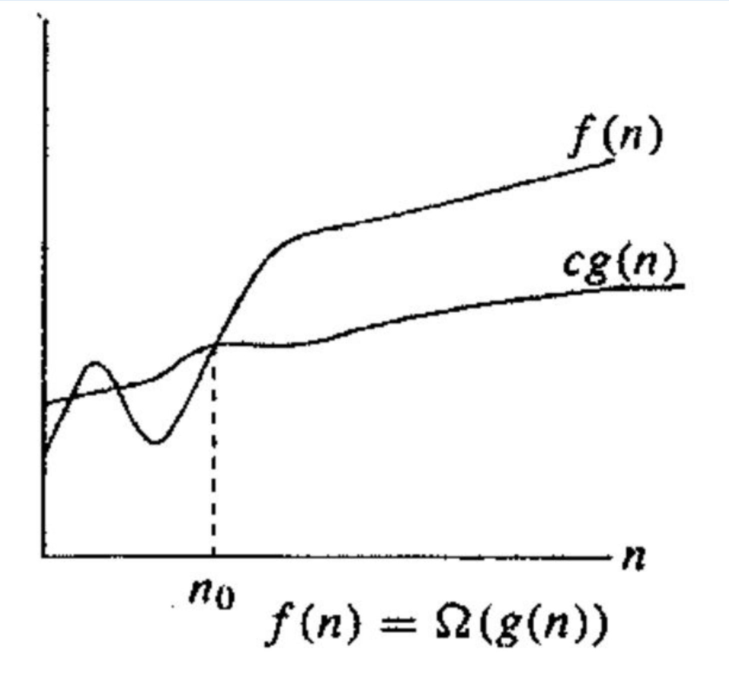


f(n) bị “chặn trên” bởi cg(n) khi n -> dương vô cùng

Kí hiệu Omega lớn (Big-Ω)

- f(n) = Ω(g(n)) nếu tồn tại hằng số c và n0 sao cho f(n) >= cg(n) với mọi n >= n0.
- O(g(n)) = {f(n) | f(n) >= c*g(n) với một hằng số c và n0 nào đó và n >= n0}.
- Ω(g(n)) là xấp xỉ cận dưới về số phép toán cơ bản của f(n)

Kí hiệu Omega lớn (Big-Ω)



f(n) bị chặn dưới bởi cg(n) khi n tiến tới dương vô cùng.

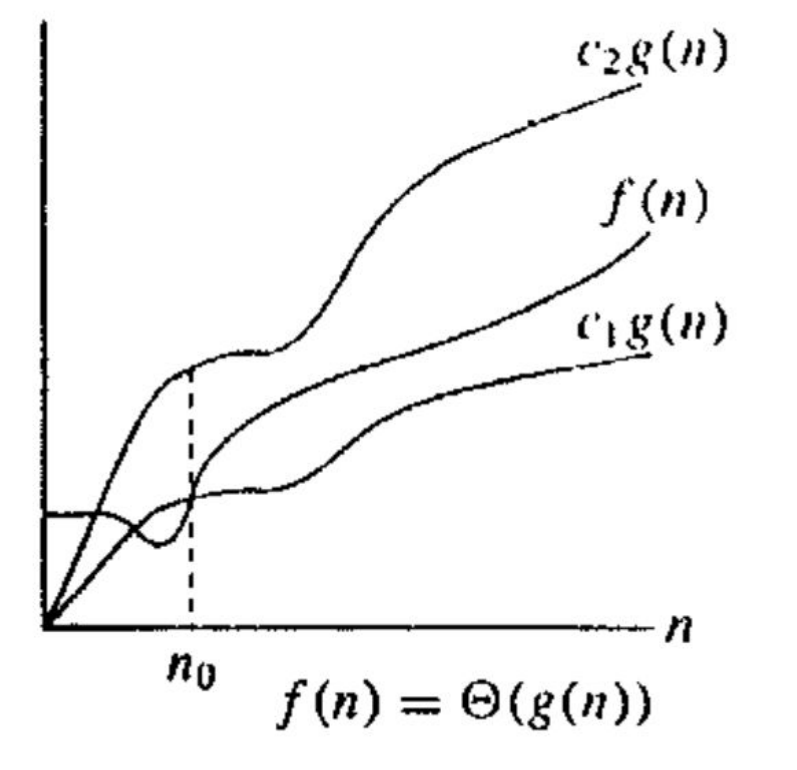
Kí hiệu Theta lớn (Big-ϴ)

- f(n) = ϴ(g(n)) nếu tồn tại hằng số c1, c2 và n0 sao cho c1*g(n) <= f(n) <= c2*g(n) với mọi n >= n0.

- ϴ(g(n)) = {f(n) | c1*g(n) <= f(n) <= c2*g(n) với một hằng số c, n1, n2 nào đó và n >= n0}.

- ϴ(g(n)) là xấp xỉ chặt trên và dưới về số phép toán cơ bản của f(n)

Kí hiệu Theta lớn (Big-Theta)



f(n) bị chặn dưới bởi cg(n) khi n tiến tới dương vô cùng.

Ví dụ về cách chứng minh cận của độ phức tạp

Ví dụ: Xác định các hằng số trong các định nghĩa để chứng minh các đánh giá tiệm cận sau đây:

– $2n^2 = O(n^3)$. Ta cần có $2n^2 \leq cn^3 \Rightarrow 2 \leq cn \Rightarrow c = 1$ và $n_0 = 2$.

– $n^2 = O(n^2)$. Ta cần có $n^2 \leq cn^2 \Rightarrow 1 \leq c \Rightarrow c = 1$ và $n_0 = 1$.

– $1000n^2 + 1000n = O(n^2)$. Ta cần có $1000n^2 + 1000n \leq cn^2 \Rightarrow c = 1001$ và $n_0 = 1$.

– $n = O(n^2)$. Ta cần có $n \leq cn^2 \Rightarrow 1 \leq cn \Rightarrow c = 1$ và $n_0 = 1$.

Ví dụ về cách chứng minh cận của độ phức tạp

– $5n^2 = \Omega(n)$. Ta cần tìm c, n_0 sao cho: $0 \le cn \le 5n^2 \; \forall \; n \ge n_0$ hay $cn \le 5n^2$
$\Rightarrow c = 1$ và $n_0 = 1$.

– $100n + 5 \ne \Omega(n^2)$. Giả sử tồn tại c, n_0 sao cho: $0 \le cn^2 \le 100n + 5 \; \forall \; n \ge n_0$.
Ta có: $100n \le 100n + 5n$, suy ra $cn^2 \le 105n$ hay $n(cn - 105) \le 0$. Do n và c là các số dương nên từ đó suy ra $n \le 105/c$. Đó là điều không thể xảy ra, do n không thể nhỏ hơn hằng số.

Chú ý: Giá trị của n_0 và c *không phải là duy nhất* trong chứng minh công thức tiệm cận.

Chú ý: Giá trị của n_0 và c *không phải là duy nhất* trong chứng minh công thức tiệm cận.

Ví dụ, ta cần chứng minh $100n + 5 = O(n^2)$. Ta có:

$$100n + 5 \le 100n + n = 101n \le 101n^2 \text{ với mọi } n \ge 5.$$

Như vậy, $n_0 = 5$ và $c = 101$ là các hằng số cần tìm. Ta cũng có:

$$100n + 5 \le 100n + 5n = 105n \le 105n^2 \text{ với mọi } n \ge 1,$$

do đó $n_0 = 1$ và $c = 105$ cũng là các hằng số cần tìm.

Khi chứng minh các công thức tiệm cận ta chỉ cần tìm các hằng c và n_0 nào đó thỏa mãn bất đẳng thức trong định nghĩa công thức tiệm cận.

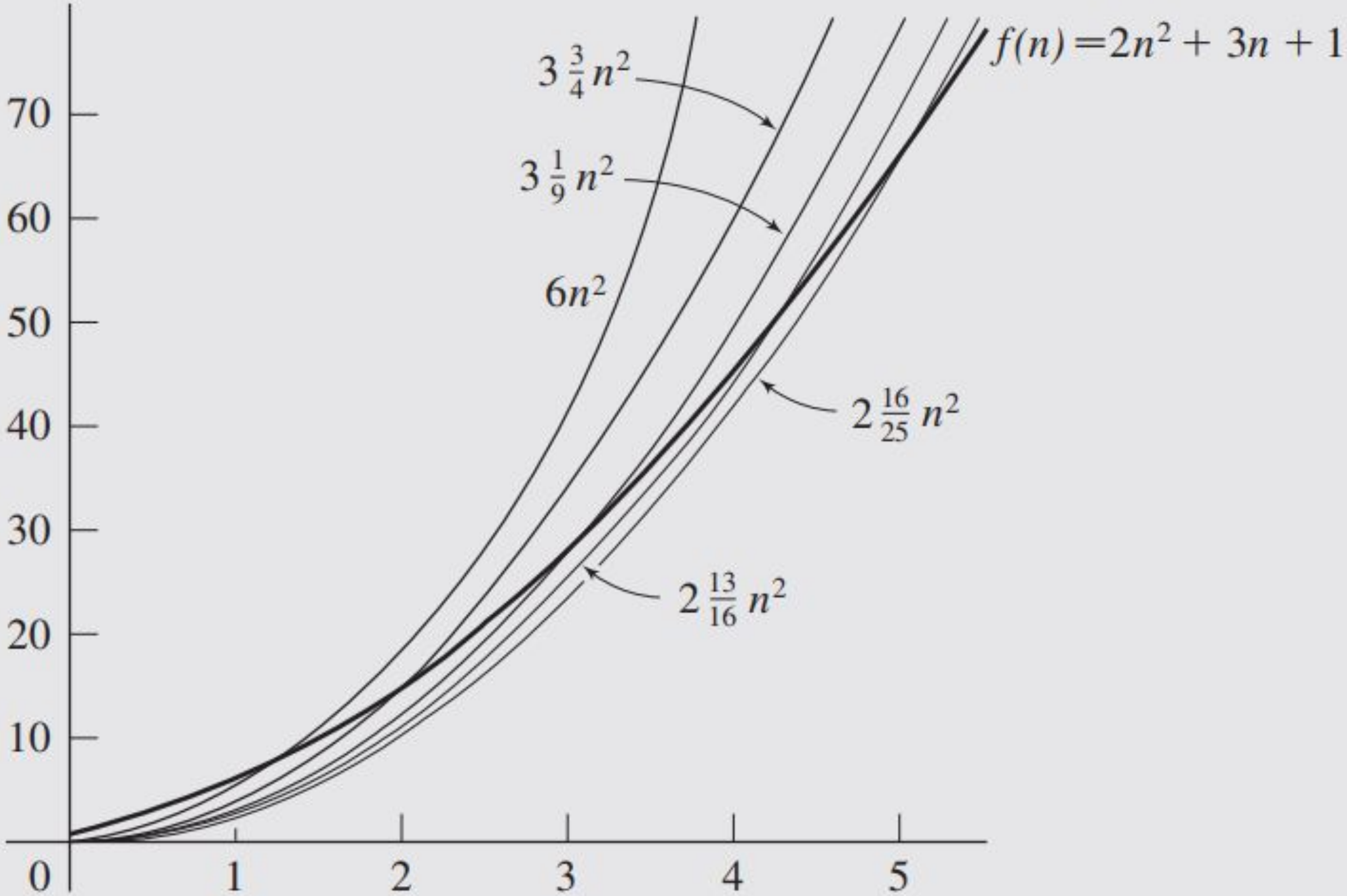
Ví dụ về cách chứng minh cận của độ phức tạp

Có nhiều giá trị c và N thỏa mãn

Different values of c and N for function $f(n) = 2n^2 + 3n + 1 = O(n^2)$ calculated according to the definition of big-O.

c	≥ 6	$\geq 3\frac{3}{4}$	$\geq 3\frac{1}{9}$	$\geq 2\frac{13}{16}$	$\geq 2\frac{16}{25}$	$\ldots$	$\rightarrow$	2
N	1	2	3	4	5	$\ldots$	$\rightarrow$	∞

Các giá trị c và N khác nhau.


$f(n) = 2n^2 + 3n + 1$
$3\frac{3}{4}n^2$
$3\frac{1}{9}n^2$
$6n^2$
$2\frac{16}{25}n^2$
$2\frac{13}{16}n^2$
70
60
50
40
30
20
10
0
1
2
3
4
5

Độ tăng của một số hàm phổ biến

Tên gọi của một số đánh giá trong ký hiệu O:

– $O(1)$: hằng số (constant).

– $O(\log n)$: logarithmic.

– $O(n)$: tuyến tính (linear).

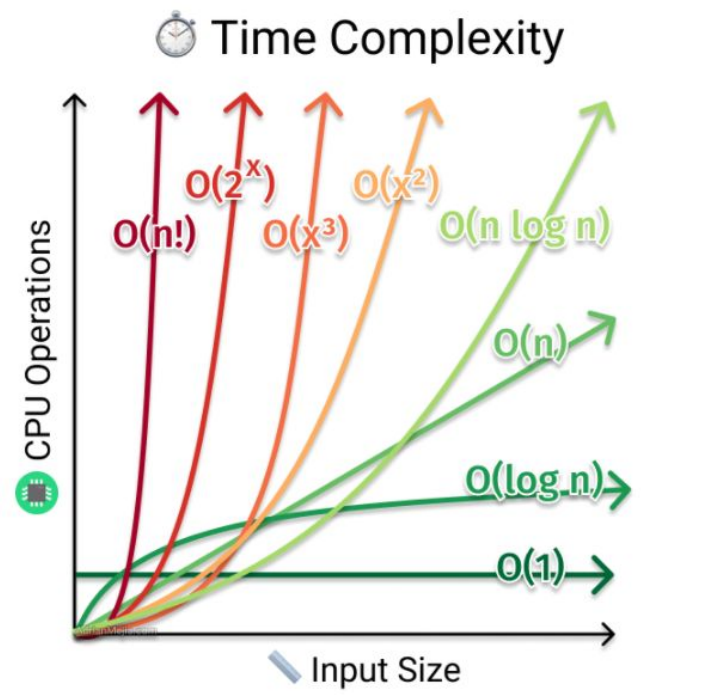
– $O(n \log n)$: trên tuyến tính (superlinear).

– $O(n^2)$: bình phương (quadratic).

– $O(n^3)$: bậc ba (cubic).

– $O(a^n)$: hàm mũ (exponential) ($a > 1$).

– $O(n^k)$: đa thức (polynomial) ($k \geq 1$).



https://chidokun.github.io/2021/07/complexity-of-time/

Số phép toán	n = 10	Thời gian	n = 100	Thời gian
$\log(n)$	3.32	3.3×10^{-8} giây	6.64	6×10^{-8} giây
$n\log(n)$	33.2	3.3×10^{-7} giây	664	6.6×10^{-6} giây
n^2	100	10^{-6} giây	10000	10^{-4} giây
n^3	1×10^{3}	10^{-5} giây	1×10^{6}	10^{-2} giây
e^n	2.2×10^{4}	2×10^{-4} giây	2.69×10^{43}	$> 10^{26}$ thế kỷ

Giả sử máy tính thực thi 10^8 phép tính trên 1s.

Ví dụ về độ tăng của các hàm

Số phép toán	n = 10000	Thời gian	$n = 10^6$	Thời gian
$\log(n)$	13.3	10^{-6} giây	19.9	$<10^{-5}$ giây
$n\log(n)$	1.33×10^5	10^{-3} giây	1.99×10^7	2×10^{-1} giây
n^2	1×10^8	1 giây	1×10^{12}	**2.77 giờ**
n^3	1×10^{12}	2.7 giờ	1×10^{18}	**115 ngày**
e^n	8.81×10^{4342}	$>10^{4327}$ thế kỷ		

Thời gian chạy sẽ rất lâu nếu thuật toán không hiệu quả => thiết kế thuật toán khác.

Ví dụ về độ tăng của các hàm

Ví dụ: Với mỗi cặp hàm sau đây, hoặc $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, hoặc $f(n) = \Theta(g(n))$. Hãy xác định quan hệ nào là đúng.

f (n)	*g(n)*	*So sánh*
f(n) = log n2	g(n) = log n + 5	$f(n) = \Theta(g(n))$
f(n) = n	g(n) = log n2	$f(n) = \Omega(g(n))$
f(n) = log log n	g(n) = log n	$f(n) = O(g(n))$
f(n) = n	g(n) = log2 n	$f(n) = \Omega(g(n))$
f(n) = n log n + n	g(n) = log n	$f(n) = \Omega(g(n))$
f(n) = 10	g(n) = log 10	$f(n) = \Theta(g(n))$
f(n) = 2n	g(n) = 10n2	$f(n) = \Omega(g(n))$
f(n) = 2n; g(n) = 3n	g(n) = 3n	$f(n) = O(g(n))$

So sánh độ tăng/quan hệ giữa các hàm

1. So sánh hai hàm sau đây trong ký hiệu O:

a) $N\log N$ và N^2 : $N\log N = O(N^2)$ hay $N^2 = O(N\log N)$?

b) $N\log N$ và N : $N\log N = O(N)$ hay $N = O(N\log N)$?

2. Đưa ra đánh giá trong ký hiệu O cho các hàm sau đây:

a) $\mathrm{N}^2 + \mathrm{N}\log\mathrm{N} = \mathrm{O}(\ldots\ldots)$.

b) $\mathrm{N} + \mathrm{N}\log\mathrm{N} = \mathrm{O}(\ldots\ldots)$.

c) $\mathrm{N}^2 + \log\mathrm{N} = \mathrm{O}(\ldots\ldots)$.

Bài tập làm quen

3. Cho hàm:

$$T(n) = 100n^{3/2} + 500n + 1000$$

xác định với đối số nguyên dương n.

a) Chứng minh rằng $T(n) \in O(n^2)$.

b) Chứng minh rằng $T(n) \notin \Omega(n^2)$.

Bài tập làm quen

Đánh giá thời gian chạy của thuật toán

Đánh giá thời gian chạy của thuật toán

- Đánh giá bằng cách phân rã thuật toán.
- Thuật toán gồm nhiều các bước/thuật toán con.
- Bước/thuật toán nào có độ phức tạp lớn nhất thì đó chính là độ phức tạp của thuật toán ban đầu.

Một số tính chất cơ bản

- **Quy tắc nhân hằng số:** Nếu f(n) là O(g(n)) thì a*f(n) cũng là O(g(n)) với a là hằng số
  - $f(n) = 2n^2+5$ là $O(n^2)$ thì $7*f(n) = 7(2n^2+5) = 14n^2+35$ cũng là $O(n^2)$.
- **Quy tắc cộng:** Nếu f(n) = O(g(n)) và d(n)=O(e(n)) thì f(n) + d(n) = O(max(g(n), e(n)))
  - $f(n) = n$ i.e $O(n)$ và $d(n) = n^2$ i.e $O(n^2)$ thì $f(n) + d(n) = n + n^2$ i.e $O(n^2)$
- **Quy tắc nhân (lồng nhau):** f(n)=O(g(n)) và d(n)=O(e(n)) thì f(n) * d(n) = O(g(n) * e(n))
  - $f(n) = n$ i.e $O(n)$ và $d(n) = n^2$ i.e $O(n^2)$ thì $f(n) * d(n) = n * n^2 = n^3$ i.e $O(n^3)$
- Chứng minh các tính chất này bằng định nghĩa

7. **Đánh giá thời gian tính của các đoạn chương trình sau:**

a)

```
sum = 0;
for( i = 0; i < n; i++)
    for( j = 0; j < n * n; j++)
        sum++;
```

b)

```
sum = 0;
for( i = 0; i < n; i++)
    for( j = 0; j < i; j++)
        sum++;
```

Phép toán cơ bản là sum++. Độ phức tạp O(1)

Đếm xem phép toán sum++ được thực hiện bao nhiêu lần theo hàm của của n.

Đánh giá độ phức tạp của các đoạn chương trình sau c)

```
sum = 0;
for( i = 0; i < n; i++)
    for( j = 0; j < i*i; j++)
        for( k = 0; k < j; k++)
            sum++;
```

d)

```
sum = 0;
for( i = 0; i < n; i++)
    sum++;
val = 1;
for( j = 0; j < n*n; j++)
    val = val * j;
```

Đánh giá độ phức tạp của các đoạn chương trình sau

Tính toán độ phức tạp thuật toán

```
int fun(int n)
{
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j < n; j += i)
        {
            // Some O(1) task
        }
    }
}
```

An interesting time complexity question - GeeksforGeeks

Tính toán độ phức tạp thuật toán

```
void fun(int n, int k)
{
    for (int i = 1; i <= n; i++)
    {
        int p = pow(i, k);
        for (int j = 1; j <= p; j++)
        {
            // Some O(1) work
        }
    }
}
```

[Time Complexity of Loop with Powers - GeeksforGeeks]

Tính toán độ phức tạp thuật toán

```
void fun(int n)
{
    int j = 1, i = 0;
    while (i < n)
    {
        // Some O(1) task
        i = i + j;
        j++;
    }
}
```

[Time Complexity where loop variable is incremented by 1, 2, 3, 4 .. - GeeksforGeeks]

Định lý thợ (master theorem)

Định lý thợ (master theorem)

- Dùng để giúp tính toán độ phức tạp của một số thuật toán đệ quy/chia để trị.
    - Giải một bài toán bằng cách giải một dãy các bài toán con và kết hợp kết quả.
    - E.g. tìm giá trị lớn nhất của 1 dãy số A[l..r] bằng đệ quy
        - Nếu l==r thì trả về A[l]
        - Tính m = (l+r)/2
        - Tìm m1 là max của A[l..m] bằng đệ quy (bài toán con)
        - Tìm m2 là max của A[m+1..r] bằng đệ quy (bài toán con)
        - Trả về max(m1,m2) (kết hợp kết quả)
- [Giải Thuật Lập Trình · master theorem (giaithuatlaptrinh.com)]()

Problem 1: giải hệ thức truy hồi:

$$T(n) = aT(\tfrac{n}{b}) + f(n) \qquad (1)$$

trong đó a, b là các hằng số dương.

Problem 2: giải hệ thức truy hồi:

$$T(n) = \sum_{i=1}^{k} a_i T(\tfrac{n}{b_i}) + f(n) \qquad (2)$$

trong đó $a_i, b_i, i = 1, \ldots, k$ là các hằng số dương.

(1): chia bài toán ban đầu ra thành a bài toán con kích thước n/b; và tốn f(n) thao tác để tìm ra kết quả.
e.g. thuật toán sắp xếp mergesort, quicksort (thuật toán chia để trị)
(2): không tìm hiểu ở đây và nó không có công thức để giải.

Công thức truy hồi

Định lý thợ

Định lý thợ (master theorem) là một công cụ giúp ta giải các hệ thức truy hồi có dạng trong **Problem 1**. Định lý dài và khó nhớ và theo mình bạn đọc cũng không cần nhớ làm gì. Chỉ cần nhớ dạng bài toán mà định lý này có thể áp dụng để giải. Nếu có thể thì chỉ cần nhớ phương pháp chứng minh định lý.

Master Theorem: Cho hệ thức truy hồi:

$$T(n) = aT(\tfrac{n}{b}) + f(n)$$

1. Nếu $af(n/b) = \kappa f(n)$ với $\kappa < 1$, ta có $T(n) = \Theta(f(n))$.
2. Nếu $af(n/b) = Kf(n)$ với $K > 1$, ta có $T(n) = \Theta(n^{\log_b a})$.
3. Nếu $af(n/b) = f(n)$, ta có $T(n) = \Theta(f(n)\log_b n)$.

Điều kiện áp dụng: a > 1, b > 1.

Định lý thợ

Định lý thợ

Định lý thợ rút gọn: Giả sử $a \geq 1$ và $b > 1$, $c > 0$ là các hằng số. Xét $T(n)$ là công thức đệ quy:

$$T(n) = a\,T(n/b) + c\,n^k$$

xác định với $n \geq 0$.

1. Nếu $a > b^k$, thì $T(n) = \Theta(n^{\log_b a})$.
2. Nếu $a = b^k$, thì $T(n) = \Theta(n^k \lg n)$.
3. Nếu $a < b^k$, thì $T(n) = \Theta(n^k)$.

Ví dụ 1: $T(n) = 3T(n/4) + cn^2$

Trong ví dụ này: $a = 3$, $b = 4$, $k = 2$.

Do $3 < 4^2$, ta có tình huống 3, nên $T(n) = \Theta(n^2)$.

Ví dụ 2: $T(n) = 2T(n/2) + n^{0.5}$

Trong ví dụ này: $a = 2$, $b = 2$, $k = 1/2$.

Do $2 > 2^{1/2}$ nên ta có tình huống 1. Vậy $T(n) = \Theta(n^{\log_b a}) = \Theta(n)$.

Ví dụ

Ví dụ 3: $T(n) = 16T(n/4) + n$

$a = 16$, $b = 4$, $k = 1$.

Ta có $16 > 4$, vì thế có tình huống 1. Vậy $T(n) = \Theta(n^2)$.

Ví dụ 4: $T(n) = T(3n/7) + 1$

$a = 1$, $b = 7/3$, $k = 0$.

Ta có $a = b^k$, suy ra có tình huống 2. Vậy $T(n) = \Theta(n^k \log n) = \Theta(\log n)$.

Định lý thợ - Lưu ý

- Không thể áp dụng định lý thợ nếu:
  - T(n) không phải là hàm đơn điệu, e.g. sin(x)
  - f(n) không phải là đa thức, e.g. T(n) = 2T(n/2) + 2^n
    - Lúc này phải sử dụng các phương pháp khác để tính T(n), e.g. kiến thức môn toán rời rạc.
  - b không phải là hằng số, e.g. T(n) = T(sqrt(n)) + n
- Định lý thợ không giúp giải phương trình truy hồi mà chỉ đưa ra ước lượng tiệm cận.