Phức tạp thời gian

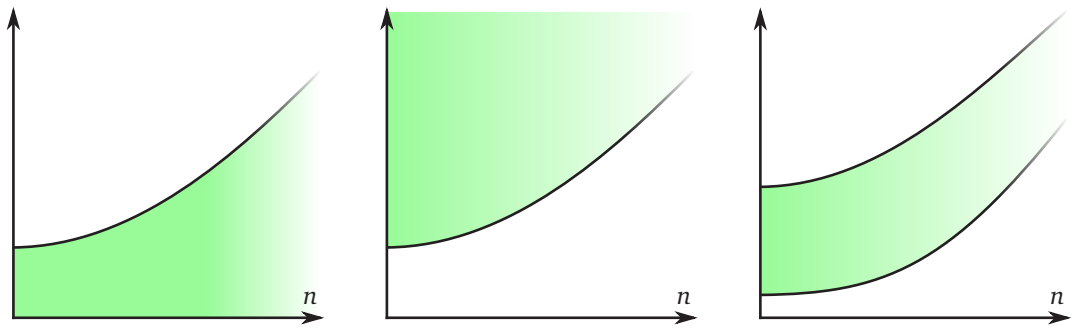
Draft of August 23, 2021

0 Giới thiệu

Một công việc quan trọng khi thiết kế chương trình máy tính là cân đo thời gian chạy. Một chương trình thao tác với dữ liệu nhỏ, như sắp xếp điểm của học sinh trong một lớp, có thể hoàn thành tức thì trên các thiết bị phổ biến hiện nay. Nhưng với dữ liệu to, như sắp xếp hàng tỷ giao dịch của ngân hàng, thời gian thực thi bắt đầu trở thành vấn đề. Chương trình sẽ chạy nhanh và êm hơn nếu được chạy trên máy tính có cấu hình mạnh hơn. Cũng nói thêm, kể cả khi chạy trên cùng một máy tính, một chương trình được viết bằng ngôn ngữ X có thể chạy nhanh hơn khi được viết bằng ngôn ngữ Y. Điều này phụ thuộc vào ti tỉ thứ bên trong trình biên dịch, thông dịch của ngôn ngữ.

Việc đo thời gian thực thi của chương trình dựa trên phần cứng hay ngôn ngữ lập trình quá nhọc. Thứ hợp lý nhất chúng ta có thể dựa vào để đo thời gian là kích thước của dữ liệu. Ta chỉ quan tâm đến thời gian chạy khi dữ liệu *lớn*. (Chương trình sẽ chạy đủ nhanh trên dữ liệu nhỏ, khác biệt thời gian tạo ra không đáng kể.) Bất kì thứ gì độc lập với kích thước dữ liệu, hay nói cách khác là hằng số, sẽ được bỏ qua. Điều quan trọng ở đây là thời gian chạy *thay đổi như thế nào* khi kích thước dữ liệu tăng.

Khi phân tích thời gian chạy của chương trình, hay thuật toán, ta quan tâm tốc độ tăng lên (growth rate) của số lượng các bước tính toán cho đầu vào. Tốc độ tăng lên này chính là *độ phức tạp thời gian* của thuật toán. Độ phức tạp thời gian mang tính ***tiệm cận*** (asymptotic bound). Có ba loại tiệm cận được đề cập ở đây: cận chặn trên (upper bound), cận chặn dưới (lower bound), và cận hẹp (tight bound).



a) Chặn trên b) Chặn dưới c) Cận hẹp

Figure 1: Minh họa đồ thị của các tiệm cận. Biến n (lớn vô cùng) là kích thước đầu vào. Đồ thị thời gian chạy của thuật toán sẽ nằm trong vùng màu xanh, được chặn bởi các cận.

1 Ví dụ về số lượng các bước tính toán

Ta kí hiệu kích thước đầu vào của thuật toán là biến số tự nhiên n; n có thể là độ dài mảng, hoặc số lượng từ trong cuốn từ điển, hoặc đơn giản chỉ là một biến số... Nhiều bài toán cũng yêu cầu đầu vào có nhiều hơn một biến, như nhân hai số có n và m chữ số, tìm ước chung lớn nhất; khi đó phân tích của chúng ta sẽ theo các biến này. Ta gọi số lượng các bước tính toán cho đầu vào cỡ n là một hàm số, thường kí hiệu là $f(n)$ hoặc $T(n)$.

Việc "đếm" số lượng các bước tính toán phụ thuộc vào mô hình tính (trừu tượng) mà ta chọn. Nếu không nói gì thêm, ta ngầm hiểu độ phức tạp thời gian được tính trên mô hình máy truy cập ngẫu nhiên (random-access machine, RAM). Mô hình này giúp ta hiểu về cách thuật toán hoạt động trên một máy tính thật và thuật toán độc lập với thiết bị chạy chúng. Theo RAM, quy ước để "đếm" số lượng bước tính toán như sau:

- Các lệnh đơn giản như cộng, trừ, nhân[1], chia, điều kiện if, truy cập bộ nhớ, sẽ cần chính xác một bước tính.
- Vòng lặp không được xem là câu lệnh đơn giản. Nó là lệnh phức được cấu thành từ nhiều câu lệnh đơn.

1.1 Tìm số lớn nhất

Bây giờ ta thảo luận một vấn đề: Cho một mảng A có n số, tìm số lớn nhất trong mảng. Giải pháp đơn giản nhất, có lẽ là duyệt từ đầu đến hết mảng, và duy trì số lớn nhất ở mỗi lần duyệt. Thuật toán FindMax của chúng ta thực hiện $1+1+n+n-1+n-1=3n$ bước tính *tối thiểu* để

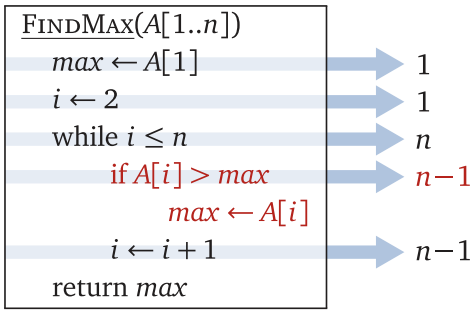


Figure 2: Số lượng bước tính toán *tối thiểu* của FindMax.

tìm số lớn nhất. Biến i chạy từ 2 đến n là $n-1$ bước, với mỗi bước thực hiện phép so sánh $A[i]$ với max, tổng cộng $2(n-1)$ bước. Vòng lặp while kiểm tra i từ lúc $i=2$ cho đến $i=n+1>n$ thì mới dừng lặp, tốn n bước kiểm tra. Nếu bạn đọc để ý, phép gán $max \leftarrow A[i]$ được thực thi *nhiều nhất* là $n-1$ lần, và số bước tính $f(n)$ chính xác là $3n \le f(n) \le 3n+(n-1)$.

Phép so sánh $i \le n$ và phép gán $i \leftarrow i+1$ chẳng qua là "thủ tục" để lặp. Thực tế khi phân tích độ phức tạp thời gian, ta không cần để ý đến bản thân vòng lặp, mà chỉ quan tâm những thứ bên trong nó. Ở ví dụ trên, ta có thể tính được độ phức tạp thời gian chỉ dựa vào số lượng bước so sánh $A[i]$ với max.

1.2 In ma trận cỡ $m \times n$

Cho ma trận A có m hàng và n cột, việc của chúng ta là in ra (màn hình) toàn bộ phần tử của nó. Ý tưởng khá đơn giản: xét từ trên xuống, ở từng hàng một, in ra tất cả các phần tử trong hàng

[1] Trong hầu hết các bộ xử lí, phép nhân thực chất tốn thời gian hơn là cộng. Uhm. Nhưng RAM là lựa chọn cân bằng giữa những khái niệm cơ sở và sự đơn giản.

đó. Thuật toán PrintMatrix dưới đây viết theo ý tưởng trên. Với mỗi một giá trị của r thì thực hiện in ra n phần tử trên hàng r. Do r chạy từ 1 đến m nên tổng cộng thuật toán thực hiện $m \cdot n$ bước tính. (Hoặc ta có thể suy luận theo kiểu: ma trận có tổng cộng $m \cdot n$ phần tử, nên in ra cũng tốn $m \cdot n$ bước. But that's boring.)

```
PrintMatrix(A[1..m, 1..n])
    for r ← 1 to m
        for c ← 1 to n
            Print(A[r, c])
```

1.3 Lũy thừa nhanh

Cho một số a và một số tự nhiên n, phép lũy thừa a^n theo Hán Việt nghĩa là nhân chồng chất lên. Ta có thể tính bằng một vòng lặp đơn giản với $n-1$ bước tính. Nhưng với một chút chỉnh sửa ở định nghĩa, ta sẽ có một công thức lũy thừa mới để tính nhanh hơn.

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ (a^2)^{n/2} & \text{if } n > 0 \text{ and } n \text{ is even} \\ (a^2)^{\lfloor n/2 \rfloor} \cdot a & \text{otherwise} \end{cases}$$

```
FastPower(a, n)
    result ← 1
    while n ≥ 1
        if n is odd
            result ← result · a
        n ← ⌊n/2⌋
        a ← a · a
    return result
```

Ví dụ này khá tương đồng với ví dụ tìm số lớn nhất, ngoại trừ số lần lặp không tăng tuyến tính (linearly) theo n. Ở FindMax, thay vì i mỗi lần lặp tăng thêm 1, hay tương đương là n giảm đi 1, thì ở FastPower mỗi lần lặp n giảm đi $\lceil n/2 \rceil$, tức khoảng một nửa. Nếu n là lũy thừa của 2, như $2, 4, 8, 16, \ldots$ thì số lần lặp chính xác là $\log_2 n$.

Bởi vì mục đích của chúng ta là phân tích thời gian chạy chứ không phải "đếm" bước tính, những ví dụ trên chỉ cho bạn đọc có cái nhìn chung. (Và "đếm" cũng khá chung chung.) Việc xác định *chính xác* số lượng bước tính là không cần thiết. Không phải tất cả thuật toán đều đáng để phân tích sâu sắc. Nghiên cứu quá chi tiết dễ khiến ta sa vào thuật toán "lý thuyết" xa rời các ứng dụng cụ thể.

2 Phức tạp thời gian

2.1 Phân tích tiệm cận của hàm số

Ở phần này, chúng ta thảo luận về phương pháp so sánh hai hàm số một cách tiệm cận. Phân tích của chúng ta cần giả sử kích thước dữ liệu là lớn. Lớn bao nhiêu? We don't know. Hãy nghĩ chúng lớn đến vô cùng!

2.1.1 Tiệm cận tương tự

Hai hàm số $f, g : \mathbb{N} \to \mathbb{R}$ là tiệm cận tương tự nhau (asymtotically similar), kí hiệu $f(n) \approx g(n)$, khi và chỉ khi

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$$

với c là hằng số dương. Ở thuật toán FindMax tôi khẳng định rằng có thể tính được độ phức tạp thời gian chỉ dựa vào số lượng bước so sánh $A[i]$ với max. Bởi vì khi n tiến đến vô cùng, thì

số bước so sánh, $g(n) = n-1$, là tiệm cận tương tự với số bước tối thiểu $f(n) = 3n$, hay kể cả $f(n) = 3n + (n-1)$:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3n+(n-1)}{n-1} = \lim_{n\to\infty} \frac{3n}{n-1} + \lim_{n\to\infty} \frac{n-1}{n-1} = 4 > 0. \quad \checkmark$$

2.1.2 Tiệm cận nhỏ hơn

Cho hai hàm số $f, g : \mathbb{N} \to \mathbb{R}$, ta nói f tiệm cận nhỏ hơn (asymtotically smaller) g, kí hiệu $f(n) \ll g(n)$, khi và chỉ khi

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Để ví dụ, ta so sánh số bước tính của thuật toán lũy thừa nhanh $f(n) = \log_2 n$ và lũy thừa "chậm" $g(n) = n-1$:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log_2 n}{n-1} = \lim_{n\to\infty} \frac{(\log_2 n)'}{(n-1)'} = \lim_{n\to\infty} \frac{\frac{1}{n\ln(2)}}{1} = \frac{1}{\ln(2)} \cdot \frac{1}{\infty} = 0. \quad \checkmark$$

2.1.3 Một số hàm thường gặp

Từ hai định nghĩa tiệm cận trên, ta có thể "xếp hạng" một số hàm thường gặp khi phân tích thời gian chạy theo độ lớn của chúng như sau:

$$1 \ll \log n \ll n \ll n \log n \ll n^c \ll 2^n \ll n!,$$

với $c > 1$ là một hằng số. Và nếu bạn không phải tín đồ của limit, **nhìn vào đơn thức lớn nhất** là có thể so sánh được hai hàm; ví dụ $30n + n^4 - 1975n \ll \frac{1}{n} + 2^n - \sin(\log_5 n)$ vì $n^4 \ll 2^n$.

2.2 Kí pháp big-Oh

Kí pháp big-Oh mô tả các hàm số tăng lên nhanh như thế nào trong giới hạn, trong khi tham số của chúng có xu hướng tiến đến vô cùng. Big-Oh đơn giản hóa phân tích của chúng ta, bằng cách bỏ qua các chi tiết ngoài lề của việc so sánh tốc độ của các thuật toán. Để kí hiệu các hàm tăng trưởng không nhanh hơn $g(n)$ khi n tiến đến vô cùng, ta dùng $O(g(n))$.

Cho hai hàm số $f, g : \mathbb{N} \to \mathbb{R}$, ta nói $f(n) = O(g(n))$ khi và chỉ khi[2] đồng thời tồn tại hằng số thực dương c và n_0 ($c, n \in \mathbb{R}^+$) sao cho

$$\forall n \geq n_0, \quad 0 \leq f(n) \leq c \cdot g(n).$$

Khi $f(n) = O(n)$, sẽ tồn tại $c \cdot g(n)$ là chặn trên của $f(n)$. Ta có vài quan sát nhỏ sau đây:

- Big-Oh không quan tâm input nhỏ hơn n_0.
- Hằng số c trong $c \cdot g(n)$ cho phép chúng ta bỏ qua các hằng số nhân và cộng khi phân tích $f(n)$, ví dụ $3n - 99$ và $100n$ và $n/2 + 5$ và n là như nhau.
- **Nếu $f(n)$ tiệm cận nhỏ hơn $g(n)$, thì $f(n) = O(g(n))$.**
- Nếu cả hai đơn thức lớn nhất của chúng bằng nhau, hay chúng tiệm cận tương tự, thì $f(n) = O(g(n))$ và $g(n) = O(f(n))$. Ví dụ $9n^2 = O(n^2 + 1945)$ và $n^2 + 1945 = O(9n^2)$, nhưng cả hai đều là $O(n^2)$ vì hằng số không quan trọng. Trong trường hợp này, ta nói $f(n) = \Theta(g(n))$ cũng như $g(n) = \Theta(f(n))$ (kí pháp Θ đọc là big-Theta). Sẽ tồn tại hai hằng số c_1 và c_2 sao cho $c_2 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)$; hàm $g(n)$ tạo một cận hẹp cho $f(n)$.

[2] Nếu kí hiệu theo cách nói, thì $f(n) \in O(g(n))$ sẽ có nghĩa hơn. Tuy nhiên dấu $=$ là tiêu chuẩn được sử dụng rộng rãi vào cuối thế kỉ 19. Ở trường hợp này dấu $=$ không chỉ sự bằng nhau, tức ghi ngược lại là sai, vì $O(g(n)) \notin f(n)$.

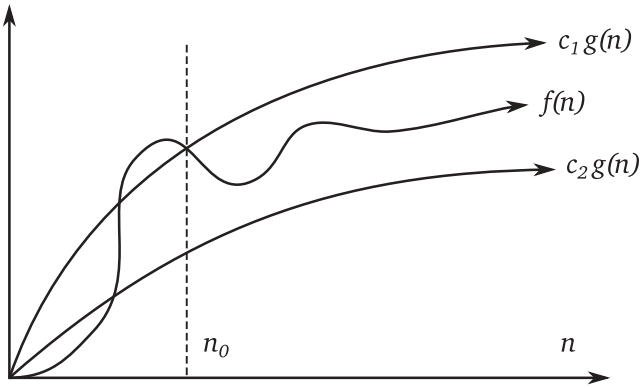


Figure 3: Độ phức tạp thời gian không quan tâm n nhỏ.

2.2.1 Một số tính chất của big-Oh. (Hệ quả?)

- Với một hằng số c bất kì thì $c = O(1)$.
- $c \cdot f(n) = O(f(n))$.
- Mỗi big-Oh đều thuộc big-Oh có cấp cao hơn nó. Ví dụ: $O(n \log n) = O(n!)$.
- Với cơ số k bất kì thì $\log_k n = O(\log n)$, vì $\log_k n = \log_a n / \log_a k = O(\log_a n)$. Điều đó có nghĩa là $O(\log_k n) = O(\log_a n)$; so why bother the base?
- Giả sử khối lệnh A có thời gian chạy là $T_A(n) = O(f(n))$ và khối lệnh B có thời gian chạy là $T_B(n) = O(h(n))$:
  - Nếu hai khối lệnh này rời rạc: $T(n) = T_A(n) + T_B(n) = \max(O(f(n)), O(h(n)))$. Big-Oh chỉ quan tâm đến đơn thức lớn nhất.
  - Nếu hai khối lệnh này lồng nhau: $T(n) = T_A(n) \cdot T_B(n) = O(f(n) \cdot h(n))$.

2.2.2 Ví dụ

Chúng ta bắt đầu với một thuật toán sắp xếp quen thuộc:

```
MagicSort(A[1..n])
    for i ← 1 to n
        for j ← 1 to n − 1
            if A[j] > A[j + 1]
                Swap(A[j], A[j + 1])
```

Vòng lặp i chạy từ 1 đến n; vòng lặp j chạy từ 1 đến $n-1$. Hai vòng lặp i và j lồng nhau nên độ phức tạp thuật toán là $n(n-1) = n^2 - n = \Theta(n^2) = O(n^2)$. Vậy độ phức tạp thời gian của MagicSort là $\Theta(n^2)$ (hoặc $O(n^2)$). Ta nói MagicSort chạy trong thời gian $\Theta(n^2)$ (hoặc $O(n^2)$). Kiểm tra:

$$\lim_{n\to\infty} \frac{n^2 - n}{n^2} = \lim_{n\to\infty} \frac{n^2}{n^2} - \lim_{n\to\infty} \frac{n}{n^2} = 1 - 0 = 1 > 0. \quad \checkmark$$

Vậy $n^2 - n$ và n^2 tiệm cận tương tự nhau, do đó MagicSort có cận hẹp là $\Theta(n^2)$ và chặn trên là $O(n^2)$. Hoặc ta có thể chọn $c = 1$, khi đó $n^2 - n \le 1n^2$ với mọi số tự nhiên n. Nhưng thông thường ta chọn c lớn để dễ nhìn thấy "sự hiển nhiên" này hơn.

Ví dụ tiếp theo là thuật toán tìm kiếm nhị phân, với một cách cài đặt khác. Dĩ nhiên, mảng tìm kiếm phải được sắp xếp sẵn.

```
WeirdBinarySearch(x, A[1..n])
    c ← 1
    j ← ⌊n/2⌋
    while j ≥ 1
        while (c + j ≤ n) and (A[c + j] ≤ x)
            c ← c + j
        j ← ⌊j/2⌋
    if A[c] = x
        return c
    else
        return NotFound
```

Tương tự FastPower, biến j chia đôi ở mỗi lần lặp, do đó vòng lặp while ngoài chạy trong thời gian $O(\log n)$. Vòng lặp while trong dừng khi có một hoặc cả hai điều kiện sai. Chúng ta không có thông tin gì về x, nên ta dựa vào phép so sánh $c + j \leq n$ để "đếm" số lần code bên trong thực thi. Hai điều cần quan tâm ở đây là khoảng cách từ c đến n, và j:

$$n - c \leq c + 2j = c + 2\lfloor n/2 \rfloor.$$

Câu lệnh gán $c \leftarrow c + j$ bên trong được thực thi nhiều nhất là 2 lần. Sau lúc này $c \geq n$, nên khi quay lại while kiểm tra thì không thể nào thỏa $c + j \leq n$ được nữa, vì j tối thiểu là 1 (đảm bảo bằng vòng lặp while ngoài). Vậy vòng lặp while trong có độ phức tạp là $O(2)$. Hai vòng lặp lồng nhau nên thuật toán WeirdBinarySearch chạy trong thời gian $O(2 \log n) = O(\log n)$.

2.2.3 Một số lưu ý về big-Oh

- Chúng ta có thể nói thuật toán FastPower chạy trong thời gian $O(n)$ hoặc $O(n^2)$ hoặc thậm chí $O(n!)$. Người ta có thể nói một thuật toán có độ phức tạp thời gian $O(n^5)$ trong khi thuật toán thực sự là $O(\sqrt{n})$. Tất cả đều đúng, nhưng không mang lại thông tin hữu ích. Mặt khác, cũng có người dùng $O(n^2)$ để nói thời gian chạy tăng lên theo n^2: *không hơn, không kém*; trong khi thực sự big-Oh chỉ cho biết thời gian chạy là *không hơn*. That's OK. Nhưng hãy sử dụng các tiệm cận làm rõ thời gian chạy của thuật toán nhất.

- Big-Oh có thể dùng để kí hiệu trường hợp xấu nhất (worst case) của thuật toán, nhưng nó cũng có thể dùng để kí hiệu rất nhiều hàm số khác, as well. Nhìn chung big-Oh dùng để làm việc với tiệm cận trên. Chúng ta có thể dùng nó cho bất kì thứ gì ta muốn. Nhiều người từ nhiều lĩnh vực dùng big-Oh với nghĩa "xấp xỉ": cân nặng $O(14)$ kilogram, nhiệt độ Celsius $O(38)$, khoảng cách $O(169.69)$ nanomet.

- Độ phức tạp thời gian của *bất kì thứ gì* với input cố định là $O(1)$. Sẽ khá vô nghĩa nếu nói big-Oh của "thuật toán sắp xếp mảng gồm 456 phần tử", hay "thuật toán tính 7 giai thừa".