The more we reduce ourselves to machines in the lower things,
the more force we shall set free to use in the higher.

— Anna C. Brackett, *The Technique of Rest* (1892)

The methods given in this paper require no foresight or ingenuity,
and hence deserve to be called algorithms.

— Edward R. Moore, "The Shortest Path Through a Maze" (1959)

1 Thuật toán

Thuật toán (algorithm) là phương pháp để tìm ra đáp án từ vấn đề đã cho, bằng cách sử dụng một dãy hữu hạn các bước đơn giản và rõ ràng có thể thực thi bằng máy tính. Ta có thể xem thuật toán là thủ tục tính toán (computational procedure).

Dữ liệu vào (input) ⇒ Thuật toán ⇒ Dữ liệu ra (output)

Một thuật toán phải có đồng thời ba tính chất sau:

Tính đúng đắn: Thuật toán dẫn đến kết quả chính xác cho mọi input có thể có của bài toán. Với mỗi input, thuật toán chỉ đưa ra một output tương ứng, không có chiều ngược lại[1].

Tính dừng: Thuật toán phải dừng lại sau một số bước.

Tính xác định: Các bước thực thi thuật toán phải rõ ràng, không được mập mờ và hiểu theo một nghĩa duy nhất.

Ngoài các tính chất bắt buộc trên, khi sử dụng thuật toán còn phải cân nhắc *tính hiệu quả:* tốc độ và bộ nhớ. Có nhiều bài toán mà đến nay máy tính phải mất hàng trăm năm, hàng triệu năm, hàng tỉ năm,... để giải. Ví dụ như các thuật toán mã hóa đều bị phá được bằng phương pháp tấn công bạo lực (brute force)[2] tuy nhiên (gần như tuyệt đối trong các trường hợp) số lần thử tất cả mật mã được tính theo cấp giai thừa! Bên cạnh đó cũng có nhiều bài toán dùng một lượng bộ nhớ rất lớn. Các thuật toán xử lí dữ liệu lớn (big data) không thể nào ngây thơ đến mức tải một lúc hết hàng trăm gigabyte dữ liệu vào RAM (ít nhất là đến thời điểm viết tài liệu này) để tính toán được. Thực tiễn cho thấy nhiều bài toán quá phức tạp để tính toán với chi phí hợp lí, đôi khi ta chỉ cần cách giải gần tối ưu, có sai số chấp nhận được. Trong trường hợp này, người ta đã mở rộng khái niệm của thuật toán bằng cách bỏ qua tính chính xác, gọi chung cách giải bài toán là *giải thuật*.

[1] Đây là tính đơn trị của hàm số trong toán học. Lưu ý rằng tính chất này không dùng cho các bài toán về xác suất, vì xác suất là công cụ để làm việc với sự ngẫu nhiên.

[2] Nếu câu này sai, đó là lỗi của TS. Thái Thanh Tùng trong *Giáo trình mật mã học & an toàn thông tin*.

Để lấy ví dụ cho thuật toán, ta thảo luận về hai thuật toán tìm vị trí cơ bản. Để đơn giản, không gian tìm kiếm trong hai ví dụ dưới đây là mảng một chiều $A[left..right]$.

Tìm kiếm tuyến tính

Ý tưởng của thuật toán này là lần lượt ghé nhà từng phần tử, nếu gặp đúng phần tử cần tìm thì trả về địa chỉ nhà của nó.

```
LinearSearch(A[left..right], target)
    for i ← left to right
        if A[i] = target
            return i
    return (−1)
```

Tìm kiếm nhị phân

Tuy nhiên, khi mảng A *được sắp xếp*, chúng ta có thể tìm kiếm nhanh hơn rất nhiều. Trật tự của phần tử sẽ dẫn đường cho việc tìm kiếm. Ta sẽ giả sử mảng A đã được sắp xếp theo trật tự không giảm, hay $A[1] \leq ... \leq A[k] \leq ... \leq A[n]$. Thông tin quan trọng ở đây là: phần tử bên trái sẽ nhỏ hơn hoặc bằng phần tử bên phải. Chọn phần tử giữa (roughly) mảng để so sánh với *target*, nếu bằng nhau thì tìm kiếm thành công; nếu *target* lớn hơn thì tìm tiếp trong đoạn nửa bên phải mảng và ngược lại tìm tiếp trong đoạn nửa trái.

```
BinarySearch(A[left..right], target)
    while left ≤ right do
        mid ← left + ⌊(right − left)/2⌋
        if target = mid
            return mid
        if target > mid
            left ← mid + 1
        else
            right ← mid − 1
    return (−1)
```

Phần cập nhật đoạn tìm kiếm mới tương đối dễ hiểu. Vì hai đoạn được chia ra bởi *mid* nên vị trí bên cạnh *mid* sẽ làm giới hạn của đoạn mới: đoạn mới bên trái là $A[left..mid-1]$, tương tự với đoạn bên phải là $A[mid+1..right]$. Dần dần, kích thước của đoạn mới sẽ thu nhỏ lại. Chúng ta sẽ sớm thấy rằng, với mảng đã được sắp xếp, tìm kiếm nhị phân chạy nhanh hơn tìm kiếm tuyến tính.

2 Mô tả thuật toán

Một mô tả hoàn chỉnh của bất kì thuật toán nào bao gồm bốn phần được đánh số sau:

2.1 Xác định vấn đề

Trước khi giải quyết vấn đề phải biết được vấn đề cần giải quyết. Các vấn đề trong thế giới thực, nhìn chung đều được mô tả bằng ngôn ngữ nói. Ngôn ngữ là một công cụ tư duy phức tạp và hơn hết dễ gây nhập nhằng về ngữ nghĩa, vì vậy hãy sử dụng các công cụ chính xác hơn để biểu diễn lại như *toán*: số, mảng, đồ thị, cây,... Hãy chú ý đến những giả định ngầm bên trong, ví dụ trong vấn đề tìm đường đi ngắn nhất, độ dài đường đi không được là số âm.

Gần như ở tất cả các cuộc thi thuật toán, đề bài đều cho một (vài) cặp input – output làm mẫu. Nhưng cuộc đời không giống cuộc thi, thuật toán lúc này có sẽ cần kiểu biểu diễn input và output đặc thù nào đó mà vấn đề tự nó không nói ra. Ví dụ một thuật toán nhân, nhìn vào ai cũng biết input là hai con số, output là một con số, nhưng nếu cần nhân hai số có hàng trăm chữ số thì cách biểu diễn này vô dụng.

2.2 Diễn giải thuật toán

Thuật toán không nên được viết dưới dạng ngôn ngữ tự nhiên thuần túy vì không thể mô tả chính xác toàn bộ những khái niệm phức tạp, như đệ quy chẳng hạn. Cũng không nên dùng mã để diễn giải ý tưởng, mã được dùng lập trình cho máy tính, không phải con người, nên nó rất khó đọc. Cách diễn giải tốt nhất là hòa trộn cả ngôn ngữ nói và ngôn ngữ lập trình lại. Khi diễn giải, hãy giữ mọi thứ *đơn giản* như bản chất của nó, không rắc rối hơn cũng chẳng phải sơ sài hơn, nếu có vòng lặp, hãy viết bằng vòng lặp; nếu có đệ quy, hãy viết theo kiểu đệ quy. Hãy để mọi thứ rõ ràng!

2.3 Chứng minh tính đúng đắn

Đứng trước một vấn đề, lời giải của chúng ta có xu hướng kết hợp của các kĩ thuật đã biết và ý tưởng mới lóe ra trong đầu. Cách giải quyết này gọi là "thuật giải"[3] tự nhiên", thường được dùng bởi vì thấy nó dẫn đến đáp án. (Rất tiếc, hầu hết ý nghĩ thế này đều sai.) Chúng ta cần một *chứng minh toán học* để chứng minh thuật giải này có đủ tiêu chuẩn của một thuật toán. Và chứng minh của các thuật toán thường sử dụng *quy nạp*.

2.4 Xem xét thời gian thực thi

Tùy từng trường hợp mà ta cần chọn thuật toán có tốc độ cho phù hợp. Nhưng làm sao để biết thuật toán đủ nhanh hay chưa? Người ta đã đưa ra một đơn vị để đo, đó là ***phức tạp thời gian*** (time complexity). Độ phức tạp thời gian không cho biết một thuật toán chạy trong bao

[3]Nhắc lại, "Thuật giải" hay "giải thuật" là một khái niệm mở rộng của thuật toán, nó không gò bó trong tính đúng đắn và tính rõ ràng. Tiêu biểu là *giải thuật di truyền.* hay *giải thuật tham lam*.

lâu mà nó ước lượng thuật toán chạy *nhanh đến mức nào*. Ý tưởng rất đơn giản: sử dụng một hàm số có tham số là kích thước của input. Bằng cách tính độ phức tạp thời gian, chúng ta có thể biết liệu thuật toán đủ nhanh hay chưa mà không cần trực tiếp chạy thử.

Hàm phức tạp thời gian được kí hiệu $O(\cdots)$ với dấu ba chấm đại diện cho kích thước input (thường thấy) hoặc một hàm[4] nào đó. Ví dụ, nếu input là một mảng, n là kích thước của mảng; nếu input là một chuỗi, n là độ dài của chuỗi,...

3 Phức tạp thời gian

Trước khi phân tích, rất mong bạn đọc biết rằng không phải tất cả thuật toán đều đáng để phân tích sâu sắc. Việc nghiên cứu quá chi tiết dễ khiến ta sa vào thuật toán "lý thuyết" xa rời các ứng dụng cụ thể. Dẫu biết rằng bạn đọc muốn đoán được khoảng thời gian chạy tương ứng với kích thước dữ liệu, nhưng ta không thể thực hiện được điều này với hầu hết các thuật toán. Vì thế, ta sẽ quan tâm đến việc tìm một ***chặn trên*** (upper bound) mà với dữ liệu bất kì thì thời gian chạy của thuật toán không vượt quá chặn trên này. Nói ngắn gọn, ta tìm độ phức tạp thời gian cho trường hợp xấu nhất của thuật toán.

Ta kí hiệu thời gian mà một khối lệnh i cần với đầu vào có kích thước n là $T_i(n)$. Ví dụ thuật toán WHATEVER bên cạnh có hai khối lệnh ở hai vòng lặp for, thời gian chạy của khối lệnh đầu là $T_1(n) = n$, thời gian chạy của khối lệnh thứ hai là $T_2(n) = 2n$.

```
WHATEVER()
    for j ← 1 to n
        WRITE("example")
    for k ← 1 to 2n
        WRITE("example")
```

Phân tích của ta cần một cách để so sánh hai hàm. Một phương pháp dùng nhiều nhất là tỉ lệ giữa chúng. Ví dụ $f(n) = n$ và $g(n) = 2n$ thì tỉ lệ giữa chúng là:

$$\frac{f(n)}{g(n)} = \frac{1}{2} = 0.5.$$

Lưu ý rằng không nhất thiết tỉ lệ này phải là một số cố định nào đó. Ta sẽ quan tâm đến tỉ lệ này khi n là một số rất lớn, tỉ lệ này sẽ dần ổn định. Thay vì chọn một số n nào đó, ta sẽ dùng ngay khái niệm *giới hạn hàm số* trong toán học.

3.1 Tương đương tiệm cận

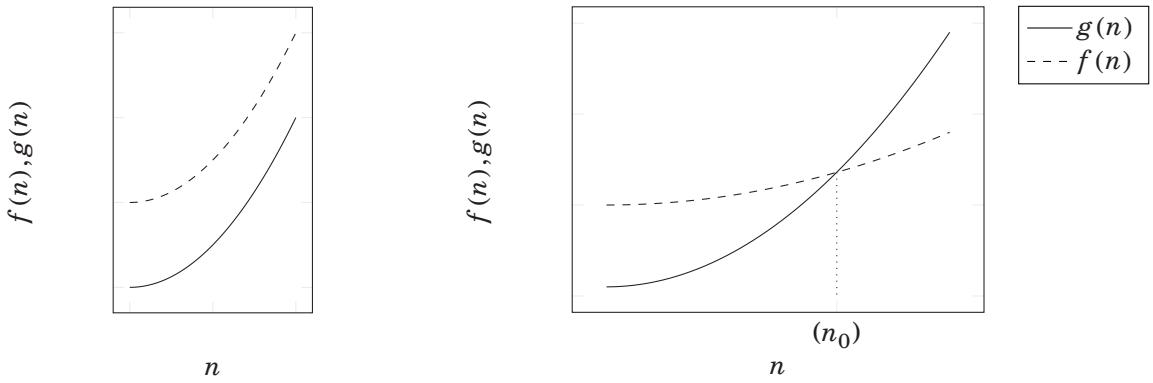
Hai hàm số $f(n)$ và $g(n)$ được gọi là *tương đương tiệm cận* khi và chỉ khi:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1 \qquad [1]$$

Nếu $f(n)$ và $g(n)$ là tương đương tiệm cận của nhau, ta kí hiệu $f(n) \sim g(n)$. Ví dụ, $f(n) = 3n$ và $g(n) = 3n + 3$ là tương đương tiệm cận của nhau:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{3n}{3n+3} = 1.$$

[4] Có nhiều lúc tham số đưa vào không phải là một con số trực tiếp mà qua kết quả của một hàm khác.



(a) Tương đương tiệm cận.

(b) $f(n)$ có $g(n)$ là chặn trên.

Figure 1

3.2 Kí pháp big-Oh

Kí pháp big-Oh được dùng để chỉ *chặn trên* (upper bound). Để kí hiệu tập hợp các hàm tăng trưởng không nhanh hơn $g(n)$ khi n tiến đến vô cùng, ta dùng $O(g(n))$. Theo toán học mà nói, $f(n) = O(g(n))$ khi và chỉ khi[5]:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty. \tag{2}$$

Định nghĩa trên có thể được phát biểu theo cách khác. $f(n) = O(g(n))$ khi và chỉ khi đồng thời tồn tại hằng số c và giá trị n_0 nhỏ nhất (xem Figure 1 (b)) sao cho với mọi $n \geq n_0$ thì:

$$f(n) \leq c \cdot g(n). \tag{3}$$

Ta có thể chứng minh được rằng $T(n) = 10n + 5 = O(n)$:

$$\begin{aligned}\lim_{n\to\infty} \frac{10n+5}{n} &= \lim_{n\to\infty} \frac{10n}{n} + \lim_{n\to\infty} \frac{5}{n} \\ &= 10 + 0 \\ &< \infty.\end{aligned}$$

Sử dụng [3] ta có thể chọn $c = 15$ và $n_0 = 1$ giữa vô vàn các giá trị thỏa mãn khác. Khi đó $\forall n \geq n_0,\ 10n + 5 \leq 15n = c \cdot n$.

Một số tính chất:

- Với một hằng số c bất kì, $c = O(1)$.
- $c \cdot f(n) = O(f(n))$.
- Mỗi big-Oh đều thuộc big-Oh có cấp cao hơn nó. Ví dụ: $O(n\log(n)) = O(n!)$.
- Với mọi số nguyên dương k, $2^n + n^k = O(2^n)$ vì $\lim_{n\to\infty} \frac{n^k}{2^n} = 0$.

[5]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 $O(g(n)) \notin f(n)$.

- Với cơ số k bất kì, $\log_k(n) = O(\log(n))$.
- Nếu $f(n) = O(g(n))$ và $g(n) = O(h(n))$ thì $f(n) = O(h(n))$.
- Giả sử khối lệnh 1 có thời gian chạy là $T_1(n) = O(f(n))$ và khối lệnh 2 có thời gian chạy là $T_2(n) = O(h(n))$:
  Nếu hai khối này rời rạc: $T(n) = T_1(n) + T_2(n) = \max(O(f(n)), O(h(n)))$.
  Nếu hai khối này lồng nhau: $T(n) = T_1(n) \cdot T_2(n) = O(f(n) \cdot h(n))$.

Một thuật toán được gọi là phức tạp thời gian **đa thức** nếu nó có độ phức tạp thời gian lớn nhất là $O(n^k)$ (với k là hằng số). Ở các ví dụ trên, tất cả đều là phức tạp đa thức trừ $O(2^n)$ và $O(n!)$. Song, vẫn có những vấn đề mà chưa có thuật toán nào giải trong thời gian đa thức, ví dụ như những vấn đề thuộc lớp **NP-hard**.

Cuối cùng, để bạn đọc có cái nhìn rõ hơn, ta sẽ tìm độ phức tạp thời gian của một thuật toán sắp xếp quen thuộc:

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

Vì hai vòng lặp i và j lồng nhau nên thuật toán có độ phức tạp là:

$$T(n) = n(n-1) = n^2 - n = O(n^2).$$

Ta có thể kiểm tra:

$$\begin{aligned}\lim_{n\to\infty} \frac{n^2-n}{n^2} &= \lim_{n\to\infty} \frac{n^2}{n^2} - \lim_{n\to\infty} \frac{n}{n^2} \\ &= 1 - 0 \\ &< \infty.\end{aligned}$$

3.3 Kí pháp big-theta

Nếu big-Oh dùng để phân tích trường hợp xấu nhất thì Big-theta Θ dùng để phân tích trường hợp trung bình. $f(n) = \Theta(g(n))$ khi và chỉ khi:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \quad \text{với } 0 < c < \infty. \tag{4}$$

Tính chất đặc trưng của big-theta là tính đối xứng:

$$\text{Nếu } f(n) = \Theta(g(n)) \text{ thì } g(n) = \Theta(f(n)). \tag{5}$$

4 Một số thuật toán toán học

4.1 Giai thừa

n giai thừa ($n!$) được định nghĩa là tích của các số nguyên dương từ 1 đến n. Trường hợp đặc biệt $0! = 1$.

```
FACTORIAL(n)
    f ← 1
    for i ← 2 to n
        f ← f · i
    return f
```

4.2 Fibonacci

Dãy số Fibonacci là dãy số mà kể từ phần tử thứ hai trở đi, mỗi phần tử bằng tổng của hai phần tử liền trước nó. Hai trường hợp cơ sở là $f_0 = 0$ và $f_1 = 1$. Dưới đây là thuật toán in ra phần tử thứ n trong dãy, giả sử đã khởi tạo mảng $f[0..n]$.

```
FIBONACCI(n)
    f[0] ← 0
    f[1] ← 1
    for i ← 2 to n
        f[i] ← f[i − 1] + f[i − 2]
    return f[n]
```

4.3 Ước chung lớn nhất

Ước chung lớn nhất (UCLN, greatest common divisor) của hai số a và b là số lớn nhất mà cả a và b đều chia hết. Mã giải dùng thuật toán Euclid như bên. Dùng UCLN ta cũng tìm được bội chung nhỏ nhất (số nhỏ nhất chia hết cho cả a và b, least common multiple):

$$LCM(a,b) = \frac{a \cdot b}{GCD(a,b)}$$

```
GCD(a, b)
    while b ≠ 0
        temp ← a
        a ← b;
        b ← temp  mod b
    return a
```

4.4 Số nguyên tố

Số nguyên tố là số *chỉ* chia hết cho 1 và chính nó. Thuật toán kiểm tra tính nguyên tố:

```
ISPRIME(n)
    if n ≤ 1
        return False
    for i ← 2 to ⌊√n⌋
        if n mod i = 0
            return False
    return True
```

4.5 Sàng nguyên tố

Sàng nguyên tố (sieve of Eratosthenes) kiểm tra tính nguyên tố của các số từ 2 đến n. Ý tưởng cơ bản của thuật toán là nếu một số là số nguyên tố thì bội của nó không phải là số nguyên tố. Thuật toán sẽ đánh dấu các bội vào một mảng. Ví dụ với $n = 20$, mảng được tạo ra như sau:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	1	0	1	0	1	1	1	0	1	0	1	1	1	0	1	0	1

Phần tử có giá trị 0 thì số chỉ mục của nó là số nguyên tố. Đây là mã giả, giả sử tất cả phần tử trong mảng $Sieve[2..n]$ được khởi tạo bằng 0:

```
SieveOfEratosthenes(n)
    for i ← 2 to n
        if Sieve[i] = 0
            m ← 2 · i
            while m ≤ n
                Sieve[m] ← 1
                m ← m + i
    return Sieve[2..n]
```

Thuật toán chỉ chạy lên đánh dấu (vòng lặp `while`) khi `i` là số nguyên tố, do đó độ phức tạp thời gian rất gần $O(n)$.