Phân tích thuật toán

MIS

Giới thiệu

Phân tích thuật toán

Để giải quyết một vấn đề thường có nhiều cách. Để giải một bài toán, có thể có nhiều cách khác nhau. Cần phải lựa chọn cách “tốt nhất” theo một nghĩa nào đó.

Phân tích thuật toán (tiếp)

Thế nào là một thuật giải “tốt”. Có thể nêu hai tiêu chuẩn sau:

1. Đơn giản, dễ hiểu, dễ lập trình.
2. Cho lời giải nhanh, dùng ít tài nguyên máy tính.

Tuy nhiên, không phải lúc nào cũng có thể đảm bảo được cả hai.

Ví dụ

Chúng ta cùng xem xét một thuật toán để tính giá trị của $1/\sqrt{x}$ (giá trị này thường xuất hiện trong lập trình 3D).

Thông thường, người ta sẽ dùng một bảng tìm kiếm để tính toán giá trị này.

Tuy nhiên, trong trò chơi Quake III Arena, một thuật toán khác đã được sử dụng

Ví dụ

```
# tính giá trị của 1/√x
def inverse_square_root(x):
    three_halfs = 1.5
    x2 = number * 0.5
    y = np.float32(x)

    i = y.view(np.int32)
    i = np.int32(0x5f3759df) - np.int32(i >> 1)
    y = i.view(np.float32)

    y = y * (three_halfs - (x2 * y * y))
    return float(y)
```

Ví dụ (tiếp)

Thuật toán mới này có ưu điểm là nhanh, tính chính xác cao so với thuật toán dùng bảng tìm kiếm.

Tuy nhiên, có thể thấy thuật toán này xuất hiện con số ‘khó hiểu’ là 0x5f3759df (số hệ 16, tương đương với 1597463007).

Phân tích thuật toán (tiếp)

Vì vậy, tùy vào bài toán, mà cần phải lựa chọn thuật toán phù hợp:

- Nếu chỉ dùng thuật giải cho một vài lần thì tiêu chuẩn 1 quan trọng hơn tiêu chuẩn thứ 2.
- Trái lại, nếu đây là một bài toán rất phổ biến, thuật giải sẽ còn được dùng nhiều lần thì tiêu chuẩn 2 quan trọng hơn tiêu chuẩn 1.

Phân tích thuật toán (tiếp)

Mục đích của việc nghiên cứu cấu trúc dữ liệu và giải thuật chính là để xây dựng các chương trình hiệu quả.

Tiêu chuẩn 2 chính là tính hiệu quả của thuật giải. Một thuật giải được gọi là hiệu quả nếu nó tiết kiệm được không gian và thời gian.

- Tiết kiệm không gian là chiếm dụng ít bộ nhớ trong thời gian thực hiện.
- Tiết kiệm thời gian là chạy nhanh.

Phân tích thuật toán (tiếp)

Tiêu chuẩn thời gian thực hiện nhanh là quan trọng hàng đầu.

Đánh giá độ phức tạp của thuật giải là đánh giá thời gian thực hiện nó.

Thời gian thực hiện chương trình

Phụ thuộc vào nhiều yếu tố:

- Cấu hình máy tính (CPU, RAM, ...)
- Ngôn ngữ lập trình
- Cấu trúc dữ liệu
- Cài đặt chi tiết

➨ Vì vậy, rất khó để tính toán chính xác thời gian thực hiện của một thuật toán. Người ta thường đưa ra ước lượng tiệm cận thời gian thực hiện.

Ký hiệu Big-O

Giới thiệu

Định nghĩa: Thời gian thực hiện hay chi phí thực hiện hay độ phức tạp chương trình là hàm của kích thước dữ liệu vào, ký hiệu **T(n)** trong đó n là kích thước hay độ lớn của dữ liệu vào.

Lưu ý

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

Đơn vị của T(n) không phải là đơn vị thời gian vật lý như giờ, phút, giây, ... mà được đo bởi số các lệnh cơ bản (basic operations) được thực hiện trên một máy tính lý tưởng.

Các lệnh cơ bản là:

- các phép toán so sánh
- các phép toán gán
- các phép toán số học

Phân loại

Người ta thường xét **T(n)** trong ba trường hợp:

- Trường hợp tốt nhất (best case scenario)
- Trường hợp trung bình (average case scenario)
- Trường hợp xấu nhất (worst case scenario)

Trong bài này, chúng ta sẽ xem xét trường hợp trung bình.

Tốc độ tăng trưởng

Xét hai hàm số:

$$T_1(n) = C_1 n$$

$$T_2(n) = C_2 n^2$$

Ở đây, C_1 và C_2 là các hằng số dương.

Ta có thể thấy, $T_2(n)$ luôn lớn hơn $T_1(n)$ khi giá trị n đủ lớn mà không phụ thuộc vào giá trị của C_1 và C_2

Tốc độ tăng trưởng (tiếp)

Như vậy, ta có thể thấy là cần có công cụ để thể hiện rõ sự tăng trưởng của T(n), loại bỏ các thành phần không quan trọng.

Ký hiệu big-O

Độ phức tạp thuật toán trong trường hợp trung bình được ký hiệu là big-O (O lớn).

Định nghĩa

Cho **T(n)** và **g(n)** là hai hàm số, ta nói $T(n) \in O(g(n))$ nếu tồn tại các số dương c và K sao cho:

$$T(n) \leq cg(n), \ \forall n \geq K$$

Cách đọc: T(n) có độ phức tạp là O(g(n)).

Nói dễ hiểu là T(n) dù có lớn đến đâu thì kể từ một giá trị n nào đó, T(n) sẽ luôn nhỏ hơn cg(n).

Ví dụ

Cho $T(n) = 4n^2 - 2n + 2$, ta có thể thấy rằng:

$2n - 2 \geq 0, \forall n \geq 1$, nên

$4n^2 - (2n - 2) \leq 4n^2, \forall n \geq 1$

Vì vậy, $4n^2 - 2n + 2 \in O(n^2)$.

Ví dụ

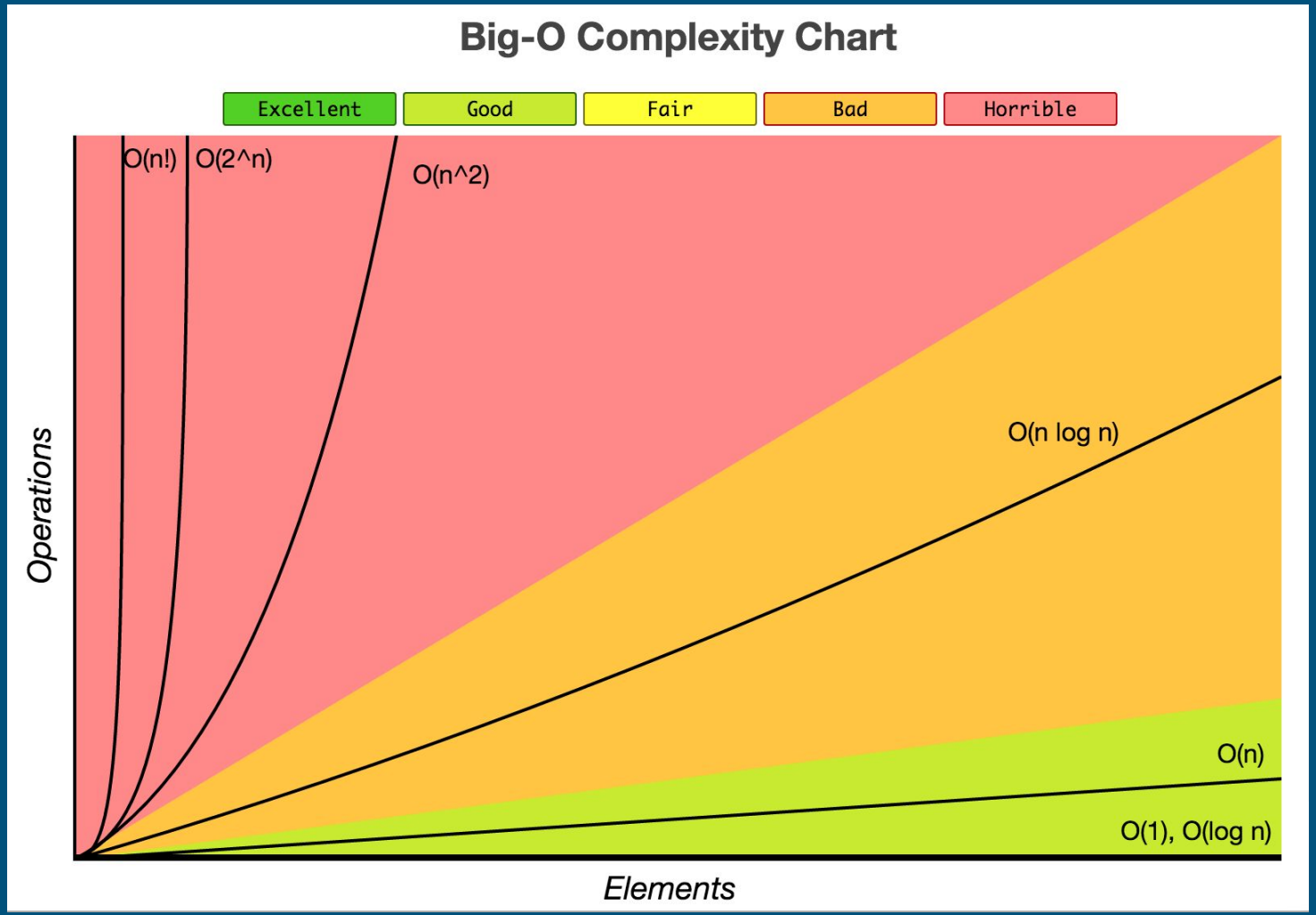
Ta có thể chứng minh được, nếu:

$$T(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$$

thì $T(n) \in O(n^k)$.

Một số ước lượng cơ bản

Hàm	Dạng hàm
Hằng số	1
Logarithm	log n
Tuyến tính	n
Bậc hai	n^2
Đa thức	n^k
Hàm mũ	2^n
Giai thừa	n!


Big-O Complexity Chart
Excellent
Good
Fair
Bad
Horrible
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(1), O(log n)
Operations
Elements

Phương pháp tính big-O

Nhắc lại, T(n) là một hàm được tính qua các lệnh cơ bản sau:

- các phép toán so sánh
- các phép toán gán
- các phép toán số học

Chúng ta thường quy ước các phép tính trên có độ phức tạp O(1).

Quy tắc tổng

Cho $T_1(n) \in O(g(n))$ và $T_2(n) \in O(h(n))$, quy tắc tổng nói rằng:

$$T_1(n) + T_2(n) \in O(\max\{g(n), h(n)\})$$

Tức là, nếu $T_1(n)$ và $T_2(n)$ thực hiện nối tiếp nhau thì độ phức tạp của cả khối sẽ lấy độ phức tạp lớn nhất.

Ví dụ

Nếu $T_1(n) \in O(\log n)$ và $T_2(n) \in O(n)$, thì

$$T_1(n) + T_2(n) \in O(\max\{\log n, n\}) = O(n)$$

do O(n) tăng trưởng nhanh hơn O(log n).

Quy tắc nhân

Nếu một thuật toán $T(n) \in O(g(n))$ được thực hiện $k(n)$ lần thì sẽ có độ phức tạp $O(k(n)g(n))$

Ví dụ

Giả sử một thuật toán có độ phức tạp O(n) được thực hiện n lần, khi đó, độ phức tạp chung sẽ là $O(n * n) = O(n^2)$

Áp dụng trong mã giả

Để ước lượng độ phức tạp của một mã giả, người ta thường chia mã giả thành từng phần nhỏ, ước lượng độ phức tạp của từng phần rồi áp dụng quy tắc cộng để đưa ra độ phức tạp của toàn bộ mã giả.

Áp dụng trong mã giả

Các phép tính số học, so sánh và phép gán được quy ước là O(1).

Áp dụng trong mã giả

Câu lệnh rẽ nhánh:

```
if ... :
      T1
else:
      T2
```

nếu $T_1(n) \in O(g(n))$ và $T_2(n) \in O(h(n))$ thì câu lệnh trên có độ phức tạp $O(\max\{g(n), h(n)\})$ (*tương tự quy tắc cộng*)

Áp dụng trong mã giả

Câu lệnh lặp

```
for ... :
      T
```

thì chúng ta xác định độ phức tạp của T và số lần lặp *trung bình* rồi áp dụng quy tắc nhân.

Ví dụ

```
#Cho thuật toán tính giai thừa
def tính_giai_thừa(n):
    if n < 2:                          #khối if
        res = 1                        #1.1
    else:                              #khối else
        res = 1                        #2.1
        for i = 2, ..., n:             #2.2
            res = res * i              #2.2.1
    return res                         #3
```

Ví dụ (tiếp)

Có thể thấy chương trình có một khối lệnh rẽ nhánh, do đó chúng ta sẽ tính độ phức tạp của từng khối.

- Khối if: chỉ bao gồm 1 phép gán, nên có độ phức tạp O(1).
- Khối else:
  - Một phép gán có độ phức tạp O(1).
  - Một vòng lặp thực hiện n - 2 + 1 = n - 1 lần một phép toán có độ phức tạp O(1), nên tổng thể vòng lặp có độ phức tạp O((n-1)*1) = O(n).

Vì vậy, khối rẽ nhánh có độ phức tạp O(max{1, n}) = O(n)