Phân tích độ phức tạp thuật toán

MADE WITH
beautiful.ai

Phân tích độ phức tạp thuật toán

- Tại sao cần phải phân tích độ phức tạp thuật toán?
- Độ phức tạp thuật toán là gì?
- Các phương pháp phân tích độ phức tạp thuật toán
- Giới thiệu về phân tích độ phức tạp thời gian khấu trừ

MADE WITH
beautiful.ai

Tại sao cần phải phân tích độ phức tạp thuật toán

Một thuật toán có thể mất đến vài tháng hay vài năm để chạy

Bạn đang chạy một thuật toán sắp xếp thời gian biểu của 20000 học sinh, sao cho số lượng nhu cầu của học sinh được đáp ứng là lớn nhất. Bạn đã đợi 12 tiếng và thuật toán của bạn chưa chạy xong, tại sao là như vậy?

- Thuật toán của bạn có lỗi, nó đang lặp vô tận ở đâu đó?
- Máy tính của bạn đã cũ kĩ lắm rồi, đừng có mà mơ nó chạy được tiếp!
- Thuật toán của bạn không chạy nổi 20000 học sinh với mớ môn học đâu :(
- Sắp xong rồi, đợi hai ngày nữa là kịp deadline

Tại sao cần phải phân tích độ phức tạp thuật toán

Khác biệt về thuật toán và công cụ tính toán

A và B cùng nhau code một bài và khi chạy trên máy, thời gian chạy chương trình A viết là 0.087s trong khi chương trình B viết chạy trong 0.143s. Khi nộp bài lên trình chấm, A bị TLE trong khi B AC. Tại sao vậy?

- Bạn dùng máy có vi xử lý 5GHz trong khi bạn ấy dùng chiếc máy cà tàn.
- Thuật toán của bạn chạy không tốt trong một số trường hợp?
- Bạn đang bị vướng ở các bộ test biên làm khó dễ :(

Tại sao cần phải phân tích độ phức tạp thuật toán

Độ tăng về thời gian chạy của thuật toán không phải tuyến tính

---

Bạn chạy một thuật toán với kích thước là 100000 mất 0.5s, với kích thước 300000 mất 1.1s. Vậy với kích thước 900000, bạn mất bao nhiêu thời gian?

- Bạn mất khoảng 2.9s?
- Chưa đo bao giờ nên chưa biết được

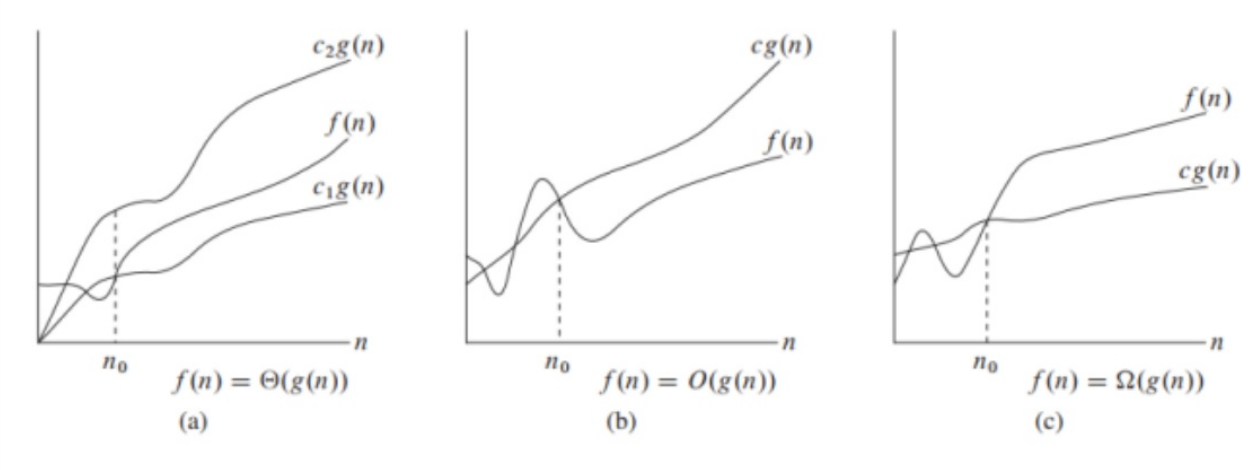
Tại sao cần phải phân tích độ phức tạp thuật toán

Để khắc phục được những điều trên, người ta đã làm gì?

- Một công cụ có thể ước tính thời gian chạy/không gian sử dụng của thuật toán
  - Độ phức tạp thuật toán
- Một công cụ có thể làm đại lượng để đo thời gian chạy/không gian sử dụng
  - Các ký hiệu như O-lớn, Theta-lớn,...

Độ phức tạp thuật toán là gì?

- Độ phức tạp thuật toán là thời gian và lượng tài nguyên cần thiết để chạy nó. Thường thì chúng ta chỉ quan tâm đến thời gian và không gian. Trong phân tích độ phức tạp lý thuyết, người ta thường dùng các ký hiệu O-lớn, Omega-lớn và Theta-lớn để biểu diễn tiệm cận
  - O(g(n)) = {f(n): tồn tại c và n0 sao cho với mọi n≥n0, 0≤f(n)≤cg(n)} O lớn
  - Ω(g(n)) = {f(n): tồn tại c và n0 sao cho với mọi n≥n0, 0≤cg(n)≤f(n)} Omega-lớn
  - Θ(g(n)) = {f(n): tồn tại c1, c2 và n0 sao cho với mọi n≥n0, 0≤c1g(n)≤f(n)≤c2g(n)} Theta-lớn



Độ phức tạp thuật toán là gì?

- Để làm quen với các ký hiệu vừa rồi, chúng ta hãy cũng nhau thực hiện một ví dụ nhỏ

  f(x) = 3x^2 + 6x. f(x) ~ O(g(x)). Tìm O(g(x)).
- Bây giờ, ta sẽ làm một số bài tập sau:
  - f(x) = 2^(n!) + n!^n ~ ?
  - So sánh giữa O(n!^n) và O(n^(2^n))

- f(x) = 2^(n!) + n!^n ~ ?

  f(x) = 2^(n!) + n!^n ~ 2^(n!) + 2^(n^2logn) ~ O(2^(n!))
- So sánh giữa O(n!^n) và O(n^(2^n))

  O(n!^n) = O(2^(nlog(n!)) = O(2^(n^2logn))

  Vậy O(n!^n) = O(2^(n^2logn)) < O(n^(2^n))

Độ phức tạp thuật toán là gì?

- Người ta thường tính độ phức tạp thuật toán theo hai mô hình sau:
  - Mô hình chi phí thông nhất: là một mô hình mà mỗi phép toán mất 1 đơn vị chi phí thời gian/không gian
  - Mô hình chi phí logarit: là một mô hình mà mỗi phép toán mất một lượng logarit(tỉ lệ với độ dài của các toán hạng) đơn vị chi phí thời gian/không gian
- Mô hình chi phí thống nhất thường được sử dụng trong hầu hết trường hợp, trong khi đó, mô hình chi phí logarit thường được sử dụng trong các bài toán liên quan đến thao tác số nguyên lớn, mã hóa, ...

Các phương pháp phân tích độ phức tạp thuật toán

- Phân tích độ phức tạp theo ba trường hợp: tốt nhất, xấu nhất và trung bình
  - Để tính độ phức tạp trong trường hợp tốt nhất và trường hợp xấu nhất, ta chỉ xét các đầu vào sao cho thuật toán chạy ít bước/nhiều bước nhất có thể.
  - Còn trường hợp trung bình thường được tính bằng các phương pháp xác suất nhằm để tính độ
  - Để hiểu rõ, ta xét thuật toán tìm kiếm nhị phân

```
def timkiem(left, right, a, x):
        mid = (left+right)/2
        if(left>right)
                return -1
        if(a[mid] > x):
                return timkiem(left, mid-1, a, x)
        if(a[mid] < x):
                return timkiem(mid+1, right, a, x)
        if(a[mid] == x):
                return mid
```

Các phương pháp phân tích độ phức tạp thuật toán

- Phân tích độ phức tạp theo ba trường hợp: tốt nhất, xấu nhất và trung bình
  - Để hiểu rõ, ta xét thuật toán tìm kiếm nhị phân

    Tại đây, để đơn giản, ta chỉ xét các dãy có độ dài là x = 2^n.

    Ta có:

    - Có 1 số có thể tìm thấy được sau 1 đơn vị thời gian
    - Có 2 số có thể tìm thấy được sau 2 đơn vị thời gian
    - Có 4 số có thể tìm thấy được sau 3 đơn vị thời gian

    ...

    - Có x/2 = 2^(n-1) số có thể tìm thấy được sau n = log2(x) đơn vị thời gian

    Vậy ta có thể dễ dàng thấy được:

    - Trường hợp tốt nhất: O(1)
    - Trường hợp xấu nhất: O(n) = O(logx)
    - Trường hợp trung bình: (1 * 1 + 2 * 2 + 3 * 4 + ... + n * 2^(n-1))/2^(n-1) = (1 * 1 + 2 * 2 + ... + log2(x) * x/2)/x ~ O(logx)

Các phương pháp phân tích độ phức tạp thuật toán

- Bởi vì tính như trên không tiện, có một số trường hợp thường gặp như sau:
  - Độ phức tạp của các vòng lặp (trường hợp xấu nhất)
  - Tuy nhiên, các vòng lặp thường lồng ghép với nhau khá là phức tạp, ta cũng có một mẹo nhỏ để tính độ phức tạp thời gian là dùng tích phân để tính.

  Quiz:

```
for i in range(0,n):
        for j in range(0,i*sqrt(i)):
                for z in range(0, 1/i*j**2*log(i*j)):
                        break
```

  Chương trình trên có độ phức tạp thời gian bao nhiêu?

  O(n^2.5)

Các phương pháp phân tích độ phức tạp thuật toán

- Bởi vì tính như trên không tiện, có một số trường hợp thường gặp như sau:
  - Độ phức tạp của hàm đệ quy (trường hợp xấu nhất):
    - Cách 1: Dùng phương pháp thế(chỉ để chứng minh, không tính được):

      Cần chứng minh $T(n) = f0(n) + T(f1(n)) + T(f2(n)) + ... \sim O(f(x))$, với $f1(n), f2(n), ... < n$

      Bước 1: Giả sử $T(n) = cf(x)$

      Bước 2: Chứng minh $T(n) = f0(n) + T(f1(n)) + T(f2(n)) + ... \leq f0(n) + cf(f1(n)) + cf(f2(n)) + ... \leq cf(n)$

      Ví dụ: CMR $T(n) = T(n/2) + 1 \sim O(\log n)$

      Giả sử $T(n) <= c\log n$

      $T(n/2) = c\log(n/2) = c\log(n) - c\log 2 <= c\log(n)$

Các phương pháp phân tích độ phức tạp thuật toán

- Bởi vì tính như trên không tiện, có một số trường hợp thường gặp như sau:
  - Độ phức tạp của hàm đệ quy (trường hợp xấu nhất):
    - Cách 1: Dùng phương pháp thế(chỉ để chứng minh, không tính được):

      Cần chứng minh T(n) = f0(n) + T(f1(n)) + T(f2(n)) + ... ~ O(f(x)), với f1(n), f2(n), ... < n

      Bước 1: Giả sử T(n) = cf(x)

      Bước 2: Chứng minh T(n) = f0(n) + T(f1(n)) + T(f2(n)) + ... ≤ f0(n) + cf(f1(n)) + cf(f2(n)) + ... ≤ cf(n)

    Quiz:

      1. Chứng minh T(n) = 2T(n/2) + O(n) ~ O(n log n)
      2. Chứng minh T(n) = T(n/5) + T(7n/10+6) + O(n) ~ O(n) (Thuật toán chọn phần tử thứ k)

    Lời giải:

      1. T(n) = 2T(n/2) + O(n) <= 2 cn/2 log n/2 + O(n) = cn log n - cn log 2 + O(n) <= cn log n với c >= 100000
      2. T(n) <= cn/5 + c(7n/10 + 6) + O(n) = 9cn/10 + 7c + O(n) <= cn với c >= 100000

Các phương pháp phân tích độ phức tạp thuật toán

- Bởi vì tính như trên không tiện, có một số trường hợp thường gặp như sau:
  - Độ phức tạp của hàm đệ quy (trường hợp xấu nhất):
    - Cách 2: Dùng định lý master:

      Định lý master giả sử hàm T ta có dạng sau: T(n) = aT(n/b) + f(n), a,b>1, f(n) ~ O(n^c)

      Ta có 3 trường hợp sau:
      - Nếu c < logb(a), T(n) ~ O(n^logb(a))
      - Nếu c = logb(a), T(n) ~ O(n^c*logn)
      - Nếu c > logb(a), T(n) ~ O(f(n))
    - Các bạn hãy tính thử ví dụ: T(n) = 4T(n/3) + n bằng định lý master

      a = 4, b = 3, c = 1

      log3(4) ~ 1.26 > c = 1 => T(n) ~ O(n)
    - Nhận thấy định lý master không thể tính được các hàm truy hồi gọi các hàm con có kích thước khác nhau, vậy chúng ta phải làm gì?

Các phương pháp phân tích độ phức tạp thuật toán

- Bởi vì tính như trên không tiện, có một số trường hợp thường gặp như sau:
  - Độ phức tạp của hàm đệ quy (trường hợp xấu nhất):
    - Cách 3: Dùng định lý Akra-Bazzi:

      Khác với định lý master, định lý Akra-Bazzi giả sử hàm T có dạng sau:

      $$T(x) = g(x) + \sum_{i=1}^{k} a_i T(b_i x + h_i(x)) \qquad \text{for } x \geq x_0.$$

      Với: ai >0, 0<bi<13, g(n) ~ O(n^c), hi(x) ~ O(x/((logx)^2))

      Lấy p là nghiệm của phương trình: $\sum_{i=1}^{k} a_i b_i^p = 1$

      Độ phức tạp: $T(x) \in \Theta\left(x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du\right)\right)$

      Ta hãy tính thử T(x) = n^2 + 7/4T(n/2) + T(3n/4)

      Ta có: a1 = 7/4, b1 = 1/2; a2 = 1, b2 = 3/4; g(n) = n^2

      7/4(1/2)^p + (3/4)^p = 1 => p = 2

      $$\begin{aligned} T(x) &= \Theta\left(x^2 \left(1 + \int_1^x \frac{u^2}{u^3} du\right)\right) \\ &= \Theta(x^2(1 + \ln x)) \\ &= \Theta(x^2 \log x) \end{aligned}$$

Các phương pháp phân tích độ phức tạp thuật toán

- Trong thực tế, người ta thường quan tâm đến độ phức tạp thuật toán trung bình và tổng thời gian chạy của chương trình hơn là trong trường hợp tế nhất. Và việc tính độ phức tạp kỳ vọng thường được áp dụng cho là các giải thuật Las Vegas(một lớp giải thuật ngẫu nhiên cho ra kết quả đúng và thời gian tuy thuộc vào đầu vào) như quicksort, quickselect, stochastic local search, ...
- Chúng ta xét một bài toán vui như sau:

  Cho một dãy A có 50% là số 0. Hãy tìm vị trí của một số khác 0.

```
def solve(arr):
        return x if arr[(x:=randint(0,len(arr)-1))]!=0 else
solve(arr)
```

  Hỏi độ phức tạp thời gian của giải thuật trên là bao nhiêu?

  T = P(chọn đúng lần 1) * 1 + P(chọn đúng tại lần 2) * 2 + ... + P(chọn đúng tại lần n) * n = 0.5 + 0.25 *2 + ... + n/2^n = 2(1-1/(2^n)-n/(2^(n+1)))

  Khi n tiến đến vô cùng, T tiến về 2. => O(1)

Các phương pháp phân tích độ phức tạp thuật toán

- Trong thực tế, người ta thường quan tâm đến độ phức tạp thuật toán trung bình và tổng thời gian chạy của chương trình hơn là trong trường hợp tế nhất. Và việc tính độ phức tạp kỳ vọng thường được áp dụng cho là các giải thuật Las Vegas(một lớp giải thuật ngẫu nhiên cho ra kết quả đúng và thời gian tuy thuộc vào đầu vào) như quicksort, quickselect, stochastic local search, ...
- Chúng ta xét một bài toán vui như sau:

  Cho một dãy A có 50% là số 0. Hãy tìm vị trí của một số khác 0.
- Bây giờ ta sẽ đi đến ví dụ về tính độ phức tạp thời gian trung bình của quicksort

  Ta biểu diễn thời gian chạy của thuật toán quicksort như sau:$T(n) = T(p) + T(n-p) + O(n)$ với p là số phần tử nhỏ hơn pivot trong mảng cần được sắp xếp.

  Nhận thấy nếu pivot là ngẫu nhiên, thì kỳ vọng $p = n/2 \Rightarrow T(n) \sim 2T(n/2) + O(n) \sim O(n \log n)$

Giới thiệu về phân tích độ phức tạp thời gian khấu trừ

- Đôi khi, đối với một giải thuật/cấu trúc dữ liệu, người ta không quan tâm đến thời gian chạy trung bình của một hàm, mà là thời gian chạy trung bình của một chuỗi các hàm. Đây là hướng tiếp cận của phân tích độ phức tạp thời gian khấu trừ
- Cấu trúc dữ liệu vector(C++)/list(python) một ví dụ điển hình của phân tích độ phức tạp thời gian khấu trừ:
  - Mỗi lần thêm vào một phần tử, nếu list đầy, gấp đôi sức chứa của list và thêm phần tử vào -> O(n)
  - Nếu list còn sức chứa, thêm phần tử vào -> O(1)

  Giả sử ta thực hiện n lần insert, ta sẽ mất log2(n) lần cần gấp đôi sức chứa và n lần thêm phần tử vào.

  T = (1 + 1 + ... + 1)n + (1 + 2 + ... + 2^(log2(n)) = n + 2^(log2(n)+1) = 3n

  Vậy thời gian thực hiện mỗi phép insert là 3n/n = 3 => O(1)
- Phương pháp trên còn gọi là phương pháp aggregate, tính trung bình thời gian thực hiện của một chuỗi n hàm.

BTVN

1. Phân tích độ phức tạp thời gian thuật toán quickselect.

2. Ngoài phương pháp aggregate còn có phương pháp potiental. Hãy dùng phương pháp đó để tính độ phức tạp của hàm thêm vào trong cấu trúc dữ liệu vector.

3. Các bạn hẳn còn nhớ bài toán 8 hậu, rằng tìm một cách đặt 8 quân hậu sao cho không con nào ăn được nhau. Nếu thuật toán của của ta có các bước như sau:

Bước 1: Khởi tạo bàn cờ.

Bước 2: Nếu đã đặt đủ quân hậu, xuất đáp án.

Bước 3: Tìm tất cả vị trí trong hàng có để đặt được quân hậu sao cho không quân hậu nào ăn được quân hậu khác.

Bước 4: Nếu không có vị trí nào thỏa mãn, đi đến bước 1.

Bước 5: Nếu có, chọn ngẫu nhiên một vị trí, di chuyển đến hàng tiếp theo và quay lại bước 3.

Thì kỳ vọng bạn sẽ phải khởi tạo bao nhiêu bàn cờ