Giải Thuật Lập Trình

Nơi tổng hợp và chia sẻ những kiến thức liên quan tới giải thuật nói chung và lý thuyết khoa học máy tính nói riêng.

‹ Cặp ghép II: Thuật toán Hopcroft-Karp -- Matching II: Hopcroft-Karp algorithm • Một số ứng dụng của cặp ghép trong đồ thị hai phía -- Applications of Bipartite Matching ›

Phân tích thuật toán -- Algorithm analysis

October 17, 2017 in Regular | 1 comment

Bài này mình hướng đến các bạn mới bắt đầu học thuật toán với mục đích giúp các bạn làm quen với một số khái niệm như $O(.), o(.), \Omega(.), \Theta(.)$. Mình đã có một bài riêng (http://www.giaithuatlaptrinh.com/?p=27) định nghĩa các khái niệm này một cách hình thức. Bài này mình bổ sung các định nghĩa đó bằng những giải thích trực quan hơn.

Khi phân tích một thuật toán, ta thường quan tâm đến **bộ nhớ** và **thời gian tính toán**. Khi chúng ta đã thành thạo phân tích một trong hai tài nguyên thì phân tích tài nguyên còn lại không phải quá khó khăn. Do đó, bài viết này mình sẽ chỉ tập trung nói về phân tích thời gian tính toán của một thuật toán.

Phân tích một thuật toán về cơ bản là đếm **số thao tác cơ bản** mà thuật toán đó thực hiện. Định nghĩa chính xác thế nào là một thao tác cơ bản là việc không tầm thường (http://www.giaithuatlaptrinh.com/?p=1719). Tuy nhiên, để đơn giản, ta tạm coi các thao tác cơ bản ở đây là: cộng, trừ, nhân, chia, so sánh và mỗi thao tác cơ bản này mất 1 đơn vị thời gian. Do đó, đôi khi ta cũng coi số thao tác cơ bản như là một **ước lượng thô** của thời gian tính toán. Mình nói ước lượng thô là vì thời gian thực phụ thuộc rất nhiều vào máy tính (hay mô hình tính toán) mà ta sử dụng. Cùng 1 triệu phép nhân số 32 bít nhưng thời gian tính toán của phần cứng khác nhau có thể khác nhau. Tuy nhiên, ta vẫn chấp nhận cách ước lượng thô này vì nó sẽ làm phép phân tích thuật toán đơn giản hơn, loại bỏ sự phụ thuộc phần cứng ra khỏi phép phân tích.

Tại sao dùng Big-O?

Cách giải thích tốt nhất có lẽ là minh họa thông qua ví dụ.

Ví dụ 1: phân tích thời gian của thuật toán sau:

```
SumOfArray(A[1, 2, ..., n]):
    s ← 0
    for i ← 1 to n
        s ← s + A[i]
    return s
```

Thuật toán SumOfArray nhận đầu vào là một mảng $A[1, \ldots, n]$ với n phần tử và trả lại tổng $A[1] + A[2] + \ldots + A[n]$.

Ta sẽ đếm số phép cộng của thuật toán trước. Mỗi lần lặp, thuật toán SumOfArray thực hiện một phép cộng để cập nhật s. Do ta có n vòng lặp, thuật toán thực hiện n phép cộng để tìm s.

Thuật toán SumOfArray chỉ thực hiện n phép cộng? Không hẳn thế. Nếu bạn lập trình bằng ngôn ngữ C chẳng hạn, thì mỗi lần thực hiện vòng lặp for, bạn phải tăng biến đếm i lên 1. Do đó, ta phải tính cả số phép cộng thực hiện trên biến i mà không phải chỉ với biến s. Số phép cộng tăng lên $2n$.

Nếu coi mỗi phép cộng mất 1 đơn vị thời gian thì thời gian của thuật toán SumOfArray có phải là $2n$ đơn vị thời gian? Cũng không hẳn như vậy. Sau mỗi lần lặp, thuật toán còn phải so sánh i với n để kiểm tra điều kiện kết thúc vòng lặp. Do đó, thuật toán sử dụng thêm n phép so sánh. Vẫn chưa hết, thuật toán còn sử dụng n phép gán với biến s và n phép gán với biến i. Tóm lại, thuật toán sử dụng khoảng $5n$ phép toán cơ bản nếu bạn thực thi bằng C.

Nếu bạn thực thi đoạn code trên bằng ngôn ngữ khác C, số lượng phép toán cơ bản có thể nhiều hơn như vậy. Tưởng tượng với đoạn code rất đơn giản trên mà thực hiện đếm chính xác số lượng phép toán cơ bản đã không tầm thường thì với các đoạn chương trình phức tạp hơn thì ta làm thế nào? Big-O $O(.)$ sẽ giúp bạn đếm đơn giản hơn!

Thay vì đếm chính xác, $O(.)$ cho phép ta đếm **tương đối** số thao tác cơ bản. Theo phân tích ở trên, dù ta thực thi thuật toán SumOfArray bằng bất kì ngôn ngữ nào thì số lượng phép toán cơ bản đều không quá $C \cdot n$ với một hằng số C nào đó. Hằng số C có thể là 5 hoặc 10 hoặc 4. Ta gọi nó là hằng số là vì giá trị của nó không phụ thuộc vào n. Theo định nghĩa hình thức của big-O (http://www.giaithuatlaptrinh.com/?p=27) , ta có thể nói:

> Thuật toán SumOfArray có thời gian $O(n)$.

Tóm lại, khi ta nói một thuật toán có độ phức tạp $O(f(n))$, ta muốn nói số lượng phép toán cơ bản mà thuật toán sử dụng không quá $C \cdot f(n)$ với một hằng số C nào đó khi n **đủ lớn**, bất chấp ngôn ngữ lập trình sử dụng. Như vậy, khi dùng $O(.)$ để phân tích thuật toán, ta có thể loại bỏ sự phụ thuộc vào **ngôn ngữ lập trình** (hay thực thi mức thấp) của thuật toán.

Ví dụ 2: Phân tích thuật toán.

```
BubbleSort(A[1, 2, ..., n]):
   for i ← 1 to n − 1
      for j ← i + 1 to n
         if A[i] > A[j]
            tmp ← A[i]
            A[i] ← A[j]
            A[j] ← tmp
   return A[1, 2, ..., n]
```

Có lẽ bạn đọc cũng nhận ra thuật toán BubbleSort thực hiện sắp xếp một mảng đầu vào theo chiều tăng dần. Để phân tích thuật toán này, ta chỉ cần đếm số phép so sánh các phần tử của mảng $A[1, \ldots, n]$ (số lần kiểm tra điều kiện của câu lệnh if). Tại sao? Ngoại trừ khi $n = 1$, mỗi khi ta

thực hiện thao tác bất kì (gán, đổi chỗ, v.v), ta đều thực hiện một phép so sánh. Do đó, chỉ cần đếm số phép so sánh và sử dụng khái niệm $O(.)$ là ta coi như đã xong.

Do mỗi lần lặp ta sử dụng một phép so sánh, thay vì đếm số phép so sánh, ta đếm số lần lặp. Với mỗi i cố định, vòng lặp trong cùng có $n-(i+1)+1=n-i$ lần lặp. Do đó, tổng số lần lặp là:

$$\sum_{i=1}^{n-1} n-i = \sum_{j=1}^{n-1} j = n(n-1)/2 = n^2/2 - n/2 \qquad (1)$$

Như vậy, độ phức tạp của thuật toán BubbleSort là $O(n^2/2-n/2)$.

Theo định nghĩa của $O(.)$ (http://www.giaithuatlaptrinh.com/?p=27), ta có thể loại bỏ hằng nhân tử hằng số trong biểu thức. Do đó, ta có thể viết $O(n^2/2-n/2)=O(n^2-n)=O(n^2)$. Tổng kết lại, ta có:

Lemma 1: Độ phức tạp của thuật toán BubbleSort là $O(n^2)$.

Từ phân tích ví dụ 2, ta có thể thấy rằng kí hiệu $O(.)$ cho phép chúng ta biểu diễn thời gian tính toán bằng các biểu thức đơn giản hơn. Không chỉ có vậy, $O(.)$ còn cho phép chúng ta phát triển các công cụ phân tích mạnh hơn, tổng quát hơn nhưng lại dễ hiểu hơn. Để minh họa điểm này, ta xét ví dụ sau.

Ví dụ 3: Phân tích thuật toán.

```
MergeSort(A[1, 2, ..., n]):
  if n > 1
    m ← ⌊n/2⌋
    MergeSort(A[1, 2, ..., m])
    MergeSort(A[m + 1, 2, ..., n])
    Merge(A[1, 2, ..., n], m)
```

```
Merge(A[1, 2, ..., n]):
  i ← 1; j ← m + 1
  for k ← 1 to n
    if j > n
      B[k] ← A[i]; i ← i + 1
    else if i > m
      B[k] ← A[j]; j ← j + 1
    else if A[i] > A[j]
      B[k] ← A[i]; i ← i + 1
    else
      B[k] ← A[j]; j ← j + 1
  for k ← 1 to n
    A[k] ← B[k]
```

Không khó để nhận ra thuật toán MergeSort thực hiện sắp xếp một mảng theo chiều tăng dần, sử dụng phương pháp chia để trị (http://www.giaithuatlaptrinh.com/?p=48). Phương pháp thường dùng để

phân tích thuật toán kiểu chia để trị là giải hệ thức truy hồi. Ta gọi $T(n)$ là độ phức tạp của thuật toán MergeSort khi mảng đầu vào có n phần tử. Độ phức tạp của hai thủ tục gọi đệ quy lần lượt là $T(\lfloor n/2 \rfloor)$ và $T(\lceil n/2 \rceil)$. Phần còn lại của phép phân tích là tính độ phức tạp của thủ tục Merge.

Nếu bạn đếm chính xác số thao tác cơ bản của thủ tục Merge là một điều không hề dễ dàng vì có các lệnh điều kiện **if-else**. Tuy nhiên, sử dụng $O(.)$, ta có thể khẳng định số thao tác cơ bản là $O(n)$ vì ta chỉ lặp n lần và mỗi lần lặp ta thực hiện $O(1)$ phép tính cơ bản. Do đó, ta có:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n) \qquad (2)$$

Phương trình (2) trông khá đẹp, nhưng giải nó không hề dễ vì có các hàm $\lceil \cdot \rceil$ và $\lfloor \cdot \rfloor$. Tuy nhiên, $O(.)$ cho phép chúng ta bỏ $\lceil \cdot \rceil$ và $\lceil \cdot \rceil$ ra khỏi phương trình (chứng minh tính chất này một cách tỉ mỉ không phải đơn giản, nhưng vẫn có thể làm được). Ta được:

$$T(n) = 2T(n/2) + O(n) \qquad (3)$$

Giải phương trình (3), ta chỉ cần áp dụng định lí thợ, thu được $T(n) = O(n \log n)$.

Rõ ràng thuật toán MergeSort trông khá phức tạp nhưng $O(.)$ cho phép chúng ta phân tích nó một cách khá đơn giản.

Khi nào Big-O gần với thực tế?

Trợ lại bài toán sắp xếp, ta đã thấy hai thuật toán trình bày trong bài này là BubbleSort và MergeSort với độ phức tạp lần lượt là $O(n^2)$ và $O(n \log n)$. Ta có thể kết luận, thuật toán BubbleSort mặc dù đơn giản hơn rất nhiều (về mặt thực thi) so với MergeSort nhưng khi n đủ lớn thì nó sẽ chậm hơn rất nhiều so với MergeSort. Để thấy sự khác biệt thực tế, mình khuyến khích bạn đọc kiểm tra hiệu năng của hai thuật toán với mảng 10 ngàn phần tử (có lẽ bạn sẽ phải đợi khá lâu thì BubbleSort mới thực hiện xong còn MergeSort chỉ cần không đến 1s).

Trong trường hợp bài toán sắp xếp ở trên, $O(.)$ cho chúng ta một cái nhìn khá chính xác giữa phân tích lý thuyết và thời gian chạy thực tế, mặc dù nó chỉ là một ước lượng khá thô. Ước lượng $O(.)$ nói chung thể hiện khá chính xác thực tế nêu sự khác biệt giữa hai thuật toán mà chúng ta đang so sánh là **đa thức** (polynomial separation) hoặc **siêu đa thức** (super-polynomial separation).

> **(Super-)Polynomial Separation:** Sự khác biệt giữa hai thuật toán A và B được gọi là đa thức (siêu đa thức) nếu tồn tại (với mọi) hằng số c sao cho $T_A(n) \geq n^c T_B(n)$, hoặc ngược lại, khi n đủ lớn. Ở đây $T_A(n)$ và $T_B(n)$ lần lượt là độ phức tạp của thuật toán A và B.

Bài tập 1: Chứng minh rằng sự khác biệt giữa BubbleSort và MergeSort là đa thức.

Một ví dụ khác đó là bài toán TSP. Thuật toán quay lui áp dụng cho bài toán TSP có độ phức tạp $O(n!)$ trong khi thuật toán quy hoạch động có

thời gian $O(2^n n^2)$. Sự khác biệt giữa hai thuật toán này là siêu đa thức (quy hoạch động nhanh hơn). Trong thực tế, chỉ cần $n \geq 20$ là ta đã thấy sự khác biệt giữa hai thuật toán này rồi.

Bài tập 2: Chứng minh rằng sự khác biệt giữa thuật toán quay lui và quy hoạch động cho bài toán TSP là siêu đa thức.

Nếu lấy 1s làm chuẩn thì Table 2 dưới đây thể hiện (tương đối) giới hạn đầu vào của bài toán.

Độ phức tạp	$O(n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
Giới hạn	$n \sim 1000000$	$n \sim 1000$	$n \sim 100$	$n \sim 20$

Table 2: Giới hạn đầu vào tương ứng với độ phức tạp khi lấy 1s làm chuẩn. Nếu $n = 10000$ mà thuật toán có độ phức tạp $O(n^2)$ thì thực tế có lẽ mất khoảng 100s thuật toán mới thực hiện xong.

Sử dụng bảng 2, không cần code ta cũng có thể ước lượng được tương đối thời gian chạy thực tế của thuật toán. Đó là lý do mà bạn cần phải biết big-O.

Cẩn thận với Big-O!

Khi sự khác biệt giữa hai thuật toán thấp hơn đa thức thì không nên coi big-O là một tiêu chuẩn so sánh trong **thực tế**. Ở đây mình nhấn mạnh thực tế vì về mặt lý thuyết, những thuật toán có big-O nhỏ hơn thường có những cái nhìn mới về bài toán hoặc những công cụ lý thuyết mới giúp ta có thể giải quyết được nhiều bài toán khác.

Một ví dụ nổi bật là bài toán tìm phần tử thứ k của một mảng (chưa sắp xếp). Bằng cách sắp xếp mảng đầu vào, ta có thể tìm phần tử thứ k trong thời gian $O(n \log n)$. Tuy nhiên, thuật toán Blum-Floyd-Pratt-Rivest-Tarjan có thể giải bài toán này trong thời gian $O(n)$. Liệu thuật toán BFPRT có thực sự nhanh hơn thuật toán sắp xếp? Câu trả lời là có, nhưng khi n khá lớn. Nếu bạn cài đặt không tối ưu thì thuật toán của bạn có khi vẫn chậm hơn sắp xếp khi $n = 10^6$.

Điều này có thể giải thích đơn giản là do các hàm như $\log n$ tăng khá chậm so với n. Hầu hết các bài toán thực tế, n nằm trong khoảng 10^6 cho đến 10^9, tương ứng là khoảng của $\log n$ từ 20 đến 30 (ta giả sử logarithm có cơ số 2). Nếu hằng số ẩn sau big O của thuật toán $O(n)$ lớn hơn hằng số ẩn sau $O(n \log n)$ khoảng 20-30 lần thì trong thực tế ta sẽ không phân biệt được thời gian tính toán của hai thuật toán này.

Ngay cả khi so sánh hai thuật toán có độ phức tạp tương ứng là $O(n \log n)$ và $O(n)$, $O(.)$ cũng không phải là tiêu chuẩn tin cậy thì so sánh hai thuật toán có cùng độ phức tạp thì $O(.)$ càng không phải là một tiêu chuẩn tin cậy. Khi hai thuật toán có cùng big-O thì thường ta sẽ dựa vào kinh nghiệm thực tế để kết luận. Tất nhiên cũng có những lý thuyết phát triển để so sánh các thuật toán như vậy, nhưng phép phân tích sẽ trở nên vô cùng phức tạp.

Các kí hiệu khác

Đã có big-O thì tại sao chúng ta lại cần các kí hiệu $o(.), \Theta(.), \Omega(n)$? Phần này mình cố gắng làm rõ ý nghĩa của các kí hiệu này. Trước hết, các bạn phải xem lại định nghĩa toán học tỉ mỉ của các kí hiệu này tại đây.

Theo phân tích ở trên, thuật toán BubbleSort có độ phức tạp $O(n^2)$. Theo định nghĩa của $O(.)$, sẽ không sai nếu ta nói BubbleSort có độ phức tạp $O(n^3)$. Chỉ là $O(n^3)$ chưa thể hiện đúng nhất độ phức tạp của thuật toán BubbleSort mà thôi. Thông thường, khi ta nói độ phức tạp của thuật toán là $O(T(n))$ thì ta cố gắng tìm biểu thức $T(n)$ gần nhất với độ phức tạp thực sự của thuật toán mà ta có thể chứng minh được. Các kí hiệu $o(.), \Theta(.), \Omega(.)$ sẽ giúp chúng ta biểu diễn về mặt lý thuyết biểu thức $O(.)$ có phản ánh đúng hiệu năng của thuật toán hay không. Để minh họa, ta xét thêm một ví dụ nữa.

Ví dụ 4: Tìm kiếm nhị phân.

```
BinarySearch(A[1, 2, ..., n], a):
  if n = 1
    return n
  else
    m ← ⌊n/2⌋
    if a < A[m]
      return BinarySearch(A[1, 2, ..., m − 1], a)
    else if a > A[m]
      return BinarySearch(A[m + 1, 2, ..., n], a)
    else
      return m
```

Thuật toán BinarySearch tìm kiếm vị trí của phần tử có giá trị a trong mảng $A[1, \ldots, n]$ đã sắp xếp theo chiều tăng dần. Trong giả mã trên, ta giả sử a luôn xuất hiện trong $A[1, \ldots, n]$.

Không khó để thấy thuật toán trên có độ phức tạp $O(\log n)$ (bạn đọc có thể xem phân tích tại đây). Tuy nhiên, không phải lúc nào thuật toán cũng cần tới $O(\log n)$ thao tác. Ví dụ khi $A[1, \ldots, 2n] = [1, 2, \ldots, 2n]$ và $a = n$ thì chỉ cần $O(1)$ thao tác thôi ta đã tìm được vị trí của phần tử a rồi. Câu hỏi đặt ra là liệu cận trên $O(\log n)$ đã chặt (tight) chưa? Sử dụng kí hiệu $o(.)$, ta có thể phát biểu câu hỏi tương đương là liệu thời gian của thuật toán BinarySearch có phải là $o(\log n)$?

Câu trả lời là không. **Với mọi** $n \geq 1$ và mọi mảng $A[1, 2, \ldots, n]$ đã sắp xếp, luôn **tồn tại** a sao cho thuật toán BinarySearch cần **ít nhất** $\lfloor \log n \rfloor$ phép so sánh để có thể xác định được vị trí của a trong mảng $A[1, 2, \ldots, n]$. Ta nói thuật toán BinarySearch có độ phức tạp $\Omega(\log n)$. Không khó để chứng minh rằng (coi như bài tập cho bạn đọc) $a = A[n]$ sẽ thỏa mãn phát biểu trên.

Như vậy, cận trên độ phức tạp của thuật toán BinarySearch là $O(\log n)$ và cận dưới độ phức tạp của thuật toán BinarySearch là $\Omega(\log n)$, ta nói thuật toán BinarySearch có độ phức tạp $\Theta(\log n)$. Kí hiệu $\Theta(.)$ dùng khi biểu thức trong cận trên $O(.)$ và cận dưới $\Omega(.)$ trùng với nhau.

To-do: Update