Cách giải thích thú vị

Big O là gì?

Với khái niệm Big O trừu cmn tượng thì cần có những ví dụ cụ thể - sinh động để tui hiểu và nhớ lâu.

Khi em hay một người con gái khác hỏi, tui có thể lục lại những mảnh ký ức này.

Tình huống

Giả sử tui có 1 ổ cứng di động Samsung chứa rất nhiều file phim JAV full HD không che. Tui muốn gửi file này đến cho một người bạn của mình ở Hà Nội. Tui cần gửi những file đến bạn mình càng nhanh càng tốt vì nếu không có JAV file, không biết anh ấy sẽ sống như thế nào nữa, tình thế rất cấp bách rồi. Theo bạn, tui phải làm sao ri?

Bạn tui đang cần gấp những file này!!!

Giải thuật 1 là có thể gửi qua Internet, tui up lên một cloud nào đó: Fshare, Goolge Drive, Mediafire. Rồi gửi link cho anh bạn của tui. Nhưng JAV full HD mà hàng ngàn bộ, ít nhất gì cũng 3,4 TB rồi.

Thời gian tui up lên rồi thời gian anh bạn tui down xuống, cộng với việc cá mập cắn cáp thì đến Tết cũng chưa xong. Trong khi bạn tui đang cần những file JAV này rất gấp.

Giải thuật thứ 2 là gửi luôn cái ổ cứng qua đường hàng không. Tui sẽ chuyển phát nhanh qua đường hàng không cho anh ấy.

Với 2 giả định này, anh em hãy cùng tui phân tích xem nào.

Time Complexity (Độ phức tạp thời gian)

*Đây chính là khái niệm của Big O. Tui có thể mô tả thời gian chạy của giải thuật (hay còn gọi là **runtimes**) đưa file JAV cho bạn tui như sau*

- **Internet Transfer**: O(s) , trong đó s là kích thước của các file JAV full HD. Điều này có nghĩa là file càng lớn thì càng mất nhiều thời gian, file 5TB khác với file 10TB. Trong đó s chỉ là ký hiệu cho kích thước của file, bạn muốn đặt là gì cũng được. Để đơn giản tui chỉ phân tích độ phức tạp dựa trên kích thước của file. Tuy nhiên khi thực tế, gửi file qua Internet còn phụ thuộc và tốc độ mạng cũng như…….....kích thước của con cá mập.
- **Airplane Transfer**: O(1), bây giờ tốc độ gửi file là một hằng số và hằng số này bằng với tốc độ của máy bay. Dù các file JAV có 1TB hay 100TB thì thời gian vận chuyển file cũng nhiêu đó. Tại sao tui là để O(1) mà không phải O(69) hay O(96). Lát nữa tui sẽ giải thích phía dưới.

Từ khúc này trở đi, khi tui nói **runtimes** có nghĩa đang nói đến **biểu thức** để **mô tả** độ phức tạp của thuật toán nha.

Có nhiều **runtimes** khác nhau. Những runtimes bạn thường thấy là O(log N), O(N log N),

O(N), O(N^2), O(2^N) và nhiều loại khác nữa.

Runtimes có thể chứa nhiều biến. Như độ phức tạp của Internet Transfer ở trên. Nếu tính cả những yếu tố tốc độ mạng, hay kích thước của cá mập thì độ phức tạp bây giờ là O(s.v/m), trong đó s là tốc độ mạng và m là kích thước cá mập (Cá mập càng to, cắn cáp càng nhanh, mạng càng chậm^.^)

Tui lấy thêm một ví dụ nữa nha. Sơn một bức tường chiều dài là w, cao là h thì Big O (wh), nếu chủ nhà muốn sơn nhiều lớp cho bền, 1 lớp vôi, 1 lớp sơn , 1 lớp sơn bóng ,vv với số lớp cần sơn ký hiệu là l thì Big O lúc này = O (lwh)

Recap:

- Big O để **mô tả** độ phức tạp của thuật toán. Ký hiệu O(biểu thức nào đó, cách xác định được biểu thức trong ngoặc này xem phía dưới)
- Runtimes có thể được tính theo nhiều biến khác nhau

Big 0, Big Theta, and Big Omega

"Trời đã sinh ra Big O, sao còn sinh ra **Big Theta, and Big Omega** ?" – Chu Du

Trong học thuật, người ta còn thêm các khái niệm là **Big Theta** O**(**Θ) **, and Big Omega** O**(**Ω)

Chi vậy, wi sờ mo, why ?

"Có lẽ càng sinh ra nhiều khái niệm thì mới có cái để học để nghiên cứu đúng hem? Có thì học đi thắc mắc làm ri? Có ngon thì tự ngâm cứu ra khái niệm rồi bắt tụi tui học nè" – Nhà bác học đại tài Nôbita

- Big O: Trong học thuật dùng để mô tả cận trên của thời gian. Ví dụ thuật toán in tất cả các phần tử của mảng có N phần tử thì có O(N), nó cũng có thể được mô tả là O(N^2), O(N^3), vv miễn là lớn hơn O(N)
- Big Omega: Ω là khái niệm để mô tả cận dưới, trái lại với Big O. In tất cả phần tử trong mảng có Ω(N). Cũng có thể là Ω(logN), hoặc Ω(1) miễn là nhỏ hơn
- Big Theta: Θ: trong học thuật thì Θ có nghĩa gồm cả O và Ω. Là sao? Nếu một thuật toán có cả Ω(N) và O(N) thì có Θ(N) luôn.

Tuy nhiên, trong đời thường thì người ta **gộp** Θ và O lại thành một. Thật khó hiểu khi mà giải thuật in tất cả phần tử của mảng có O(N^2), đúng không nào

Túm váy lại:

- Ngoài Big O còn có Big Theta, Big Omega
- Trong học thuật, hàm lâm sẽ phân biệt rạch ròi 3 cái này
- Đời thường thì coi như O = Θ

Best Case, Worst Case, and Expected Case

Ví dụ bạn có một sổ điện thoại:

Bạn có một sổ điện thoại có 1 000 000 số được sắp xếp theo thứ tự bảng chữ cái rồi (A->Z). Bạn muốn tìm số của người có tên là *Khoa Đẹp Trai*. Đương nhiên là bỏ qua trường hợp bạn đoán được từ K nằm ở đâu rồi lật ra đúng chỗ đó nha, ở đây tui đang nói là tìm theo một **giải thuật** nào đó, chứ lật lụi ai biết đường nào đâu mà tính.

Giải thuật tìm kinh điển nhất là lật ngay số thứ 500 000 và so sánh liên lạc đó với *Khoa Đẹp Trai*, nếu đúng thì chúng mừng, bạn chỉ mất 1 lần tìm kiếm.

Nếu không thì có 2 trường hợp xảy ra, một là tên của số điện thoại thứ 500 000 *đứng trước* chữ Khoa hoặc *đứng sau* chữ Khoa thì ta tiếp tục chia đôi phần còn lại của quyển sổ điện thoại để tìm tiếp như bước 1.

Giải thuật này được gọi là **binary search**.

Giả sử quyển sổ điện thoại chỉ có 3 số: Bảo Bê Đê, Khoa Đẹp Trai, Vũ Gay.

Thì ta mất **nhiều nhất** 2 lần so sánh.

Giả sử quyển sổ điện thoại có 7 số, ta mất nhiều nhất 3 lần

Giả sử quyển sổ điện thoại có 15 số, ta mất nhiều nhất 4 lần

Giả sử quyển sổ điện thoại có 1 000 000 số, ta mất nhiều nhất 20 lần

Theo lý thuyết:

Best Case: trường hợp tốt nhất cái tên cần tìm nằm ngay chính giữa O(1)

Expected Case: O(log n) nếu bạn vẫn chưa hiểu tại sao O(log n) thì có thể xem thêm phần Logarit runtime ở bài viết sau.

Worst Case: O(log n)

Trường hợp quyển sổ điện thoại này cũng 1000 000 số nhưng **chưa được sắp xếp** theo thứ tự aplabet. Giải thuật lúc này là bạn tìm số thứ nhất sau đó nếu không trùng với tên “Khoa Đẹp Trai” thì bạn chuyển sang số thứ hai cứ như vậy thì bây giờ:

Best Case: cái tên cần tìm nằm ngay đầu tiên O(1)

Expected Case: O(n)

Worst Case: O(n)

Chúng ta sẽ ít quan tâm đến best case, bởi lẽ đời đâu như mơ đâu phải lúc nào cũng trúng ngay từ lần đầu tiên.

Vậy Best Case, Expected Case, Worst Case có liên quan gì đến **Big 0, Big Theta, and Big Omega?**

Thực sự chúng éo có mối liên hệ cụ thể nào cả

- Best Case, Expected Case, Worst Case để mô tả Big O trong những **inputs** và hoàn cảnh cụ thể. Chẳng hạn mảng đầu vào có sort chưa, số cần tìm thế nào, vv
- Big 0, and Big Omega, Big Theta mô tả cận trên, dưới của runtimes

Bỏ hằng số

Ok ok. Có lẽ đến đây đã quá dài rồi. Tuy nhiên còn khá nhiều vấn đề chúng ta cần đề cập. Tui muốn mí bạn nghỉ giải lao vài phút rồi tiếp tục đọc tiếp. Đừng vì chán nản mệt mỏi mà vẫn ráng đọc, cố quá là quá cố đó.

Tiếp tục thôi. Bỏ hằng số là sao?

Tức là O(2N) = O(N) = O(669N)

O(N^2 /2) = O(9696*N^2)

Tại sao lại cho bỏ như vậy? Bởi vì định nghĩa Big O là như vậy. Cùng coi định nghĩa của nó nà:

f(n)=O(g(n)) (f is Big-O of g) or $f \preceq g$ if there exist constants N and c so that for all $n \geq N$, $f(n) \leq c \cdot g(n)$.

Thôi bỏ đi. Nhìn định nghĩa càng khó hiểu thêm!

Big O bỏ qua hằng số bởi vì nó **được tạo ra** để mô tả **độ tăng phức tạp trong thời gian dài**, chứ éo phải để tính chính xác cụ thể là chạy mất bao nhiêu giây.

Cái hàm đó tuyến tính thì nhân hằng số nó cũng là tuyến tính, hàm là logarit thì nhân tuyến tính cũng là logarit, hàm mũ thì nhân hằng số nó cũng là mũ.

Nhưng tại sao không tính thời gian chạy cụ thể ra đi? Tui muốn hiểu tại sao cơ?

Ok ok, thích thì chiều.

Câu trả lời ngắn gọn là để tính được thời gian chạy chính xác **rất khó**.

Cùng xem 2 đoạn code dưới đây:

```
int min = Integer.MAX_VALUE;

int max = Integer.MIN_VALUE;

 for (int x : array)

  if (x < min) min = X;

  if (x > max) max = X;

}
```

```
int min = Integer.MAX_VALUE;

int max = Integer.MIN_VALUE;

  for (int x : array) {

    if (x < min)

    min = X;

  }

 for (int x : array) {

    if (x > max)

    max = X;

}
```

Theo bạn thì đoạn code nào chạy nhanh hơn? Đoạn code một thì 1 vòng lặp, đoạn code 2 thì 2 vòng lặp. Nhưng đoạn code 1 cứ một vòng lặp thực hiện 2 dòng code. Nếu đi sâu vào mức hợp ngữ (assembly) thì phép nhân yêu cầu nhiều lệnh hơn phép cộng nên chưa chắc là O(N) đã nhanh hơn O(2N).

Nói túm lại là để đo được chính cmn xác thời gian chạy của thuật toán phụ thuộc vào:

- Tốc độ của cái máy tính, máy PC khác với máy chủ của NASA
- Kiến trúc máy tính
- Complier xử dụng là gì
- Chi tiết phân tầng kiến trúc bộ nhớ: thanh ghi, cache , ram các kiểu con đà điểu

Chung quy lại là có quá nhiều yếu tố, tui cũng đâu phải chuyên gia gì về mấy cái này, có phân tích tới già cũng không hiểu hết được.

Chỉ cần master mấy món này là bạn muốn đo thuật toán chính xác đến từng ms rồi

Ví dụ bạn muốn tập chạy xe máy để đi phượt. Bạn sẽ éo cần quan tâm là xe đó chạy chính xác bao nhiêu km/h tại tùy thuộc vào hoàn cảnh, tình huống bạn chạy. Cũng éo cần quan tâm dung tích xilanh nó thế nào, động cơ 4 thì ra sao.

Mục đích của Big O để so sánh nhanh chậm của thuật toán, từ đó viết thuật toán, hay áp dụng thuật toán để làm phần mềm phục vụ users chứ không cần quan tâm sâu mấy cái bên dưới.

Chừng nào bạn muốn tạo ra xe máy, hay sáng tạo ra công cụ để mô tả độ phức tạp thay cho Big O thì hãy nghiên cứu sâu các kiến thức khác!

Recap:

- Bỏ hằng số trong runtimes đi

Giữ lại đại lượng lớn nhất

$$O(N^2 + 3N + 5) = O(N^2)$$

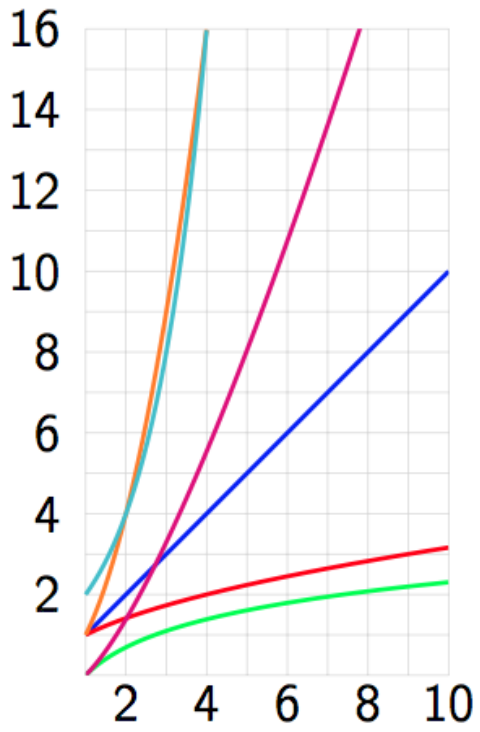
$$O(N + logN) = O(N)$$

$$O(69 * 2^N + 1000 * N^100) = O(2^N)$$

Cũng như quy tắc bỏ hằng số thì với ta chỉ để lại hàm lớn nhất về độ tăng phức tạp theo thời gian.

Dưới đây là độ phức tạp tăng dần theo thời gian, đương nhiên còn nhiều không liệt kê được hết

$$\log n \prec \sqrt{n} \prec n \prec n \log n \prec n^2 \prec 2^n$$



Nhưng điều đó không có nghĩa ta trong runtimes không thể có phép cộng,

Vẫn có thể có O(A^2+B) trong trường hợp ta không biết cụ thể độ tăng của A, B thế nào.

Đọc đến đây có lẽ bạn cũng hiểu tại sao lại dùng O(1) mà không phải O(6969696) hay O(96) rồi đúng không nào?

Thuật toán nhiều phần: Cộng và nhân

Phần này giống đếm bên xác suất nè. Giả sử có 2 đoạn code sau

```
 for (int a : arrA) {

    print(a );

 }

for (int b : arrB) {

   print(b) ;

}
```

```
 for (int a : arrA) {

    for (int b : arrB) {

    print( a + " , " + b);

   }

}
```

Với đoạn code 1 có: O(A+B)

Với đoạn code 2 có: O(A∗B)

Quy tắc nhớ:

- Nếu thuật toán chia ra trường hợp, làm hết cái này xong, làm tiếp cái khác thì cộng
- Nếu trong mỗi lần làm cái này, phải làm cái khác thì nhân.

Reference:

Bài viết tham khảo từ các quyển sách nổi tiếng về giải thuật, các khóa học giải thuật, và các bài trả lời trên stackoverflow cùng với countless đêm viết bài của tác giả.

Data Structures And Algorithms Made Easy Narasimha Karumanchi

The Algorithm Design Manual

Cracking the Coding Interview, 6th Edition 189 Programming Questions and Solutions

Grokking Algorithms

http://stackoverflow.com/questions/22188851/why-is-constant-always-dropped-from-big-o-analysis

http://cs.stackexchange.com/questions/23703/why-is-constant-always-dropped-from-big-o-analysis

http://softwareengineering.stackexchange.com/questions/110395/theta-notation-on-constant-time-why-we-use-the-1