Hiểu đơn giản về khái niệm Big O trong lập trình

I. Một ngày nắng đẹp với Jim

Thử tưởng tượng một ngày nắng đẹp, bạn ngồi uống café cùng một lập trình viên tên Jim ở Google, câu chuyện xoay quanh chủ đề công nghệ và Jim thảo luận với bạn về những thuật ngữ cơ bản trong lập trình. Bạn sẽ trả lời ra sao khi gã hỏi "Này anh bạn, cậu hiểu thế nào về khái niệm Big O".

Bài dịch của tác giả nhằm mục đích chia sẻ kiến thức với các lập trình viên trẻ, và hy vọng phần nào sẽ hữu ích với các bạn trên chặng đường tìm hiểu kiến thức công nghệ đầy thú vị này.

II. Hiểu đơn giản về Big O

Khái niệm Big O hoạc với tên gọi khác trong tiếng Việt là "độ phức tạp của thuật toán" là thuật ngữ thường dùng để chỉ khoảng thời gian tiêu hao để chạy một thuật toán. Các lập trình viên thường sử dụng Big O như một phương tiện để so sánh mức độ hiệu quả của nhiều cách xử lý khác nhau cho cùng một vấn đề.

Nói đến đây, chắc nhiều bạn nghĩ ngay đến số mili giây in ra sau kết thúc chạy chương trình, oh bạn hiểu vấn đề nhanh đấy, nhưng mới chỉ được 1 nửa thôi. Cố gắng đọc hết bài viết để hiểu chính xác về thuật ngữ Big O nhé!.

Khái niệm Big O có thể đúc kết thành một câu ngắn gọn như sau: **Thời gian chạy nhanh như thế nào, còn tùy thuộc vào giá trị đầu vào -input, vì giá trị input sẽ lớn dần lên khi chương trình chạy.**

Nghe có vẻ lý thuyết quá phải không? Nhưng thực sự đây là một khái niệm rất thú vị, và bạn không cần quan tâm đến những cách tiếp cận quá chuyên sâu như với các khái niệm phương trình toán học ở bậc phổ thông. Chỉ hiểu khái niệm Big O một cách đơn giản như sau:

1. **Thời gian chạy nhanh như thế nào** – Sẽ rất khó để xác định chính xác được thời gian chạy của một thuật toán. Vì mỗi máy tính với cấu hình khác nhau sẽ cho ra một kết quả khác nhau tùy thuộc vào CPU, bộ nhớ của máy. Nên, thay vì đặt câu hỏi "thời gian tiêu hao chính xác là bao nhiêu?", ta nên sử dụng Big O để tiếp cận vấn đề theo hướng "Thời gian chạy nhanh như thế nào?".

2. **Còn tùy thuộc vào giá trị input**. Nếu đo một thuật toán chạy chính xác mất bao nhiêu thời gian, bạn có thể đo bằng các đơn vị thời gian cụ thể như giây hoạc mili giây, nhưng với Big O, ta sẽ sử dụng đơn vị đo input - ở đây gọi là "n". Vì vậy các lập trình viên thường lấy đơn vị dựa trên độ phức tạp của thuật toán với input là "n" để tính thời gian chạy chương trình.

3. **Vì giá trị input sẽ lớn dần lên khi chương trình chạy.** Khi bắt đầu chạy chương trình mọi người thường nghĩ sẽ thật tốn công sức nếu người viết một thuật toán phức tạp để xử lý giá trị đầu vào nhỏ, nhưng qua từng bước của thuật toán đó, giá trị ban đầu vốn nhỏ lại dần trở nên lớn hơn thì mọi chuyện lại hoàn toàn khác, chẳng ai còn quan tâm đến độ phức tạp của thuật toán đó nữa. Với phép phân tích Big O, lập trình viên chỉ quan tâm đến yếu tố khác bị tiêu hao (thời gian, dung lượng lưu trữ) khi giá trị đầu vào lớn dần. Đấy chính là lý do mà "phép phân tích Big O" đôi khi còn được gọi là "phép tính tiệm cận".

Okay, lý thuyết suông như vậy đủ rồi, ta hãy cùng xem qua một vài ví dụ dễ hiểu (Java code):

III. Big O và độ phức tạp (độ tiêu hao) của thời gian

Ví dụ 1:

```
public static void printFirstItem(int[] items) {

    System.out.println(items[0]);

}
```

Giải thích: Phương thức ở trên chạy trong một khoảng thời gian được tính bằng O(1) (bằng giá trị input cụ thể là n = 1) – hay còn được gọi là "phép tính bất biến".

Ví dụ 2:

```
 public static void printAllItems(int[] items) {

    for (int item : items) {

        System.out.println(item);

    }

}
```

Giải thích: Vẫn với phương thức như trong ví dụ 1, ta chỉnh số phần tử trong mảng là 1000, sử dụng vòng lặp, phương trình sẽ chạy trong một khoảng thời gian được tính bằng O(1000).

Ví dụ 3:

Như vậy, nếu O(1) hoặc O(1000) với input là một con số cụ thể, ta sẽ biết được đầu ra – output là bao nhiêu lần chương trình in lệnh, ta gọi đó là "phép tính bất biến". Nhưng nếu đơn vị thời gian chỉ được viết là O(n) thì ta sẽ gọi đó là "phép tính tiệm cận" do n còn tùy thuộc vào giá trị input tùy chỉnh.

Từ phép tính tiệm cận O(n), ta truyền vào giá trị input cụ thể, sử dụng mảng 2 chiều, in kết quả cả 2 input (firstItem, secondItem) tương ứng với từng mảng, ta có phép tính O(*n* ^2) "input là đơn vị n bình phương", tham khảo ví dụ sau:

```
public static void printAllPossibleOrderedPairs(int[]
items) {

    for (int firstItem : items) {

        for (int secondItem : items) {

            System.out.println(firstItem + ", " +
secondItem);

        }

    }

}
```

Mở rộng phép tính O(n)

Ví dụ 4:

Ta dùng một số nguyên dương để làm giá trị lớn nhất cho vòng lặp, cũng tạo ra được một phương trình để tính giá trị thời gian theo phép tính tiệm cận O(n) như với cách sử dụng mảng, tham khảo ví dụ sau:

```
public static void sayHiNTimes(int n) {

    for (int i = 0; i < n; i++) {

        System.out.println("hi");

    }

}

public static void printAllItems(int[] items) {

    for (int item : items) {

        System.out.println(item);

    }

}
```

Ví dụ 5:

Tính đơn vị thời gian tiệm cận bằng cách xác định độ phức tạp của thuật toán, ví dụ sau giúp ta xác định phép tính thời gian theo phương trình O(2n):

```
public static void printAllItemsTwice(int[] items) {

    for (int item : items) {
```

```
        System.out.println(item);

    }

    // once more, with feeling

    for (int item : items) {

        System.out.println(item);

    }

}
```

Ví dụ 6:

Hoạc mở rộng phép tính O(n) theo cách khác ta có phép tính thời gian theo phương trình O(1+n/2 + 100):

```
public static void
printFirstItemThenFirstHalfThenSayHi100Times(int[] items)
{

    System.out.println(items[0]);

    int middleIndex = items.length / 2;

    int index = 0;

    while (index < middleIndex) {
```

```
        System.out.println(items[index]);

        index++;

    }

    for (int i = 0; i < 100; i++) {

        System.out.println("hi");

    }

}
```

IV. Hiểu đơn giản về “best case” và “worst case”

Hiểu đơn giản là là khi ta chạy chương trình thì thường thời gian chạy sẽ nằm trong một biên độ thời gian nhất định, ta gọi thời gian chạy nhanh nhất có thể tính được là “best case” và nhìn chung, “worst case” ám chỉ thời gian chạy theo mặc định của phép tính O(n). Nghiên cứu ví dụ sau và đọc giải thích bạn sẽ hiểu rõ về vấn đề này:

```
public static boolean contains(int[] haystack, int needle)
{

    // does the haystack contain the needle?

    for (int n : haystack) {

        if (n == needle) {

            return true;

        }
```

```
    }

    return false;

}
```

Giả sử haystack là một mảng gồm 100 chữ số từ 1 đến 100, needle là một số ngẫu nhiên nằm từ 1 đến 100. Vậy, rõ ràng "best case" ở đây sẽ là O(n) = O(1) - trường hợp xảy ra khi ngay khi bắt đầu vòng chạy, thuật toán mò ngay được needle trong haystack, và "worst case" là O(n) = 100 - trường hợp xảy ra khi vòng lặp chạy và phải đến cuối vòng thuật toán mới mò được needle.

Nói gọn lại, ta có thể hiểu O(n) được coi là "worst case" và O(1) sẽ được coi là "best case". Với một số thuật toán khác, lập trình viên có thể tính toán cụ thể được giá trị trung bình "average case".

V. Big O và độ phức tạp (độ tiêu hao) của bộ nhớ

Trong nhiều trường hợp, khi thiết kế thuật toán, ta lại đặt mục tiêu tiết kiệm bộ nhớ hơn so với tiết kiệm thời gian chạy chương trình. Trong lập trình thì độ phức tạp (độ tiêu hao) bộ nhớ - memory complexity cũng quan trọng ngang ngửa với độ phức tạp (độ tiêu hao) thời gian – time complexity khi chạy chương trình. Trong rất nhiều tình huống, lập trình viên sẽ phải cân nhắc thuật toán gây ra tình trạng bù trừ giữa 2 yếu tố trên.

Quy tắc đơn giản để xác định mức độ tiêu hao bộ nhớ với phương thức Big O là giá trị truyền vào càng nhiều thì thuật toán sẽ sử dụng càng nhiều bộ nhớ để xử lý. Xem 2 ví dụ dưới đây.

Giá trị đầu vào là n=1 với phương thức O(1), 1 dòng lệnh sẽ được in ra khi chạy chương trình:

```
public static void sayHiNTimes(int n) {

    for (int i = 0; i < n; i++) {

        System.out.println("hi");

    }
```

```
}
```

Giá trị đầu vào là n = một số rất to, số lệnh nhiều n lần được in ra khi chạy chương trình:

```
public static String[] arrayOfHiNTimes(int n) {

    String[] hiArray = new String[n];

    for (int i = 0; i < n; i++) {

        hiArray[i] = "hi";

    }

    return hiArray;

}
```

VI. Hạn chế của Big O và lời khuyên cho những lập trình viên trẻ

Trước khi viết một thuận toán, là một lập trình viên trẻ, bạn hãy luôn tạo thói quen cân nhắc đến yếu tố thời gian và dung lượng bộ nhớ tiêu hao khi thuật toán đó được chạy. Dần dần, ta sẽ tạo dựng được thói quen và luôn tìm được phương án tối ưu trước mọi vấn đề cần giải quyết trong lập trình.

Phép phân tích tiệm cận hay Big O là một công cụ cực kỳ hiệu quả, nhưng vẫn có những lưu ý sau:

Giá trị bất biến: Rõ ràng là Big O chỉ giải quyết vấn đề dựa trên tối ưu hóa phương thức code, bỏ qua các giá trị bất biến như đơn vị thời gian, size tiêu hao thực của bộ nhớ. Điều này rất bất lợi trong thời đại công nghệ phần cứng đang phát triển nhanh như vũ bão, tôi đang sử dụng một chiếc máy tính đời cổ để chạy một chương trình tốn đến 5 giờ, tôi miệt mài tốn công sức nghiên cứu tối ưu cách viết code để cải thiện chương trình chạy nhanh hơn 30 phút, thì cũng không bằng tôi ra tiệm sửa máy và nâng cấp phần cứng chiếc máy lên, khi đó máy tôi chạy chương trình nhanh hơn tận 4 tiếng.

Phương án tối ưu hóa code nóng vội: Đôi khi sử dụng phương thức Big O để tối ưu hóa code có thể tiết kiệm được thời gian chạy chương trình và giảm tiêu hao bộ nhớ, nhưng đồng thời sẽ khiến code khó đọc hoạc rất tốn thời gian suy nghĩ để viết. Đối với những lập trình viên trẻ, họ nên học cách viết code một cách tường minh, rõ ràng, dễ đọc, dễ hiểu và dễ bảo dưỡng dù chương trình họ viết ra có thể không được tiết kiệm tài nguyên như các lập trình viên lành nghề khác.

Nói như vậy không có nghĩa là các lập trình viên trẻ chả cần phải quan tâm gì đến Big O, vì bất kỳ một lập trình viên nào (dù trẻ, lành nghề, chuyên gia hay mới bước vào nghề) cũng luôn phải chú ý cân bằng các yếu tố thời gian chạy chương trình, thời gian chạy thuật toán, bộ nhớ tiêu hao, khả năng bảo dưỡng, tính tường minh của những dòng code.