Big O Notation Rules

Table of contents

Có bốn quy luật về Complexity như sau:

Rule 1

Nếu như một thuật toán thực hiện một dãy gồm n bước thực thi, thì độ phức tạp sẽ là $O(n)$.

Giả dụ ta có một thuật toán tìm kiếm số nhỏ nhất tuyến tính sau đây.

```
void findMin(int* a,int n)
{
    int min=a[0];
    for(int i=0; i<n; i++)
    {
        if(a[i]<min){
            min=a[i];
        }
    }
    cout<<min<<endl;
}
```

Vòng lặp thực hiện quét qua n phần tử của mảng, độ phức tạp của nó hiển nhiên sẽ là O(n). Giả sử ta tìm kiếm một số cụ thể nào đó, ví dụ số x, trong mảng a, thì Complexity sẽ phụ thuộc vào vị trí của x. Nếu x càng ở gần đầu mảng (nếu ta duyệt từ đầu mảng), thì độ phức tạp sẽ giảm so với Worst Case.

Rule 2

Nếu như thuật toán thực hiện một tác vụ tốn a bước, sau đó thực hiện một tác vụ khác tốn b bước. Thì tổng Complexity sẽ à $O(a+b)$ bước.

```
void findMin(int* a,int n)
{
    int min=a[0];//O(1)
    for(int i=0; i<n; i++)//O(n)
    {
        if(a[i]<min){
            min=a[i];
        }
    }
    cout<<min<<endl;//O(1)
}
```

Ví dụ trong chương trình trên, các dòng khai báo bên ngoài vòng lặp tính là một bước, nó sẽ có độ phức tạp hằng số O(1). Vậy tổng độ phức tạp của hàm *findMin*; là $O(1+n+1)$, tức là $O(n+2)$. Ngoài ra, còn có trường hợp đặc biệt nếu chương trình thực hiện a, b bước với $a=b$, thì độ phức tạp sẽ chỉ là $O(a)$ hoặc $O(b)$.

```
void findMin(int* a,int* b,int n)
{
    for(int i=0; i<n; i++)//O(n)
    {
        cout<<a[i]<<" ";
    }
    for(int i=0; i<n; i++)//O(n)
    {
        cout<<b[i]<<" ";
    }
}
```

Đoạn code sau khi tối ưu hóa sẽ trở thành:

```
void findMin(int* a,int* b,int n)
{
    for(int i=0; i<n; i++)//O(n)
    {
        cout<<a[i]<<" ";
        cout<<b[i]<<" ";
    }
}
```

Lưu ý rằng độ phức tạp trong trường hợp này không phải $O(2n)$ mà chỉ đơn giản là $O(n)$.

Rule 3

Nếu một thuật toán có độ phức tạp là $O(a+b)$ và a lớn hơn rất nhiều so với b, thì độ phức tạp có thể đơn giản thành O(a).

Trong ví dụ đầu tiên thì độ phức tạp là $O(n+2)$. Trong trường hợp n quá lớn so với 2 $(n >> 2)$ thì chúng ta có thể đơn giản thuật toán thành $O(n)$.

Mở rộng ra, nếu ta có một chương trình có tổng độ phức tạp là $O(n^2+n)$, rõ ràng ta có bất đẳng thức:

$O(n) \leq O(n^2+n) \leq O(n^2+n^2)$. Nếu n là một số lớn $(n^2 >> n)$ thì $O(n^2+n)$ có thể đơn giản thành $O(n)$.

Rule 4

Nếu một thuật toán có độ phức tạp là $O(a)$, và mỗi bước i từ 1 đến a lại có b bước nhỏ nữa. Thì Complexity tổng cộng là $O(a*b)$.

```
void findMin(int** a,int n)
{
    int**a= new int*[n]; //O(1)
```

```
    for(int i=0; i<n; i++)//O(n*n)
    {
        for(int j=0;j<n;j++){
            cout<<a[i][j]<<endl;
        }
    }
}
```

Ví dụ này, có hai vòng lặp lồng nhau. Với mỗi vòng lặp của chỉ số i lại có n vòng lặp của chỉ số j. Do cả a và b đều là n nên Complexity tổng cộng, bao gồm cả dòng cấp phát bộ nhớ đầu, sẽ là $O(n^2 + 1)$. Trong trường hợp dữ liệu đầu vào là lớn, $O(n^2 + 1)$ có thể rút gọn thành $O(n^2)$.

Practices

> Thuật toán dưới đây có độ phức tạp là $O(1)$

```
for(int i = 0; i < n; i++)
{
    i += n / 2;
}
```

Do nhảy hai lần là đến n nên độ phức tạp luôn là một hằng số (số 2).

> Thuật toán dưới đây có độ phức tạp là $O(log_c(n))$

```
for(int i = 1; i < n; i*=c)
    // some O(1) expressions or statements
```

Mục đích của chúng ta là đi xác định số lần lặp. Đặt số lần lặp trong vòng lặp là k (khác n do không phải bước nhảy tuyến tính), ta thấy giá trị của i là $0, c, c^2, c^3, c^4, \dots c^k$. Giá trị cuối phải bé hơn hoặc bằng n.

Ta cho $n = c^k$ và lấy log hai vế theo cơ số c, ta được $log_c(n) = log_c(c^k) = k$. Như vậy với k lần lặp thì độ phức tạp là $O(log_c(n))$.

Cụ thể hơn, để i = 1 trở thành 16, c = 2, thì cần $k = log_2(16) = 4$ lần lặp. K ở đây có ý nghĩa là số lần lặp để giá trị i bằng hoặc nhỏ hơn n khi kết thúc vòng lặp. Nói cách khác, đối với các thuật toán có bước nhảy phi tuyến tính thì số lần lặp bằng Complexity là một hàm số logarith.

> Thuật toán dưới đây có độ phức tạp là $O(log_2(n))$

```
for(int i = n; i > 0; i/=2)
    // some O(1) expressions or statements
```

Trường hợp này là trường hợp khác của trường hợp trên, thay vì nhân cho 2 thì ta lại chia cho 2

> Thuật toán dưới đây có độ phức tạp là $O(n^2)$

```
var value = 0;
for (var i = 0; i < n; i++) for (var j = 0; j < i; j++) value += 1;
```

Có n lần lặp theo biến i, mỗi lần lặp sẽ lặp i lần theo biến j, tổng chuỗi sẽ có dạng 1 + 2 + 3 + 4 + 5 + ... + n. Tổng này sẽ có giá trị $\frac{n(n+1)}{2}$.

> Thuật toán dưới đây có độ phức tạp là $O(log_c(log_2(n)))$

```
for (int i = 2; i <=n; i = pow(i, c))
{
    // some O(1) expressions or statements
}
```

Giá trị của i lần lượt là $2^c, (2^c)^c = 2^{c^2}, 2^{c^3}, \ldots, 2^{c^{log_c(log_2(n))}}$. Giá trị cuối phải bằng hoặc nhỏ hơn n mà ta có giá trị cuối là $2^{c^{log_c(log_2(n))}} = 2^{log_2(n)} = n$, thỏa mãn yêu cầu.

Như vậy có $log_c(log_2(n))$ vòng lặp và do đó độ phức tạp là $O(log_c(log_2(n)))$. Nếu k là 2 thì có độ phức tạp $O(log_2(log_2(n)))$.

Trường hợp khác:

```

```

```
// func() is any constant root function
for (int i = n; i > 1; i = func(i))
{
   // some O(1) expressions or statements
}
```

Độ phức tạp cũng tương tự, là $O(log_k(log_2(n)))$.

> Thuật toán dưới đây có độ phức tạp là $O(\sqrt{n})$

```
void function(int n)
{
    int i = 1, s = 1;
    while (s <= n)
    {
        i++;
        s += i;
        printf("*");
    }
}
```

Thoạt nhìn, có thể thấy giá trị của i là 1, 2, 3... Nhưng giá trị của $s_i = s_{i-1} + i$, tức là giá trị thứ i của s là tổng các giá trị i trước đó.

Mục đích của chúng ta vẫn là đi tìm số lần lặp của vòng lặp. Ta gọi k là số lần lặp. Tổng s lớn hơn n sẽ dừng vòng lặp, nghĩa là nếu $s = 1 + 2 + 3 + 4 + \ldots + k = \frac{k(k+1)}{2}$ > n thì dừng.

Cho $\frac{k(k+1)}{2} = n$, lấy căn hai vế ta được $k = \sqrt{n}$. Suy ra Complexity $O(\sqrt{n})$.

> Thuật toán dưới đây có độ phức tạp là $O(n^5)$

```
void function(int n)
{
    int count = 0;

    // executes n times
    for (int i=0; i<n; i++)

        // executes O(n*n) times.
        for (int j=i; j< i*i; j++)
```

```
            if (j%i == 0)
            {
                // executes j times = O(n*n) times
                for (int k=0; k<j; k++)
                    printf("*");
            }
    }
```

Điều kiện trong vòng lặp thứ hai để lặp trên những số chia hết cho i từ i đến i^2. Ví dụ từ i bằng 2 thì sẽ có hai số là 2 và 4. Nếu i = 5 thì có năm số: {5,10,15,20,25}.

Như vậy với số i bất kỳ thì có i số từ i đến i^2 chia hết cho nó. Và mỗi số như vậy được lặp thêm j lần nữa.

Chưa chắc câu này lắm.