

Trước khi chúng ta đi sâu vào chủ đề Phân tích độ phức tạp của thuật toán, chúng ta hãy hiểu rõ về thuật toán thực sự là gì.

What is the algorithm?

Thuật toán là một bộ quy tắc / hướng dẫn cụ thể mà máy tính sẽ tuân theo để giải quyết một vấn đề cụ thể. Nói cách khác, chúng ta cần nói cho máy tính biết cách xử lý dữ liệu, để chúng ta có thể hiểu ý nghĩa của nó.

Thuật toán là phương pháp để giải quyết một bài toán, công việc cụ thể

What is Algorithmic Analysis?

Chúng ta muốn xử lý số lượng lớn dữ liệu và muốn thực hiện nhanh nhất có thể. Đây là nơi phân tích thuật toán xuất hiện.

Khi thực hiện phân tích thuật toán, chúng ta muốn đánh giá hiệu suất của thuật toán theo kích thước đầu vào của nó.

Chúng ta muốn xử lý số lượng lớn dữ liệu và thực hiện nhanh nhất có thể.

Yếu tố đầu tiên được quan tâm là số lượng đầu vào phải được xử lý. Do đó, nhiều dữ liệu bằng thời gian hơn. Một yếu tố khác mà bạn có thể muốn xem xét quan trọng, là tốc độ của máy chủ.

1 bài toán có rất nhiều thuật toán khác nhau

-Vì vậy phải phân tích thuật toán để lựa chọn 1 thuật toán tốt nhất

-cải tiến thuật toán hiện có để nó tốt hơn

When is an Algorithm considered best?

- Thực hiện đúng
- Tốn ít bộ nhớ
- Thực hiện nhanh

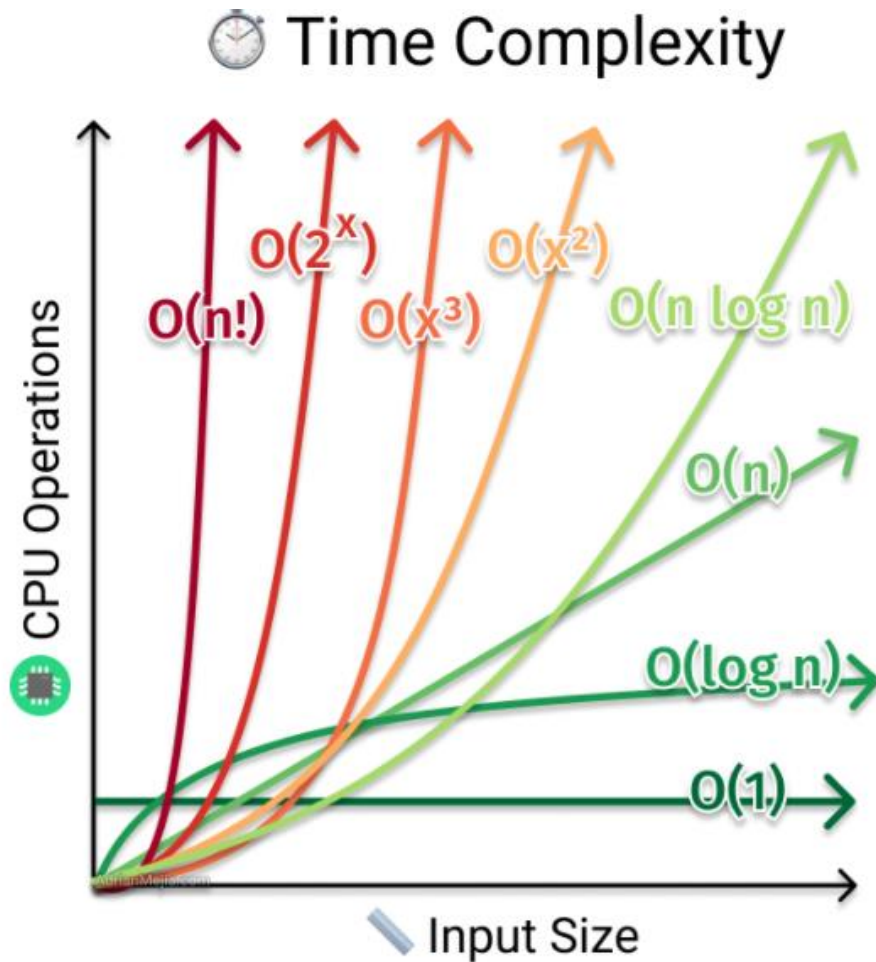
Thời gian thực hiện chương trình

- Là một hàm của kích thước dữ liệu vào, ký hiệu $T(n)$ trong đó n là kích thước (độ lớn) của dữ liệu vào.
- **Ví dụ :** Chương trình tính tổng của n số có thời gian thực hiện là $T(n) = cn$ trong đó c là một hằng số.
- Thời gian thực hiện chương trình là một hàm không âm, tức là $T(n) \geq 0 \forall n \geq 0$.

- Đơn vị đo của $T(n)$ không phải là giờ, phút, giây...
- Là một lệnh được thực hiện trong một máy tính lý tưởng.
- **Ví dụ:** $T(n) = Cn$ thì có nghĩa là chương trình ấy cần Cn chỉ thị thực thi.

Có thể được xác định từ hai yếu tố chính:

- Thời gian thực hiện từng lệnh. Trong đó, những phép toán thường được đánh giá như phép so sánh và phép gán.
- Tần suất thực hiện của các lệnh đó. Thường được đánh giá phụ thuộc độ lớn của dữ liệu.



2.1. Constant $\{O(1)\}$

Một chương trình có độ phức tạp hằng số $O(1)$ nếu nó chỉ chạy đúng một số lượng thao tác, cho dù có tăng kích thước đầu vào. Vì vậy, nó không phụ thuộc vào số lượng phần tử đầu vào N . Những thao tác gán hay phép tính cộng trừ nhân chia đều có độ phức tạp $O(1)$.

Vd: bảng băm, mỗi thao tác lấy dữ liệu đều tốn một số lượng thao tác như nhau, cho dù có tăng số lượng key trong bảng băm

2.2. Logarithmic

Chương trình có độ phức tạp logarit $O(\log N)$ sẽ chậm hơn độ phức tạp hằng số khi tăng kích thước đầu vào. Trong vòng lặp nếu sau mỗi bước lặp thì thao tác còn lại giảm đi thì các bạn có thể ngầm dự đoán nó có độ phức tạp logarit.

Binary Search là một thuật toán điển hình, sau mỗi bước lặp, kích thước mảng để xét giảm đi một nửa nên có độ phức tạp $O(\log_2 N)$. Nếu thuật toán bạn đang viết giảm đi gấp 3 sau mỗi bước lặp thì có thể xem nó có độ phức tạp $O(\log_3 N)$.

Cơ số của hàm \log không quan trọng, việc thay đổi cơ số chỉ tương đương với việc nhân với một hằng số. Nên việc giảm đi một nửa hay giảm đi gấp 3 thì cũng xem là $O(\log N)$.

2.3. Linear

Độ phức tạp tuyến tính $O(N)$ ám chỉ rằng nếu tăng gấp đôi kích thước đầu vào thì thời gian chạy cũng tăng gấp đôi, nghĩa là tăng tuyến tính theo kích thước của N . Thông thường sẽ là một vòng for duyệt hết N phần tử.

Nếu chương trình bạn đang viết chỉ duyệt nửa mảng nhưng mà khi N tăng gấp đôi mà số lần lặp cũng tăng gấp đôi thì nó cũng có độ phức tạp $O(N)$ nhé.

2.4. Linearithmic

Độ phức tạp $O(N \log N)$ biểu thị thời gian chạy cho kích thước N là $N \log N$. Nghĩa là nhiều hơn $O(N)$ nhưng nhỏ hơn $O(N^2)$.

Chẳng hạn, một chương trình bạn đang viết có 2 vòng for lồng nhau. Vòng for ngoài duyệt đến N , vòng for bên trong sau mỗi lần lặp thì giảm đi, thì có thể xem là $O(N \log N)$.

Nhiều thuật toán thuộc nhóm chia để trị cũng có độ phức tạp $O(N \log N)$ như Merge sort hay Quick sort.

2.5. Quadratic

$O(N^2)$ biểu thị thời gian chạy là N^2 cho dữ liệu kích thước N . Thông thường chúng ta viết 2 vòng for lồng nhau mà đều duyệt đến N thì có thể coi là $O(N^2)$ như một số thuật toán sort: Bubble sort, Selection sort, ...

Nếu duyệt mảng N phần tử, các thuật toán có độ phức tạp $O(N^2)$ thì thường là phải xét tất cả các cặp dữ liệu trong mảng này.

2.6. Cubic

Tương tự $O(N^2)$, thuật toán có độ phức tạp $O(N^3)$ thường được viết như 3 vòng for lồng nhau và kiểm tra tất cả các cặp 3 phần tử.

*Khi đánh giá thuật toán ta **thường** đánh giá thuật toán trong trường hợp xấu nhất



TRƯỜNG HỢP TỐT NHẤT, XẤU NHẤT VÀ TRUNG BÌNH

UNIVERSITY

Tìm kiếm tuần tự

Search(x, a, n)

0	1	2	3	4	5	6	7
5	6	3	10	1	4	7	9

Seach(5, a, 8): Trường hợp tốt nhất
Seach(10, a, 8): Trường hợp TB
Seach(15, a, 8): Trường hợp xấu nhất

LỰA CHỌN THUẬT TOÁN NHƯ THẾ NÀO?

Cùng một bài toán, có 2 thuật toán P1 và P2 với thời gian thực hiện là $T1(n) = 100n^2$ và $T2(n) = 5n^3$.

Chọn thuật toán nào?

Khi $n < 20$ thì $T2 < T1$. Khi $n > 20$ thì $T1 < T2$

Chọn P1 vì hầu hết các trường hợp $T1 < T2$

Dùng thời gian để lựa chọn rất khó khăn vì phải so sánh 2 đa thức.

Ta thường xét như này:

KHÁI NIỆM ĐỘ PHỨC TẠP CỦA THUẬT TOÁN

Chỉ xét n^2 và n^3 thì rất dễ dàng lựa chọn vì $n^2 < n^3$


Độ phức tạp thời gian của thuật toán

Ký hiệu O (ô lớn)

$T1(n) = O(n^2)$, $T2(n) = O(n^3)$

Có những ký hiệu khác ω và Θ nhưng thường sử dụng nhất là O

$O(C \cdot f(n)) = O(f(n))$ với C là hằng số. Đặc biệt $O(C) = O(1)$



ĐPT ĐA THỨC VÀ ĐPT HÀM MŨ

- Các độ phức tạp thường gặp:
- $1, \log_2 n, n, n \log_2 n, n^2, n^3, 2^n, n!, n^n$.
- ĐPT đa thức \Rightarrow chấp nhận được.
- ĐPT hàm mũ \Rightarrow cải tiến/ tìm thuật toán khác.
- Trong cách viết, ta thường dùng $\log n$ thay thế cho $\log_2 n$.

Nên chọn đa thức \rightarrow

Nếu là hàm mũ thì nên tìm thuật toán khác tốt hơn

QUY TẮC XÁC ĐỊNH ĐỘ PHỨC TẠP

- Độ phức tạp tính toán của giải thuật: $O(f(n))$
- Việc xác định độ phức tạp tính toán của giải thuật trong thực tế có thể tính bằng một số quy tắc đơn giản sau:
 - **Quy tắc bỏ hằng số:**
 $T(n) = O(c \cdot f(n)) = O(f(n))$ với c là một hằng số dương
 - **Quy tắc lấy max:**
 $T(n) = O(f(n) + g(n)) = O(\max(f(n), g(n)))$
 - **Quy tắc cộng:**
 $T_1(n) = O(f(n)) \quad T_2(n) = O(g(n))$
 $T_1(n) + T_2(n) = O(f(n) + g(n))$
 - **Quy tắc nhân:**
Đoạn chương trình có thời gian thực hiện $T(n) = O(f(n))$
Nếu thực hiện $k(n)$ lần đoạn chương trình với $k(n) = O(g(n))$ thì độ phức tạp sẽ là $O(g(n) \cdot f(n))$

QUY TẮC CỘNG VÀ QUY TẮC NHÂN

$T1(n)$ và $T2(n)$ là thời gian thực hiện của hai đoạn chương trình P1 và P2.

$T1(n)=O(f(n))$, $T2(n)=O(g(n))$

Quy tắc cộng: P1 và P2 nối tiếp nhau
 $T(n)=O(\max(f(n),g(n)))$.

Quy tắc nhân: P1 và P2 lồng nhau là
 $T(n) = O(f(n).g(n))$.

QUI TẮC CHUNG TÍNH ĐPT CHƯƠNG TRÌNH KHÔNG CÓ CT CON

Các lệnh gán, nhập, xuất, return: $O(1)$

Một chuỗi tuần tự các lệnh: Quy tắc cộng

Cấu trúc IF: $\max(\text{đk}, CV1, CV2)$

Cấu trúc vòng lặp là tổng (trên tất cả các lần lặp) thời gian thực hiện thân vòng lặp.

Trình tự đánh giá:

- Nối tiếp: Từ trên xuống
- Lồng nhau: Từ trong ra

– Quy tắc bỏ hằng số:

$T(n) = O(c.f(n)) = O(f(n))$ với c là một hằng số dương

Xác định được số lần lặp

- `for(i=a; i<=b; i++)`: số lần lặp = $b-a+1$
- `for(i=1; i<=n; i= 2*i)`
- Sau lần lặp thứ 1: $i= 2$
- Sau lần lặp thứ 2: $i= 4$
- ...
- Sau lần lặp thứ k : $i= 2^k$
- Lặp kết thúc khi $i= 2^k= n$
- $k= \log_2 n \Rightarrow$ số lần lặp = $\log_2 n$

Ví dụ:

```
/*1*/ Sum=0;  
/*2*/ for (i=1; i<=n; i=i*2) {  
/*3*/     scanf ("%d", &x);  
/*4*/     Sum=Sum+x; }
```

- 1: $O(1)$
- 3 và 4: $O(1)$
- 2 thực hiện $\log_2 n$ lần \Rightarrow 2: $O(\log n)$
- $T(n) = O(\log n)$

Dùng quy tắc + (max...)

-3,4 nối tiếp nhau nên dùng quy tắc $\max(3,4)O(1) \rightarrow O(1) \rightarrow O(\log n) * O(1) = O(\log n)$

-1,2 dùng quy tắc + $\max(1,2) \rightarrow O(\log n)$

VÍ DỤ 2: SẮP XẾP “NỔI BỌT”

```
void Sort(int a[],int n){
    int i,j,temp;
    /*1*/for(i= 0; i<=n-2; i++)
    /*2*/  for(j=n-1; j>=i+1;j--)
    /*3*/    if (a[j] < a[j-1]) {
    /*4*/      temp=a[j-1];
    /*5*/      a[j-1]= a[j];
    /*6*/      a[j]= temp;
    }
}
```

- 4, 5, 6: $O(1)$
- 3: $O(1)$
- 2 lặp
 $(n-1)-(i+1)+1 = n-i-1$ lần
- 2: $O(n-i-1)$
- 1 chính là toàn chương trình

$$T(n) = \sum_{i=0}^{n-2} (n-i-1) = \frac{n(n-1)}{2} = O(n^2)$$

3-4-> $O(1)$ QUY TẮC NHÂN 2($O(1)*O(n-i-1)$)

1 lồng 2 nên tính tổng

Số lần lặp không xác định

Quy tắc đánh giá từ trên xuống và từ trong vòng lặp ra

TÌM KIẾM TUẦN TỰ

```

search(int x,int a[],
int n){
    int i;
    int found;
    // i=0;
    // found=0;
    // while(i<=n-1&&!found)
    //     if (x==a[i])
    //         found=1;
    //     else i++;
    // return found;
}

```

- 1 • 1, 2 và 7: $O(1)$
- 2 • 5 và 6: $O(1) \Rightarrow$
- 3 • $O(1)$
- } • Trong trường hợp xấu nhất 3 thực hiện n lần $\Rightarrow 3: O(n)$

1-2-3 nối tiếp nhau dùng quy tắc $\max(1,2,3) \rightarrow 3$

```

s = 1; p = 1;
for (i=1; i<=n; i++) {
    p = p * x / i;
    s = s + p;
}

```

\Rightarrow Vậy độ phức tạp của thuật toán là $O(n)$

```

for (i= 1; i<=n; i++)
    for (j= 1; j<=n; j++)
        //Lệnh

```

\Rightarrow Dùng quy tắc nhân ta có $O(n^2)$

```

for (i= 1; i<=n; i++)
    for (j= 1; j<=m; j++)
        //Lệnh

```

\Rightarrow Dùng quy tắc nhân ta có $O(n*m)$

```
for (i= 1;i<=n;i++)
    for (j= 1;j<=m;j++) {
        for (k= 1;k<=x;k++)
            //lệnh
        for (h= 1;h<=y;h++)
            //lệnh
    }
=> O(n*m* max (x,y))
```

```
for (i= 1;i<=n;i++)
    for (j= 1;j<=m;j++) {
        for (k= 1;k<=x;k++)
            //lệnh
        for (h= 1;h<=y;h++)
            //lệnh
    }
=> O(n*m* max (x,y))
```