

TÀI LIỆU LÝ THUYẾT CTDL & GT

Phân Tích Thuật Toán

Nội dung

- Giới thiệu thuật toán và phân tích thuật toán
- Tiêu chí và cách thức đánh giá thuật toán
 - Các tiêu chí cơ bản
 - $Big - O, Big - \Omega, Big - \Theta$
- Vận dụng phương pháp đánh giá

Thuật toán là gì?

- Thuật toán là gì?

- *Thuật toán là các bước cần thực hiện để giải quyết một bài toán.*

- Ví dụ: bài toán “chiên trứng gà”

- *Bước 1: lấy cái chảo.*

- *Bước 2: lấy chai dầu ăn*

- *Có chai dầu ăn không?*

- » *Nếu có, đổ nó vào chảo*

- » *Nếu không, đi mua nó không?*

- *Nếu có, thì đi mua*

- *Nếu không, dẹp món trứng chiên.*

- *Bước 3: bật bếp ga lên*

Tại sao cần phân tích thuật toán?

- *Có bao nhiêu cách để đi từ thành phố Hồ Chí Minh đến Hà Nội?*
 - Máy Bay
 - Tàu hỏa
 - Thuyền
 - Xe khách
 - Xe buýt
 - Đường mòn Hồ Chí Minh...
- *Tại sao không tối ưu duy nhất một cách đi?*
 - Phụ thuộc vào hoàn cảnh, điều kiện, tính sẵn sàng, sự thuận tiện, tiện nghi, ...



Tại sao cần phân tích thuật toán? (tt)

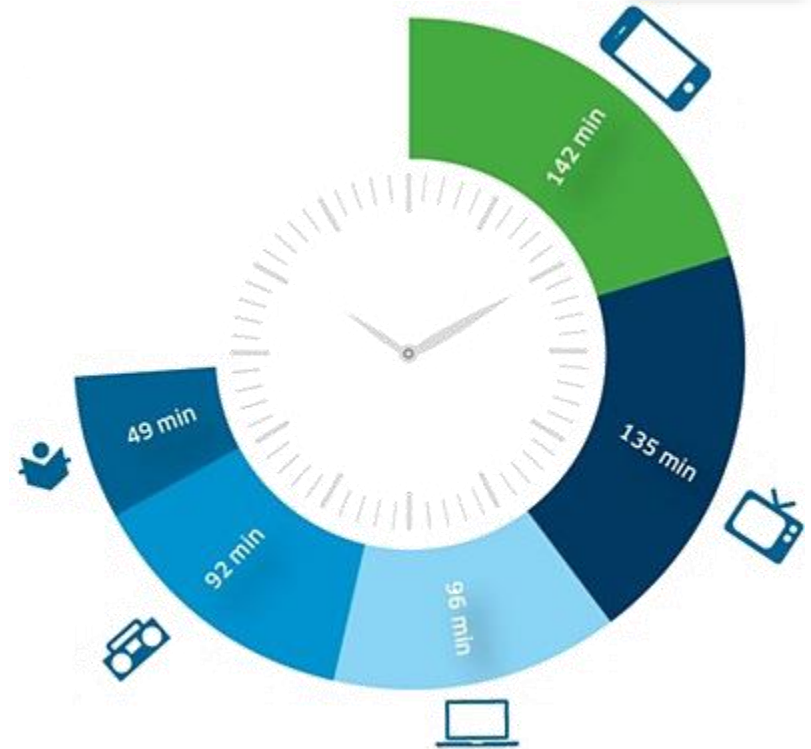
- Trong khoa học máy tính, cũng tồn tại rất nhiều thuật toán để giải quyết cùng một vấn đề.
 - Ví dụ?
 - Bài toán sắp xếp: insertion sort, selection sort, quick sort, ...
 - Giải thuật nén: Huffman, Nén RLE, ...
- Việc phân tích thuật toán giúp xác định thuật toán nào hiệu quả về khía cạnh không gian, thời gian, ...
 - Bao lâu thì chương trình chạy xong?
 - Tại sao chương trình lại bị văng bộ nhớ?
 - ...

Đối tượng của phân tích thuật toán

- Đối tượng xem xét:

- Thời gian chạy (*)
- Bộ nhớ tiêu tốn (*)
- Tính dễ hiểu
- Tính ổn định
- Tính chịu lỗi
- ...

- Đối tượng chính được quan tâm là *thời gian thực thi*.



Phân tích thời gian thực thi

- Phân tích thời gian thực thi:
 - Xem xét *thời gian xử lý* tăng lên như thế nào khi *kích thước của bài toán* tăng lên.
 - Kích thước bài toán thường nằm ở *kích thước đối số vào* (input):
 - Kích thước của mảng
 - Bậc đa thức
 - Số phần tử của ma trận
 - Số lượng bit thể hiện của input
 - Số đỉnh và cạnh của đồ thị
 - ...



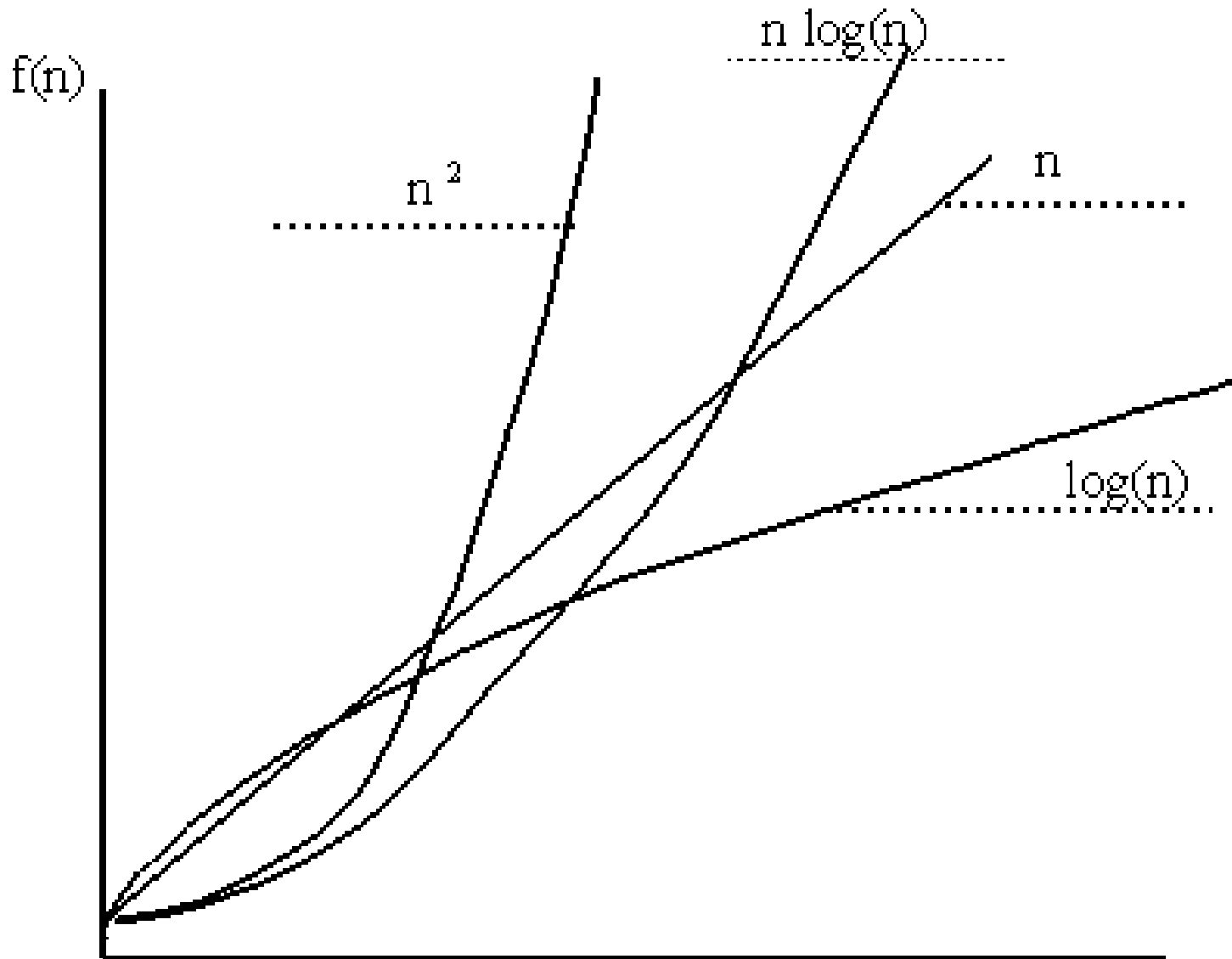
So sánh các thuật toán ntn?

- Độ đo:
 - Thời gian chạy?
 - Không tốt vì nó phụ thuộc vào cấu hình mỗi máy.
 - Số dòng lệnh thực hiện?
 - Không tốt vì nó phụ thuộc vào ngôn ngữ lập trình cũng như phong cách của mỗi lập trình viên.
- Vậy có cách so sánh nào độc lập với máy, phong cách lập trình, ...?
 - *Xem thời gian thực thi của một chương trình là một hàm của kích thước đối số n .*
$$f(n) = \dots$$

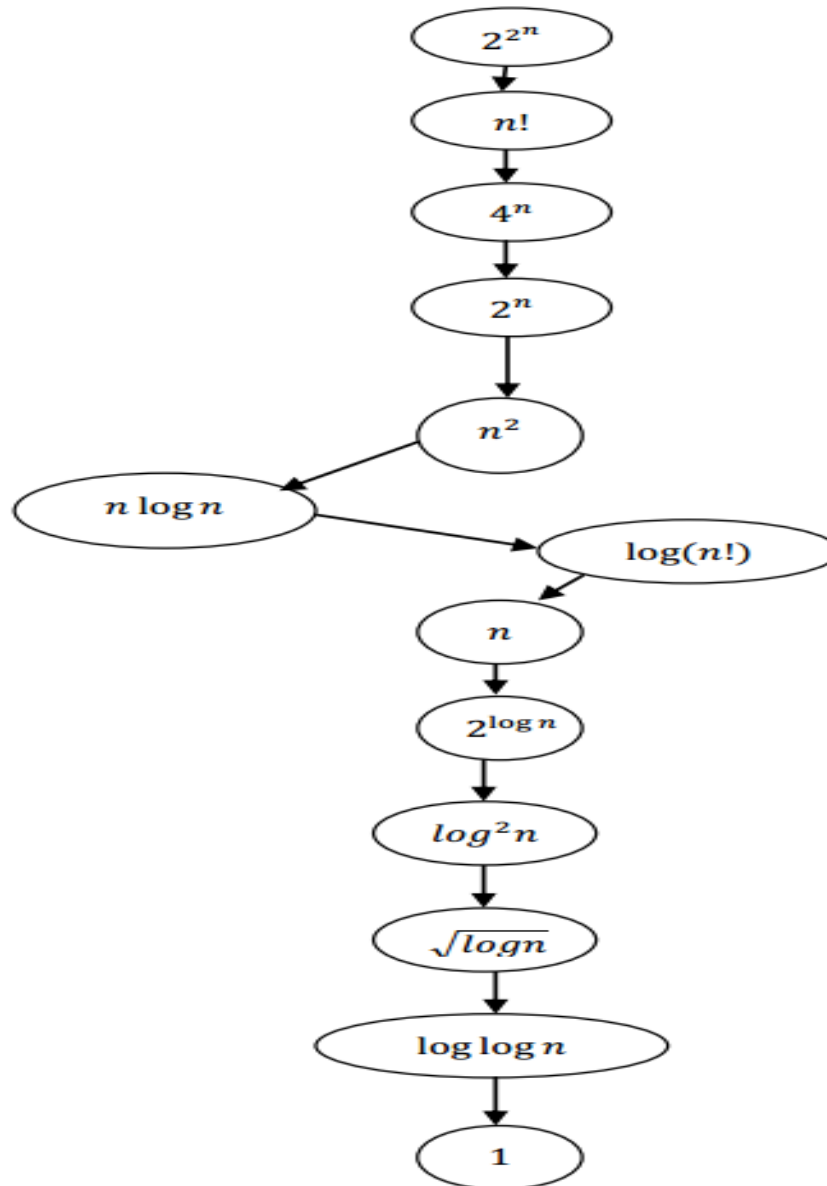
Tỉ lệ tăng

- **Tỉ lệ tăng** (growth rate) là tỉ lệ mà chi phí của thuật toán tăng khi kích thước của đối số tăng.
- Tỉ lệ tăng thường *phụ thuộc vào đối số tăng nhanh nhất* và giả định đối số cần xử lý rất lớn ($n \rightarrow \infty$) :
 - Ví dụ 1: khi thu nhập hàng tháng vài tỷ thì một vài ngàn trở nên số lẻ, không đáng kể.
 - Ví dụ 2: một thuật toán có hàm thời gian thực thi
$$f(n) = n^{100} + n^2 + 10000000 \approx n^{100}$$
- Như vậy khi so sánh thuật toán thường người ta sẽ dựa trên tỉ lệ tăng.

Tỉ lệ tăng (tt)



Một số so sánh tỉ lệ tăng



*Tỉ lệ tăng
giảm*

Loại phân tích

- Không chỉ phụ thuộc vào kích thước input, mà *việc bố trí, cấu trúc, tính chất* của dữ liệu *input*
 - mỗi thuật toán có thể có nhiều biểu thức thời gian.
 - tỉ lệ tăng cũng khác nhau hay thời gian thực thi khác nhau.
- *VD: số phép so sánh cần thực hiện để tìm giá trị 1?*

1	25	6	5	2	37	40
40	25	6	5	2	37	1

Loại phân tích

- Có ba loại phân tích:
 - Trường hợp tốt nhất (best case, Big-Ω):
 - Bố trí dữ liệu input làm cho thuật toán chạy nhanh nhất.
 - Thường không thực tế.
 - Trường hợp trung bình (average case):
 - Bố trí dữ liệu ngẫu nhiên
 - Khó dự đoán được phân phối.
 - Trường hợp xấu nhất (worst case, Big-O):
 - Bố trí dữ liệu input gây cho thuật toán chạy chậm nhất.
 - Đảm bảo tuyệt đối chặn trên.

$$\textit{Lower Bound} \leq \textit{Average Time} \leq \textit{Upper Bound}$$

Big-O

- Để đánh giá trường hợp xấu nhất hay chặn trên (Upper Bound), người ta đưa ra khái niệm **Big-O**.

O

$$\textit{Lower Bound} \leq \textit{Average Time} \leq \textit{Upper Bound}$$

Lịch sử Big-O

- Ký hiệu Big-O được giới thiệu năm 1894 bởi **Paul Bachmann** (Đức) trong cuốn sách *Analytische Zahlentheorie* (“Analytic Number Theory”) (tái bản lần 2)
- Ký hiệu này (sau đó) được phổ biến rộng rãi bởi nhà toán học **Edmund Landau**, nên còn gọi là ký hiệu Landau (Landau notation), hay Bachmann-Landau notation
- **Donald Knuth** là người đưa ký hiệu này vào ngành Khoa học máy tính (Computer Science) năm 1976 – “Big Omicron and big Omega and big Theta” - ACM SIGACT News, Volume 8, Issue 2

Big-O (tt)

- Cho hàm thời gian thực thi của một thuật toán là $f(n)$
 - Ví dụ: $f(n) = n^4 + 100n^2 + 10n + 50$
- Gọi hàm chặn trên của $f(n)$ là $g(n)$ khi n đủ lớn. Nghĩa là không có giá trị n đủ lớn nào làm cho $f(n)$ vượt $cg(n)$ (c : hằng số)
 - Ví dụ: $g(n) = n^4$
- Lúc này, $f(n)$ được gọi là Big-O của $g(n)$. Hay $g(n)$ là tỉ lệ tăng cực đại của $f(n)$

Big-O (tt)

- Định nghĩa:

- Cho $f(n)$ và $g(n)$ là hai hàm số
- Ta nói: $f(n) = O(g(n))$, nếu tồn tại số dương c và n_0 sao cho:

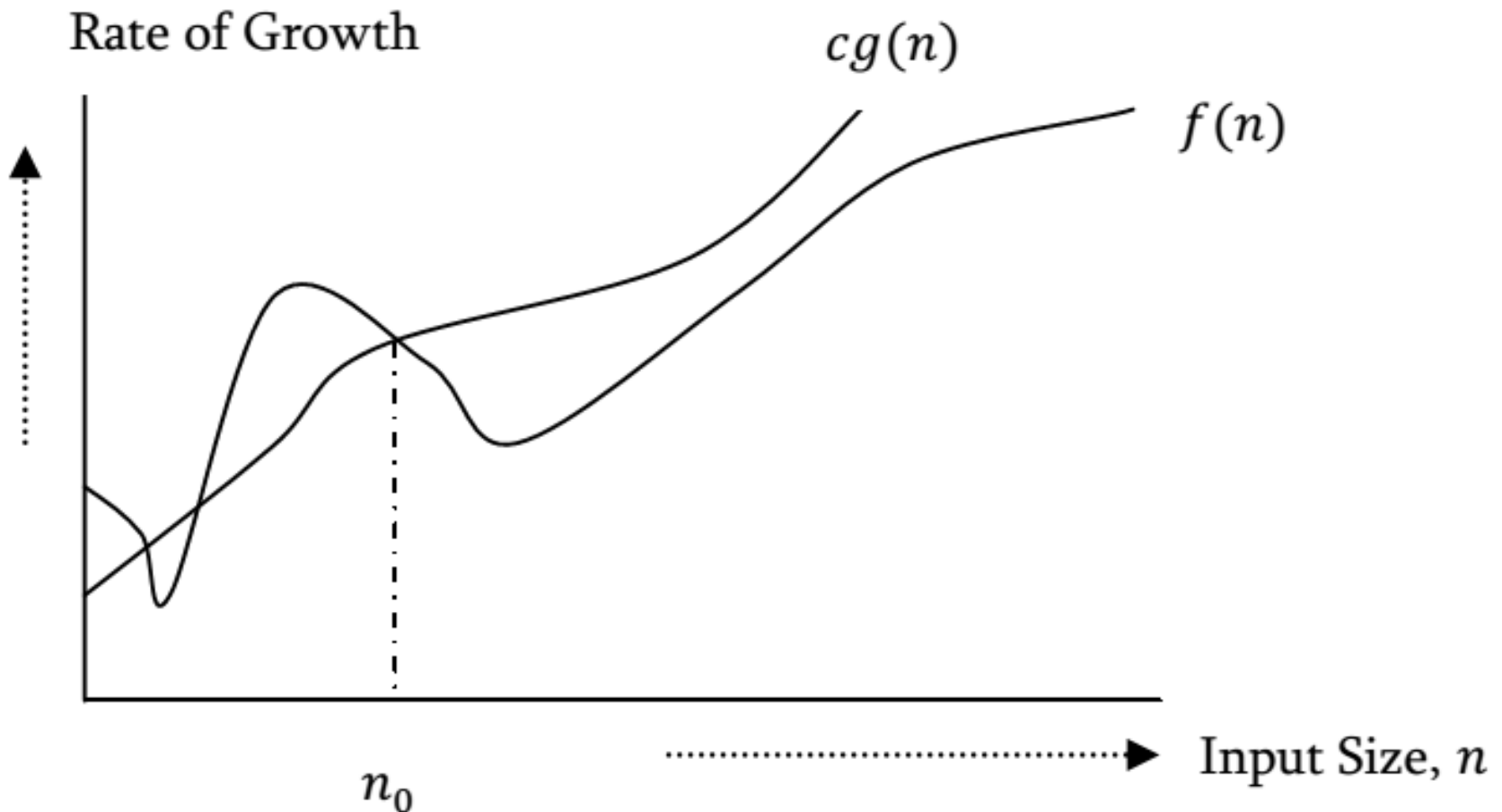
$$0 \leq f(n) \leq cg(n)$$

với $n \geq n_0$

- Phát biểu: f là Big-O của g nếu tồn tại số dương c sao cho f không thể lớn hơn $c \cdot g$ khi n đủ lớn

→ $g(n)$ được gọi là giới hạn trên (upper bound) của $f(n)$

Minh họa Big-O



Đối với n nhỏ ($n < n_0$) ta thường không quan tâm vì nó thường không quan trọng và không thể hiện được độ phức tạp của thuật toán.

Ví dụ Big-O

- Giả sử có hai giải thuật P1, P2:
 - P1: $f_1(n) = 100n^2$
 - P2: $f_2(n) = 5n^3$
- Nhận xét với:
 - $n = 5$: $P1 > P2$?
 - $n = 21$: $P1 > P2$?
- Hàm chặn trên:
 - P1: $g_1(n) = n^2$
 - P2: $g_2(n) = n^3$

Phân tích thuật toán với Big-O

- Nhận xét:

- Mỗi thuật toán có nhiều hàm số biểu diễn thời gian ứng với từng bố trí của input dẫn đến các trường hợp thực thi tốt nhất, trung bình và xấu nhất.
- $g(n)$ là chặn trên của tất cả các hàm đó.

- Như vậy so sánh các thuật toán, ta chỉ cần so sánh $g(n)$ hay Big-O của nó.

- Ví dụ: Thay vì sử dụng độ phức tạp của giải thuật là $2n^2 + 6n + 1$, ta sẽ sử dụng giới hạn (chặn) trên của độ phức tạp của giải thuật là n^2

Tồn tại một hay nhiều hàm $g(n)$?

- Nếu tồn tại $g'(n) > g(n)$, thì $g'(n)$ có đảm bảo điều kiện Big-O không?
- Những lưu ý:
 - $g(n)$ tốt nhất là hàm nhỏ nhất vẫn thỏa Big-O.
 - Càng đơn giản càng tốt (?)

Bài tập Big-O

- Tìm hàm chặn trên cùng với giá trị c và n_0 của các hàm sau:

a) $f(n) = 3n + 8$

b) $f(n) = n^2 + 1$

c) $f(n) = n^4 + 100n^2 + 50$

d) $f(n) = 2n^3 - 2n^2$

e) $f(n) = n$

f) $f(n) = 410$

- Gợi ý đáp án: $(g(n), c, n_0)$

a. $(n, 4, 8)$

b. $(n^2, 2, 1)$

c. $(n^4, 2, 100)$

d. $(n^3, 2, 1)$

e. $(n^2, 1, 1)$

f. $(1, 1, 1)$

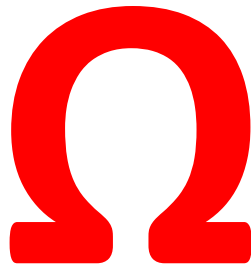
Liệu có đáp án khác không?

Bài tập Big-O

- *Xác định $O(g(n))$ của các hàm sau đây:*
 - $f(n) = 10$
 - $f(n) = 5n + 3$
 - $f(n) = 10n^2 - 3n + 20$
 - $f(n) = \log n + 100$
 - $f(n) = n \log n + \log n + 5$
- *Phát biểu nào là đúng?*
 - $2^{n+1} = O(2^n)$?
 - $2^{2n} = O(2^n)$?

Big-Ω

- *Chặn dưới* (lower bound) hay *trường hợp tốt nhất* được thể hiện bằng Big-Ω.



$$\textit{Lower Bound} \leq \textit{Average Time} \leq \textit{Upper Bound}$$

Big-Ω (tt)

- Cho hàm thời gian thực thi của một thuật toán là $f(n)$
 - Ví dụ: $f(n) = 100n^2 + 10n + 50$
- Gọi hàm chặn dưới của $f(n)$ là $g(n)$ khi n đủ lớn. Nghĩa là không có giá trị n đủ lớn nào làm cho $cg(n)$ vượt $f(n)$ (c : hằng số)
 - Ví dụ: $g(n) = n^2$
- Lúc này, $f(n)$ được gọi là Big-Ω của $g(n)$. Hay $g(n)$ là tỉ lệ tăng cực tiểu của $f(n)$

Big-Ω (tt)

- Định nghĩa:

- Cho $f(n)$ và $g(n)$ là hai hàm số
- Ta nói: $f(n) = \Omega(g(n))$, nếu tồn tại số dương c và n_0 sao cho:

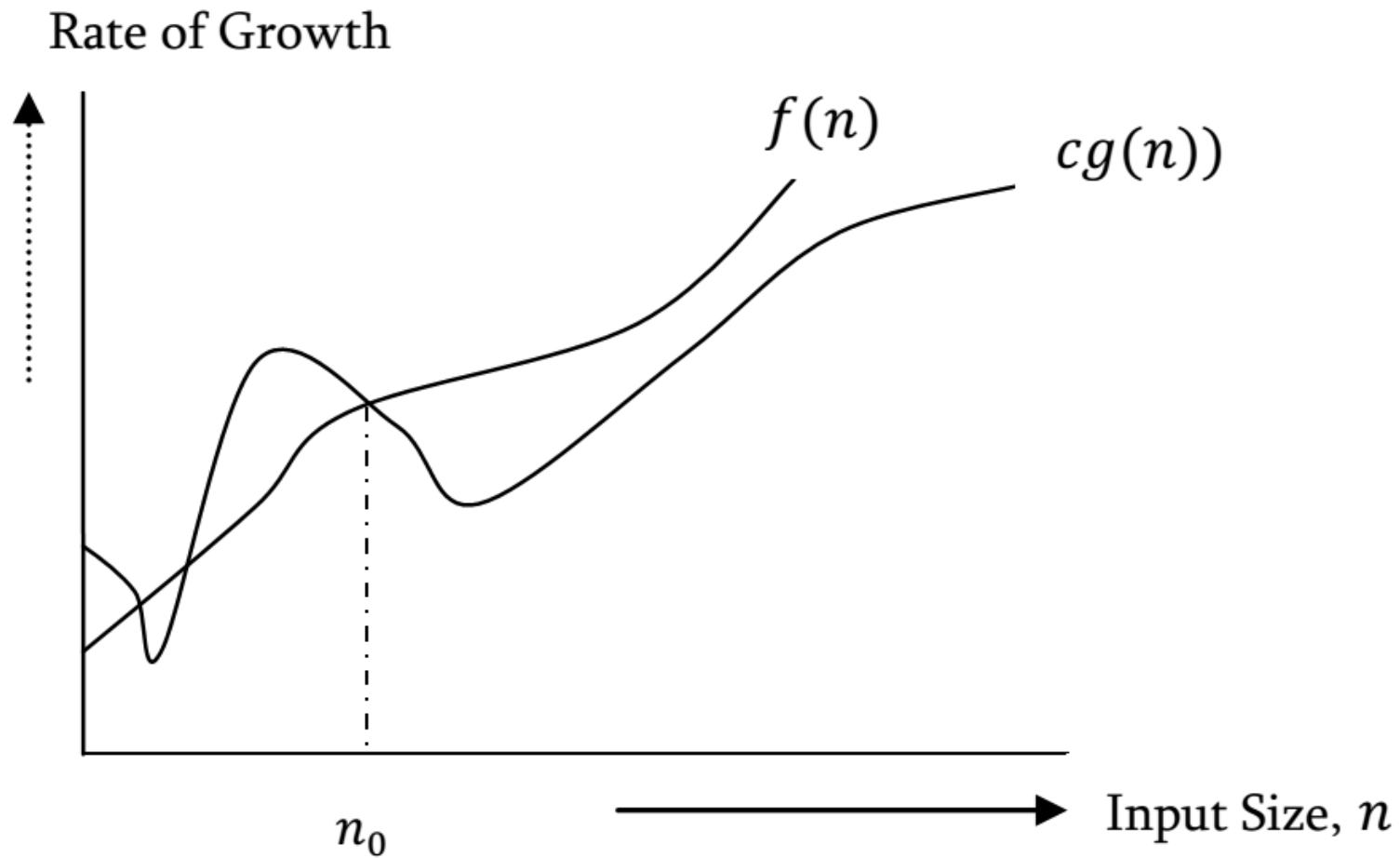
$$0 \leq cg(n) \leq f(n)$$

với $n \geq n_0$

- Phát biểu: f là Big-Ω của g nếu tồn tại số dương c sao cho f không thể nhỏ hơn $c \cdot g$ khi n đủ lớn

→ $g(n)$ được gọi là giới hạn dưới (lower bound) của $f(n)$

Minh họa Big-Ω



Bài tập Big-Ω

- Tìm hàm chặn dưới của các hàm sau:

a) $f(n) = 5n^2$

b) $f(n) = n$

c) $f(n) = n^3$

- Gợi ý đáp án:

a. $\Omega(n)$

b. $\Omega(\log n)$

c. $\Omega(n^2)$

Big-Θ

- **Big-Θ** được sử dụng để xác định xem *giới hạn dưới* và *giới hạn trên* giống nhau không.

$$\text{Lower Bound } (\Omega) \leq \text{Average Time} \leq \text{Upper Bound } (O)$$



- Nếu Ω và Θ giống nhau, thì **tỉ lệ tăng** (growth rate) của trường hợp trung bình cũng tương tự như vậy.

Big- Θ (tt)

- Định nghĩa:

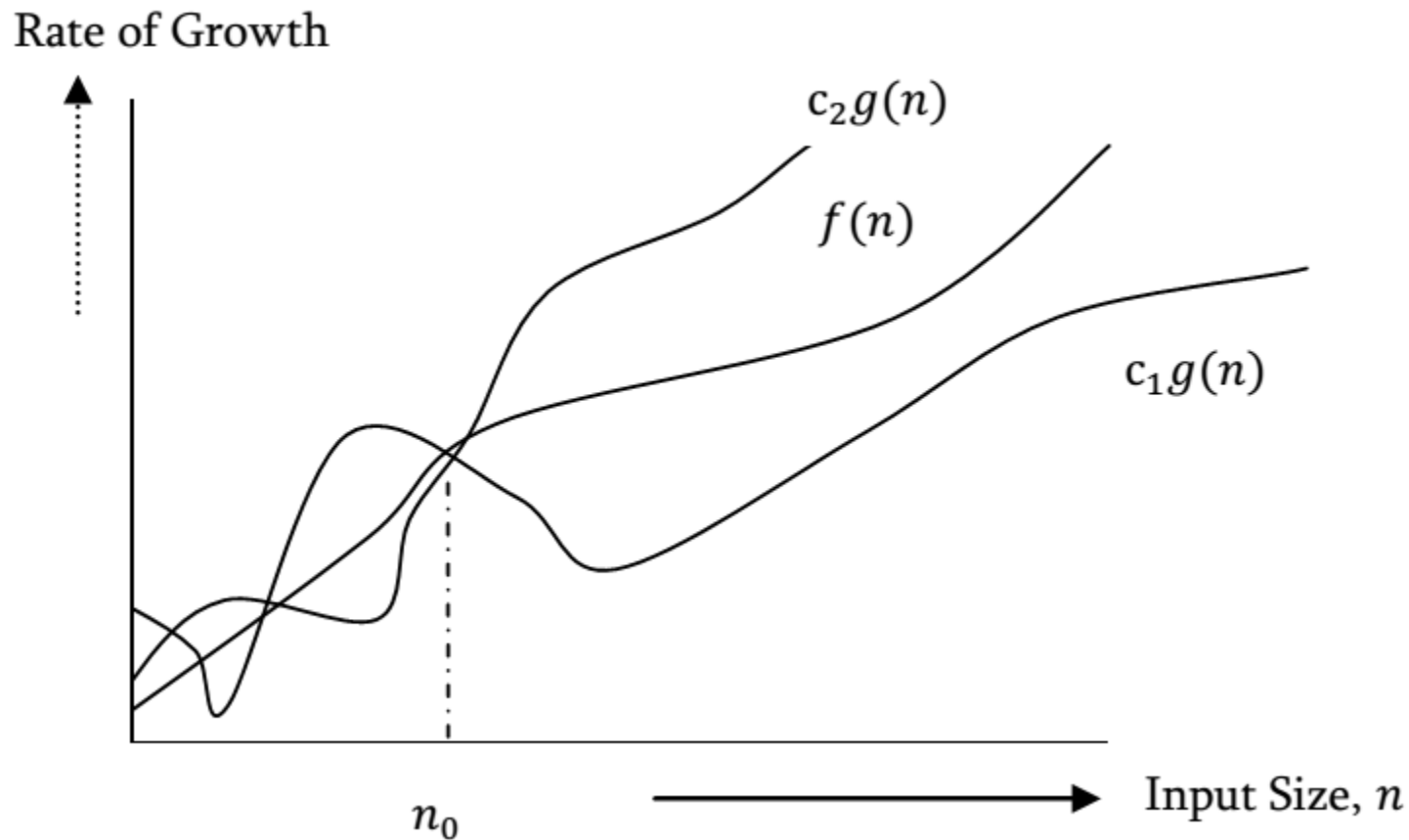
- Cho $f(n)$ và $g(n)$ là hai hàm số
- Ta nói: $f(n) = \Theta(g(n))$, nếu tồn tại số dương c_1 , c_2 và n_0 sao cho:

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

với $n \geq n_0$

→ $g(n)$ được gọi là *giới hạn chặt* (tight bound) của $f(n)$

Minh họa Big-Θ



Bài tập Big- Θ

1) Tìm hàm giới hạn chặt của hàm sau:

$$f(n) = \frac{n^2}{2} - \frac{n}{2}$$

2) Chứng minh:

a. $n \neq \theta(n^2)$

b. $6n^3 \neq \theta(n^2)$

c. $n \neq \theta(\log n)$

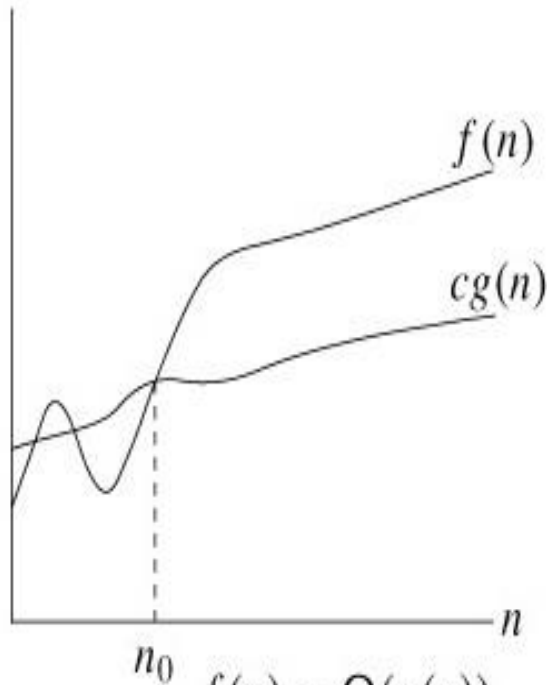
• Gợi ý đáp án:

1) $\theta(n^2), c_1 = \frac{1}{5}, c_2 = 1, n_0 = 1$

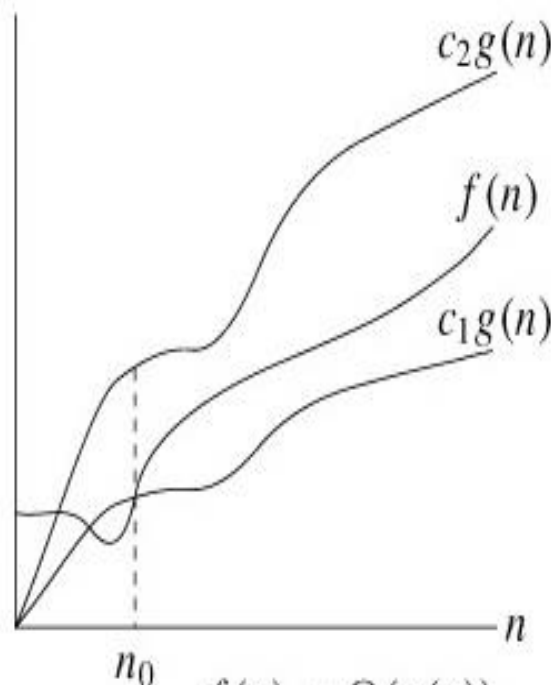
2)

a. $n \leq \frac{1}{c_1}$ (không thể) ...

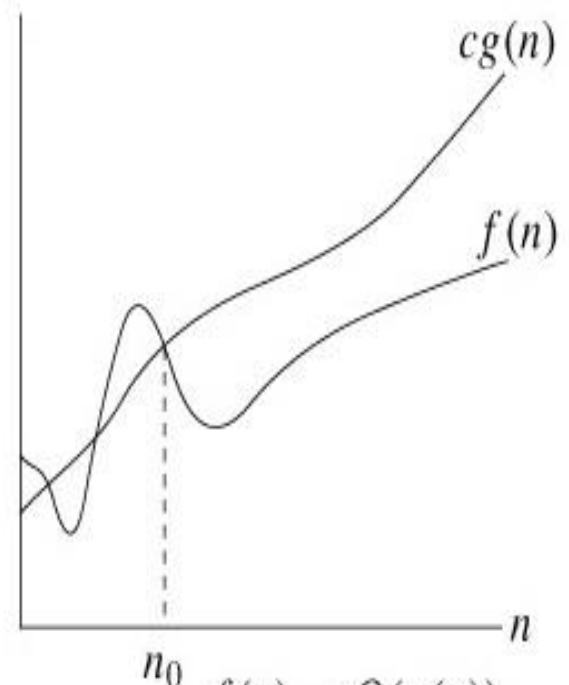
Tóm tắt Big-O, Big-Ω, Big-Θ



$f(n) = \Omega(g(n))$
(a)



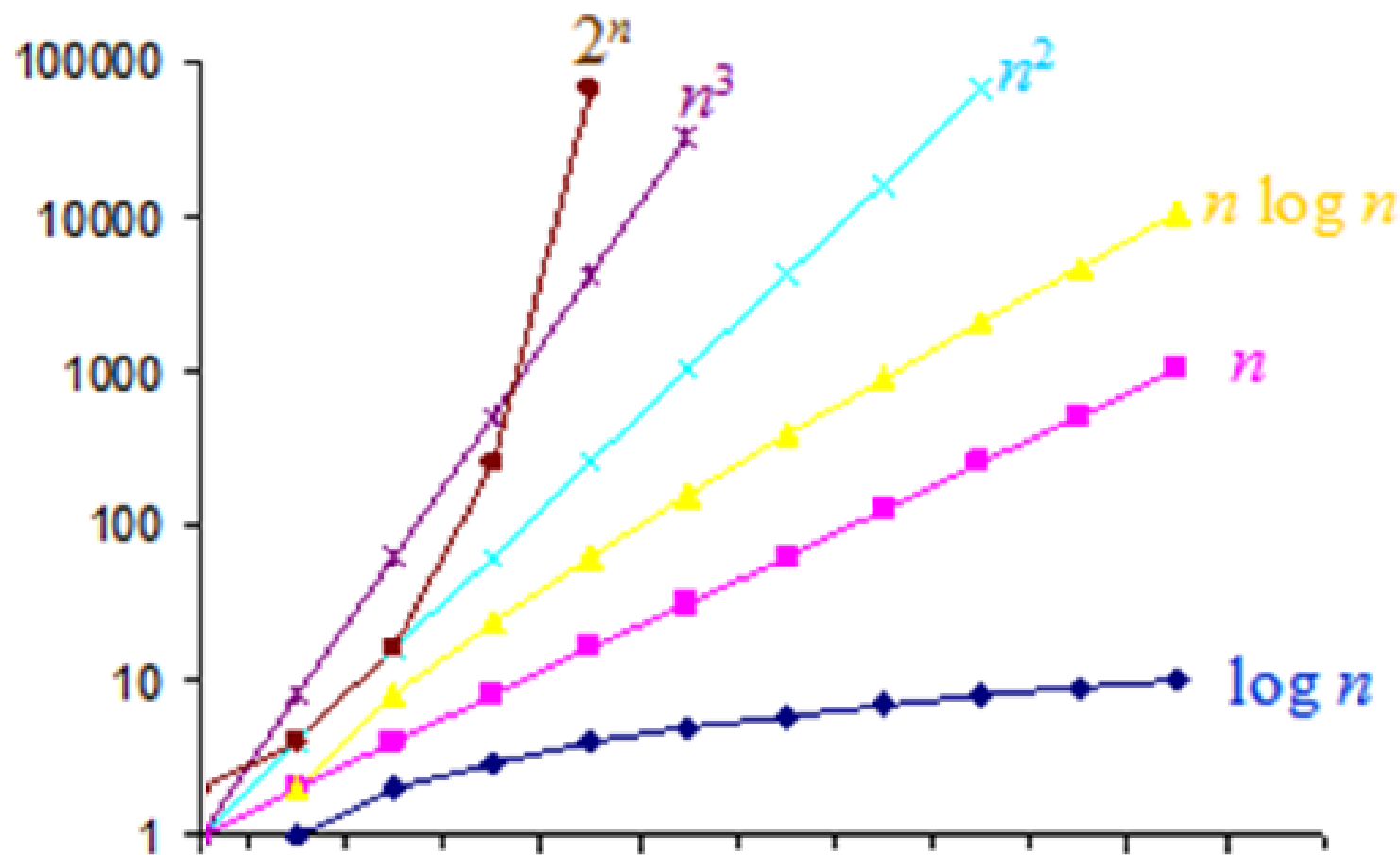
$f(n) = \Theta(g(n))$
(b)

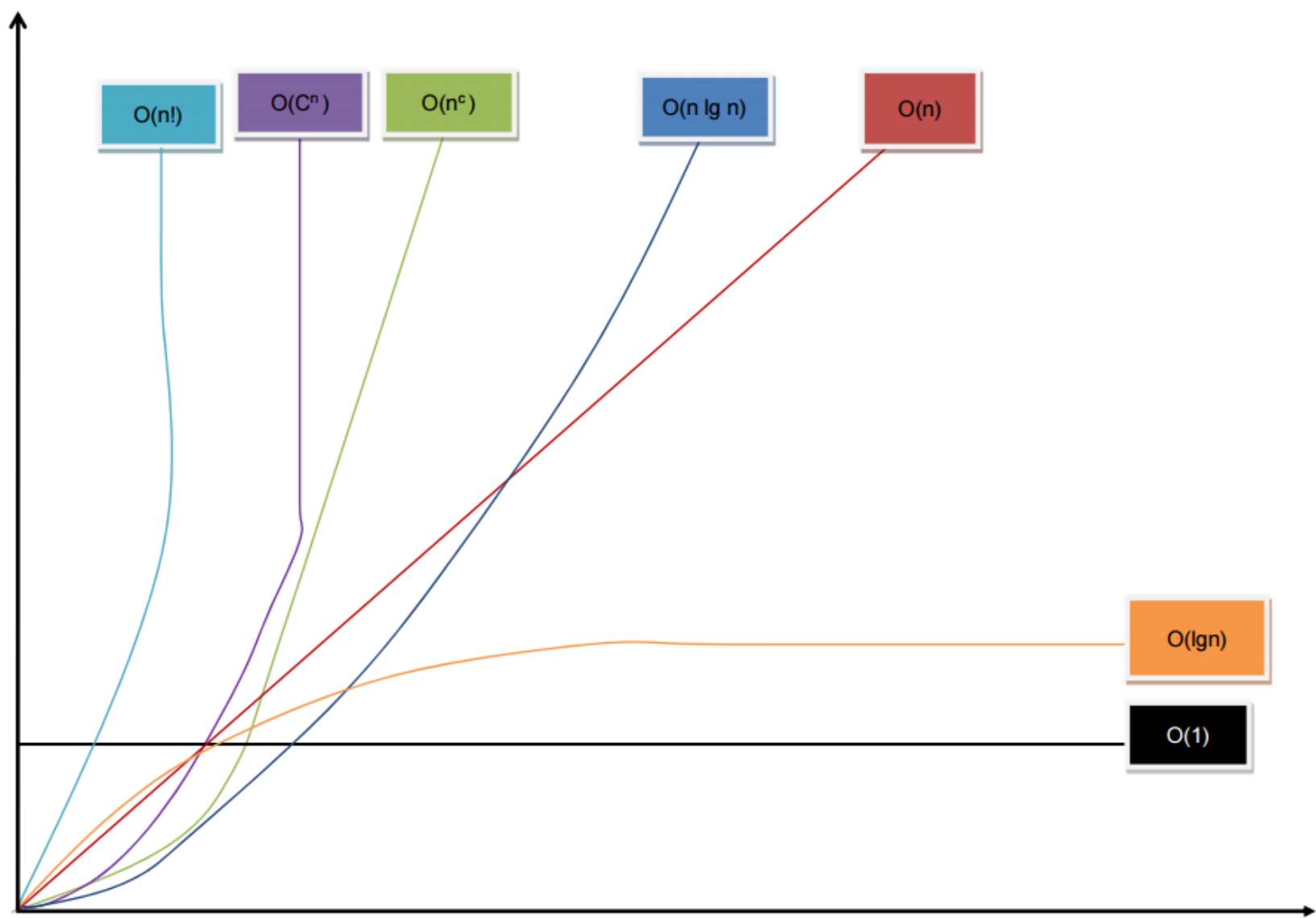


$f(n) = O(g(n))$
(c)

Nhìn chung, thực tế người ta chỉ quan tâm đến giới hạn trên (Big-O), còn giới hạn dưới (Big-Ω) không quan trọng, giới hạn chặt (Big-Θ) chỉ xem xét khi giới hạn trên và dưới giống nhau.

So sánh một số hàm số





Bài tập tổng hợp Big-O, Big-Ω, Big-Θ

- Cho các cặp hàm sau, xác định các mối quan hệ giữa chúng và giải thích. Biết rằng, các quan hệ có thể có là $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$ hoặc $f(n) = \Theta(g(n))$
 - a) $f(n) = \log n^2$; $g(n) = \log n + 5$
 - b) $f(n) = \sqrt{n}$; $g(n) = \log n^2$

Một số độ phức tạp của TT

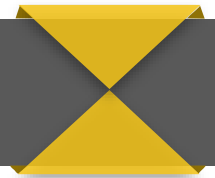
- Các thuật toán phổ biến

Algorithm	Worst case	Typical case
Simple greedy	$O(n)$	$O(n)$
Sorting	$O(n^2)$	$O(n \lg n)$
Shortest paths	$O(2^n)$	$O(n)$
Linear programming	$O(2^n)$	$O(n)$
Dynamic programming	$O(2^n)$	$O(2^n)$
Branch-and-bound	$O(2^n)$	$O(2^n)$

Thời gian chạy

n	$O(n)$	$O(n \lg n)$	$O(n^2)$	$O(n^3)$	$O(n^{10})$	$O(2^n)$
10	.01 μ s	.03 μ s	.10 μ s	1 μ s	10 s	1 μ s
50	.05 μ s	.28 μ s	2.5 μ s	125 μ s	3.1 y	13 d
100	.10 μ s	.66 μ s	10 μ s	1 ms	3171 y	10^{13} y
1,000	1 μ s	10 μ s	1 ms	1 s	10^{13} y	10^{283} y
10,000	10 μ s	130 μ s	100 ms	16.7 min	10^{23} y	
100,000	100 μ s	1.7 ms	10 s	11.6 d	10^{33} y	
1,000,000	1 ms	20 ms	16.7 min	31.7 y	10^{43} y	

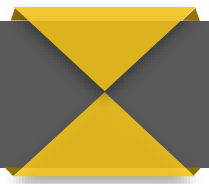
Phương Pháp Tính Độ Phức Tạp



Coding example #1

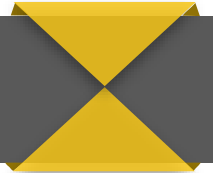
```
for ( i=0 ; i<n ; i++ )  
    m += i;
```


Coding example #2



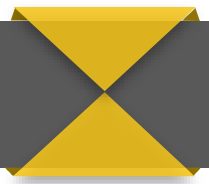
```
for ( i=0 ; i<n ; i++ )  
    for( j=0 ; j<n ; j++ )  
        sum[i] += entry[i][j];
```

Coding example #3



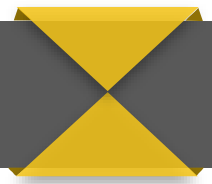
```
for ( i=0 ; i<n ; i++ )  
    for( j=0 ; j<i ; j++ )  
        m += j;
```

Coding example #4



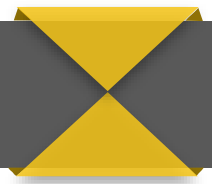
```
i = 1;
while (i < n) {
    tot += i;
    i = i * 2;
}
```

Example #4: equivalent # of steps?



```
i = n;  
while (i > 0) {  
    tot += i;  
    i = i / 2;  
}
```

Coding example #5



```
for ( i=0 ; i<n ; i++ )  
    for( j=0 ; j<n ; j++ )  
        for( k=0 ; k<n ; k++ )  
            sum[i][j] += entry[i][j][k];
```

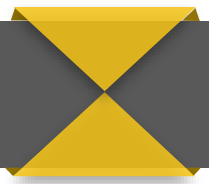
Coding example #6



```
for ( i=0 ; i<n ; i++ )  
    for( j=0 ; j<n ; j++ )  
        sum[i] += entry[i][j][0];
```

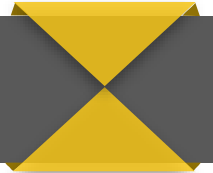
```
for ( i=0 ; i<n ; i++ )  
    for( k=0 ; k<n ; k++ )  
        sum[i] += entry[i][0][k];
```

Coding example #7



```
for ( i=0 ; i<n ; i++ )  
    for( j=0 ; j< sqrt(n) ; j++ )  
        m += j;
```

Coding example #8

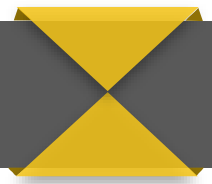


```
for ( i=0 ; i<n ; i++ )  
    for( j=0 ; j< sqrt(995) ; j++ )  
        m += j;
```


Coding example #8 : Equivalent code

```
for ( i=0 ; i<n ; i++ )  
{  
    m += j ;  
    m += j ;  
    m += j ;  
    ...  
    m += j ;    // 31 times  
}
```

Coding example #9

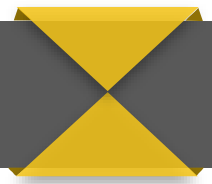


```
int total(int n)
    for( i=0 ; i < n; i++)
        subtotal += i;

main()
    for ( i=0 ; i<n ; i++ )
        tot += total(i);
```

Coding example #9: Equivalent code

```
for ( i=0 ; i<n ; i++ ) {  
    subtotal = 0;  
    for( j=0 ; j < i; j++)  
        subtotal += j;  
    tot += subtotal;  
}
```



// Input: int A[N], array of N integers
// Output: Sum of all numbers in array A

```
int Sum(int A[], int N)
{
    int s=0;
    for (int i=0; i< N; i++)
        s = s + A[i];
    return s;
}
```

VD

// Input: int A[N], array of N integers
// Output: Sum of all numbers in array A

```
int Sum(int A[], int N){
```

```
    int s=0; ← ①
```

```
    for (int i=0; i< N; i++) ← ②, ③, ④
```

```
        s = s + A[i]; ← ⑤, ⑥, ⑦
```

```
    return s; ← ⑧
```

```
}
```

1,2,8: Once

3,4,5,6,7: Once per each iteration
of for loop, N iteration

Total: $5N + 3$

The *complexity function* of the
algorithm is : $f(N) = 5N + 3$

Estimated running time for different values of N:

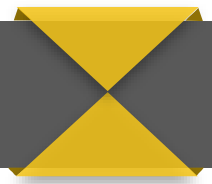
$N = 10 \Rightarrow 53$ steps

$N = 100 \Rightarrow 503$ steps

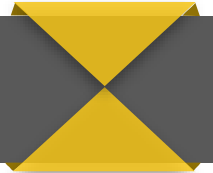
$N = 1,000 \Rightarrow 5003$ steps

$N = 1,000,000 \Rightarrow 5,000,003$ steps

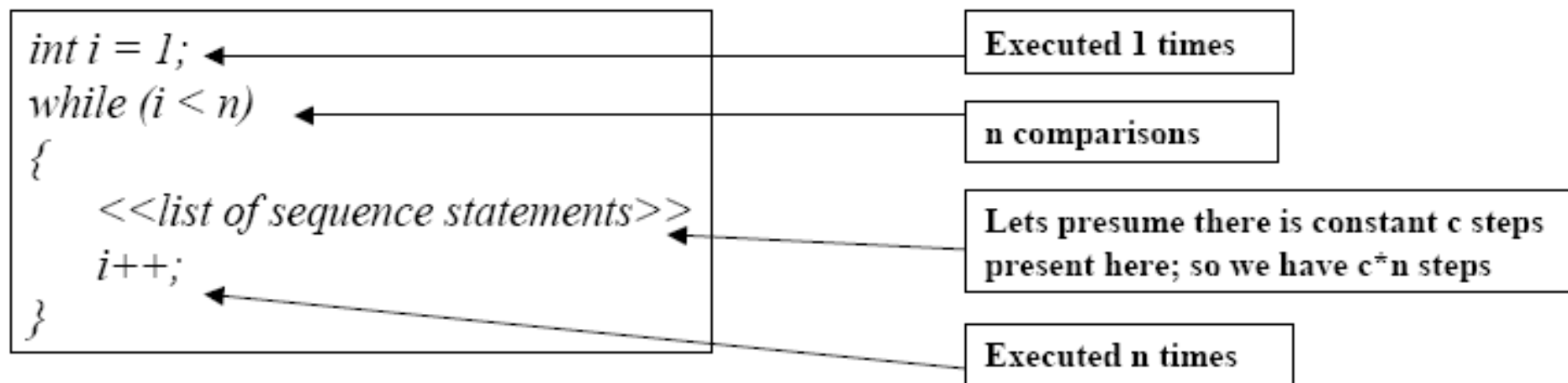
As N grows, the number of steps grow in *linear* proportion to N for this function “*Sum*”




```
int sum = 0,j;  
  for (j=0; j < N; j++)  
    sum = sum +j;
```

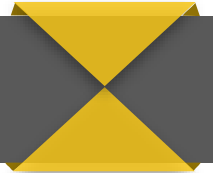


```
int sum =0, j;  
  for (j=0; j < 100; j++)  
    sum = sum +j;
```



```
int j,k;  
  for (j=0; j<N; j++)  
    for (k=N; k>0; k--)  
      sum += k+j;
```




```
int j,k;  
    for (j=0; j < N; j++)  
        for (k=0; k < j; k++)  
            sum += k+j;
```

PP tính độ phức tạp

```
bool Search (int a[], int n, int x)
{
    #1  int i = 0; O(1)
    #2  bool flag = false; O(1)
    #3  while (i < n && flag == false) O(1)
        {
            #4      if (a[i] == x) O(1)
                    flag = true; O(1)
                    else
                        i ++; O(1)
        }
    #5  return flag; O(1)
}
```


PP tính độ phức tạp

- Lệnh {1}, {2}, {3} và {5} nối tiếp nhau, do đó độ phức tạp của hàm Search chính là độ phức tạp lớn nhất trong 4 lệnh này.
- Lệnh {1}, {2} và {5} : $O(1) \rightarrow$ độ phức tạp của hàm Search chính là độ phức tạp của lệnh {3}.
- Vòng trong lệnh {3} là lệnh {4}. Lệnh {4} có độ phức tạp $O(1)$.
- Lệnh {3}: vòng lặp không xác định, nhưng worst case (tất cả các phần tử của mảng a đều khác x, ta phải xét hết tất cả các $a[i]$, i có các giá trị từ 1 đến n) thì vòng lặp {3} thực hiện n lần, do đó lệnh {3} tốn $O(n)$. Vậy ta có $T(n) = O(n)$.



```
for (j=0; j < N; j++)  
    for (k =0; k < j; k++)  
        sum = sum + j*k;  
for (l=0; l < N; l++)  
    sum = sum -l;  
System.out.print("sum is now"+sum);
```

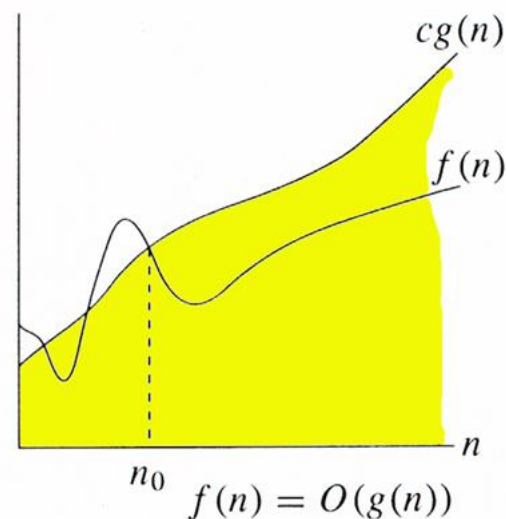
} $O(N^2)$
}
}
} $O(1)$



```
double power( double x, int n) {  
    if ( n == 0)  
        return 1.0;                // base case  
    //else  
        return power(x, n-1)*x;    // recursive case  
}
```

Phân tích tiệm cận

- Với một thuật toán $f(n)$, chúng ta luôn muốn tìm một hàm khác $g(n)$ mà xấp xỉ $f(n)$ có giá trị cao hơn.
- Đường cong biểu diễn $g(n)$ gọi là **đường cong tiệm cận** (asymptonic curve).
- Bài toán tìm $g(n)$ gọi là **phân tích tiệm cận** (asymptotic analysis)




Cách phân tích tiệm cận – Vòng lặp

*Thời gian chạy của **vòng lặp** thường là thời gian thực thi của các câu lệnh bên trong vòng lặp nhân với số lần lặp.*

Ví dụ:

```
for (i=1; i<=n; i++)  
{  
    m = m+2;           //thời gian hằng c  
}
```

Thời gian chạy $= c \times n = cn = O(n)$


$$\sum_{i=1}^{X(n)} (T_0(n) + T_i(n))$$

$X(n)$: Số vòng lặp

$T_0(n)$: Điều kiện lặp

$T_i(n)$: Thời gian thực hiện vòng lặp thứ i

Vòng lặp lồng nhau

Đối với vòng lặp lồng, ta phân tích từ trong ra ngoài. Tổng thời gian là tích của kích thước tất cả vòng lặp.

Ví dụ:

```
for (i=1; i<=n; i++)  
{  
    for (j=1; j<=n; j++)  
    {  
        k = k+1;           //thời gian hằng c  
    }  
}
```

Thời gian chạy $= c \times n \times n = cn^2 = O(n^2)$

Câu lệnh tuần tự

Đối với các *lệnh tuần tự nhau*, để tính thời gian chạy, ta cộng dồn độ phức tạp thời gian ở mỗi lệnh.

Ví dụ:

```
x = x + 1; //thời gian hằng  $c_0$ 
for (i=1; i<=n; i++)
{
    m = m + 2; //thời gian hằng  $c_1$ 
}
for (i=1; i<=n; i++)
{
    for (j=1; j<=n; j++)
    {
        k = k+1;           //thời gian hằng  $c_2$ 
    }
}
```

Thời gian chạy $= c_0 + c_1n + c_2n^2 = O(n^2)$

Câu lệnh rẽ nhánh

Thường ta xét trường hợp xấu nhất, thời gian chạy bằng phần chạy thời gian lâu nhất của lệnh rẽ nhánh.

if (điều kiện) $\rightarrow T_0(n)$

 lệnh 1 $\rightarrow T_1(n)$

else

 lệnh 2 $\rightarrow T_2(n)$

Thời gian: $T_0(n) + \max (T_1(n), T_2(n))$

Câu lệnh rẽ nhánh (tt)

Ví dụ:

```
//test: constant  $c_0$ 
if (length() != otherStack.length())
{
    return false; //then part: constant  $c_1$ 
}
else
{
    // else part: (constant + constant) * n
    for (int n = 0; n < length(); n++)
    {
        if (!list[n].equals(otherStack.list[n]))
            //constant  $c_2$ 
            return false; //  $c_3$ 
    }
}
```

Thời gian chạy $= c_0 + c_1n + (c_2 + c_3)n = O(n)$

Quy tắc cộng và quy tắc nhân

- **Quy tắc cộng:**

Nếu $f_1 = O(g_1)$ và $f_2 = O(g_2)$ thì $f_1 + f_2 = O(\max\{g_1, g_2\})$

Nếu ta thực hiện một số thao tác theo thứ tự, thời gian thực thi bị chi phối bởi thao tác tốn nhiều thời gian nhất.

- **Quy tắc nhân:**

Nếu $f_1 = O(g_1)$ và $f_2 = O(g_2)$ thì $f_1 * f_2 = O(g_1 * g_2)$

Nếu ta lặp lại một thao tác một số lần, tổng thời gian thực thi là thời gian thực thi của thao tác nhân với số vòng lặp

Một số tính chất

- *Tính bắc cầu:*

$$f(n) = O(g(n)) \text{ và } g(n) = O(h(n))$$

$$\rightarrow f(n) = O(h(n))$$

- *Tính phản xạ:*

$$f(n) = O(f(n))$$

Tóm tắt tính độ phức tạp cơ bản

- Phát biểu đơn giản (đọc, ghi, gán)
 - $O(1)$
- Các phép tính đơn giản (+ - * / == > >= < <=)
 - $O(1)$
- Chuỗi các phát biểu/phép tính đơn giản
 - Quy tắc cộng
- Vòng lặp for, do, while
 - Quy tắc nhân

Vậy trường hợp gọi hàm đồng cấp?

Hàm A gọi Hàm B

Hàm B gọi Hàm C

...

- Độ phức tạp của chuỗi thao tác gọi hàm đồng cấp?

$$O(n) = \max\{O_A(n), O_B(n), O_C(n) \dots\}$$

Bài tập tính Big-O

- Tính chi phí giới hạn trên (big-O) cho các giải thuật sau:

VD1: for (i = 0; i < n; i++)
 for (j = 0; j < n; j++)
 b[i][j] += c;

VD2: for (i = 0; i < n; i++)
 if (a[i] == k) return 1;
 return 0;

VD3: for (i = 0; i < n; i++)
 for (j = i+1; j < n; j++)
 b[i][j] -= c;

Bài tập tính Big-O (tt)

- VD 4,5,6: cộng, nhân và chuyển vị ma trận

```
for(i = 0; i < n; i++)  
    for(j = 0; j < n; j++)  
        a[i][j] = b[i][j] + c[i][j];
```

```
for(i = 0; i < n; i++)  
    for(j = 0; j < n; j++)  
        for(k = a[i][j] = 0; k < n; k++)  
            a[i][j] += b[i][k] * c[k][j];
```

```
for(i = 0; i < n - 1; i++)  
    for(j = i+1; j < n; j++) {  
        tmp = a[i][j];  
        a[i][j] = a[j][i];  
        a[j][i] = tmp;  
    }
```

Độ phức tạp log

- Một thuật toán có *độ phức tạp $O(\log n)$* nếu nó mất thời gian hằng số để giảm kích thước bài toán bởi một phân số (thường là $\frac{1}{2}$).
- Ví dụ:

```
for (i=1; i<=n; )  
  
    {  
  
        i=i*2;  
  
    }
```

Giả sử vòng lặp thực hiện được k lần.

Một số bài tập thêm

```
s = 0;
for (i=0; i<=n;i++){
    p =1;
    for (j=1;j<=i;j++)
        p=p * x / j;
    s = s+p;
}
```

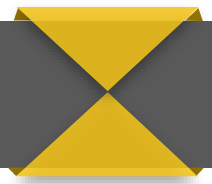
Bt (tt)

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

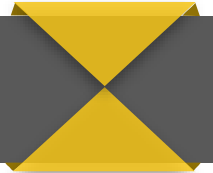
Bt (tt)

- $s = n * (n - 1) / 2;$

Bt (tt)

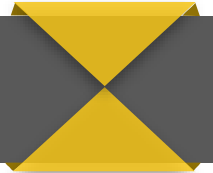


```
for (i= 1;i<=n;i++)  
    for (j= 1;j<=m;j++)  
        x += 2;
```

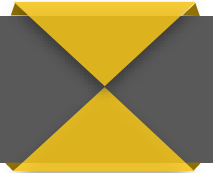


```
(1)  for (i = 0 ; i < n ; i++)  
(2)      for (j = 0 ; j < n ; j++)  
(3)          A[i][j] = 0;
```

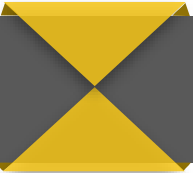
```
(4)  for (i = 0 ; i < n ; i++)  
(5)      A[i][i] = 1;
```



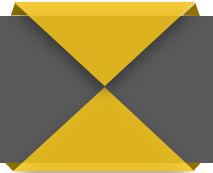
```
for (i = 0 ; i < n ; i++)  
    for (j = 0 ; j < n ; j++)  
        if (i == j)  
            A[i][j] = 1;  
        Else  
            A[i][j] = 0;
```



```
sum = 0;
for ( i = 0; i < n; i ++ )
    for ( j = i + 1; j <= n; j ++ )
        for ( k = 1; k < 10; k ++ )
            sum = sum + i * j * k ;
```



```
sum = 0;
for ( i = 0; i < n; i ++ )
    for ( j = i + 1; j <= n; j ++ )
        for ( k = 1; k < m; k ++ ) {
            x = 2*y;
            sum = sum + i * j * k ;
        }
```



```
sum = 0;
for ( i = 0; i < n; i ++ )
    for ( j = i + 1; j <= n; j ++ )
        for ( k = 1; k < m; k ++ ) {
            x = 2*y;
            sum = sum + i * j * k ;
        }
```

Bt (tt)

```
for (i= 1;i<=n;i++)  
    u = u + 2;  
for (j= 1;j<=m;j++)  
    u = u * 2;
```

```
for (i= 1;i<=n;i++) {  
    for (u= 1;u<=m;u++)  
        for (v= 1;v<=n;v++)  
            h = x*100;  
    for j:= 1 to x do  
        for k:= 1 to z do  
            x = h/100;  
}
```



```
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  
    }
```

Bt (tt)

```
for (i= 1;i<=n;i++)  
    for (u= 1;u<= m;u++)  
        for (v= 1;v<=n;v++)  
            //lệnh  
  
for (j= 1;j<=x;j++)  
    for (k= 1;k<=z;k++)  
        //lệnh
```

Bt (tt)

$$P(x) = x^m + a_{m-1}x^{m-1} + \dots + a_1x + a_0$$

$$Q(x) = b_nx^n + b_{n-1}x^{n-1} + \dots + b_1x + b_0$$

if (m < n) p = m; else p = n;

for (i=0; i <= p; i++)

 c[i] = a[i] + b[i];

if (p < m)

 for (i=p+1; i <= m; i++) c[i] = a[i];

else

 for (i=p+1; i <= n; i++) c[i] = b[i];

while (p > 0 && c[p] == 0) p = p-1;

Bt (tt)

$$P(x) = x^m + a_{m-1}x^{m-1} + \dots + a_1x + a_0$$

$$Q(x) = b_nx^n + b_{n-1}x^{n-1} + \dots + b_1x + b_0$$

$$p = m + n;$$

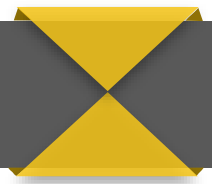
```
for (i=0;i<=p;i++) c[i] = 0;
```

```
for (i=0;i<=m;i++)
```

```
    for (j=0;j<=n;j++)
```

```
        c[i+j] = c[i+j] + a[i] + b[j];
```

Bt (tt)



Tìm hiểu nâng cao

- Phương pháp chia để trị
- Độ phức tạp của đệ qui
- ...

Kết luận

- Mặc dù việc phân tích độ phức tạp thuật toán cho ra sự so sánh chính xác nhất giữa các thuật toán nhưng về thực tế nó có thể khiến cho lập trình viên khó khăn hoặc không thể tìm được Big-O đối với một số chương trình có độ phức tạp cao.
- Do đó, người ta thường quay lại với phương pháp chạy thực nghiệm (đo thời gian chạy) với cùng một nền tảng phần cứng, ngôn ngữ, bộ dữ liệu. Đồng thời cũng thử thay đổi kích thước cũng như loại bộ dữ liệu khác nhau.

A large, stylized yellow 'X' shape is centered on a dark gray background. The 'X' is composed of two overlapping triangles, with a slight 3D effect suggested by a darker yellow shadow on the right side of each triangle. The text 'The End.' is written in a white, sans-serif font, centered within the intersection of the 'X'.

The End.