HOA SEN UNIVERSITY

www.hoasen.edu.vn

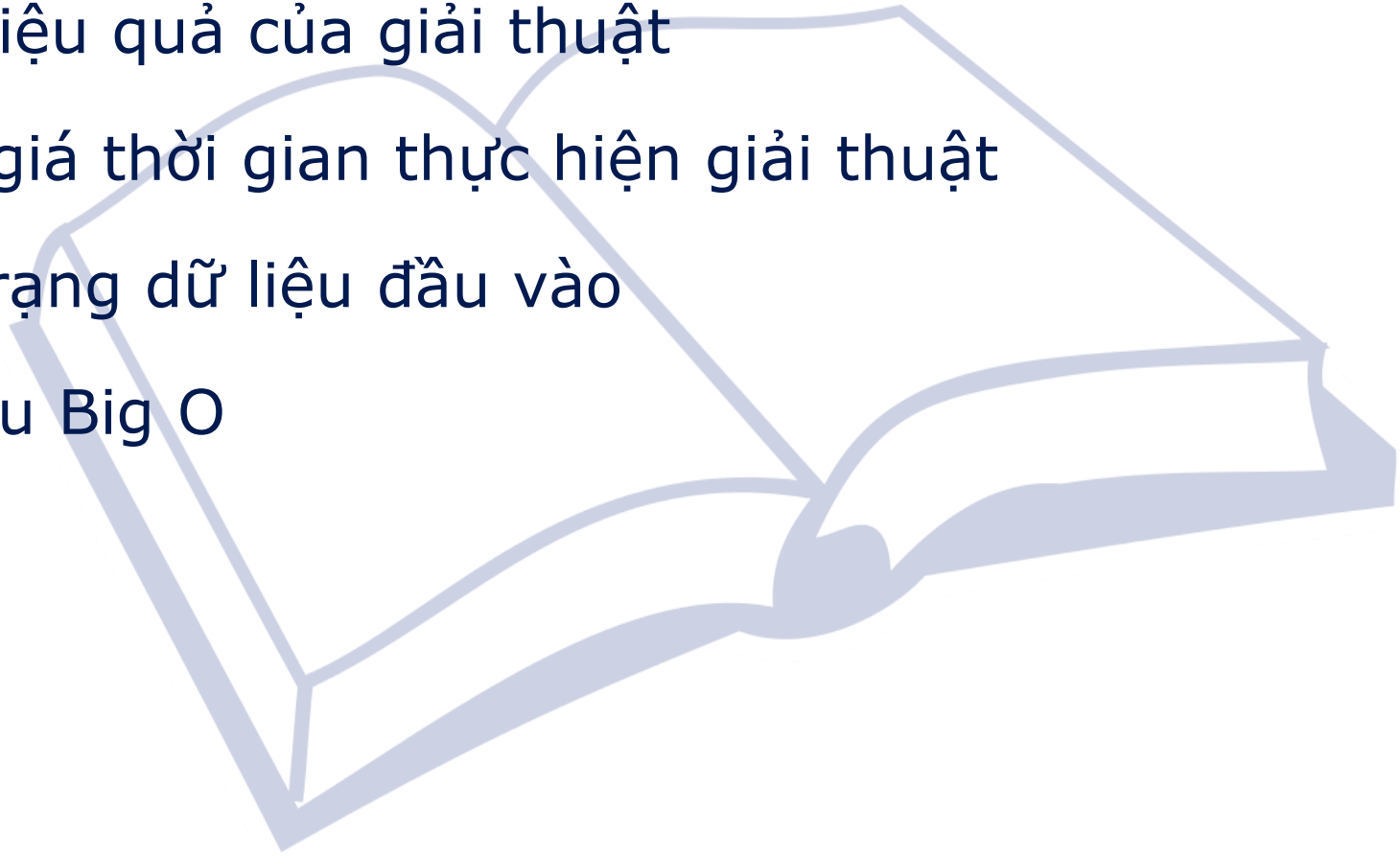
Cấu Trúc Dữ Liệu & Giải Thuật

Data Structures & Algorithms

GV: Phan Hồng Trung

Bài 3 – Độ phức tạp của giải thuật

- Tính hiệu quả của giải thuật
- Đánh giá thời gian thực hiện giải thuật
- Tình trạng dữ liệu đầu vào
- Ký hiệu Big O



Tính hiệu quả của giải thuật

- Để giải một bài toán, có thể có nhiều cách khác nhau → cần phải lựa chọn cách "tốt nhất" theo một nghĩa nào đó.
- Có 2 tiêu chuẩn để đánh giá một giải thuật "tốt":
  - TC1: Đơn giản, dễ hiểu, dễ lập trình.
  - TC2: Cho lời giải nhanh, sử dụng ít tài nguyên máy tính.
- Thường không thể đảm bảo đồng thời cả 2 tiêu chuẩn trên → phải tùy theo trường hợp mà áp dụng TC1 hay TC2:
  - Nếu chỉ dùng giải thuật cho một vài lần thì TC1 quan trọng hơn TC2.
  - Nếu đây là một bài toán rất phổ biến, giải thuật được dùng nhiều lần thì TC2 quan trọng hơn TC1.

Tính hiệu quả của giải thuật

- Mục đích của việc nghiên cứu CTDL&GT là xây dựng các chương trình "hiệu quả" → TC2 chính là tính "hiệu quả" của giải thuật.
- Một giải thuật được gọi là "hiệu quả" nếu nó tiết kiệm được không gian và thời gian:
  - Tiết kiệm không gian là chiếm dụng ít bộ nhớ trong thời gian thực hiện.
  - Tiết kiệm thời gian là chạy nhanh.
- Tiêu chuẩn thời gian thực hiện nhanh là quan trọng hàng đầu → đánh giá độ phức tạp của giải thuật là đánh giá thời gian thực hiện nó.

Đánh giá thời gian thực hiện giải thuật

- **Tính độc lập:** cần đánh giá thời gian thực hiện sao cho:
  - Không phụ thuộc máy, ngôn ngữ lập trình, trình biên dịch.
  - Không cần phải triển khai chương trình thực hiện giải thuật.
  - Chỉ dựa vào bản thân giải thuật.
- **Các phép toán sơ cấp:** là những phép toán mà thời gian thực hiện nó "đủ ngắn" (không vượt quá một hằng số nào đó):
  - Các phép tính số học.
  - Các phép tính logic.
  - Các phép nhảy, gán.
- **Kích thước dữ liệu đầu vào:** thường dùng một số nguyên dương n để thể hiện kích thước này.

Đánh giá thời gian thực hiện giải thuật

- Một giải thuật T áp dụng để giải bài toán có kích thước n sẽ cần tổng số T(n) các phép toán sơ cấp, trong đó T(n) là một hàm phụ thuộc vào n.
- Hàm số T(n) là đặc trưng cho hiệu quả của giải thuật T.

Tình trạng dữ liệu đầu vào

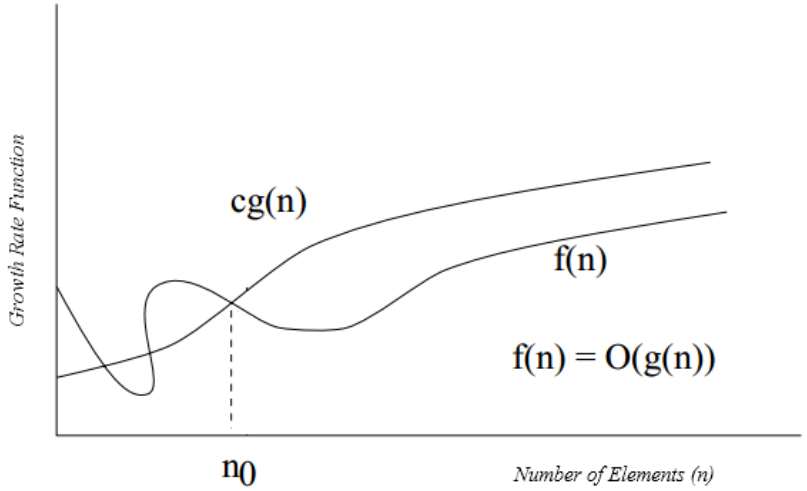
- Hai yếu tố quyết định thời gian thực hiện giải thuật:
  - Số lượng dữ liệu đầu vào
  - Tình trạng dữ liệu đầu vào
- Ví dụ 1: trong bài toán sắp xếp một dãy số, nếu dãy đã có sẵn thứ tự mong muốn hoặc gần như thế thì việc sắp xếp sẽ nhanh hơn trường hợp một dãy bất kỳ.
- Ví dụ 2: bài toán tìm kiếm tuần tự phần tử X trong một dãy số cho sẵn, số lần so sánh:
  - Trường hợp tốt nhất (best case): 1
  - Trường hợp xấu nhất (worst case): n
  - Trường hợp trung bình (average case): (n+1)/2

Tình trạng dữ liệu đầu vào

- Hợp lý nhất là dùng ước lượng thời gian thực hiện trung bình $\mathbf{T_{avg}}$ để so sánh & đánh giá giải thuật.
- Nếu tính thời gian thực hiện trung bình quá khó khăn, có thể đánh giá căn cứ vào trường hợp xấu nhất, tức là dùng $\mathbf{T_{max}}$.

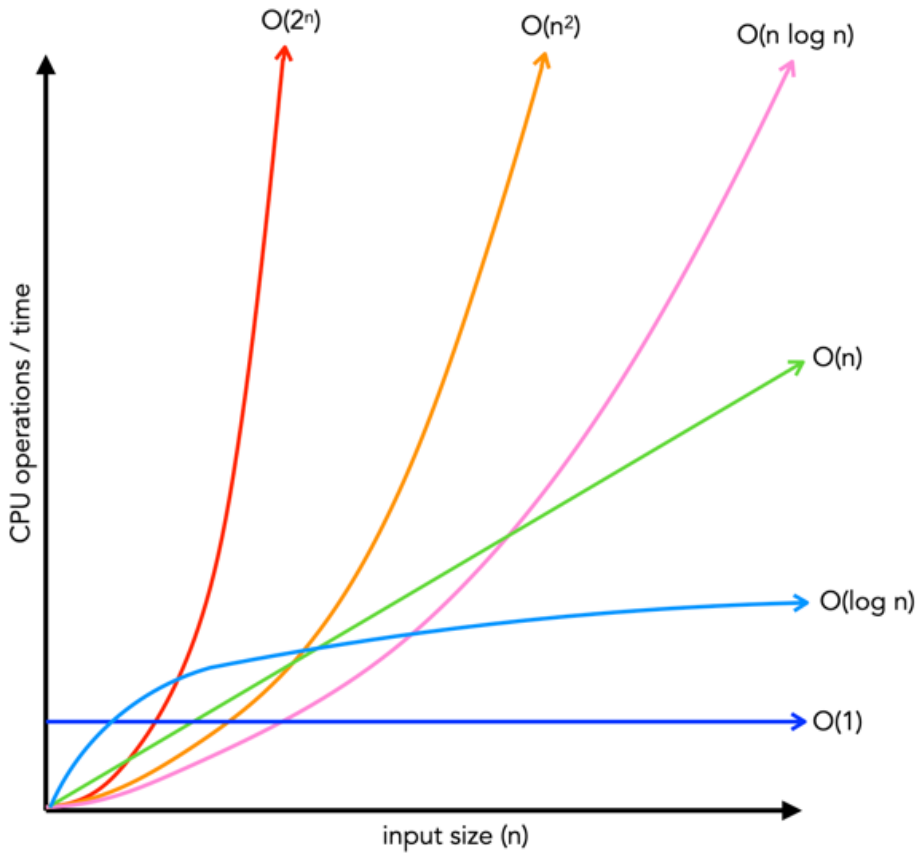
Ký hiệu Big O

- Định nghĩa:
  - Giả sử f(n), g(n) là 2 hàm số không âm, đồng biến theo n.
  - Ta nói "**f(n) là Big O của g(n)**" và viết **f(n) = O(g(n))**, khi và chỉ khi tồn tại hằng số C và hằng số n_0 để **f(n) ≤ C.g(n)**, với mọi n ≥ n_0.
  - Ta nói f(n) có cấp lớn không vượt quá g(n) vì f(n) có tăng tới đâu đi nữa cũng không thể vượt quá g(n).



Ký hiệu Big O

- Trong khoa học máy tính, ký hiệu Big O được sử dụng để đánh giá độ phức tạp của giải thuật.
- Big O càng nhỏ thì giải thuật càng tốt, tức chạy càng nhanh.
- Big O được dùng như một công cụ để phân loại và so sánh mức độ hiệu quả của các giải thuật.
- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$



Ví dụ

- Ví dụ:

  $f(n) = 2n^2 + 3n + 5$; $g(n) = n^2$

  $f(n) \leq 2n^2 + 3n^2 + 5n^2$, với mọi $n \geq 1$

  $f(n) \leq 10n^2 = 10.g(n)$, với mọi $n \geq 1$

  Ta viết: $f(n) = O(g(n)) = O(n^2)$

Ví dụ

```
int sum(int n)
{
    int s = 0; // O(1)
    for (int i = 1; i <= n; i++) // n lần
        s += i; // O(1)
    return s; // O(1)
}
```

- f(n) = O(1) + n.O(1) + O(1)
  = n.O(1) + 2.O(1)
  = a.n + b (đặt a = O(1), b = 2.O(1))
  = O(n)

Ví dụ

$$1 + 2 + 3 + 4 + \ldots + n = \frac{(1+n).n}{2}$$

```
void interchangeSort(int *a, int n)
{
    for (int i = 0; i < n - 1; i++)      // i đi từ 0 đến n - 2
        for (int j = i + 1; j < n; j++) // n - i - 1 lần
            if (a[i] > a[j])             // O(1)
            {
                int temp = a[i]; // O(1)
                a[i] = a[j];     // O(1)
                a[j] = temp;     // O(1)
            }
}
```

for(i = a; i < b; i++) → số lần lặp: b – a
for(i = 1; i < 5; i++) → số lần lặp: 4

- f(n) $= \sum_{i=0}^{n-2}((n-i-1).4.O(1))$
  $= 4.O(1).\sum_{i=0}^{n-2}(n-i-1)$
  $= 4.O(1).((n-1)+(n-2)+\cdots+1)$
  $= 4.O(1).\mathrm{n}\frac{(n-1)}{2}$
  $= O(n^2)$


Q
A