CMC UNIVERSITY

CHƯƠNG 1.
GIỚI THIỆU CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

CMC UNIVERSITY

Nội dung

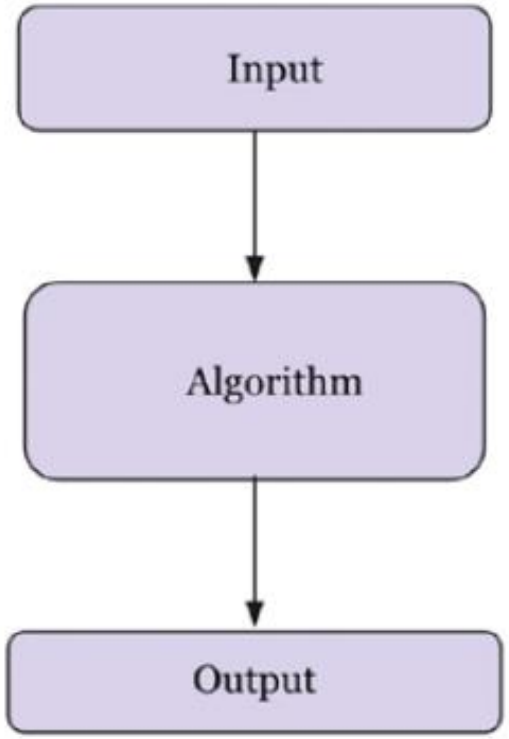
- ✓ Các khái niệm cơ bản
- ✓ Kiểu dữ liệu trừu tượng
- ✓ Độ phức tạp tính toán
- ✓ Đệ qui

CMC UNIVERSITY

Các khái niệm cơ bản

Thuật toán là gì?

Trong thuật ngữ lập trình máy tính, thuật toán là một dãy các câu lệnh chặt chẽ và rõ ràng xác định một trình tự các thao tác trên dữ liệu vào sao cho sau một số hữu hạn bước thực hiện ta đạt được kết quả mong muốn.



Ví dụ: Một thuật toán cộng hai số:

- Lấy hai số đầu vào
- Cộng hai số bằng toán tử +
- Hiển thị kết quả

CMC UNIVERSITY

Thuật toán là gì?

Đánh giá một thuật toán tốt

- Đầu vào và đầu ra phải được xác định chính xác.
- Mỗi bước trong thuật toán phải rõ ràng và không mơ hồ.
- Các thuật toán nên hiệu quả nhất trong số nhiều cách khác nhau để giải quyết vấn đề.
- Một thuật toán không nên bao gồm mã máy tính. Thay vào đó, thuật toán nên được viết theo cách mà nó có thể được sử dụng trong các ngôn ngữ lập trình khác nhau

CMC UNIVERSITY

Thuật toán là gì?

Ví dụ:

- Thuật toán cộng hai số
- Thuật toán tìm số lớn nhất trong ba số
- Thuật toán tìm nghiệm của phương trình bậc hai
- Thuật toán tìm giai thừa
- Thuật toán kiểm tra số nguyên tố
- Thuật toán tính dãy Fibonacci

Tại sao phải nghiên cứu thuật toán?

Tóm tắt dưới đây là một số lý do quan trọng để nghiên cứu các thuật toán:

- Cần thiết cho khoa học máy tính và kỹ thuật
- Quan trọng trong nhiều lĩnh vực khác (chẳng hạn như sinh học tính toán, kinh tế học, sinh thái học, truyền thông, vật lý, v.v.)
- Chúng đóng một vai trò trong đổi mới công nghệ
- Chúng cải thiện khả năng giải quyết vấn đề và tư duy phân tích

Có hai khía cạnh có tầm quan trọng hàng đầu trong việc giải quyết một vấn đề nhất định:

- Đầu tiên, chúng ta cần một cơ chế hiệu quả để lưu trữ, quản lý và truy xuất dữ liệu, cần thiết để giải quyết một vấn đề (điều này nằm trong cấu trúc dữ liệu);
- Thứ hai, chúng tôi yêu cầu một thuật toán hiệu quả là một tập hợp hữu hạn các hướng dẫn để giải quyết vấn đề đó.

Tại sao phải nghiên cứu thuật toán?

Do vậy, việc nghiên cứu cấu trúc dữ liệu và thuật toán là chìa khóa để giải quyết mọi bài toán bằng chương trình máy tính. Một thuật toán hiệu quả cần có những đặc điểm sau:

- Nó phải càng cụ thể càng tốt
- Mỗi lệnh phải được xác định chính xác
- Không được có bất kỳ lệnh nào mơ hồ nào
- Tất cả các lệnh của thuật toán phải được thực thi trong một khoảng thời gian hữu hạn và với số bước hữu hạn
- Cần có đầu vào và đầu ra rõ ràng để giải quyết bài toán
- Mỗi lệnh của thuật toán phải tích hợp trong việc giải bài toán đã cho

Tại sao phải nghiên cứu thuật toán?

Hãy xem xét một ví dụ về thuật toán (tương tự) để hoàn thành một nhiệm vụ trong cuộc sống hàng ngày của chúng ta; chúng ta hãy lấy ví dụ về việc chuẩn bị một tách trà. Thuật toán chuẩn bị một tách trà có thể bao gồm các bước sau:

- Đổ nước vào ấm
- Đặt ấm lên bếp và châm lửa
- Thêm gừng đập dập vào nước ấm
- Cho lá trà vào ấm
- Thêm sữa
- Khi nước bắt đầu sôi, cho đường vào
- Sau 2-3 phút, trà có thể được phục vụ

Tại sao phải nghiên cứu thuật toán?

- Quy trình trên là một trong những cách pha trà khả thi. Theo cách tương tự, giải pháp cho một vấn đề trong thế giới thực có thể được chuyển đổi thành thuật toán, thuật toán này có thể được phát triển thành phần mềm máy tính bằng ngôn ngữ lập trình.
- Vì có thể có một số giải pháp cho một vấn đề nhất định, nên nó phải hiệu quả nhất có thể khi được triển khai bằng phần mềm. Với một vấn đề, có thể có nhiều hơn một thuật toán đúng, được định nghĩa là thuật toán tạo ra chính xác đầu ra mong muốn cho tất cả các giá trị đầu vào hợp lệ.
- Chi phí thực hiện các thuật toán khác nhau có thể khác nhau; nó có thể được đo bằng thời gian cần thiết để chạy thuật toán trên hệ thống máy tính và dung lượng bộ nhớ cần thiết cho nó.

CMC UNIVERSITY

Cấu trúc dữ liệu

Cấu trúc dữ liệu là gì?

- Cấu trúc dữ liệu là một kho lưu trữ được sử dụng để lưu trữ và sắp xếp dữ liệu. Đó là cách sắp xếp dữ liệu trên máy tính để có thể truy cập và cập nhật dữ liệu một cách hiệu quả.
- Tùy thuộc vào yêu cầu và dự án, điều quan trọng là chọn cấu trúc dữ liệu phù hợp cho dự án. Ví dụ: nếu muốn lưu trữ dữ liệu tuần tự trong bộ nhớ, thì có thể sử dụng cấu trúc dữ liệu Array.

CMC UNIVERSITY

Cấu trúc dữ liệu

Các kiểu cấu trúc dữ liệu

Về cơ bản, cấu trúc dữ liệu được chia thành hai loại:

- Cấu trúc dữ liệu tuyến tính (Linear data structure)
- Cấu trúc dữ liệu phi tuyến (Non-linear data structure)

Cấu trúc dữ liệu tuyến tính

- Trong cấu trúc dữ liệu tuyến tính, các phần tử được sắp xếp theo thứ tự lần lượt. Vì các phần tử được sắp xếp theo thứ tự cụ thể nên chúng rất dễ thực hiện.
- Tuy nhiên, khi độ phức tạp của chương trình tăng, cấu trúc dữ liệu tuyến tính có thể không phải là lựa chọn tốt nhất do tính phức tạp trong vận hành.
- Cấu trúc dữ liệu tuyến tính phổ biến là:

CMC UNIVERSITY

Cấu trúc dữ liệu

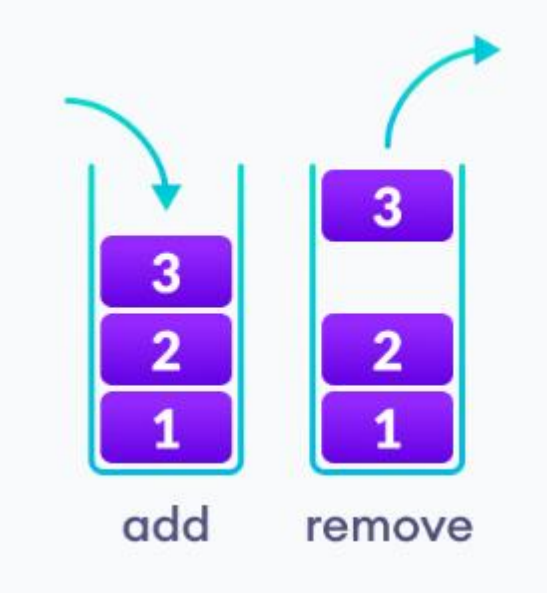
1. Mảng (Array)

Trong mảng, các phần tử trong bộ nhớ được sắp xếp thành bộ nhớ liên tục. Tất cả các phần tử của mảng đều có cùng kiểu. Và loại phần tử có thể được lưu trữ dưới dạng mảng được xác định bởi ngôn ngữ lập trình.

2. Ngăn xếp (Stack)

Trong cấu trúc dữ liệu ngăn xếp, các phần tử được lưu trữ theo nguyên tắc LIFO. Nghĩa là phần tử cuối cùng được lưu trữ trong ngăn xếp sẽ bị xóa trước tiên.

Nó hoạt động giống như một chồng đĩa, trong đó chiếc đĩa cuối cùng được giữ trên chồng sẽ được lấy ra trước.



Cấu trúc dữ liệu

3. Hàng đợi (Queue)

- Không giống như ngăn xếp, cấu trúc dữ liệu hàng đợi hoạt động theo nguyên tắc FIFO trong đó phần tử đầu tiên được lưu trong hàng đợi sẽ bị xóa trước tiên.
- Nó hoạt động giống như việc xếp hàng người ở quầy bán vé, nơi người đầu tiên xếp hàng sẽ nhận được vé trước.

4. Danh sách liên kết (Linked List)

- Trong cấu trúc dữ liệu danh sách liên kết, các phần tử dữ liệu được kết nối thông qua một loạt các nút. Và mỗi nút chứa các mục dữ liệu và địa chỉ của nút tiếp theo.

CMC UNIVERSITY

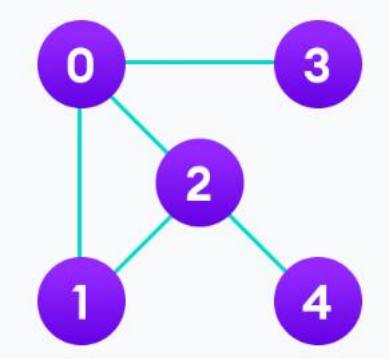
Cấu trúc dữ liệu

Cấu trúc dữ liệu phi tuyến

- Không giống như cấu trúc dữ liệu tuyến tính, các phần tử trong cấu trúc dữ liệu phi tuyến tính không theo bất kỳ trình tự nào. Thay vào đó chúng được sắp xếp theo cách phân cấp trong đó một phần tử sẽ được kết nối với một hoặc nhiều phần tử.
- Cấu trúc dữ liệu phi tuyến tính được chia thành cấu trúc dữ liệu đồ thị và cây.

1. Đồ thị

- Trong cấu trúc dữ liệu đồ thị, mỗi nút được gọi là đỉnh và mỗi đỉnh được kết nối với các đỉnh khác thông qua các cạnh.
- Cấu trúc dữ liệu dựa trên đồ thị phổ biến:
  - Cây khung và Cây khung tối thiểu
  - Các thành phần liên thông mạnh
  - Ma trận kề
  - Danh sách kề



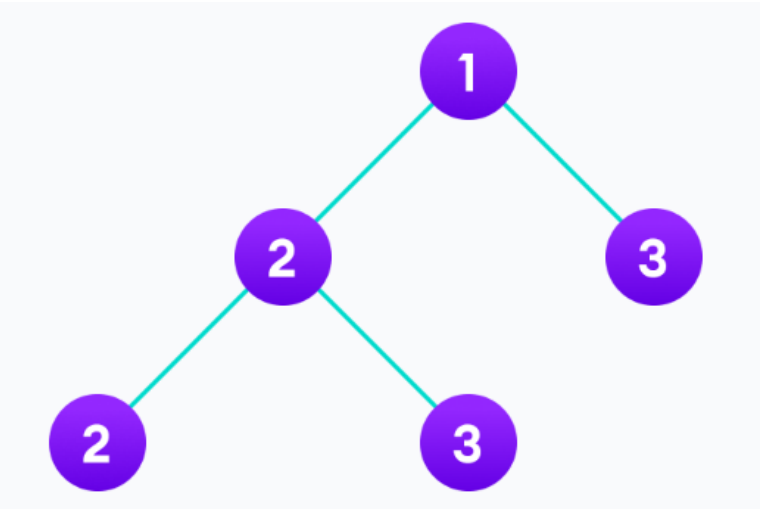
Cấu trúc dữ liệu

2. Cây

- Tương tự đồ thị, cây cũng kaf một tập hợp các đỉnh và canh. Tuy nhiên, trong cấu trúc dữ liệu cây, chỉ có thể có một cạnh giữa hai đỉnh.

Các cấu trúc dựa vào cây phổ biến

- Cây nhị phân (Binary Tree)
- Cây tìm kiếm nhị phân (Binary Search Tree)
- Cây AVL
- B-Tree
- B+ Tree
- Cây đỏ đen (Red-Black Tree)



CMC UNIVERSITY

Cấu trúc dữ liệu

Cấu trúc dữ liệu tuyến tính	Cấu trúc dữ liệu phi tuyến
Các mục dữ liệu được sắp xếp theo thứ tự liên tục, nối tiếp nhau	Các mục dữ liệu được sắp xếp theo thứ tự không tuần tự (kiểu phân cấp).
Tất cả các mục có mặt trên một tầng.	Các mục dữ liệu có mặt ở các tầng khác nhau.
Nó có thể được duyệt trên một lần chạy. Nghĩa là, nếu chúng ta bắt đầu từ phần tử đầu tiên, chúng ta có thể duyệt qua tất cả các phần tử một cách tuần tự trong một lần duyệt	Nó đòi hỏi phải chạy nhiều lần. Nghĩa là, nếu chúng ta bắt đầu từ phần tử đầu tiên thì có thể không duyệt được tất cả các phần tử trong một lần duyệt

CMC UNIVERSITY

Cấu trúc dữ liệu

Cấu trúc dữ liệu tuyến tính	Cấu trúc dữ liệu phi tuyến tính
Việc sử dụng bộ nhớ không hiệu quả	Các cấu trúc khác nhau sử dụng bộ nhớ theo những cách hiệu quả khác nhau tùy thuộc vào nhu cầu.
Độ phức tạp thời gian tăng theo kích cỡ dữ liệu.	Độ phức tạp thời gian vẫn giữ nguyên
Ví dụ: Mảng, ngăn xếp, hang đợi	Ví dụ: Cây, đồ thị, bản đồ

Hiểu cách hoạt động của từng cấu trúc dữ liệu sẽ giúp ta có thể chọn cấu trúc dữ liệu phù hợp cho dự án của mình → viết code hiệu quả về bộ nhớ và thời gian.

CMC UNIVERSITY

Kiểu dữ liệu trừu tượng

CMC UNIVERSITY

Kiểu dữ liệu trừu tượng

- Kiểu dữ liệu (Data Type) về cơ bản là một loại dữ liệu có thể được sử dụng trong các chương trình máy tính khác nhau. Nó biểu thị kiểu như số nguyên, float, v.v., không gian như số nguyên sẽ chiếm 4-byte, ký tự sẽ chiếm 1-byte không gian bộ nhớ, v.v
- Kiểu dữ liệu trừu tượng (Abstract Data Type - ADT) là kiểu dữ liệu đặc biệt, có hành vi được xác định bởi một tập hợp các giá trị và tập hợp các thao tác (phép toán). Từ khóa "Abstract" được sử dụng vì chúng ta có thể sử dụng các kiểu dữ liệu này, chúng ta có thể thực hiện các thao tác khác nhau. Nhưng cách thức hoạt động của những thao tác đó hoàn toàn bị ẩn với người dùng. ADT được tạo bằng các kiểu dữ liệu nguyên thủy, nhưng logic thao tác bị ẩn.
- Ví dụ ADT: Stack, Queue, List, …

CMC UNIVERSITY

Kiểu dữ liệu trừu tượng

Phép toán trên Stack:

- isFull(), kiểm tra stack có đầy hay không
- isEmpty(), kiểm tra stack có rỗng hay không
- push(x), đẩy x vào stack
- pop(), xóa một phần tử khỏi đỉnh của stack
- peek(), lấy phần tử ở đỉnh của stack
- size(), hàm trả ra số phần tử của stack

CMC UNIVERSITY

Kiểu dữ liệu trừu tượng

Phép toán trên Queue:

- isFull(), kiểm tra queue có đầy hay không
- isEmpty(), kiểm tra queue có rỗng hay không
- insert(x), bổ sung x vào đuôi queue
- delete(), xóa một phần tử khỏi đầu của queue
- size(), hàm trả ra số phần tử của queue

CMC UNIVERSITY

Kiểu dữ liệu trừu tượng

Phép toán trên danh sách (List)

- size(), hàm trả ra số phần tử trong danh sách
- insert(x), hàm chèn một phần tử vào danh sách
- remove(x), hàm xóa một phần tử khỏi danh sách
- get(i), hàm lấy phần tử ở vị trí *i*
- replace(x, y), hàm thay thế giá trị *x* bởi giá trị *y*

CMC UNIVERSITY

Độ phức tạp tính toán

Phân tích hiệu năng của thuật toán

Khi thiết kế một thuật toán hiệu quả cần lưu ý:

1. Thuật toán phải chính xác và phải tạo ra kết quả như mong đợi cho tất cả các giá trị đầu vào hợp lệ

2. Thuật toán phải tối ưu theo nghĩa là nó phải được thực thi trên máy tính trong giới hạn thời gian mong muốn, phù hợp với yêu cầu không gian bộ nhớ tối ưu.

Phân tích hiệu suất của thuật toán là rất quan trọng để quyết định giải pháp tốt nhất cho một vấn đề nhất định. Nếu hiệu suất của thuật toán nằm trong yêu cầu về thời gian và không gian mong muốn, thì thuật toán đó là tối ưu.

Một trong những phương pháp phổ biến để ước tính hiệu suất của một thuật toán là thông qua phân tích độ phức tạp của nó. Phân tích thuật toán giúp ta xác định thuật toán nào hiệu quả nhất về thời gian và không gian tiêu thụ.

Phân tích hiệu năng của thuật toán

Hiệu năng của thuật toán được đo bởi kích thước của dữ liệu đầu vào, *n*, và thời gian và không gian bộ nhớ được thuật toán sử dụng.

Độ phức tạp thời gian

Độ phức tạp về thời gian của thuật toán là lượng thời gian mà thuật toán sẽ thực hiện trên hệ thống máy tính để tạo ra đầu ra..

- Thời gian chạy của thuật toán phụ thuộc vào kích thước đầu vào; khi kích thước đầu vào, *n*, tăng lên thì thời gian chạy cũng tăng lên. Kích thước đầu vào được đo bằng số lượng mục trong đầu vào

Ví dụ: Một thuật toán sắp xếp có thời gian chạy tăng khi sắp xếp 5,000 phần tử so với khi sắp xếp 50 phần tử.

- Thời gian được đo bằng các thao tác chính sẽ được thực hiện trong thuật toán (chẳng hạn như các thao tác so sánh), trong đó các thao tác chính là các câu lệnh chiếm một lượng thời gian đáng kể trong quá trình thực hiện thuật toán.

Ví dụ: thao tác chính của thuật toán sắp xếp là thao tác so sánh sẽ chiếm phần lớn thời gian chạy, so với phép gán hoặc bất kỳ thao tác nào khác

CMC UNIVERSITY

Phân tích hiệu năng của thuật toán

- Yêu cầu về không gian của một thuật toán được đo bằng bộ nhớ cần thiết để lưu trữ các biến, hằng và lệnh trong quá trình thực hiện chương trình.
- Lý tưởng nhất là các thao tác chính này không nên phụ thuộc vào phần cứng, hệ điều hành hoặc ngôn ngữ lập trình được sử dụng để triển khai thuật toán.
- Cần một lượng thời gian không đổi để thực thi từng dòng mã; tuy nhiên, mỗi dòng có thể mất một khoảng thời gian khác nhau để thực thi

Ví dụ 1:

Code	Time required (cost)
`if (n == 0 || n == 3) {`	c_1
`    System.out.println("data");`	c_2
`} else {`	c_3
`    for (int i = 0; i < n; i++) {`	c_4
`        System.out.println("structure");`	c_5
`    }`	
`}`	

CMC UNIVERSITY

Phân tích hiệu năng của thuật toán

Thời gian chạy của thuật toán là tổng thời gian thực hiện tất cả các câu lệnh.

Trong đoạn code trên, giả sử câu lệnh 1 chạy hết thời gian c_1, câu lệnh 2 chạy hết thời gian c_2, …. Do vậy, nếu câu lệnh thứ i chạy hết thời gian không đổi c_i và nếu câu lệnh thứ i được thực hiện n lần, thì đoạn code cần thời gian $c_i n$.

Tổng thời gian chạy thuật toán $T(n)$ với giá trị cho trước n (giả sử giá trị n khác 0 hoặc 3) sẽ như sau:

$$T(n) = c_1 + c_3 + c_4 \times n + c_5 \times n$$

Nếu giá trị n bằng 0 hoặc 3, thì thời gian chạy của thuật toán sẽ là:

$$T(n) = c_1 + c_2$$

CMC UNIVERSITY

Phân tích hiệu năng của thuật toán

Do đó, thời gian chạy cần thiết của một thuật toán không chỉ phụ thuộc vào kích thước của dữ liệu đầu vào mà còn phụ thuộc đầu vào là gì ?

Với ví dụ trên:

- Trường hợp tốt nhất khi đầu vào bằng 0 hoặc bằng 3 và trong trường hợp đó, thời gian chạy của thuật toán là hằng số.
- Trường hợp tồi nhất, giá trị của n khác 0 và 3, thì thời gian chạy của thuật toán được biểu diễn là $a \times n + b$, là hàm tuyến tính của n.

CMC UNIVERSITY

Phân tích hiệu năng của thuật toán

Ví dụ 2: tìm kiếm tuyến tính

```java
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static int lS(List<Integer> L, int x) {
        for (int i = 0; i < L.size(); i++) {
            if (L.get(i) == x) {
                return i;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        List<Integer> L = new ArrayList<>();
        L.add(3); L.add(4); L.add(1); L.add(6); L.add(14);
        int x = 4;
        System.out.println("Position of x is: " + lS(L, x));
    }
}
```

Output

```
Position of x is: 1
```

CMC UNIVERSITY

Phân tích hiệu năng của thuật toán

- Thời gian chạy của thuật toán trường hợp tồi nhất (worst-case) là độ phức tạp cận trên; đó là thời gian chạy tối đa của một thuật toán thực thi với bất kỳ đầu vào đưa ra.

Ví dụ: Trong bài toán tìm kiếm tuyến tính, trường hợp xấu nhất xảy ra khi phần tử cần tìm kiếm được tìm thấy trong phép so sánh cuối cùng hoặc không tìm thấy trong danh sách. Trong trường hợp này, thời gian chạy cần thiết sẽ phụ thuộc tuyến tính vào độ dài của danh sách.

- Thời gian chạy của thuật toán trường hợp trung bình (average-case) là thời gian chạy trung bình cần thiết để một thuật toán thực thi. Trong phân tích này, ta tính toán mức trung bình trong thời gian chạy cho tất cả các giá trị đầu vào có thể có.

Ví dụ: trong tìm kiếm tuyến tính, số lượng phép so sánh ở tất cả các vị trí sẽ là *1* nếu phần tử cần tìm kiếm được tìm thấy ở chỉ mục thứ 0; và tương tự, số phép so sánh là *2, 3, …,* đến *n*, tương ứng, với các phần tử được tì thấy ở vị trí chỉ mục *1, 2, 3, … (n-1)*.

Do vậy, thời gian chạy thuật toán trường hợp trung bình: $T(n) = \frac{1+2+\cdots+n}{n} = \frac{n(n+1)}{2n}$

CMC UNIVERSITY

Phân tích hiệu năng của thuật toán

- Thời gian chạy thuật toán trường hợp tốt nhất (best-case) thời gian tối thiểu cần thiết để chạy thuật toán; là giới hạn dưới của thời gian chạy cần thiết cho một thuật toán

Ví dụ: trong bài toán tìm kiếm tuyến tính, phần tử cần tìm sẽ được tìm thấy trong lần so sánh đầu tiên.

Độ phức tạp không gian

- Độ phức tạp về không gian của thuật toán ước tính yêu cầu bộ nhớ để thực thi nó trên máy tính nhằm tạo ra đầu ra dưới dạng một hàm của dữ liệu đầu vào.
- Khi thực hiện thuật toán trên hệ thống máy tính, cần phải lưu trữ dữ liệu đầu vào cùng với dữ liệu trung gian và tạm thời trong cấu trúc dữ liệu được lưu trữ trong bộ nhớ của máy tính.

CMC UNIVERSITY

Phân tích hiệu năng của thuật toán

Ví dụ 3

```
1 public class Main {
2     public static void main(String[] args) {
3         int[] numbers = {2, 3, 5, 8};
4         for (int i = 0; i < numbers.length; i++) {
5             int square = numbers[i] * numbers[i];
6             System.out.println(square);
7         }
8     }
9 }
```

The output of the code is:

```
4
9
25
64
```

Trong đoạn code này, thuật toán sẽ yêu cầu cấp phát bộ nhớ cho số lượng mục trong danh sách đầu vào.

Số phần tử đầu vào là *n*, yêu cầu không gian tăng theo kích cỡ đầu vào, do vậy, độ phức tạp không gian của thuật toán *O(n)*

Giả sử có 2 thuật toán tìm kiếm, một thuật toán có độ phức tạp không gian *O(n)* và thuật toán kia là *O(nlogn)*.

Thuật toán đầu tiên là tốt hơn thuật toán thứ hai về không gian bộ nhớ yêu cầu.

CMC UNIVERSITY

Kí hiệu tiệm cận

- Để phân tích độ phức tạp về thời gian của một thuật toán, tốc độ tăng trưởng (cấp độ tăng trưởng) rất quan trọng khi kích thước đầu vào lớn
- Khi kích thước đầu vào trở nên lớn, chúng tôi chỉ xem xét các số hạng bậc cao hơn và bỏ qua các số hạng bậc không đáng kể
- Trong phân tích tiệm cận, chúng tôi phân tích hiệu quả của các thuật toán đối với các kích thước đầu vào lớn khi xem xét các số hạng có cấp độ tăng trưởng cao hơn và bỏ qua các hằng số nhân và các số hạng bậc thấp hơn.

CMC UNIVERSITY

Kí hiệu tiệm cận

Các ký hiệu tiệm cận sau đây thường được sử dụng để tính độ phức tạp thời gian chạy của thuật toán:

- θ: biểu thị độ phức tạp thời gian chạy thuật toán trong trường hợp tồi nhất với một giới hạn chặt.
- O: biểu thị độ phức tạp về thời gian chạy thuật toán trong trường hợp tồi nhất với giới hạn trên, đảm bảo rằng hàm không bao giờ tăng nhanh hơn giới hạn trên.
- Ω: biểu thị giới hạn dưới của thời gian chạy thuật toán. Nó đo lượng thời gian tốt nhất để thực hiện thuật toán

CMC UNIVERSITY

Kí hiệu tiệm cận

Kí hiệu Theta

- Hàm sau đây mô tả thời gian chạy trong trường hợp tồi nhất cho ví dụ 1:

$$T(n) = c_1 + c_3 + c_4 \times n + c_5 \times n$$

Ở đây, đối với kích thước đầu vào lớn, thời gian chạy trong trường hợp tồi nhất sẽ là $\theta(n)$, tức là độ phức tạp về thời gian của thuật toán có cấp độ tăng trưởng (gọi tắt là cấp độ) là n

- Chúng tôi thường coi một thuật toán là hiệu quả hơn một thuật toán khác nếu thời gian chạy trong trường hợp xấu nhất của nó có cấp độ tăng trưởng thấp hơn.

θ biểu thị thời gian chạy trong trường hợp tồi nhất đối với một thuật toán có giới hạn chặt. Với một hàm cho trước $F(n)$, độ phức tạp thời gian chạy trong trường hợp tồi nhất tiệm cận có thể được định nghĩa như sau:

$$T(n) = \theta(F(n))$$

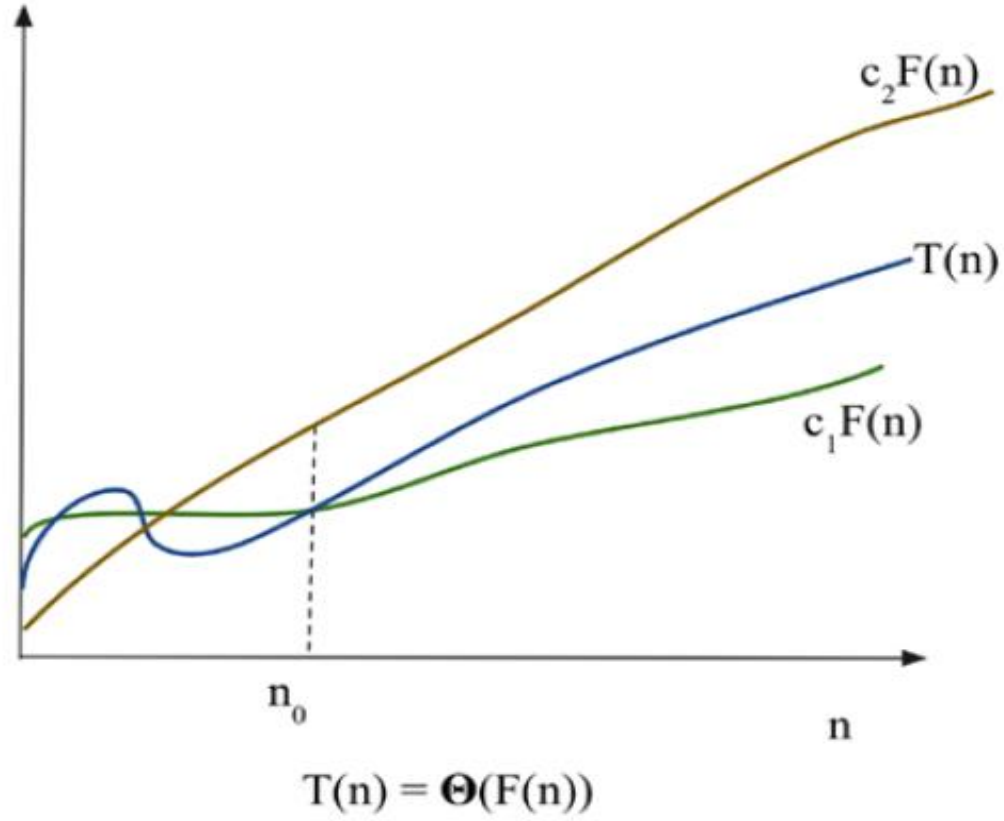
khi và chỉ khi tồn tại các hằng số n_0, c_1, và c_2 sao cho:

$$0 \leq c_1(F(n)) \leq T(n) \leq c_2(F(n)) \text{ với mọi } n \geq n_0$$

Kí hiệu tiệm cận

- Hàm $T(n)$ thuộc tập các hàm $\Theta(F(n))$ nếu tồn tại các hằng số dương c_1 và c_2 sao cho giá trị của $T(n)$ luôn nằm trong khoảng giữa $c_1F(n)$ và $c_2F(n)$ với mọi giá trị lớn của n. Nếu điều kiện này là đúng, thì ta nói $F(n)$ tiệm cận chặt đối với $T(n)$

Hình bên chỉ ra giá trị của $T(n)$ luôn nằm giữa $c_1F(n)$ và $c_2F(n)$ với các giá trị của n lớn hơn n_0



Kí hiệu tiệm cận

Ví dụ 4: cho hàm $f(n)= n^2 + n$

Để xác định độ phức tạp thời gian với định nghĩa ký hiệu Θ, trước tiên ta phải xác định các hằng số c_1 , c_2, n_0 sao cho

$$0 \le c_1 n^2 \le n^2 + n \le c_2 n^2 \quad \text{với mọi } n \ge n_0$$

Chia cho n^2, ta được:

$$0 \le c_1 \le 1 + 1/n \le c_2 \quad \text{for all } n \ge n_0$$

Bằng cách chọn $c_1 = 1$, $c_2 = 2$, $n_0 = 1$, điều kiện sau có thể thỏa định nghĩa của Θ

$$0 \le n^2 \le n^2 + n \le 2n^2 \quad \text{với mọi } n \ge 1$$

Suy ra: $f(n) = \Theta(g(n))$, nghĩa là $f(n) = \Theta(n^2)$

CMC UNIVERSITY

Kí hiệu tiệm cận

Kí hiệu O lớn (Big O)

- Ta thấy rằng θ bị giới hạn tiệm cận từ cạnh trên và cạnh dưới của hàm, trong khi kí hiệu Big O đặc trưng cho độ phức tạp thời gian chạy trường hợp tồi nhất, vốn chỉ là cận trên tiệm cận của hàm. Kí hiệu O được định nghĩa như sau.

Cho một hàm $F(n)$, $T(n)$ là một Big O của hàm $F(n)$, được định nghĩa như sau:

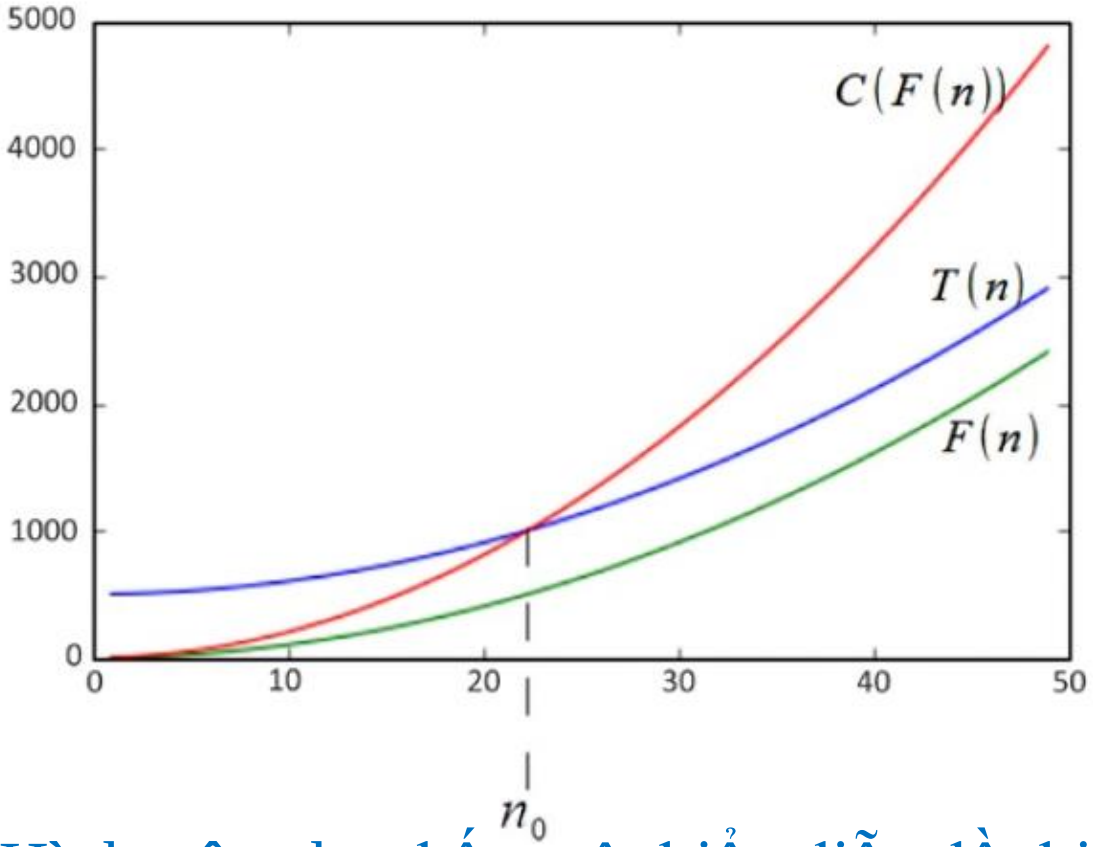
$$T(n) = O(F(n))$$

khi và chỉ khi tồn tại các hằng số n_0 và c sao cho:

$$T(n) \leq c(F(n)) \text{ với mọi } n \geq n_0$$

- Trong kí hiệu Big O, bội số hằng của $F(n)$ là cận trên tiệm cận của $T(n)$, và các hằng số dương n_0 và c phải sao cho tất cả các giá trị của n lớn hơn n_0 luôn nằm trên hoặc dưới hàm $c.F(n)$.

Kí hiệu tiệm cận



Trong kí hiệu *O*, *O(F(n))* thực sự là một tập hợp các hàm bao gồm tất cả các hàm có tốc độ tăng trưởng bằng hoặc nhỏ hơn *F(n)*. Ví dụ, $O(n^2)$ cũng bao gồm *O(n)*, *O(log n)*, ...

Tuy nhiên, Big *O* nên mô tả một hàm càng sát càng tốt, ví dụ, $F(n) = 2n^3 + 2n^2 + 5$ là $O(n^4)$, tuy nhiên, *F(n)* chính xác hơn là $O(n^3)$.

Hình trên cho thấy một biểu diễn đồ thị của hàm *T(n)* với giá trị *n* thay đổi.

Ta có thể thấy rằng $T(n) = n^2 + 500 = O(n^2)$, với $c = 2$ và n_0 xấp xỉ *23*

CMC UNIVERSITY

Kí hiệu tiệm cận

Time Complexity	Name
O(1)	Constant
O(logn)	Logarithmic
O(n)	Linear
O(nlogn)	Linear-logarithmic
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential

Log_2n	n	$nlog_2n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	2 147 483 648

Các hàm 2^n, n!, n^n được gọi là các hàm mũ. Một thuật toán có thời gian thực hiện hàm mũ sẽ rất chậm. Các hàm n^3, n^2, $nlog_2n$, n, log_2n được gọi là các hàm đa thức. Các thuật toán có thời gian thực hiện ở mức đa thức thường chấp nhận được.

CMC UNIVERSITY

Kí hiệu tiệm cận

Kí hiệu Omega

- Kí hiệu Ω mô tả giới hạn tiệm cận dưới của thuật toán.
- Kí hiệu Ω tính toán độ phức tạp thời gian chạy trong trường hợp tốt nhất của thuật toán.
- Kí hiệu Ω là một tập hợp các hàm sao cho có các hằng số dương n_0 và c sao cho với mọi giá trị n lớn hơn n_0, $T(n)$ luôn nằm trên hoặc phía trên hàm $c.F(n)$.

$$T(n) = \Omega\,(F(n))$$

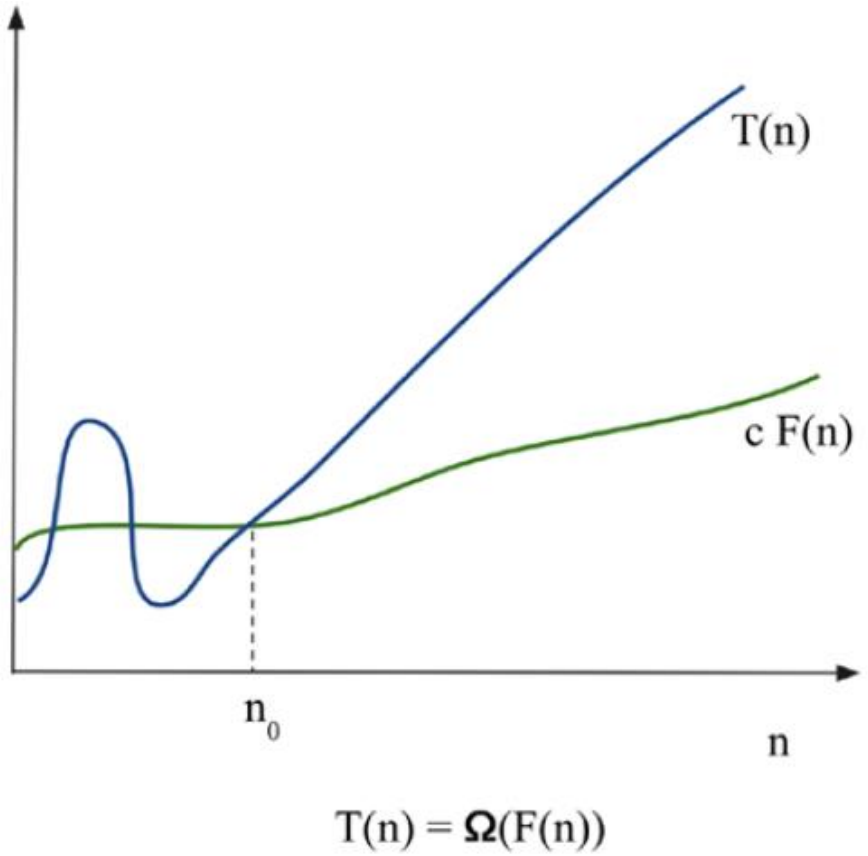
khi và chỉ khi các hằng n_0 và c có mặt, thì:

$$0 \leq c(F(n)) \leq T(n) \quad \text{với mọi} \quad n \geq n_0$$

CMC UNIVERSITY

Kí hiệu tiệm cận

- Nếu thời gian chạy của một thuật toán là Ω(F(n)), nghĩa là thời gian chạy của thuật toán ít nhất là một hằng số nhân của F(n) với các giá trị đủ lớn của kích thước đầu vào (n).
- Kí hiệu Ω đưa ra giới hạn dưới về độ phức tạp thời gian chạy trong trường hợp tốt nhất của một thuật toán đã cho. Nghĩa là thời gian chạy của một thuật toán đã cho sẽ ít nhất là F(n) mà không phụ thuộc vào đầu vào.



Kí hiệu tiệm cận

Ví dụ 5: Tìm F(n) với hàm $T(n) = 2n^2 + 3$ sao cho $T(n) = \Omega(F(n))$

Giải: Sử dụng kí hiệu Ω, điều kiện cho giới hạn dưới là:

$$c.F(n) \leq T(n)$$

Điều kiện này đúng với mọi giá trị của n lớn hơn 0, và c =1

$$0 \leq cn^2 \leq 2n^2 + 3, \text{ với mọi } n \geq 0$$

$$2n^2 + 3 = \Omega(n^2)$$

$$F(n) = n^2$$

Tính độ phức tạp thời gian của thuật toán

- Để phân tích một thuật toán liên quan đến thời gian chạy trường hợp tốt nhất, tồi nhất và trung bình của thuật toán, không phải lúc nào cũng có thể tính các thời gian này cho mọi hàm hoặc thuật toán đã cho.
- Vấn đề quan trọng cần xác định độ phức tạp thời gian chạy trong trường hợp tồi nhất của thuật toán trong các tình huống thực tế; do đó, chúng ta tập trung vào tính giới hạn trên Big O để tính độ phức tạp thời gian chạy trong trường hợp tồi nhất của thuật toán.
- Việc xác định độ phức tạp về thời gian của bất kỳ thuật toán nào cũng có thể dẫn đến các vấn đề phức tạp. Tuy nhiên, trong thực tế, đối với một số thuật toán có thể phân tích bằng một số quy tắc đơn giản.

Tính độ phức tạp thời gian của thuật toán

Qui tắc tổng

- Giả sử $T_1(n)$ và $T_2(n)$ là thời gian thực hiện của 2 đoạn chương trình P_1, P_2, ở đó $T_1(n) = O(f(n))$, $T_2(n) = O(g(n))$, thì thời gian thực hiện của P_1 và P_2 lần lượt sẽ là:

$$T_1(n) + T_2(n) = O(\max(f(n), g(n)))$$

Ví dụ 6: Trong một chương trình có 3 bước thực hiện mà thời gian thực hiện của mỗi bước là $O(n^2)$, $O(n^3)$ và $O(n\log_2 n)$ tương ứng, thời gian thực hiện của 2 bước đầu là $O(\max(n^2, n^3)) = O(n^3)$. Thời gian thực hiện chương trình sẽ là

$$O(\max(n^3, n\log_2 n)) = O(n^3).$$

Một ứng dụng khác của quy tắc này:nếu $g(n) \le f(n)$, $\forall n \ge n_0$ thì $O(f(n) + g(n))$ cũng là $O(f(n))$. Ví dụ, $O(n^4 + n^2) = O(n^4)$, $O(n + \log_2 n) = O(n)$.

Tính độ phức tạp thời gian của thuật toán

Qui tắc tích

- Giả sử $T_1(n)$ và $T_2(n)$ là thời gian thực hiện của hai đoạn chương trình P_1, P_2, ở đó $T_1(n) = O(f(n))$, $T_2(n) = O(g(n))$, thì thời gian thực hiện các đoạn lồng nhau P_1 và P_2 sẽ là:

$$T_1(n)\, T_2(n) = O(f(n)g(n))$$

Ví dụ 7: *for (int j = 0; j < n; j++) {*

x = x + 1;

}

Câu lệnh gán có thời gian thực hiện là c (hằng số), do vậy nó có giá trị O(1).

Câu lệnh lặp có thời gian thực hiện O(n.1) = O(n).

Tính độ phức tạp thời gian của thuật toán

Qui tắc tích

Ví dụ 8: *for (int i = 0; i < n; i++) {*

```
        for (int j = 0; j < n; j++) {
            x = x + 1;
        }
    }
```

Câu lệnh trên được đánh giá $O(n.n) = O(n^2)$

Ứng dụng khác của qui tắc này: $O(cf(n)) = O(f(n))$. Ví dụ, $O(n^2/2) = O(n^2)$

Tính độ phức tạp thời gian của thuật toán

Ví dụ 9: Tìm độ phức tạp thời gian chạy trong trường hợp tồi nhất của đoạn mã Java sau:

```
for (int i = 0; i < n; i++) {
        System.out.println("Java");
}
```

$T(n) = cn = O(n)$

```
for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            System.out.println("run");
        }
```

$T(n) = O(n^2)$

```
for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            System.out.println("run fun");
            break;
        }
```

$T(n) = O(n)$

```
for (int i = 0; i < n; i++) {
        System.out.println("Java");
    }
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++) {
        System.out.println("run fun");
    }
```

$T(n) = c_1 n + c_2 n^2 = O(n^2)$

Computing the time complexity of an algorithm

Ví dụ 10: Tìm độ phức tạp thời gian trong trường hợp tồi nhất của đoạn mã Java sau :

```
if (n == 0) {
    System.out.println("data");
} else {
    for (int i = 0; i < n; i++) {
        System.out.println("structure");
    }
}
```

$T(n) = c_1 + c_2 n = O(n)$

```
int i = 1;
int j = 0;
while (i * i < n) {
    j = j + 1;
    i = i + 1;
    System.out.println("data");
}
```

$T(n) = O(\sqrt{n})$

```
int i = 0;
for (i = n / 2; i < n; i++) {
    int j = 1;
    while (j + n / 2 <= n) {
        int k = 1;
        while (k < n) {
            k *= 2;
            System.out.println("data");
            j += 1;
        }
    }
}
```

$T(n) = O(n*n*\log_2 n) = O(n^2 \log_2 n)$

CMC UNIVERSITY

Computing the time complexity of an algorithm

Ví dụ 11: So sánh độ phức tạp thời gian trong trường hợp tồi nhất của các đoạn mã Java sau :

```
1  /** Returns true if there is no element common to all three arrays. */
2  public static boolean disjoint1(int[ ] groupA, int[ ] groupB, int[ ] groupC) {
3    for (int a : groupA)
4      for (int b : groupB)
5        for (int c : groupC)
6          if ((a == b) && (b == c))
7            return false;                    // we found a common value
8    return true;                             // if we reach this, sets are disjoint
9  }
```

$T(n) = O(n^3)$

```
1  /** Returns true if there is no element common to all three arrays. */
2  public static boolean disjoint2(int[ ] groupA, int[ ] groupB, int[ ] groupC) {
3    for (int a : groupA)
4      for (int b : groupB)
5        if (a == b)                  // only check C when we find match from A and B
6          for (int c : groupC)
7            if (a == c)              // and thus b == c as well
8              return false;          // we found a common value
9    return true;                     // if we reach this, sets are disjoint
10 }
```

$T(n) = O(n^2)$

CMC UNIVERSITY

Computing the time complexity of an algorithm

Ví dụ 11: So sánh độ phức tạp thời gian trong trường hợp tồi nhất của đoạn mã Java sau :

```
1  /** Returns true if there are no duplicate elements in the array. */
2  public static boolean unique1(int[ ] data) {
3    int n = data.length;
4    for (int j=0; j < n−1; j++)
5      for (int k=j+1; k < n; k++)
6        if (data[j] == data[k])
7          return false;                    // found duplicate pair
8    return true;                           // if we reach this, elements are unique
9  }
```

$T(n) = O(n^2)$

```
1  /** Returns true if there are no duplicate elements in the array. */
2  public static boolean unique2(int[ ] data) {
3    int n = data.length;
4    int[ ] temp = Arrays.copyOf(data, n);      // make copy of data
5    Arrays.sort(temp);                         // and sort the copy
6    for (int j=0; j < n−1; j++)
7      if (temp[j] == temp[j+1])                // check neighboring entries
8        return false;                          // found duplicate pair
9    return true;                               // if we reach this, elements are unique
10 }
```

$T(n) = O(n\log n)$

CMC UNIVERSITY

Computing the time complexity of an algorithm

Ví dụ 11: So sánh độ phức tạp thời gian trong trường hợp tồi nhất của đoạn mã Java sau :

```
1  /** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
2  public static double[ ] prefixAverage1(double[ ] x) {
3    int n = x.length;
4    double[ ] a = new double[n];                    // filled with zeros by default
5    for (int j=0; j < n; j++) {
6      double total = 0;                             // begin computing x[0] + ... + x[j]
7      for (int i=0; i <= j; i++)
8        total += x[i];
9      a[j] = total / (j+1);                         // record the average
10   }
11   return a;
12 }
```

$T(n) = O(n^2)$

```
1  /** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
2  public static double[ ] prefixAverage2(double[ ] x) {
3    int n = x.length;
4    double[ ] a = new double[n];          // filled with zeros by default
5    double total = 0;                     // compute prefix sum as x[0] + x[1] + ...
6    for (int j=0; j < n; j++) {
7      total += x[j];                      // update prefix sum to include x[j]
8      a[j] = total / (j+1);               // compute average based on current sum
9    }
10   return a;
11 }
```

$T(n) = O(n)$

CMC UNIVERSITY

Đệ qui

CMC UNIVERSITY

Đệ qui

Thuật toán đệ quy gọi chính nó lặp đi lặp lại để giải quyết vấn đề cho đến khi một điều kiện nhất định được đáp ứng. Mỗi cuộc gọi đệ quy tự quay vòng các cuộc gọi đệ quy khác.

Một hàm đệ quy có thể nằm trong một vòng lặp vô hạn; do đó, yêu cầu mỗi hàm đệ quy phải tuân thủ các thuộc tính nhất định. Cốt lõi của hàm đệ quy là hai loại trường hợp:

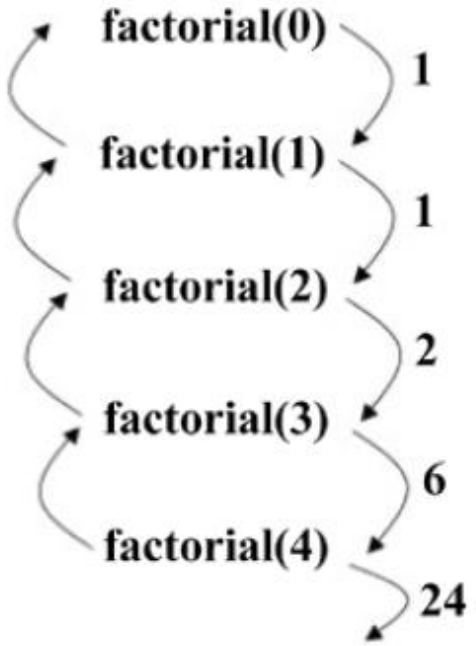
1. Các trường hợp cơ sở: Các trường hợp này cho phép đệ quy biết khi nào kết thúc, nghĩa là đệ quy sẽ bị dừng khi điều kiện cơ sở được đáp ứng

2. Các trường hợp đệ quy: Hàm gọi chính nó một cách đệ quy và chúng ta tiến dần đến việc đạt được các tiêu chí cơ sở

Đệ qui

- Một bài toán đơn giản có thể áp dụng một cách tự nhiên giải pháp đệ quy là tính giai thừa. Thuật toán giai thừa đệ quy xác định hai trường hợp: trường hợp cơ sở khi $n = 0$ (điều kiện kết thúc) và trường hợp đệ quy khi $n > 0$ (lời gọi hàm).
- Việc thực hiện đệ quy của hàm giai thừa:

```
1  public static int factorial(int n) throws IllegalArgumentException {
2    if (n < 0)
3      throw new IllegalArgumentException( );     // argument must be nonnegative
4    else if (n == 0)
5      return 1;                                  // base case
6    else
7      return n * factorial(n–1);                 // recursive case
8  }
```



CMC UNIVERSITY

Bài tập

Yêu cầu đối với sinh viên (theo nhóm 3 sinh viên):

- Tìm hiểu các thuật toán đệ quy và thủ tục đệ quy

- Viết báo cáo bằng ppt và trình bày các nội dung sau: Khái niệm thuật toán đệ quy, quy trình đệ quy, mối quan hệ giữa đệ quy và quy nạp toán học

- Viết chương trình có đệ quy và không đệ quy để giải từng bài toán sau: tính chuỗi Fibonacci, tìm ước chung lớn nhất của 2 số nguyên dương, tìm tất cả các hoán vị của n phần tử của dãy số. Phân tích và đánh giá độ phức tạp thời gian của thuật toán. So sánh hiệu quả của thuật toán đệ quy và không đệ quy


CMC UNIVERSITY
THANK YOU