ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

NHẬP MÔN LẬP TRÌNH

Bài thực hành 07

ĐỆ QUY

Giảng viên thực hành: Lê Đức Khoan

Thành phố Hồ Chí Minh, 05/2025

Mục lục

1 Khái niệm

Đệ quy (recursion) là một kỹ thuật lập trình trong đó một hàm tự gọi lại chính nó để giải quyết bài toán bằng cách chia nhỏ thành các bài toán con có cấu trúc tương tự. Một hàm đệ quy thường gồm hai phần: **trường hợp cơ sở** để dừng việc gọi đệ quy, và **trường hợp đệ quy** để tiếp tục phân rã bài toán.

Trong lập trình, đệ quy rất hữu ích để giải quyết các bài toán có cấu trúc phân cấp hoặc lặp lại theo kiểu tự nhiên, ví dụ như: duyệt cây (cây nhị phân, cây tổng quát), duyệt đồ thị (DFS), thuật toán chia để trị (như Merge Sort, Quick Sort), giải bài toán tối ưu hoá (như Fibonacci, knapsack), và các bài toán quay lui (như tổ hợp, hoán vị, n-queens, Sudoku).

Dù đệ quy giúp viết mã ngắn gọn và trực quan hơn cho nhiều bài toán, nhưng nếu không kiểm soát tốt, nó có thể gây ra tràn ngăn xếp hoặc giảm hiệu suất. Vì vậy, trong một số trường hợp, đệ quy cần được kết hợp với kỹ thuật ghi nhớ (memoization) hoặc chuyển đổi thành vòng lặp để tối ưu hiệu quả.

2 Các bước giải bài toán đệ quy

1. **Xác định bài toán có thể chia nhỏ được không:** Kiểm tra xem bài toán có thể phân rã thành các bài toán con có cùng bản chất với bài toán ban đầu hay không?
2. **Xác định trường hợp cơ sở (base case):** Đây là điều kiện dừng của hàm đệ quy. Nếu thiếu trường hợp cơ sở, chương trình có thể rơi vào vòng lặp vô hạn và gây lỗi tràn ngăn xếp.
3. **Xác định bước đệ quy (recursive case):** Gọi lại chính hàm đang viết với đầu vào nhỏ hơn hoặc đơn giản hơn, sao cho bài toán tiến dần đến trường hợp cơ sở.
4. **Kết hợp kết quả từ các lời gọi đệ quy:** Nếu có nhiều lời gọi đệ quy, cần kết hợp chúng để tạo ra kết quả cuối cùng (ví dụ: cộng lại, chọn giá trị lớn nhất, nối chuỗi,...).
5. **Kiểm tra với dữ liệu nhỏ:** Thử nghiệm với các đầu vào nhỏ như $n = 0, 1, 2$ để kiểm tra tính đúng đắn và theo dõi luồng gọi hàm.

Ví dụ: Viết chương trình tính giai thừa sử dụng đệ quy.

```cpp
#include <iostream>
using namespace std;

int factorial(int n) {
    if (n == 0 || n == 1) {       // Based case
        return 1;
    } else {                      // Call recursion
        return n * factorial(n - 1);
    }
}
```

Phân tích bài toán và giải thuật:

1. **Xác định bài toán có thể chia nhỏ được không:** Bài toán tính giai thừa $n!$ có thể chia nhỏ thành bài toán con là $n \times (n-1)!$, tức là cùng kiểu bài toán nhưng với giá trị nhỏ hơn.

2. **Xác định trường hợp cơ sở (base case):** Khi $n = 0$ hoặc $n = 1$, ta có $n! = 1$. Đây là điều kiện dừng của hàm đệ quy.
3. **Xác định bước đệ quy (recursive case):** Khi $n > 1$, ta gọi lại chính hàm `factorial(n - 1)` và nhân với n.
4. **Kết hợp kết quả từ các lời gọi đệ quy:** Mỗi lần lời gọi đệ quy trả về, ta nhân kết quả đó với n để xây dựng dần giá trị $n!$.
5. **Kiểm tra với dữ liệu nhỏ:** Khi thử với $n = 0$, hàm trả về 1. Với $n = 2$, ta có:

$$2! = 2 \times factorial(1) = 2 \times 1 = 2$$

Qua đó, ta thấy hàm hoạt động đúng cho các đầu vào nhỏ.

3 So sánh đệ quy và vòng lặp

Tiêu chí	Đệ quy	Vòng lặp
Cơ chế	Hàm tự gọi chính nó, dựa trên **ngăn xếp (stack)** để lưu trạng thái.	Sử dụng cấu trúc lặp (for, while) để thực thi khối code nhiều lần.
Hiệu suất	Chậm hơn do tốn chi phí gọi hàm và quản lý ngăn xếp.	Nhanh hơn vì không có chi phí gọi hàm.
Khả năng đọc code	Dễ đọc với các bài toán có cấu trúc đệ quy (VD: cây, phân chia để trị).	Dễ hiểu hơn với các bài toán đơn giản, lặp tuần tự.
Quản lý bộ nhớ	Tốn nhiều bộ nhớ do stack lưu trạng thái hàm. Có thể gây **stack overflow**.	Tối ưu hơn, không phụ thuộc vào stack.
Tính linh hoạt	Phù hợp cho bài toán **phân cấp** hoặc **chia nhỏ vấn đề**.	Phù hợp xử lý tuần tự, tác vụ lặp đơn giản.

4 Sử dụng đề quy và vòng lặp

Khi nào nên dùng đệ quy?

- Khi bài toán có **cấu trúc đệ quy** (VD: duyệt cây, Fibonacci, giai thừa).
- Code cần **ngắn gọn, dễ hiểu** và không quan tâm đến hiệu suất tối ưu.
- Đảm bảo **độ sâu đệ quy nhỏ**, tránh stack overflow.
- Ngôn ngữ hỗ trợ **tối ưu hóa đuôi (tail recursion)** (VD: Haskell, Scheme).

Khi nào nên dùng vòng lặp?

- Khi cần **hiệu suất cao**, xử lý số lượng lớn vòng lặp.
- Bài toán **không có cấu trúc đệ quy** (VD: duyệt mảng, tính tổng).
- Tránh nguy cơ **stack overflow** với dữ liệu đầu vào lớn.
- Cần kiểm soát trực tiếp biến và trạng thái lặp.

5 Các điểm lưu ý khi làm việc với đệ quy

1. **Xác định điều kiện dừng (điều kiện cơ sở)**
   - Đây là điều kiện giúp kết thúc quá trình đệ quy.
   - Nếu không có điều kiện dừng, hàm sẽ gọi lặp vô hạn và gây tràn ngăn xếp (stack overflow).
   - Điều kiện dừng phải được xác định rõ ràng và đúng đắn.
2. **Gọi đệ quy với đầu vào tiến gần hơn đến điều kiện dừng**
   - Mỗi lần gọi đệ quy phải làm cho bài toán tiến gần hơn đến điều kiện cơ sở.
   - Nếu không, hàm sẽ không bao giờ kết thúc.
3. **Tránh đệ quy thừa hoặc không cần thiết**
   - Một số bài toán có thể được giải bằng vòng lặp với hiệu suất tốt hơn.
   - Đệ quy không cần thiết có thể làm tăng độ phức tạp và gây tốn bộ nhớ.
4. **Hiểu rõ ngăn xếp hàm (call stack)**
   - Mỗi lời gọi đệ quy được lưu trên ngăn xếp.
   - Sử dụng đệ quy sâu có thể gây ra lỗi tràn ngăn xếp (stack overflow).
5. **Cân nhắc sử dụng kỹ thuật tối ưu hoá**
   - **Ghi nhớ (memoization):** Lưu kết quả của các lời gọi trước để tránh tính toán lặp lại.
   - **Đệ quy đuôi (tail recursion):** Một số ngôn ngữ tối ưu hóa loại đệ quy này để tránh tràn ngăn xếp.
6. **So sánh với phương pháp lặp**
   - Đối với các bài toán đơn giản như tính tổng, giai thừa, phương pháp lặp thường hiệu quả hơn.
   - Nên lựa chọn giữa đệ quy và vòng lặp dựa trên độ phức tạp và tính rõ ràng của thuật toán.

6 Yêu cầu bài nộp

Sinh viên được cung cấp một thư mục mã nguồn **lab_07** chứa thư viện, các định nghĩa hàm. Mã nguồn đã bao gồm đầy đủ các định nghĩa hàm cho tất cả các bài tập trong Lab 07. Sinh viên thực hiện thực các hàm theo prototype đã được định nghĩa ở file **.cpp** và có thể viết thêm hàm nếu cần thiết.

Chú ý: Sinh viên không được tự ý sửa đổi bất kỳ thành phần nào của mã nguồn mẫu. Nếu mã nguồn biên dịch không thành công khi chấm thì sẽ nhận điểm 0 cho bài tập đó.

Khi nộp bài sinh viên **đổi tên thư mục lab_07 thành MSSV và zip toàn bộ mã nguồn** thành file **MSSV.zip** với MSSV là mã số sinh viên được nhà trường cung cấp. Các thành phần ở trong thư mục phải giống như thư mục được cung cấp ban đầu. Không thêm bất kỳ file nay thư mục nào khác.

Tên file zip mẫu:

```
24123456.zip
```

Tất cả các trường hợp làm sai yêu cầu sẽ nhận điểm 0 cho bài thực hành. Vì thế sinh viên cần đọc kỹ và thực hiện đúng yêu cầu.

7 Hướng dẫn chạy file thực hành lab_07

Tổ chức thư mục của lab_07 như sau:

```
lab_06
   |-- recursion
   |        |-- recursion.h
   |        |-- recursion.cpp
   |
   |-- main.cpp
   |-- Makefile
```

Vì có nhiều file được include vào trong file main để chạy chương trình do đó để nhanh chóng compile và chạy chương trình. Trong bài lab này chúng ta sẽ tiếp cận với giải pháp dùng Makefile để compile và chạy chương trình. Gõ vào terminal để kiểm tra version của Make:

```
make -v
```

Nếu không xuất hiện version thì máy chưa có Make. Khi đó, các bạn có thể tham khảo hướng dẫn cài đặt Make. Sau khi đã cài xong, các bạn đi đến thư mục đang chưa file **Makefile** và có thể thực hiện các câu lệnh sau:

- **make all**: Compile code ở tất cả các file để tạo ra file thực thi (**.exe**). Đây là default command nên nếu bạn gõ **make** thì nó sẽ tự động hiểu là **make all**.
- **make clean**: Để xoá bỏ file thực thi vừa tạo.
- **make run**: Thực hiện compile code và chạy chương trình.

Các câu lệnh trong Makefile bao gồm:

```
1    # Compiler and flags
2    CXX = g++
3    CXXFLAGS = -std=c++17 -Wall -g -I recursion
4
5    # Name of execution file
```

```
6   TARGET = main
7
8   # List of source files
9   SRC = main.cpp recursion/recursion.cpp
10
11  # Default command
12  all:
13          $(CXX) $(CXXFLAGS) $(SRC) -o $(TARGET)
14
15  # Clean command
16  clean:
17          rm -f $(TARGET)
18
19  # Compile and run command
20  run: all
21          ./$(TARGET)
```

- **Câu lệnh số 2**: Khai báo compiler là **g++**.
- **Câu lệnh số 3**: Khai báo thêm các flags khi compile code. Với các tham số sau **-I** chỉ các thư mục chưa file **.h**. Nếu có thêm nhiều thư viện tự định nghĩa ta sẽ thêm tiếp các cặp **-I library_directory**. Mỗi thư mục sẽ đi với một ký tự **-I**.
- **Câu lệnh số 6**: Khai báo tên file **Thực thi** được tạo ra khi compile mã nguồn.
- **Câu lệnh số 13**: Compile mã nguồn và tạo ra file thực thi. Với câu lệnh tương tự trong những tuần trước: **g++ main.cpp -o main**. Câu lệnh này sẽ được chạy khi ta gọi: **make all** hoặc **make**.
- **Câu lệnh số 17**: Xoá file thực thi vừa được tạo ra. Lệnh được thực thi khi gọi: **make clean**.
- **Câu lệnh số 21**: Thực hiện compile và chạy file thực thi. Lệnh được thực thi khi gọi: **make run**. Khi gọi lệnh này sẽ thực hiện hai việc liên tục là **make all** và **./main**. Với **make all** đã được trình bày ở trên và **./main** là để chạy file thực thi.

8 Bài tập

Chú ý: Giải các bài tập dưới đây sử dụng đệ quy. Sinh viên tự sao chép các hàm từ file **.h** sang file **.cpp** để hiện thực.

8.1 Tính giai thừa

Viết hàm đệ quy để tính giai thừa của một số nguyên dương n.

$$n! = n * (n-1) * (n-2) * ... * 1$$

```
1    int compute_factorial(int n);
```

8.2 Tính số Fibonaci thứ n

Viết hàm đệ quy để tính số Fibonacci thứ n, với công thức:

$$\mathbf{F(0)} = 0, \mathbf{F(1)} = 1$$

$$\mathbf{F(n)} = \mathbf{F(n-1)} + \mathbf{F(n-2)}$$

```
1    int fibonaci(int n);
```

8.3 In các số từ 1 đến n giảm dần

Viết chương trình đệ quy in các số từ 1 đến n theo thứ tự giảm dần và cách nhau bằng khoảng trắng.

```
1    void print_reverse(int n);
```

8.4 Đảo ngược chuỗi

Viết chương trình đệ quy đảo ngược chuỗi của một chuỗi cho trước.

```
string reverse_string(const string& s);
```

8.5 Tìm ước chung lớn nhất (GCD) sử dụng đệ quy Euclid

Viết chương trình đệ quy tìm ước chung lớn nhất của hai số nguyên dương a và b sử dụng thuật toán Euclid:

$$GCD(a, b) = GCD(b, a \mod b)$$

$$GCD(a, 0) = a$$

```
int gcd(int a , int b);
```

8.6 Đếm số chữ số

Viết chương trình đệ quy đếm số chữ số của một số nguyên dương n.

```
int count_digits(long long n);
```

8.7 Kiểm tra mảng đối xứng

Viết chương trình đệ quy kiểm tra tính đối xứng của một mảng.

```
bool is_symetric(const int arr[], int left, int right);
```

8.8 Kiểm tra chuỗi đối xứng

Viết chương trình đệ quy kiểm tra tính đối xứng của chuỗi.

```
bool is_symetric_string(const string& s);
```

8.9 Sinh tất cả các hoán vị chuỗi

Viết chương trình đệ quy in ra tất các chuỗi hoán vị của một chuỗi cho trước. Chuỗi ban đầu đảm bảo gồm các ký tự duy nhất, không có trùng lặp.

```
void print_permutation(const string & s);
```

8.10 Đếm đường đi trong ma trận

Viết chương trình đệ quy đếm số đường đi từ góc trên bên trái đến góc dưới bên phải (chỉ được đi sáng phải hoặc xuống dưới) sao cho tổng trên đường đi bằng s.

```
int count_path(int arr[][MAX_COLS], int rows, int cols, int s);
```