**BỘ GIÁO DỤC VÀ ĐÀO TẠO**

**TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP. HCM KHOA ĐIỆN – ĐIỆN TỬ**



**TIỂU LUẬN CẤU TRÚC RỜI RẠC**

**Ứng dụng của đệ quy trong lập trình**

**Nhóm sinh viên**

**Nhữ Đình Thành – 20119283**

**Chu Trương Phương Hiền – 20119266**

**GVHD: Phan Học**

Tp. Hồ Chí Minh, tháng 9 năm 2024

**Bảng phân công công việc**

|  |  |
| --- | --- |
| **Nội dung** | **Thành viên** |
| Tìm hiểu khái niệm và cách hoạt động của đệ quy trong lập trình | Chu Trương Phương Hiền |
| Nghiên cứu một vài bài toán sử dụng đệ quy | Nhữ Đình Thành |
| Minh họa đệ quy giúp giải quyết các bài | Chu Trương Phương Hiền |
| Độ phức của các thuật toán đệ quy | Nhữ Đình Thành |

**Mục lục**

[**1. Giới thiệu** 5](#_Toc183338439)

[**1.1. Định nghĩa đệ quy trong lập trình** 5](#_Toc183338440)

[**1.2. Vai trò của đệ quy trong các giải thuật tối ưu** 5](#_Toc183338441)

[**1.3. Ưu và nhược điểm của đệ quy trong ứng dụng lập trình** 5](#_Toc183338442)

[**2. Cách hoạt động của đệ quy trong lập trình** 6](#_Toc183338443)

[**2.1. Cấu trúc của hàm đệ quy** 6](#_Toc183338444)

[**2.2. Các loại đệ quy** 6](#_Toc183338445)

[**2.2.1. Đệ quy tuyến tính (Linear Recursion)** 6](#_Toc183338446)

[**2.2.2. Đệ quy nhị phân (Binary Recursion)** 7](#_Toc183338447)

[**2.2.3. Đệ quy đuôi (Tail Recursion)** 7](#_Toc183338448)

[**3. Nghiên cứu một vài bài toán sử dụng đệ quy** 8](#_Toc183338449)

[**3.1. Bài toán Fibonacci** 8](#_Toc183338450)

[**3.2. Bài toán Tháp Hà Nội** 10](#_Toc183338451)

[**3.3. Sắp xếp sử dụng đệ quy** 12](#_Toc183338452)

[**3.3.1. Quick Sort** 12](#_Toc183338453)

[**3.3.2. Merge Sort** 14](#_Toc183338454)

[**4. Tính toán độ phức tạp các bài toán** 16](#_Toc183338455)

[**4.1. Bài toán fibonacci** 16](#_Toc183338456)

[**4.1.1. Độ phức tạp thời gian** 16](#_Toc183338457)

[**4.1.2. Độ phức tạp không gian** 16](#_Toc183338458)

[**4.1.3. So sánh với các thuật toán không đệ quy** 16](#_Toc183338459)

[**4.2. Bài toán tháp Hanoi** 17](#_Toc183338460)

[**4.2.1. Độ phức tạp thời gian** 17](#_Toc183338461)

[**4.2.2. Độ phức tạp không gian** 17](#_Toc183338462)

[**4.2.3. So sánh với các thuật toán không đệ quy** 18](#_Toc183338463)

[**4.3. Bài toán Quick Sort** 18](#_Toc183338464)

[**4.3.1. Độ phức tạp thời gian** 18](#_Toc183338465)

[**4.3.2. Độ phức tạp không gian** 18](#_Toc183338466)

[**4.3.3. So sánh với các thuật toán không đệ quy** 19](#_Toc183338467)

[**4.4. Bài toán Merge Sort** 19](#_Toc183338468)

[**4.4.1. Độ phức tạp thời gian** 19](#_Toc183338469)

[**4.4.2. Độ phức tạp không gian** 19](#_Toc183338470)

[**4.4.3. So sánh với các thuật toán không đệ quy** 20](#_Toc183338471)

[**5. Kết luận** 20](#_Toc183338472)

# **1. Giới thiệu**

## **1.1. Định nghĩa đệ quy trong lập trình**

Đệ quy là kỹ thuật lập trình khi một hàm tự gọi lại chính nó để giải quyết bài toán bằng cách chia nhỏ thành các phần đơn giản hơn. Nguyên tắc "chia để trị" được áp dụng, trong đó bài toán lớn được phân chia thành các bài toán con tương tự, tiếp tục cho đến khi đạt đến trường hợp cơ bản (base case) mà có thể giải quyết mà không cần đệ quy nữa. Một đặc điểm quan trọng của đệ quy là mỗi lần gọi phải tiến dần đến trường hợp cơ bản, nếu không chương trình sẽ bị rơi vào vòng lặp vô tận.

## **1.2. Vai trò của đệ quy trong các giải thuật tối ưu**

Đệ quy là công cụ quan trọng trong nhiều thuật toán tối ưu, đặc biệt là các thuật toán chia để trị. Nó cho phép chia nhỏ bài toán lớn thành các bài toán con, giúp tăng hiệu quả xử lý và đơn giản hóa việc giải quyết các vấn đề phức tạp.

Đệ quy đặc biệt hữu ích khi làm việc với các cấu trúc dữ liệu như cây và đồ thị. Trong cây, các thuật toán duyệt như pre-order, in-order, và post-order đều dựa vào đệ quy để xử lý tuần tự từng nhánh con, giúp quản lý các cây dữ liệu lớn dễ dàng hơn. Tương tự, trong đồ thị, đệ quy được sử dụng trong thuật toán tìm kiếm theo chiều sâu (DFS) để duyệt qua các đỉnh và cạnh, đảm bảo không bỏ sót bất kỳ phần tử nào. Nhờ vậy, đệ quy giúp xử lý bài toán phức tạp một cách trực quan và hiệu quả hơn.

## **1.3. Ưu và nhược điểm của đệ quy trong ứng dụng lập trình**

Ưu điểm của đệ quy:

* Đơn giản hóa giải thuật: Đệ quy giúp làm giải thuật dễ hiểu và ngắn gọn, giải quyết các bài toán phức tạp bằng cách chia nhỏ thành các bài toán con tương tự.
* Hiệu quả trong chia để trị: Đệ quy hữu ích trong các bài toán có cấu trúc phân tầng như cây và đồ thị, tự động hóa quá trình xử lý và kết hợp kết quả để giải quyết bài toán lớn hơn.
* Tính trực quan: Đệ quy rất tự nhiên trong các bài toán có cấu trúc phân nhánh, như duyệt cây hoặc bài toán đồ thị.

Nhược điểm của đệ quy:

* Tiêu tốn bộ nhớ: Đệ quy cần lưu trữ trạng thái của mỗi lời gọi hàm trong ngăn xếp, có thể dẫn đến tràn bộ nhớ khi đệ quy lồng nhau nhiều lần.
* Hiệu suất kém: Đệ quy có thể kém hiệu quả khi không tối ưu, như bài toán tính dãy Fibonacci không tối ưu có thể gây lặp lại phép tính không cần thiết.
* Khó kiểm soát và gỡ lỗi: Gỡ lỗi đệ quy có thể khó khăn, đặc biệt khi không có điều kiện dừng rõ ràng, dễ dẫn đến vòng lặp vô hạn hoặc lỗi khó phát hiện.

# **2. Cách hoạt động của đệ quy trong lập trình**

## **2.1. Cấu trúc của hàm đệ quy**

Một hàm đệ quy gồm hai phần: trường hợp cơ sở và trường hợp đệ quy.

* Trường hợp cơ sở: Đây là điều kiện giúp dừng đệ quy. Nếu không có trường hợp cơ sở, hàm sẽ gọi lại vô hạn, dẫn đến tràn bộ nhớ.
* Trường hợp đệ quy: Hàm tự gọi lại chính nó, giải quyết từng bước của bài toán và tiếp tục với dữ liệu mới cho đến khi điều kiện dừng đạt được.

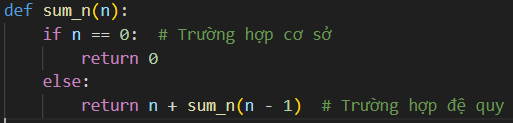
Trong hàm tính giai thừa, khi đầu vào là 0 hoặc 1, hàm trả về 1 mà không gọi lại. Mỗi lần gọi đệ quy, giá trị n giảm dần về trường hợp cơ sở, đảm bảo không xảy ra lặp vô hạn.

## **2.2. Các loại đệ quy**

### **2.2.1. Đệ quy tuyến tính (Linear Recursion)**

Đệ quy tuyến tính là loại đệ quy mà trong đó hàm chỉ tự gọi chính nó một lần trong mỗi lần thực thi. Đây là loại đệ quy đơn giản và thường gặp nhất. Đệ quy tuyến tính được sử dụng trong các bài toán có cấu trúc đơn giản, như tính giai thừa, tìm số Fibonacci, hay tính tổng dãy số.

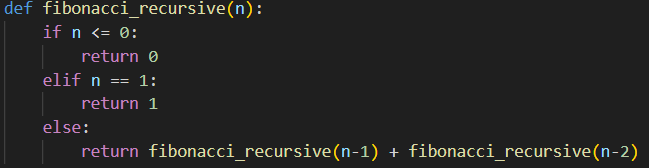
Ví dụ: Hàm tính tổng của một dãy số từ 1 đến n có thể được viết đệ quy như sau:



Trong ví dụ này, mỗi lần gọi hàm sum(n - 1) sẽ đưa giá trị n giảm dần về 0. Đệ quy tuyến tính thường ít tiêu tốn bộ nhớ hơn các loại đệ quy phức tạp hơn, nhưng vẫn có thể gặp giới hạn nếu n quá lớn.

### **2.2.2. Đệ quy nhị phân (Binary Recursion)**

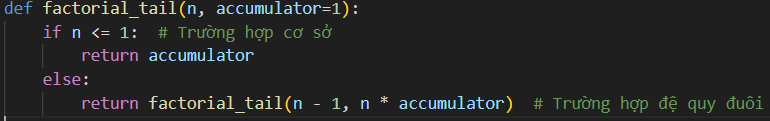
Đệ quy nhị phân là loại đệ quy trong đó hàm tự gọi lại chính nó hai lần hoặc nhiều hơn. Thường gặp trong các bài toán chia dữ liệu thành hai phần hoặc liên quan đến cây nhị phân, như tìm kiếm trong cây hoặc tính dãy Fibonacci. Dãy Fibonacci được định nghĩa là F(n) = F(n-1) + F(n-2), và hàm Fibonacci sẽ tự gọi lại hai lần cho mỗi giá trị của n. Nhược điểm của đệ quy nhị phân là độ phức tạp tính toán tăng nhanh, vì các giá trị nhỏ hơn được tính lại nhiều lần, làm giảm hiệu quả với bài toán lớn nếu không tối ưu hóa.

****

### **2.2.3. Đệ quy đuôi (Tail Recursion)**

Đệ quy đuôi là loại đệ quy mà trong đó hàm gọi lại chính nó ngay ở cuối, mà không cần thực hiện thêm bất kỳ phép tính nào sau khi quay lại từ lời gọi đệ quy. Đệ quy đuôi có thể tối ưu hóa tốt hơn đệ quy thông thường, vì nó không cần lưu trữ các lời gọi đệ quy trước đó trên stack, nhờ đó giảm thiểu nguy cơ lỗi tràn bộ nhớ.

Đệ quy đuôi cho bài toán tính giai thừa có thể được triển khai với tham số accumulator để tích lũy kết quả trong mỗi lần gọi. Accumulator lưu trữ kết quả tích lũy, giúp tính giai thừa mà không cần tính lại. Đệ quy đuôi tối ưu bộ nhớ, giúp xử lý bài toán lớn mà không lo tràn bộ nhớ.



| **Loại đệ quy** | **Số lần gọi lại** | **Đặc điểm** | **Ứng dụng** |
| --- | --- | --- | --- |
| **Đệ quy tuyến tính** | Gọi lại một lần, dễ kiểm soát và tránh tràn bộ nhớ. | Đơn giản, phù hợp bài toán có quy trình rõ ràng như tính tổng, giai thừa. | Tính tổng, giai thừa, đếm phần tử. |
| **Đệ quy nhị phân** | Gọi lại nhiều lần, phân nhánh nhanh, tăng độ phức tạp. | Phân nhánh, thích hợp cho cây nhị phân, dãy Fibonacci, nhưng kém hiệu quả nếu không được tối ưu. | Cây nhị phân, dãy Fibonacci, thuật toán tìm kiếm nhị phân. |
| **Đệ quy đuôi** | Gọi lại một lần, không giữ trạng thái trước, giúp tối ưu bộ nhớ. | Tối ưu bộ nhớ, hiệu quả cho bài toán yêu cầu tính toán lặp lại mà không lo tràn bộ nhớ. | Tính giai thừa, phép tính lặp đơn giản với tham số tích lũy. |

# **3. Nghiên cứu một vài bài toán sử dụng đệ quy**

## **3.1. Bài toán Fibonacci**

Bài toán Fibonacci là một trong những bài toán cổ điển trong lý thuyết số, được sử dụng rộng rãi trong nhiều lĩnh vực như lập trình, sinh học, và kinh tế học. Dãy số Fibonacci được định nghĩa bằng công thức sau:

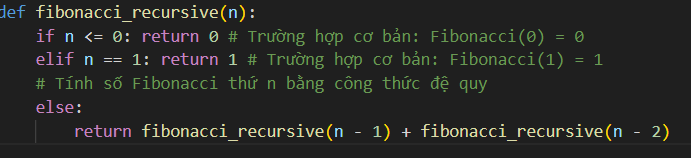
F(0) = 0

F(1) = 1

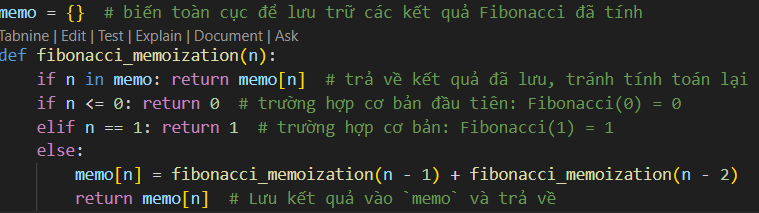
F(n) = F(n-1) + F(n-2) (với n ≥ 2)

Dãy số Fibonacci bắt đầu từ 0, 1, và các số tiếp theo được tính bằng tổng của hai số trước đó. Ví dụ: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

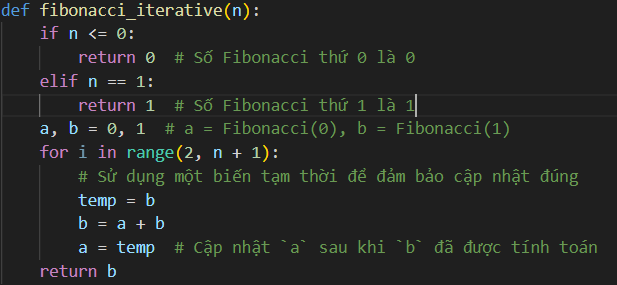
Cách tiếp cần Fibonacci theo đệ quy:

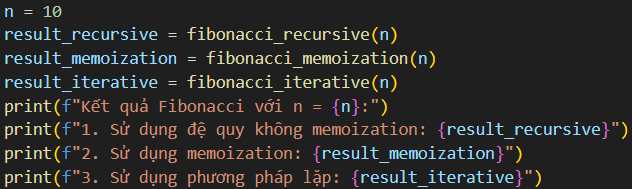


Mặc dù hàm này dễ hiểu và triển khai, nhưng nó có độ phức tạp thời gian là O(điều này có nghĩa là thời gian tính toán sẽ tăng rất nhanh khi n tăng. Để cải thiện hiệu suất, có thể sử dụng kỹ thuật memoization để lưu trữ kết quả của các phép tính đã thực hiện. Kỹ thuật này giúp giảm độ phức tạp thời gian xuống O(n).

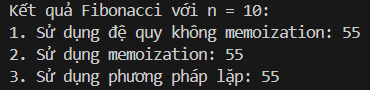


Nếu không sử dụng đệ quy, có thể viết hàm theo cách bên dưới bằng cách lặp trong vòng lặp:





Kết quả cho số Fibonacci thứ 10:



Kết luận

Đệ quy: Phương pháp đệ quy đơn giản và dễ hiểu nhưng không hiệu quả cho n lớn do độ phức tạp thời gian O(Các phép tính bị lặp lại rất nhiều, dẫn đến thời gian chạy lâu. Tuy nhiên, có thể

Không đệ quy: Phương pháp không đệ quy sử dụng vòng lặp có độ phức tạp thời gian O(n) và không tiêu tốn nhiều bộ nhớ (O(1) không gian). Đây là lựa chọn tốt hơn cho các trường hợp n lớn.

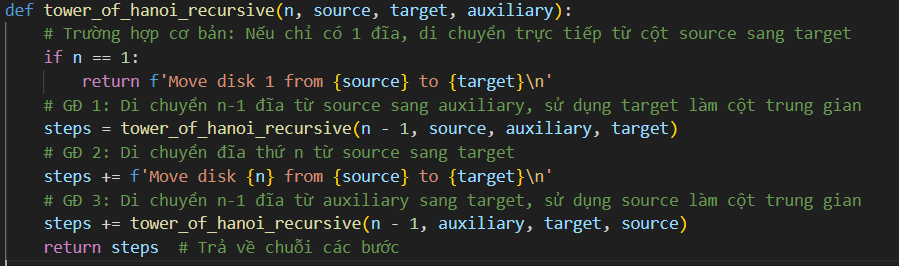
## **3.2. Bài toán Tháp Hà Nội**

Bài toán bao gồm ba cột và một số đĩa có kích thước khác nhau. Mục tiêu là di chuyển tất cả các đĩa từ cột A sang cột C, tuân theo các quy tắc sau:

* Chỉ một đĩa có thể được di chuyển một lần.
* Một đĩa chỉ có thể được đặt lên một đĩa lớn hơn hoặc trên cột trống.
* Không được đặt một đĩa lớn lên một đĩa nhỏ.

Quy trình giải quyết bài toán Tháp Hà Nội với n đĩa có thể được mô tả như sau:

* Di chuyển n-1 đĩa từ cột A sang cột B, sử dụng cột C làm trung gian.
* Di chuyển đĩa lớn nhất từ cột A sang cột C.
* Di chuyển n-1 đĩa từ cột B sang cột C, sử dụng cột A làm trung gian.

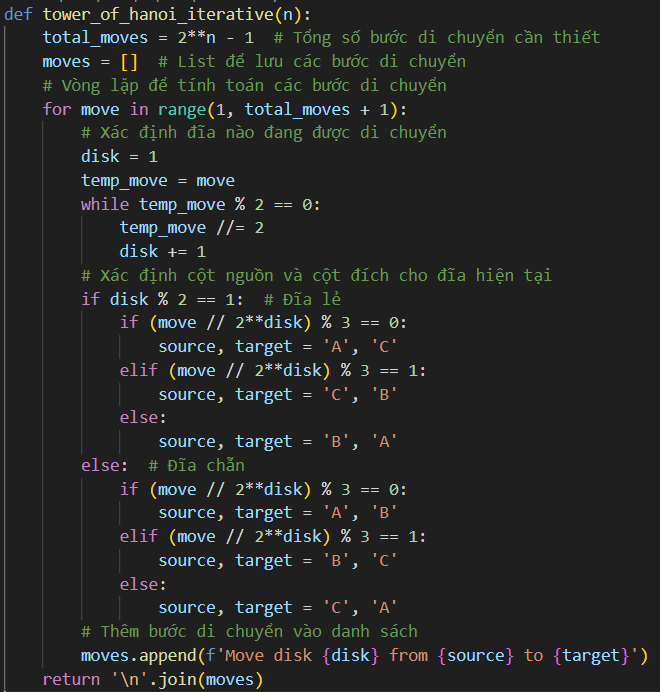


Đây là một thuật toán đệ quy để giải quyết bài toán Tháp Hà Nội với các tham số:

* n: số đĩa cần di chuyển
* source: cột nguồn (cột chứa các đĩa ban đầu)
* target: cột đích (cột cần chuyển đĩa đến)
* auxiliary: cột phụ (cột trung gian để hỗ trợ di chuyển)

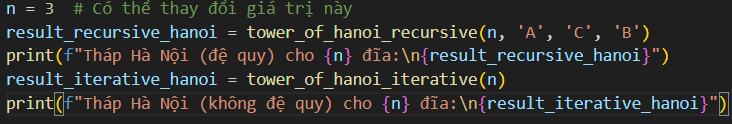
Thuật toán này có độ phức tạp là O(với n là số đĩa. Điều này có nghĩa là số bước di chuyển sẽ tăng gấp đôi mỗi khi ta thêm một đĩa mới.

Tiếp cận theo hướng không phải đệ quy:



Xác định số đĩa di chuyển: Sử dụng biến temp\_move và lặp phép chia temp\_move //= 2 cho đến khi không chia hết cho 2, số lần chia là số thứ tự đĩa cần di chuyển.

Xác định cột nguồn và đích: Dựa vào số thứ tự đĩa và giá trị của move để xác định source và target, giống như logic trong hàm get\_poles.



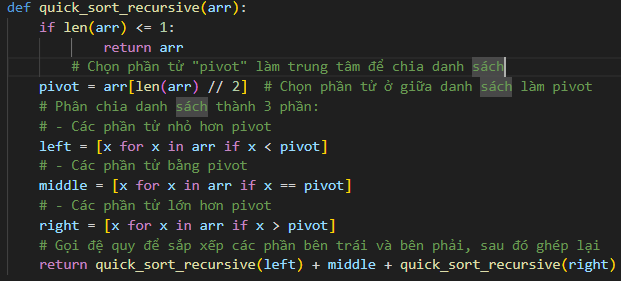
|  |  |
| --- | --- |
|  |  |

Phiên bản không đệ quy hiệu quả về bộ nhớ, sử dụng vòng lặp thay vì stack frame. Thời gian vẫn là nhưng không cần bộ nhớ stack lớn như phương pháp đệ quy, mặc dù phương pháp đệ quy dễ hiểu và dễ triển khai.

## **3.3. Sắp xếp sử dụng đệ quy**

### **3.3.1. Quick Sort**

Sắp xếp là một trong những bài toán quan trọng trong khoa học máy tính. Có nhiều thuật toán sắp xếp khác nhau, và một trong những thuật toán sử dụng đệ quy phổ biến nhất là Quick Sort (Sắp xếp nhanh).



Hàm này sẽ trả về danh sách đã được sắp xếp. Độ phức tạp trung bình của Quick Sort là O(n log n), nhưng trong trường hợp xấu nhất, độ phức tạp có thể lên đến O(n²). Nếu pivot không được chọn tốt. Ví dụ về trường hợp pivot không được chọn tốt là khi danh sách đã được sắp xếp (theo chiều tăng hoặc giảm), hoặc khi danh sách chứa các giá trị giống nhau.

Nếu danh sách đã sắp xếp tăng dần, chọn pivot ở giữa (như trong hàm quick\_sort\_recursive) sẽ dẫn đến một phân hoạch rất mất cân bằng vì mọi phần tử nhỏ hơn pivot sẽ nằm ở bên trái, và phần bên phải sẽ chỉ còn rất ít (hoặc ngược lại).

arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

sorted\_arr = quick\_sort\_recursive(arr)

Trong trường hợp này:

* Khi pivot là phần tử giữa (5 hoặc 6 tùy vào cách chia), danh sách sẽ được chia thành [1, 2, 3, 4] và [5, 6, 7, 8, 9, 10].
* Tiếp tục chia như vậy, sẽ tạo ra một chuỗi các phân hoạch mất cân bằng, dẫn đến việc tốn nhiều bước hơn và độ phức tạp thời gian đạt O(n²).

Nếu danh sách chứa các giá trị giống nhau, phân hoạch sẽ không hiệu quả vì các phần tử sẽ không được chia nhỏ ra mà chỉ dồn về một bên hoặc giữ nguyên ở phần middle.

arr = [5, 5, 5, 5, 5, 5, 5, 5, 5, 5]

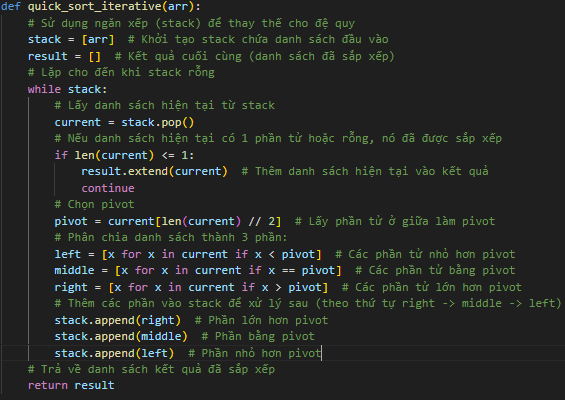
sorted\_arr = quick\_sort\_recursive(arr)

Trong trường hợp này:

* Mỗi lần chọn pivot, danh sách chỉ được phân hoạch thành một phần middle lớn mà không chia nhỏ mảng.
* Kết quả là không có tiến triển đáng kể trong việc thu nhỏ mảng qua mỗi lần đệ quy, và thời gian tính toán trở thành O(n²).

Để tránh trường hợp xấu nhất này, một số kỹ thuật chọn pivot khác đã được phát triển như là chọn pivot ngẫu nhiên giúp giảm khả năng gặp phải chuỗi phân hoạch không cân bằng khi các giá trị đầu vào có mẫu nhất định hoặc chọn median của ba phần tử tức là chọn giá trị trung vị trong ba phần tử (thường là phần tử đầu, giữa và cuối).

Tiếp cận theo hướng không đệ quy sử dụng vòng lặp



Thực hiện in ra kết quả 2 phương án đệ quy và không đệ quy cho 1 mảng ngẫu nhiên 5 phần tử:

array\_size = 5  # Có thể thay đổi kích thước mảng

arr = random.sample(range(10000), array\_size)  # Tạo mảng ngẫu nhiên

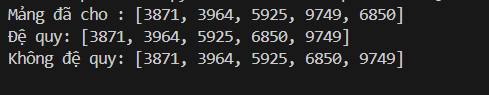
print("Mảng đã cho : " + str(arr))

sorted\_recursive = quick\_sort\_recursive(arr)

sorted\_iterative = quick\_sort\_iterative(arr)

print(f"Đệ quy: {sorted\_recursive}")

print(f"Không đệ quy: {sorted\_iterative}")



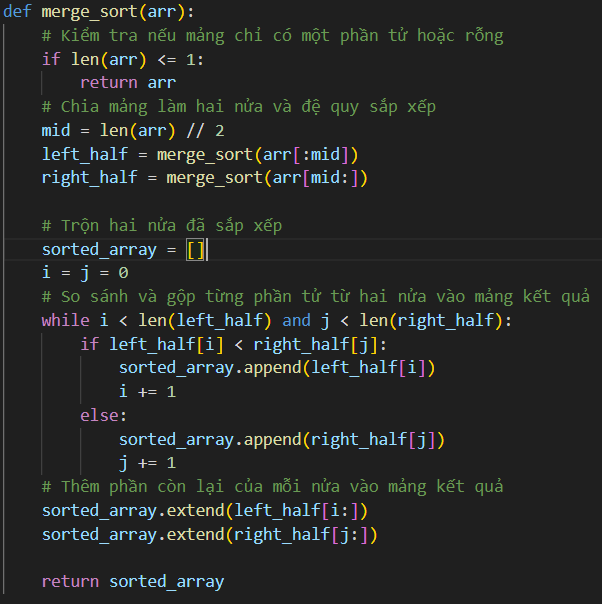
### **3.3.2. Merge Sort**

Bài toán Merge Sort là một thuật toán sắp xếp phổ biến dựa trên kỹ thuật chia để trị (divide and conquer). Thuật toán này chia mảng thành các phần nhỏ hơn, sắp xếp từng phần, rồi gộp lại để tạo ra một mảng đã được sắp xếp.

Bài toán Merge Sort được chia thành 3 bước chính:

1. Chia (Divide): Chia mảng thành hai nửa bằng nhau cho đến khi mỗi phần chỉ còn một phần tử.
2. Trị (Conquer): Sắp xếp từng nửa đã chia bằng cách gộp chúng lại một cách có thứ tự.
3. Kết hợp (Combine): Gộp hai nửa đã sắp xếp thành một mảng đã được sắp xếp hoàn chỉnh.

Tiếp cận bài toán Merge Sort theo hướng đệ quy:



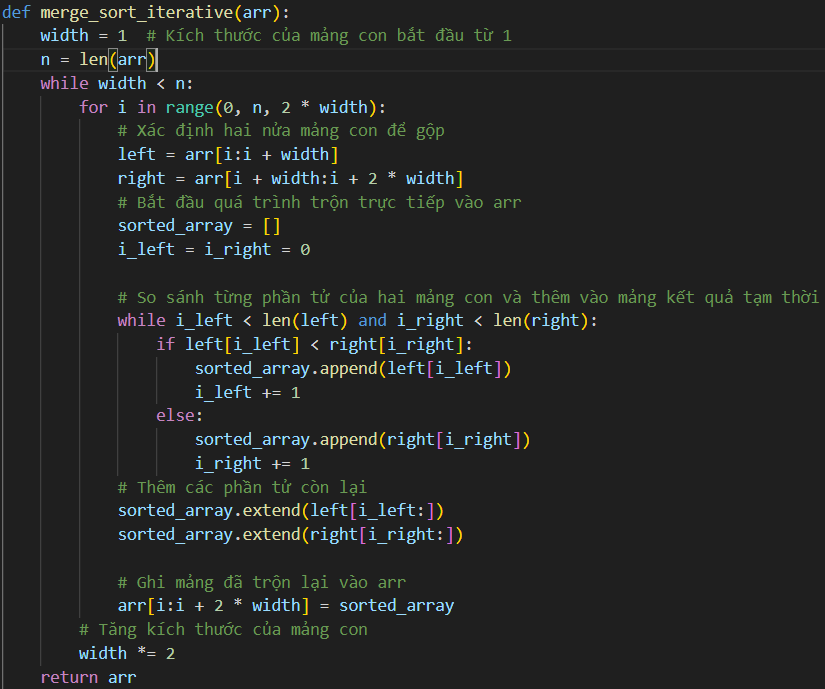
Điều kiện cơ bản: Nếu mảng có độ dài nhỏ hơn hoặc bằng 1, trả về mảng ngay lập tức vì nó đã được sắp xếp.

Chia mảng: Tìm chỉ số giữa (mid), chia mảng làm hai nửa, rồi đệ quy gọi merge\_sort trên từng nửa.

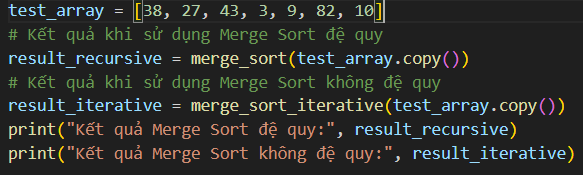
Trộn mảng: Tại bước gộp hai mảng con đã sắp xếp, duyệt qua từng phần tử, so sánh và sắp xếp vào mảng sorted\_array.

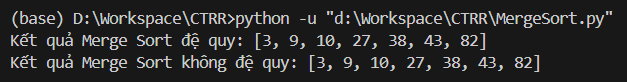
Hoàn tất: Trả về mảng sorted\_array đã được sắp xếp hoàn toàn.

Tiếp cận bài toán Merge Sort theo hướng không đệ quy:



Trong phương pháp này, thay vì chia mảng và trộn bằng cách gọi đệ quy thực hiện lần lượt các bước trộn từ các mảng con có kích thước nhỏ nhất đến lớn nhất. Điều này được thực hiện bằng cách sử dụng các vòng lặp để tăng dần kích thước của các mảng con cho đến khi đạt được mảng gốc đã sắp xếp.





# **4. Tính toán độ phức tạp các bài toán**

## **4.1. Bài toán fibonacci**

### **4.1.1. Độ phức tạp thời gian**

Thuật toán đệ quy Fibonacci tính giá trị F(n) theo công thức:

F(n) = F(n−1) + F(n−2)

Mỗi lời gọi hàm tính lại các giá trị Fibonacci trước đó, dẫn đến tính toán lặp lại nhiều lần.

Độ phức tạp thời gian của thuật toán là:

T(n) = T(n−1) + T(n−2) + O(1)

Trong đó, T(n) là số lời gọi đệ quy, O(1) là chi phí mỗi bước tính.

Thuật toán Fibonacci đệ quy tạo ra cây nhị phân với O lời gọi và không hiệu quả khi n tăng lên.

### **4.1.2. Độ phức tạp không gian**

Độ phức tạp không gian của thuật toán Fibonacci đệ quy phụ thuộc vào độ sâu của ngăn xếp khi gọi đệ quy. Mỗi lời gọi lưu trữ các giá trị của n trên ngăn xếp, tạo thành chuỗi đệ quy sâu O(n). Do đó, độ phức tạp không gian là O(n).

### **4.1.3. So sánh với các thuật toán không đệ quy**

Thuật toán Fibonacci có thể cải tiến bằng cách sử dụng vòng lặp thay vì đệ quy, giúp giảm độ phức tạp thời gian và không gian. Thuật toán không đệ quy sử dụng một vòng lặp để tính dãy Fibonacci, với độ phức tạp thời gian O(n) và không gian O(1).

| **Thuật toán** | **Thời gian** | **Không gian** | **Đặc điểm** |
| --- | --- | --- | --- |
| **Đệ quy** | O( | O(n) | Chậm, tốn nhiều thời gian và không gian khi n lớn. |
| **Không đệ quy** | O(n) | O(1) | Hiệu quả hơn cả về thời gian lẫn không gian, đặc biệt với n lớn. |

Thuật toán không đệ quy hiệu quả hơn về cả thời gian và không gian so với thuật toán đệ quy, đặc biệt khi n lớn.

## **4.2. Bài toán tháp Hanoi**

### **4.2.1. Độ phức tạp thời gian**

Công thức đệ quy của bài toán Tháp Hà Nội là:

T(n) = 2T(n−1) + O(1)

Trong đó:

* T(n) là thời gian di chuyển n đĩa.
* 2T(n−1) là thời gian di chuyển n−1 đĩa qua cọc trung gian và đích.
* O(1) là thời gian di chuyển một đĩa.

Giải công thức đệ quy, ta có:

T(n) = - 1.

Vậy độ phức tạp thời gian của thuật toán là O

### **4.2.2. Độ phức tạp không gian**

Thuật toán Tháp Hà Nội đệ quy sử dụng ngăn xếp để lưu trữ lời gọi hàm đệ quy. Với mỗi lần gọi, số lượng đĩa giảm 1, nên độ sâu ngăn xếp tối đa là n. Do đó, độ phức tạp không gian là O(n).

### **4.2.3. So sánh với các thuật toán không đệ quy**

| **Thuật toán** | **Độ phức tạp thời gian** | **Độ phức tạp không gian** | **Đặc điểm** |
| --- | --- | --- | --- |
| **Đệ quy** | O() | O(n) | Thực hiện 2n−1 bước, yêu cầu lưu trữ trạng thái cho từng cuộc gọi đệ quy. |
| **Không đệ quy** | O() | O(n) | Mô phỏng bước đệ quy bằng ngăn xếp, nhưng chiều sâu ngăn xếp thấp hơn. |

## **4.3. Bài toán Quick Sort**

### **4.3.1. Độ phức tạp thời gian**

Trong trường hợp trung bình, nếu điểm chốt chia mảng thành hai phần bằng nhau, thời gian chạy Quick Sort tuân theo phương trình đệ quy:

T(n) =

T(n) = O(n log n)

Trong trường hợp xấu nhất, khi điểm chốt chia mảng thành hai phần không:

T(n) = T(n−1) + O(n)

T(n) =

Kết luận

Trường hợp trung bình: O(n log n)

Trường hợp xấu nhất:

### **4.3.2. Độ phức tạp không gian**

Độ phức tạp không gian của Quick Sort phụ thuộc vào độ sâu của các lời gọi đệ quy và việc lưu trữ thông tin về điểm chốt và hai phần của mảng tại mỗi mức.

**Trường hợp trung bình và tốt nhất** nếu điểm chốt chia đều mảng, độ sâu đệ quy tối đa sẽ là O(logn). Mỗi lời gọi đệ quy cần lưu trữ một số thông tin nhất định, nên độ phức tạp không gian trung bình và tốt nhất là O(logn).

**Trường hợp xấu nhất** nếu điểm chốt không tốt, Quick Sort sẽ phải xử lý từng phần tử, dẫn đến độ sâu đệ quy tối đa là O(n). Do đó, độ phức tạp không gian xấu nhất sẽ là O(n).

### **4.3.3. So sánh với các thuật toán không đệ quy**

| **Thuật toán** | **Độ phức tạp thời gian trung bình** | **Độ phức tạp thời gian xấu nhất** | **Độ phức tạp không gian trung bình** | **Độ phức tạp không gian xấu nhất** |
| --- | --- | --- | --- | --- |
| **Đệ quy** | O(n log n) | O() | O(log n) | O(n) |
| **Không đệ quy** | O(n log n) | O() | O(log n) | O(n) |

## **4.4. Bài toán Merge Sort**

### **4.4.1. Độ phức tạp thời gian**

Merge Sort gồm ba bước chính:

* Chia mảng: Chia mảng thành hai nửa, tốn O(logn) cho mỗi lần chia.
* Sắp xếp: Đệ quy sắp xếp từng nửa mảng.
* Kết hợp: O(n) để gộp các mảng con đã sắp xếp.

Giải phương trình, ta được:

T(n) = O(n log n)

### **4.4.2. Độ phức tạp không gian**

O(n): Cần không gian bổ sung để lưu mảng phụ trong quá trình trộn.

O(nlogn): Do độ sâu tối đa của stack đệ quy khi chia mảng.

Merge Sort yêu cầu không gian bổ sung O(n) để lưu các mảng con trong quá trình gộp. Mỗi lần gọi đệ quy cần một mảng phụ kích thước bằng n để thực hiện bước merge. Nên độ phức tạp không gian của thuật toán là O(n + logn).

### **4.4.3. So sánh với các thuật toán không đệ quy**

| **Thuật toán** | **Độ phức tạp thời gian** | **Độ phức tạp không gian** | **Có sử dụng ngăn xếp** |
| --- | --- | --- | --- |
| Đệ quy | O(n log n) | O(n + log n) | Có |
| Không đệ quy | O(n log n) | O(n) | Không |

Mặc dù cả hai phiên bản đều có độ phức tạp thời gian giống nhau, phiên bản không đệ quy tiết kiệm bộ nhớ và giảm nguy cơ Stack Overflow, đặc biệt khi xử lý mảng lớn.

# **5. Kết luận**

Đệ quy là công cụ quan trọng trong lập trình, giúp giải quyết bài toán phân chia và chinh phục. Có các loại đệ quy như đệ quy tuyến tính, đệ quy đuôi và đệ quy lồng nhau, mỗi loại phù hợp với các bài toán khác nhau. Đệ quy có thể giảm độ phức tạp thời gian nhưng tốn bộ nhớ do sử dụng ngăn xếp. Mặc dù không tối ưu về bộ nhớ và hiệu suất khi xử lý dữ liệu lớn, đệ quy vẫn hữu ích nhờ tính trực quan và khả năng giải quyết bài toán phân cấp. Việc chọn đệ quy hay không đệ quy phụ thuộc vào yêu cầu và tài nguyên hệ thống.