

TUYỂN TẬP



# THUẬT TOÁN CHO LẬP TRÌNH VIỆN

# 

MÚC TÁC	2
LỜI MỞ ĐẦU	4
I. TOÁN HỌC VÀ TỐI ƯU HÓA	5
1. Bài toán Tổng hai số nguyên lớn được biểu diễn dưới dạng chuỗi	6
2. Bài toán đổi tiền	8
3. Bài toán Cấp số Cộng	10
4. Hoán đổi giá trị của hai biến	12
5. Đặt tên cho cột excel	13
6. Bài toán Cộng thêm 1	14
7. Bài toán Xoay một bức ảnh	16
II. BÀI TOÁN ĐỆ QUY VÀ QUAY LUI	19
8. Đảo ngược một số nguyên	20
9. Bài toán Tính số mũ	21
10. Cộng gộp chữ số	23
11. Bài toán Tháp Hà Nội	25
12. Bài toán N quân hậu và giải pháp đệ quy	27
13. Bài toán Mã đi tuần	31
14. Thuật toán giải Sudoku	36
15. Bài toán Vượt qua Mê cung	42
16. Bài toán Phân chia Tập hợp	47
III. CẤU TRÚC DỮ LIỆU TUYẾN TÍNH	52
17. Tìm phần tử nhỏ thứ nhì trong mảng	52
18. Thuật toán tìm phần tử bị thiếu trong mảng	55
19. Loại bỏ phần tử trùng lặp	56
20. Tìm số chỉ xuất hiện 1 lần trong mảng	57
21. Tìm cặp số có tổng bằng một số cho trước	58
22. Tìm điểm cân bằng của mảng	60
23. Tìm chiều dài của dãy con tăng dần dài nhất trong mảng	62
24. Tìm phần tử thứ k lớn nhất trong một mảng.	64
IV. THUẬT TOÁN SẮP XẾP	66
25. Sắp xếp nổi bọt	67
26. Sắp xếp chèn	69
27. Thuật toán Sắp xếp chọn	71
28. Thuật toán Timsort	73
29. Gộp 2 mảng đã được sắp xếp	75

V. DANH SÁCH LIÊN KẾT	77
30. Đảo ngược các phần tử của linked list	78
31. Đảo ngược một slice bất kỳ trong linked list	81
32. Phát hiện linked list vòng	85
33. Xóa tất cả các node trùng lặp trong một linked list	88
34. Thuật toán tháo vòng cho linked list	90
35. Thuật toán tìm phần tử thứ k tính từ cuối của linked list	93
36. Biểu diễn phép cộng hai số bằng linked list: Một cách tiếp cận thú vị	95
37. Tìm vị trí chính giữa của linked list	98
38. Tìm điểm giao nhau giữa hai linked list	100
39. Thuật toán merge hai linked lists đã được sắp xếp	102
VI. CẤU TRÚC CÂY	105
40. Thuật toán tạo cây nhị phân	106
41.Thuật toán Duyệt cây nhị phân	108
42. Thuật toán tạo cây nhị phân tìm kiếm	110
43. Thuật toán Tìm kiếm phần tử trong cây nhị phân tìm kiếm	113
44. Thuật toán Cân bằng cây nhị phân tìm kiếm	115
45. Thuật toán Thêm Phần tử vào Cây Nhị Phân Tìm Kiếm	118
VII. THAO TÁC VỚI CHUỗI	120
46. Hai chuỗi cùng có các ký tự giống nhau	121
47. Tìm độ dài đoạn chuỗi con chung dài nhất	123
48. Kiểm tra một chuỗi có phải là chuỗi con của một chuỗi khác hay không	125
49. Thuật toán kiểm tra chuỗi đối xứng	127
50. Tìm chuỗi con đối xứng dài nhất	129
LỜI KẾT	132

# LỜI MỞ ĐẦU

Chào mừng bạn đến với cuốn "Cẩm nang 50 thuật toán dành cho lập trình viên" - một tài liệu quý giá dành cho những ai đang tìm kiếm một nguồn tham khảo toàn diện về các thuật toán cơ bản và nâng cao trong lĩnh vực khoa học máy tính và lập trình.

Trong thời đại công nghệ thông tin phát triển, việc nắm vững các thuật toán không chỉ là một lợi thế mà còn là yêu cầu thiết yếu đối với mọi lập trình viên. Thuật toán chính là nền tảng cho mọi ứng dụng, từ những phần mềm đơn giản đến các hệ thống phức tạp, từ ứng dụng di động đến trí tuệ nhân tạo. Hiểu và áp dụng thuật toán một cách hiệu quả sẽ giúp bạn không chỉ giải quyết các vấn đề lập trình một cách tối ưu mà còn nâng cao tư duy logic và khả năng phân tích của mình.

Cuốn cẩm nang này được biên soạn với mục đích cung cấp cho bạn một bộ sưu tập đa dạng gồm 50 thuật toán quan trọng, được lựa chọn kỹ lưỡng từ cơ bản đến nâng cao. Mỗi thuật toán được trình bày một cách chi tiết, bao gồm: Bài toán và ý tưởng cốt lõi, giải thích chi tiết về cách thuật toán hoạt động, mã nguồn minh họa bằng các ngôn ngữ lập trình phổ biến và các ứng dụng thực tế trong lập trình,...

Cho dù bạn là sinh viên ngành Công nghệ Thông tin đang tìm kiếm tài liệu học tập, hay là một lập trình viên muốn củng cố và mở rộng kiến thức của mình, cuốn cẩm nang này sẽ là người bạn đồng hành đắc lực trên con đường khám phá thế giới thuật toán đầy thú vi.

Hy vọng rằng, thông qua việc nghiên cứu và áp dụng các thuật toán trong cuốn cẩm nang này, bạn sẽ không chỉ nâng cao kiến thức chuyên môn mà còn phát triển tư duy phản biện, khả năng giải quyết vấn đề, và sự tự tin trong lập trình.

Hãy bắt đầu hành trình khám phá 50 thuật toán cùng CodeGym nhé!



# 1. BÀI TOÁN TỔNG HAI SỐ NGUYÊN LỚN ĐƯỢC BIỂU DIỄN DƯỚI DẠNG CHUỗI

### - Đề bài:

Cho hai chuỗi ký tự đại diện cho hai số nguyên lớn. Hãy tính tổng của hai số này và trả về kết quả dưới dạng chuỗi ký tự.

### - Ví dụ:

Cho hai số được biểu diễn dưới dạng chuỗi là "123456789123456789" và "987654321987654321". Tổng của hai số này là "111111111111111"

### - Ý tưởng:

- Đảo ngược chuỗi: Đảo ngược hai chuỗi số, hoặc duyệt hai chuỗi theo thứ tự từ cuối về đầu, để tiện cho việc tính toán từng chữ số từ hàng đơn vị.
- Khởi tạo: Khởi tạo một chuỗi rỗng để lưu kết quả, một biến carry để lưu chữ số nhớ.
- Tính tổng từng chữ số: Duyệt qua từng cặp chữ số tương ứng của hai chuỗi (từ hàng đơn vị đến hàng cao nhất).
- Xử lý chữ số nhớ: Công thêm carry vào tổng của hai chữ số hiện tai.
- Cập nhật kết quả và chữ số nhớ: Lấy phần dư của tổng làm chữ số mới của kết quả và cập nhật carry bằng phần nguyên của tổng chia 10.
- Xử lý chữ số nhớ cuối cùng: Nếu sau khi kết thúc vòng lặp, carry vẫn khác 0 thì thêm carry vào đầu kết quả.
- Đảo ngược kết quả: Đảo ngược lại chuỗi kết quả để được kết quả cuối cùng.

### - Triển khai:

### • Mã nguồn C++:

```
string addLargeNumbers(string num1, string num2) {
    string result = "";
    int i = num1.length() - 1, j = num2.length() - 1;
    int carry = 0;
    while (i >= 0 || j >= 0 || carry) {
        int x1 = i >= 0 ? num1[i] - '0' : 0;
        int x2 = j >= 0 ? num2[j] - '0' : 0;
        int sum = x1 + x2 + carry;
       carry = sum / 10;
       result.push_back(sum % 10 + '0');
       i--;
       j--;
    }
    reverse(result.begin(), result.end());
    return result;
}
```

### • Mã nguồn Java:

```
public String addLargeNumbers(String num1, String num2) {
    StringBuilder result = new StringBuilder();
    int i = num1.length() - 1, j = num2.length() - 1;
    int carry = 0;

while (i >= 0 || j >= 0 || carry != 0) {
        int x1 = i >= 0 ? num1.charAt(i) - '0' : 0;
        int x2 = j >= 0 ? num2.charAt(j) - '0' : 0;
        int sum = x1 + x2 + carry;
        carry = sum / 10;
        result.append(sum % 10);
        i--;
        j--;
    }

return result.reverse().toString();
}
```

### • Mã nguồn Python:

```
def add_large_numbers(num1, num2):
    result = ""
    i, j = len(num1) - 1, len(num2) - 1
    carry = 0

while i >= 0 or j >= 0 or carry:
        x1 = int(num1[i]) if i >= 0 else 0
        x2 = int(num2[j]) if j >= 0 else 0
        value = x1 + x2 + carry
        carry = value // 10
        result += str(value % 10)
        i -= 1
        j -= 1

return result[::-1]
```

### - Ưu điểm:

- · Linh hoạt: Có thể xử lý các số nguyên lớn tùy ý.
- Hiệu quả: Độ phức tạp thời gian tuyến tính O(n), trong đó n là độ dài của chuỗi số lớn hơn.

### - Nhược điểm:

• Phức tạp hơn: So với các phép tính số nguyên thông thường, thuật toán này phức tạp hơn do cần xử lý từng chữ số.

### - Ứng dụng:

- Tính toán các phép toán lớn trên số nguyên.
- Giải các bài toán liên quan đến số học.
- · Xây dựng các thư viện tính toán số lớn.

### - Lưu ý:

• Để xử lý các số âm, ta có thể chuyển đổi các số âm về dạng bù 2 trước khi thực hiện phép cộng.

- Nếu cần xử lý phần thập phân, ta có thể thêm các chữ số 0 vào phần thập phân ngắn hơn, kết hợp với tính toán riêng rẽ phần thập phân và phần nguyên sau đó ghép kết quả lại.
- Nếu cần tính tổng của nhiều số lớn, có thể sử dụng các thuật toán nâng cao hơn như thuật toán Karatsuba.

# 2. BÀI TOÁN ĐỔI TIỀN

### - Đề bài:

Cho một số tiền nhất định và mệnh giá của các tờ tiền, hãy tìm ra số lượng tờ tiền ít nhất để tạo thành số tiền amount đó.

### - Ví du:

Cho số tiền là 4, và các tờ tiền có mệnh giá [1, 2, 3]. Kết quả là chúng ta phải sử dụng ít nhất là 2 tờ tiền [2, 2] hoặc [1, 3].

### - Ý tưởng:

Thuật toán tham lam (Greedy Algorithm) giúp cung cấp một giải pháp đơn giản và hiệu quả trong nhiều trường hợp thực tế, ý tưởng là luôn chọn tờ tiền có mệnh giá lớn nhất có thể mà không vượt quá số tiền còn lại.

### - Các bước:

- Sắp xếp mảng các mệnh giá tiền theo thứ tự giảm dần.
- Khởi tạo một biến count để đếm số lượng tờ tiền đã sử dụng và đặt count bằng 0.
- Lặp qua từng mệnh giá tiền, khi số tiền còn lại lớn hơn hoặc bằng mệnh giá hiện tai:
- 1. Tăng count lên 1.
- 2. Trừ mệnh giá hiện tại khỏi số tiền còn lại.

### • Mã nguồn C++:

```
int minCoins(vector<int> coins, int amount) {
    sort(coins.rbegin(), coins.rend()); // Sắp xếp giảm dần
    int count = 0;
    for (int coin : coins) {
        while (amount >= coin) {
            amount -= coin;
            count++;
        }
        if (amount == 0) {
            break;
        }
    }
    return count;
}
```

### Mã nguồn Java

```
public static int minCoins(int[] coins, int amount) {
    Arrays.sort(coins);
    int n = coins.length;
    int count = 0;
    for (int i = n - 1; i >= 0; i--) {
        while (amount >= coins[i]) {
            amount -= coins[i];
            count++;
        }
        if (amount == 0) {
            break;
        }
    }
    return count;
}
```

### Mã nguồn Python

```
def min_coins(coins, amount):
    coins.sort(reverse=True)
    count = 0
    for coin in coins:
        while amount >= coin:
            amount -= coin
            count += 1
        if amount == 0:
            break
    return count
```

### - Giải thích:

- Sắp xếp giảm dần: Đảm bảo luôn chọn tờ tiền có mệnh giá lớn nhất trước.
- Vòng lặp: Tiếp tục trừ đi mệnh giá lớn nhất cho đến khi không thể trừ được nữa, rồi chuyển sang mệnh giá tiếp theo.
- Đếm số lượng tờ tiền: Mỗi lần trù, tăng số lượng tờ tiền lên 1.

### - Lưu ý:

- Thuật toán tham lam không luôn cho kết quả tối ưu với mọi hệ thống tiền tê.
- Để giải quyết bài toán tổng quát hơn, có thể sử dụng các thuật toán quy hoạch đông.

Ví dụ: Với amount = 937 và mảng mệnh giá tiền như trên, thuật toán sẽ trả về kết quả là 8 tờ tiền (1 tờ 500, 2 tờ 200, 1 tờ 100, 4 tờ 5).

### - Các cải tiến:

- Tối ưu hóa: Có thể sử dụng các kỹ thuật tối ưu hóa như tìm kiếm nhị phânđể tìm mệnh giá tiền phù hợp nhanh hơn.
- Xử lý trường hợp không tìm thấy cách đổi: Kiểm tra xem có thể đổi được số tiền amount hay không.

**Câu hỏi:** Nếu yêu cầu bài toán không phải là tìm ra ít số tiền nhất, mà là tìm ra tất cả các tổ hợp có thể, thì thuật toán sẽ thay đổi như thế nào?

# 3. BÀI TOÁN CẤP SỐ CỘNG

### - Mô tả bài toán:

Cho một tập hợp các số nguyên, hãy tìm dãy con dài nhất có thể tạo thành cấp số cộng.

Một cấp số cộng là một dãy số mà hiệu giữa hai số liên tiếp là một hằng số, gọi là công sai (common difference).

### - Ví dụ 1:

- Cho dãy số: [3, 6, 9, 12]
- Kết quả: [2, 6, 9, 12] vì cả 4 phần tử này tạo thành một cấp số cộng với công sai là 3.

### - Ví du 2:

- Cho dãy số: [9, 4, 7, 2, 10]
- Kết quả: [4, 7, 10], vì đây là một cấp số cộng dài nhất, có công sai là 3.

### - Ý tưởng giải bài toán:

Để giải quyết bài toán này, ta có thể sử dụng quy hoạch động (dynamic programming). Ý tưởng chính là:

- Mảng DP: Tạo một mảng DP hai chiều dp[i][j], trong đó dp[i][j] biểu diễn độ dài của dãy con cấp số cộng dài nhất kết thúc tại phần tử thứ i với công sai là j.
- Quét: Duyêt qua từng phần tử trong dãy và cập nhật mảng DP.
- Kết quả: Giá trị lớn nhất trong mảng DP sẽ là độ dài của dãy con cấp số cộng dài nhất.

### - Triển khai:

### • Mã nguồn C++

```
int longestArithmeticSubsequence(vector<int>& nums) {
   int n = nums.size();
   if (n <= 2) {
      return n;
   }
   int ans = 2;
   unordered_map<int, int> dp[n];
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < i; ++j) {
        int diff = nums[i] - nums[j];
            dp[i][diff] = dp[j].count(diff) ? dp[j][diff] + 1 :
2;
      ans = max(ans, dp[i][diff]);
    }
   }
   return ans;
}</pre>
```

### • Mã nguồn Java

### Mã nguồn Python

```
def longest_arithmetic_subsequence(nums):
    n = len(nums)
    dp = [defaultdict(int) for _ in range(n)]
    ans = 2
    for i in range(n):
        for j in range(i):
            diff = nums[i] - nums[j]
            dp[i][diff] = dp[j][diff] + 1
            ans = max(ans, dp[i][diff])
    return ans
```

### - Ứng dung thực tế:

- •Tiết kiệm và đầu tư định kỳ: Khi bạn gửi một số tiền cố định hàng tháng vào một tài khoản tiết kiệm hoặc quỹ đầu tư, tổng số tiền tích lũy sau một thời gian sẽ tạo thành một cấp số cộng. Công sai là số tiền bạn gửi thêm mỗi tháng.
- Lương thưởng tăng dần theo năm: Nếu một nhân viên nhận được mức tăng lương cố định hàng năm, tổng mức lương qua các năm sẽ hình thành một cấp số cộng.
- Quản lý tồn kho: Trong một số mô hình quản lý tồn kho, lượng hàng hóa nhập vào kho hàng kỳ có thể theo một cấp số cộng để đảm bảo đáp ứng nhu cầu tăng dần.
- Tính toán chi phí: Trong một số tình huống, chi phí có thể tăng theo cấp số cộng, chẳng han như chi phí phát sinh theo số giờ làm việc thêm giờ của nhân viên.

# 4. HOÁN ĐỔI GIÁ TRỊ CỦA HAI BIẾN

### - Đề bài:

Cho 2 số nguyên a và b. Hãy hoán đổi giá trị của a và b cho nhau mà không sử dụng đến một biến thứ ba.

### - Các cách giải:

Có một vài cách để giải quyết bài toán này, mỗi cách đều tận dụng các phép toán số học để đạt được mục tiêu.

### 1. Sử dụng phép cộng và trừ:

### - Code mẫu C++

```
// C++
a = a + b;
b = a - b;
a = a - b;
```

### - Giải thích:

- Bước 1: Gán tổng của a và b cho a.
- Bước 2: Gán hiệu của a (lúc này là tổng) và b cho b. Giá trị này chính là giá trị ban đầu của a.
- Bước 3: Gán hiệu của a (lúc này là tổng) và b (lúc này là giá trị ban đầu của a) cho a. Giá trị này chính là giá trị ban đầu của b.

Lưu ý: phương pháp này có thể gặp vấn đề tràn dữ liệu khi tổng của a và b lớn hơn độ rộng bit của kiểu dữ liệu.

### 2. Sử dụng phép XOR (Exclusive OR):

### - Code mẫu C++, Java, Python

```
// C++, Java, Python

a = a \land b;

b = a \land b;

a = a \land b;
```

### - Giải thích:

- XOR là một phép toán bitwise, trả về 1 nếu hai bit tương ứng khác nhau, ngược lại trả về 0. Cụ thể:
  - a ∧ a = 0
  - a  $\land$  0 = a
  - $a \land b = b \land a$
- Bước 1: Gán kết quả XOR của a và b cho a.
- Bước 2: Gán kết quả XOR của a (lúc này là a  $^{\circ}$  b) và b cho b. Thực chất là (a  $^{\circ}$  b)  $^{\circ}$  b = a.
- Bước 3: Gán kết quả XOR của a (lúc này là a  $^{\wedge}$  b) và b (lúc này là a) cho a. Thực chất là (a  $^{\wedge}$  b)  $^{\wedge}$  a = b.
- Phương pháp này nhanh hơn so với sử dụng phép cộng và trừ

# 5. ĐẶT TÊN CHO CỘT EXCEL

### - Đề bài:

Cho một số nguyên đóng vai trò làm số thứ tự của một cột trong một sheet excel, trả về tên cột tương ứng.

### - Ví dụ:

```
1 -> A
2 -> B
...
26 -> Z
27 -> AA
28 -> AB
```

>> Vấn đề này thực chất là bài toán chuyển đổi hệ cơ số 10 sang hệ cơ số 26.

### - Các bước:

- 1. Trừ 1: Vì cột đầu tiên (A) có số thứ tự là 1, ta trừ 1 để bắt đầu từ 0.
- 2. Chia lấy dư: Lần lượt chia số đã trừ 1 cho 26 và lấy dư. Dư sẽ cho biết chữ cái cuối cùng của tên cột.
- 3. Chia lấy nguyên: Lấy phần nguyên của phép chia trên và tiếp tục chia cho 26 cho đến khi thương bằng 0.
- 4. Chuyển số thành chữ cái: Mỗi dư thu được sẽ được chuyển thành chữ cái tương ứng (0 -> |A|, 1 -> |B|, ..., 25 -> |Z|).
- 5. Ghép chuỗi: Ghép các chữ cái thu được theo thứ tự ngược lại để tạo thành tên cột.

### - Triển khai:

### • Mã nguồn C++

```
string convertToExcelColumnName(int colNum) {
   string columnName = "";
   while (colNum > 0) {
      int remainder = colNum % 26;
      columnName = char('A' + remainder) + columnName;
      colNum = (colNum - remainder) / 26;
   }
   return columnName;
}
```

### Mã nguồn Java

```
public String convertToTitle(int columnNumber) {
   StringBuilder result = new StringBuilder();
   while (columnNumber > 0) {
      columnNumber--;
      result.insert(0, (char)('A' + columnNumber % 26));
      columnNumber /= 26;
   }
   return result.toString();
}
```

### • Mã nguồn Python

```
def convertToTitle(columnNumber):
    result = ""
    while columnNumber > 0:
        columnNumber -= 1
        result = chr(ord('A') + columnNumber % 26) + result
        columnNumber //= 26
    return result
```

- -Thuật toán trên có độ phức tạp thời gian là O(log n), khá hiệu quả cho các số thứ tự côt thông thường.
- Có thể ứng dung để:
  - Xử lý dữ liệu Excel: Tự động hóa các tác vụ liên quan đến tên cột.
  - Phát triển các ứng dụng liên quan đến bảng tính.
  - · Các bài toán liên quan đến chuyển đổi số thành chuỗi.

# 6. BÀI TOÁN CỘNG THÊM 1

### - Đề bài:

Cho một số nguyên rất lớn được biểu diễn dưới dạng một mảng các chữ số. Tính kết quả khi cộng số đó thêm 1 đơn vị. Kết quả được biểu diễn dưới dạng một mảng các chữ số.

### - Ví du:

```
Input: digits = [1,2,3]
Output: [1,2,4]
Input: digits = [4,3,2,1]
Output: [4,3,2,2]
Input: digits = [9]
Output: [1,0]
```

### - Ý tưởng:

- Xử lý từ cuối mảng: Bắt đầu từ chữ số cuối cùng (hàng đơn vị) để thực hiện phép cộng.
- Quét và cộng:
- Nếu chữ số hiện tại cộng với 1 nhỏ hơn 10, ta cập nhật giá trị của chữ số đó và dừng quá trình.
- Nếu chữ số hiện tại cộng với 1 bằng 10, ta đặt chữ số hiện tại bằng 0 và tiếp tục cộng 1 cho chữ số kế tiếp.
- Thêm chữ số 1 ở đầu: Nếu sau khi quét hết mảng mà vẫn còn phải cộng 1, ta thêm một chữ số 1 vào đầu mảng kết quả.

### - Triển khai:

### • Mã nguồn C++

```
vector<int> plusOne(vector<int>& digits) {
   int n = digits.size();
   for (int i = n - 1; i >= 0; i--) {
       digits[i]++;

       if (digits[i] < 10) {
           return digits;
       }
       digits[i]

= 0;
   }

   // Nếu tất cả các chữ số đều là 9, thêm 1 vào đầu mảng digits.insert(digits.begin(), 1);
   return digits;
}</pre>
```

### Mã nguồn Java

```
public List<Integer> plusOne(List<Integer> digits) {
   int n = digits.size();
   for (int i = n - 1; i >= 0; i--) {
      digits.set(i, digits.get(i) + 1);
      if (digits.get(i) < 10) {
        return digits;
      }
      digits.set(i, 0);
   }

   // N\u00e9u t\u00e9t c\u00e4c c\u00e4c c\u00e4c v\u00e9 d\u00e9u l\u00e4 9, th\u00e9m 1 v\u00e4c d\u00e9u m\u00e4ng d\u00e4m m\u00e4ng digits.add(0, 1);
   return digits;
}</pre>
```

### • Mã nguồn Python

```
def plusOne(digits):
    n = len(digits)
    for i in range(n - 1, -1, -1):
        digits[i] += 1
        if digits[i] < 10:
            return digits
        digits[i] = 0

# N\u00e9u t\u00e1t c\u00e1 c\u00e3c c\u00e1\u00e3 o\u00e1 d\u00e3 u l\u00e3 9, th\u00e9m 1 v\u00e3o d\u00e1 u m\u00e3ng
    digits.insert(0, 1)
    return digits</pre>
```

### - Ưu điểm:

- Hiệu quả: Thuật toán có độ phức tạp thời gian tuyến tính O(n), trong đó n là số lương chữ số.
- Dễ hiểu: Logic đơn giản, dễ theo dõi.
- Không sử dụng đệ quy: Tránh được vấn đề tràn stack khi xử lý số rất lớn.

### - Lưu ý:

- Biểu diễn số lớn: Sử dụng mảng để biểu diễn số nguyên lớn giúp tránh giới hạn của kiểu số nguyên cơ bản.
- Xử lý trường hợp đặc biệt: Khi tất cả các chữ số đều là 9, cần thêm một chữ số 1 vào đầu mảng.

### - Ứng dụng:

- Xử lý số lớn: Được sử dụng trong các bài toán liên quan đến số nguyên rất lớn.
- Thư viện toán học: Là cơ sở cho các phép toán cơ bản với số lớn.

### - Các cải tiến:

- Sử dụng linked list: Để linh hoạt hơn trong việc thêm/xóa phần tử ở đầu mảng.
- Tối ưu hóa: Có thể tối ưu hóa thêm bằng cách sử dụng các kỹ thuật như bit manipulation.

# 7. BÀI TOÁN XOAY MỘT BỰC ẢNH

### - Đề bài:

Cho một bức ảnh được thể hiện dưới dạng một mảng hai chiều n x n. Xoay bức ảnh đó 90 độ theo chiều kim đồng hồ mà không sử dụng một mảng tam nào khác.

### - Ví du:

Input: matrix = [[1,2,3],[4,5,6],[7,8,9]] Output: [[7,4,1],[8,5,2],[9,6,3]]

1	2	3	7	4	1
4	5	6	8	5	2
7	8	9	9	6	3

### - Ý tưởng:

- Xoay từng lớp: Chia ma trận thành các lớp hình vuông đồng tâm, bắt đầu từ lớp ngoài cùng.
- Di chuyển vào trong: Sau khi xoay xong một lớp, chuyển sang lớp bên trong và lặp lại quá trình.

### - Triển khai:

### • Mã nguồn C++

```
void rotate(vector<vector<int>>& matrix) {
    int n = matrix.size();
    for (int layer = 0; layer < n / 2; layer++) {</pre>
        int first = layer;
        int last = n - 1
 - layer;
        for (int i = first; i < last; i++) {</pre>
            int offset = i - first;
            int top = matrix[first][i];
            // move bottom left to top
            matrix[first][i] = matrix[last - offset][first];
            // move bottom right to top left
              matrix[last - offset][first] = matrix[last][last -
offset];
            // move bottom right to bottom left
            matrix[last][last - offset] = matrix[i][last];
            // move top to bottom right
            matrix[i][last] = top;
    }
}
```

### • Mã nguồn Java

```
public class Solution {
    public void rotate(List<List<Integer>> matrix) {
        int n = matrix.size();
        for (int layer = 0; layer < n / 2; layer++) {</pre>
            int first = layer;
            int last = n - 1 - layer;
            for (int i = first; i < last; i++) {</pre>
                int offset = i - first;
                int
top = matrix.get(first).get(i);
                // move bottom left to top
                 matrix.get(first).set(i, matrix.get(last - off-
set).get(first));
                // move bottom right to top left
                   matrix.get(last - offset).set(first, matrix-
.get(last).get(last - offset));
                // move bottom right to bottom left
                     matrix.get(last).set(last - offset, matrix-
.get(i).get(last));
                // move top to bottom right
                matrix.get(i).set(last, top);
   }
```

### • Mã nguồn Python

```
def rotate(matrix):
    n = len(matrix)
    for layer in range(n // 2):
       first = layer
        last = n - 1 - layer
        for i in range(first, last):
            offset = i - first
            top = matrix[first][i]
            # move bottom left to top
            matrix[first][i] = matrix[last - offset][first]
            # move bottom right to top left
             matrix[last - offset][first] = matrix[last][last -
offset]
            # move bottom right to bottom left
            matrix[last][last - offset] = matrix[i][last]
            # move top to bottom right
            matrix[i][last] = top
```

### - Ưu điểm:

- Hiệu quả về không gian: Không cần sử dụng thêm mảng tạm.
- Thuật toán rõ ràng: Dễ hiểu và triển khai.

### - Nhược điểm:

• Khó hình dung: Đối với những người mới bắt đầu, việc hình dung quá trình xoay có thể hơi khó khăn.

### - Lưu ý:

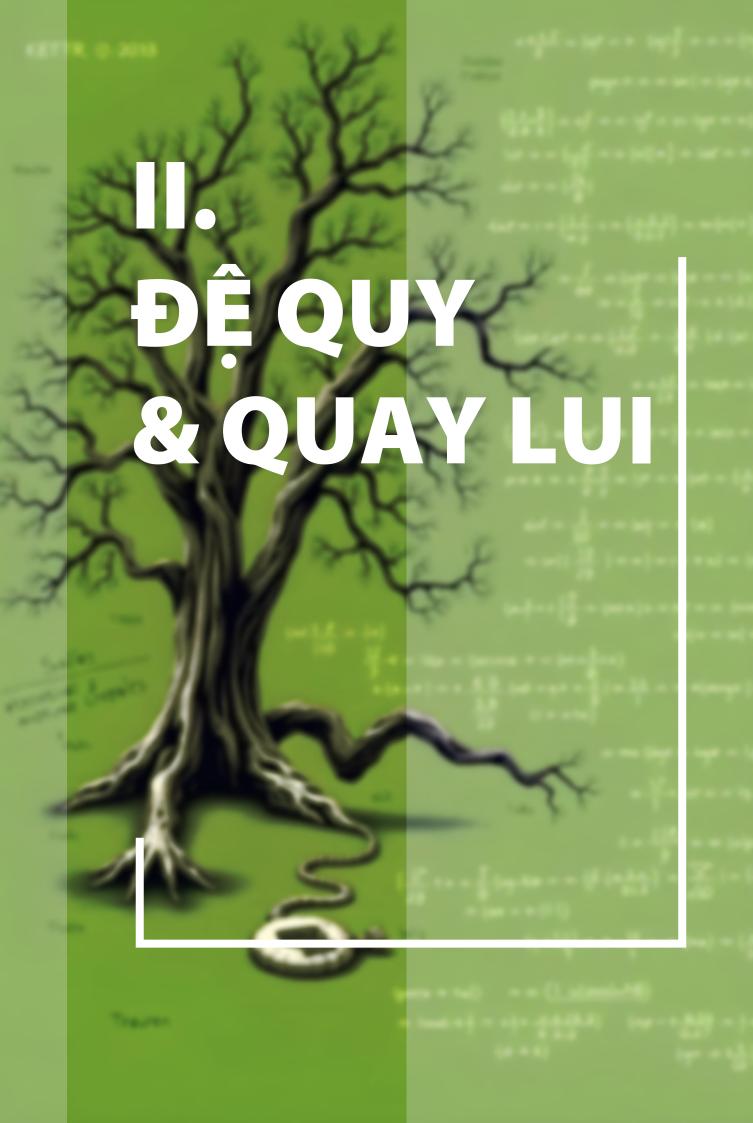
- Ma trận vuông: Thuật toán này chỉ áp dụng cho ma trận vuông.
- In-place: Thuật toán này sẽ làm thay đổi trực tiếp ma trận gốc.

### - Ứng dụng:

- Xử lý ảnh: Xoay ảnh.
- Các thuật toán liên quan đến ma trận: Nhiều thuật toán trên ma trận có thể sử dụng kỹ thuật xoay ma trận.

### - Tối ưu hóa:

• Kiểm tra kích thước: Có thể kiểm tra xem ma trận có phải là ma trận vuông hay không trước khi thực hiện thuật toán.



# 8. ĐẢO NGƯỢC MỘT SỐ NGUYÊN

### - Đề bài:

Cho số nguyên x, trả về số nguyên x được viết theo thứ tự ngược lại.

### - Ví dụ 1:

Input: x = 123 Output: 321

### - Ví dụ 2:

Input: x = -123 Output: -321

### - Ví dụ 3:

Input: x = 120 Output: 21

Mặc dù hoàn toàn có thể sử dụng vòng lặp để giải quyết bài toán này, nhưng đây là một trường hợp rất tốt để thực hành ứng dụng đệ quy.

### - Ý tưởng:

- Điều kiện dừng: Nếu số chỉ còn một chữ số, trả về chính số đó.
- Bước đệ quy:
  - · Lấy chữ số hàng đơn vị của số.
  - Tạo số mới bằng cách nhân số hiện tại với 10 rồi cộng với chữ số hàng đơn vị vừa lấy.
  - · Gọi đệ quy với số đã loại bỏ chữ số hàng đơn vị.

### - Triển khai:

### • Mã nguồn C++

```
int reverse(int x) {
   if (x == 0) return 0;

   int lastDigit = x % 10;
   int reversed = reverse(x / 10);
   return reversed * 10 + lastDigit;
}

• Mã nguồn Java
public int reverse(int x) {
   if (x == 0) return 0;

   int lastDigit = x % 10;
   int reversed = reverse(x / 10);
   return reversed * 10 + lastDigit;
}
```

### • Mã nguồn Python

```
def reverse(x):
    if x == 0:
        return 0

    last_digit = x % 10
    reversed_number = reverse(x // 10)
    return reversed_number * 10 + last_digit
```

### - Ưu điểm của cách giải đệ quy:

- Dễ hiểu: Logic rõ ràng, dễ theo dõi.
- Code gọn gàng: Sử dụng đệ quy giúp code ngắn gọn và dễ đọc.

### - Lưu ý:

- Số âm: Để xử lý số âm, ta có thể chuyển số âm thành số dương, đảo ngược số dương và sau đó đổi dấu kết quả.
- Tràn số: Nếu kết quả quá lớn hoặc quá nhỏ để biểu diễn bằng kiểu dữ liệu nguyên, có thể xảy ra lỗi tràn số.

## 9. BÀI TOÁN TÍNH SỐ MŨ

### - Đề bài:

Triển khai thuật toán tính số mũ n của số nguyên x. Ví dụ với x = 2 và n = 3 thì cho kết quả là 8 (2 mũ 3).

Đây là một bài luyện tập rất tốt để thực hành sử dụng đệ quy.

### - Ý tưởng:

- Đặt điều kiện dừng: Khi n = 0, kết quả là 1.
- Công thức đệ quy:  $x^n = x * x^n 1$

### - Triển khai:

• Mã nguồn C++

```
int power(int x, int n) {
    if (n == 0) {
        return 1;
    } else {
        return x * power(x, n - 1);
    }
}
```

### • Mã nguồn Java

```
public static int power(int x, int n) {
    if (n == 0) {
        return 1;
    } else {
        return x * power(x, n - 1);
    }
}
```

### • Mã nguồn Python

```
def power(x, n):
    if n == 0:
        return 1
    else:
        return x * power(x, n - 1)
```

### - Ưu điểm:

- · Logic rõ ràng, dễ theo dõi.
- Code gọn gàng: Sử dụng đệ quy giúp code ngắn gọn và dễ đọc.

### - Nhược điểm:

Hiệu suất: mặc dù thuật toán đơn giản, rõ ràng, dễ theo dõi, nhưng lưu ý đối các số mũ lớn có thể gây ra tràn stack hoặc chạy chậm do số lần gọi hàm đệ quy quá nhiều.

### - Tối ưu hóa:

Để cải thiện hiệu suất, có thể sử dụng đệ quy chia để trị, hoặc thay thế đệ quy thành sử dụng phép lặp để tính toán số mũ.

Ví dụ về chia để trị: Mã nguồn C++

```
int power(int x, int n) {
    if (n == 0) {
        return 1;
    }
    if (n % 2 == 0) {
        int temp = power(x, n / 2);
        return temp * temp;
    } else {
        return x * power(x, n - 1);
    }
}
```

# 10. CỘNG GỘP CHỮ SỐ

### - Đề bài:

Cho một số nguyên, lặp đi lặp lại việc tính tổng các chữ số cho tới khi kết quả chỉ bao gồm một chữ số.

### - Ví dụ 1:

```
Input: num = 38
Output: 2
```

### - Giải thích:

```
38 --> 3 + 8 --> 11
11 --> 1 + 1 --> 2 // kết quả chỉ còn một chữ số
```

### - Ví dụ 2:

```
Input: num = 0
Output: 0
```

### - Triển khai

### • Mã nguồn C++

```
int sumDigits(int n) {
    if (n < 10) {
        return n;
    }

    int sum = 0;
    while (n > 0) {
        sum += n % 10;
        n /= 10;
    }

    return sumDigits(sum);
}
```

### • Mã nguồn Java

```
public static int sumDigits(int n) {
    if (n < 10) {
        return n;
    }

    int sum = 0;
    while (n > 0) {
        sum += n % 10;
        n /= 10;
    }

    return sumDigits(sum);
}
```

### • Mã nguồn Python

```
def sum_digits(n):
    if n < 10:
        return n

sum = 0
while n > 0:
        sum += n % 10
        n //= 10

return sum_digits(sum)
```

### - Điểm nổi bật:

- Đệ quy: Sử dụng đệ quy để giải quyết bài toán một cách tự nhiên và dễ hiểu.
- Hiệu quả: Mặc dù sử dụng đệ quy, thuật toán này khá hiệu quả vì mỗi lần gọi đệ quy, số lượng chữ số của số cần xử lý đều giảm đi.
- Tính tổng các chữ số: Phần tính tổng các chữ số của một số là một thuật toán cơ bản và thường được sử dụng trong các bài toán liên quan đến số học.

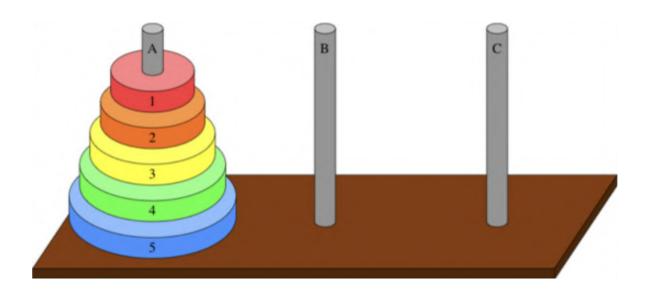
### - Ứng dụng:

- Kiểm tra số Armstrong: Số Armstrong là số mà tổng các lũy thừa các chữ số của nó (mỗi chữ số mũ bằng số lượng chữ số của số đó) bằng chính nó.
- Các bài toán liên quan đến số học: Nhiều bài toán về số học liên quan đến việc phân tích các chữ số của một số.
- Các bài tập về đệ quy: Bài toán này thường được dùng để minh họa cách sử dụng đệ quy trong lập trình.

### - Lưu ý:

Tràn số: mặc dù rất khó xảy ra, nếu số quá lớn, có thể xảy ra tràn số khi tính tổng.

# 11. BÀI TOÁN THÁP HÀ NỘI



### - Mô tả bài toán:

Tháp Hà Nội là một bài toán đệ quy cổ điển trong khoa học máy tính. Bài toán đặt ra như sau:

- Có 3 coc A, B, C.
- Trên cọc A có n đĩa có kích thước giảm dần từ dưới lên.
- Nhiệm vụ là di chuyển tất cả các đĩa từ cọc A sang cọc C, với các quy tắc sau:
  - · Mỗi lần chỉ được di chuyển một đĩa.
  - Đĩa lớn không được đặt lên đĩa nhỏ.

### - Giải thuật đệ quy

Bài toán tháp Hà Nội là một ví dụ điển hình để minh họa sức mạnh và vẻ đẹp của đệ quy trong lập trình. Lời giải bằng đệ quy là rất tự nhiên, bởi:

- 1. Chia nhỏ vấn đề: Bài toán tháp Hà Nội có thể được chia nhỏ thành các bài toán con nhỏ hơn có cùng cấu trúc. Việc di chuyển n đĩa có thể được chia thành các bước: di chuyển n-1 đĩa, di chuyển đĩa lớn nhất và di chuyển n-1 đĩa còn lại.
- 2. Điều kiện dừng: Khi chỉ còn một đĩa, việc di chuyển trở nên đơn giản và trực tiếp.
- 3. Tự gọi lại: Các bài toán con lại vẫn là bài toán tháp Hà Nội với số lượng đĩa ít hơn, cho phép chúng ta gọi đệ quy để giải quyết.

### - Các bước:

- Điều kiện dừng: Nếu chỉ có 1 đĩa, ta di chuyển trực tiếp từ cọc A sang cọc C.
- Bước đệ quy:
  - Di chuyển n-1 đĩa trên cùng từ cọc A sang cọc B (sử dụng cọc C làm trung gian).
  - Di chuyển đĩa lớn nhất từ cọc A sang cọc C.
  - Di chuyển n-1 đĩa từ cọc B sang cọc C (sử dụng cọc A làm trung gian).

### - Triển khai

### • Mã nguồn C++

```
void hanoi(int n, char from, char to, char aux) {
    if (n == 1) {
        cout << "Di chuyển đĩa " << n << " từ " << from << " sang"
        << to << endl;
            return;
        }
        hanoi(n - 1, from, aux, to);
        cout << "Di chuyển đĩa " << n << " từ " << from << " sang "
        << to << endl;
        hanoi(n - 1, aux, to, from);
    }

int main() {
    int n = 3; // Số lượng đĩa
        hanoi(n, 'A', 'C', 'B');
    return 0;
}</pre>
```

### Mã nguồn Java

```
public class HanoiTower {
    public static void towerOfHanoi(int n, char from_rod, char
to_rod, char aux_rod) {
        if (n == 1) {
            System.out.println("Move disk 1 from rod " +
                                 from rod + " to rod " + to rod);
            return;
        towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
        System.out.println("Move disk " + n +
                           " from rod " + from_rod + " to rod " +
to_rod);
       towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
    public static void main(String args[]) {
       int n = 3; // Số lượng đĩa
       towerOfHanoi(n, 'A', 'C', 'B');
}
```

### Mã nguồn Python

```
def TowerOfHanoi(n , from_rod, to_rod, aux_rod):
    if n == 1:
        print("Move disk 1 from rod", from_rod, "to rod", to_rod)
        return
    TowerOfHanoi(n-1, from_rod, aux_rod, to_rod)
    print("Move disk",n,"from rod",from_rod,"to rod",to_rod)
    TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)

n = 3
TowerOfHanoi(n, 'A', 'C', 'B')
```

### - Giải thích:

- •Tham số:
  - n: Số lương đĩa.
  - from: Cọc xuất phát.
  - to: Coc đích.
  - aux: Coc trung gian.
- Điều kiện dừng: Khi chỉ còn 1 đĩa, ta di chuyển trực tiếp.
- · Bước đê quy:
  - Di chuyển n-1 đĩa trên cùng từ cọc xuất phát sang cọc trung gian.
  - Sau đó di chuyển đĩa còn lại từ cọc xuất phát sang cọc đích.
  - Sau đó di chuyển n-1 đĩa ở cọc trung gian sang cọc đích.

# 12. BÀI TOÁN N QUÂN HẬU VÀ GIẢI PHÁP ĐỆ QUY

### - Bài toán:

Tìm tất cả các cách để đặt N quân hậu lên một bàn cờ kích thước N x N sao cho không có hai quân hậu nào đe dọa lẫn nhau.

### - Giải pháp:

Bài toán N quân hậu là một bài toán tìm kiếm toàn diện, đòi hỏi chúng ta phải duyệt qua tất cả các khả năng đặt quân hậu trên bàn cờ để tìm ra các cấu hình hợp lệ. Đệ quy là một công cụ lý tưởng cho những bài toán như vậy vì các lý do sau:

### a. Tính chất đệ quy tự nhiên:

- Chia nhỏ vấn đề: Bài toán N quân hậu có thể được chia nhỏ thành các bài toán con nhỏ hơn: đặt quân hậu vào hàng tiếp theo sau khi đã đặt các quân hậu ở các hàng trước đó.
- Điều kiện dừng: Khi đã đặt đủ N quân hậu mà không vi phạm bất kỳ quy tắc nào, ta tìm được một giải pháp.
- Tự gọi lại: Để tìm các giải pháp khác, ta quay lại hàng trước đó và thử đặt quân hâu ở một vi trí khác.

# b. Ý tưởng chính để không có hai quân hậu nào cùng nằm trên một hàng, một cột hoặc một đường chéo:

- Bởi mỗi hàng chỉ đặt được một quân hậu, ta xét từng hàng, lần lượt thử đặt quân hậu vào các cột.
- Kiểm tra xem vị trí hiện tại có hợp lệ hay không (tức là không có quân hậu nào khác đe doa).
- Nếu vị trí hợp lệ, đặt quân hậu vào đó và gọi đệ quy để đặt thêm quân hậu tiếp theo
- Nếu không tìm thấy vị trí hợp lệ nào, quay lại và thử đặt quân hậu ở cột khác của hàng trước đó.

### - Triển khai

### • Mã nguồn C++

```
bool isSafe(vector<int> &board, int row, int col) {
    // Kiểm tra hàng và các đường chéo
    for (int i = 0; i < row; i++) {
       if (board[i] == col | | abs(board[i] - col) == abs(i - row))
            return false;
    }
    return true;
void solveNQueensUtil(vector<int> &board, int row, vector<vec-</pre>
tor<int>> &result) {
    if (row == board.size()) {
       result.push_back(board);
       return;
    }
    for (int col = 0; col < board.size(); col++) {</pre>
        if (isSafe(board, row, col)) {
            board[row] = col;
            solveNQueensUtil(board, row + 1, result);
            board[row] = -1; // Backtrack
        }
    }
vector<vector<int>> solveNQueens(int n) {
    vector<int> board(n, -1);
    vector<vector<int>> result;
    solveNQueensUtil(board, 0, result);
   return result;
}
```

### • Mã nguồn Java

```
import java.util.ArrayList;
import java.util.List;
public class NQueens {
    public List<List<String>> solveNQueens(int n) {
        List<List<String>> result = new ArrayList<>();
        char[][] board = new char[n][n];
        for (char[] row : board) {
            Arrays.fill(row, '.');
        }
        solveNQueens(board, 0, result);
        return result;
    }
    private boolean isSafe(char[][] board, int row, int col) {
        // Kiểm tra hàng và các đường chéo
        for (int i = 0; i < row; i++) {
            if (board[i][col] == 'Q' ||
                (i - row == Math.abs(board[i] - col))) 
                return false;
        return true;
    }
        private void solveNQueens(char[][] board, int row,
List<List<String>> result) {
        if (row == board.length) {
            result.add(constructSolution(board));
            return;
        }
        for (int col = 0; col < board.length; col++) {</pre>
            if (isSafe(board, row, col)) {
                board[row][col] = 'Q';
                solveNQueens(board, row + 1, result);
                board[row][col] = '.';
        }
    }
    // Chuyển đổi mảng 2 chiều thành danh sách các chuỗi
    private List<String> constructSolution(char[][] board) {
        List<String> res = new ArrayList<>();
        for (char[] row : board) {
            res.add(new String(row));
       return res;
   }
}
```

### • Mã nguồn Python

```
def solveNQueens(n):
    def is_safe(row, col):
        for i in range(row):
            if board[i] == col or abs(board[i] - col) == row - i:
                return False
        return True
    def solve(row):
        if row == n:
            result.append([".Q"[c] * n for c in board])
            return
        for col in range(n):
            if is_safe(row, col):
                board[row] = col
                solve(row + 1)
                board[row] = -1
   board = [-1] * n
    result = []
    solve(0)
    return result
# Ví du:
n = 4
solutions = solveNQueens(n)
for solution in solutions:
    for row in solution:
       print(row)
    print()
```

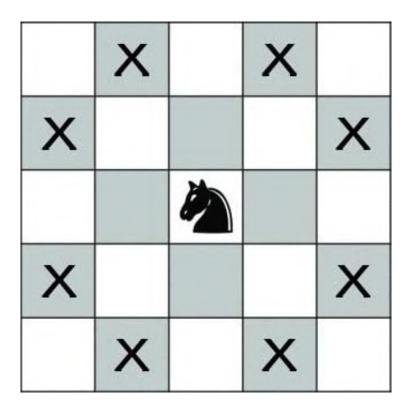
### - Giải thích:

- Hàm isSafe: Kiểm tra xem việc đặt quân hậu ở vị trí (row, col) có hợp lệ hay không bằng cách so sánh với các quân hậu đã đặt ở các hàng trước đó.
- · Hàm đệ quy:
  - Nếu đã đặt đủ N quân hậu, thêm một giải pháp vào danh sách kết quả.
  - Lặp qua các cột của hàng hiện tại, nếu vị trí hợp lệ thì đặt quân hậu vào đó và gọi đệ quy để đặt các quân hậu tiếp theo.
  - Sau khi gọi đệ quy, gỡ bỏ quân hậu để thử các khả năng khác (backtrack).

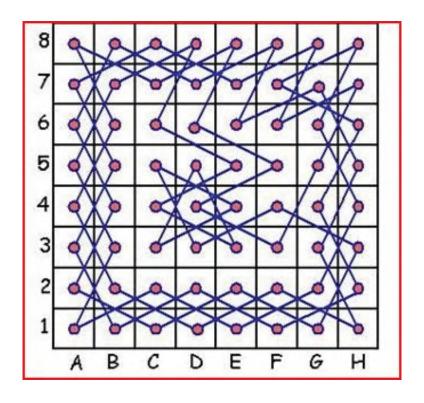
# 13. BÀI TOÁN MÃ ĐI TUẦN

### - Đề bài:

Bài toán Hiệp Sĩ yêu cầu tìm một đường đi của quân mã trên bàn cờ N x N sao cho nó thăm tất cả các ô đúng một lần. Quân mã di chuyển theo hình chữ "L", nghĩa là từ vị trí của mình nó có thể di chuyển được tới các ô có chữ X như trong hình dưới đây:



Ví dụ dưới đây biểu diễn một tuyến đường hợp lệ:



- Giải pháp: Bài toán này có tính chất đệ quy một cách tự nhiên:
  - Chia nhỏ vấn đề: Bài toán có thể được chia nhỏ thành các bài toán con nhỏ hơn: từ một ô bất kỳ, tìm đường đi đến các ô tiếp theo.
  - Điều kiện dừng: Khi đã đi qua tất cả các ô trên bàn cờ hoặc không còn nước đi hợp lê nào, quá trình tìm kiếm kết thúc.
  - Tự gọi lại: Để tìm các đường đi khác, ta quay lại một ô trước đó và thử một nước đi khác.

### - Triển khai

### • Mã nguồn C++

```
#include <iostream>
#include <vector>
using namespace std;
const int dx[] = \{2, 2, -2, -2, 1, 1, -1, -1\};
const int dy[] = \{1, -1, 1, -1, 2, -2, 2, -2\};
bool isSafe(int x, int y, int N, vector<vector<bool>>& visited) {
   return (x \ge 0 \&\& x < N \&\& y \ge 0 \&\& y < N \&\& !visited[x][y]);
void solve(int x, int y, int movei, int N, vector<vector<bool>>&
visited, vector<pair<int, int>>& path) {
    if (movei == N * N) {
        // Đã tìm được đường đi
        for (auto p : path) {
           cout << "(" << p.first << ", " << p.second << ") ";
        cout << endl;</pre>
        return;
    }
    visited[x][y] = true;
    path.push_back({x, y});
    for (int i = 0; i < 8; i++) {
        int newX = x + dx[i];
        int newY = y + dy[i];
        if (isSafe(newX, newY, N, visited)) {
            solve(newX, newY, movei + 1, N, visited, path);
        }
    }
    visited[x][y] = false;
    path.pop_back();
int main() {
    int N = 8; // Kích thước bàn cờ
    int startX = 0, startY = 0; // Điểm bắt đầu
    vector<vector<bool>> visited(N, vector<bool>(N, false));
    vector<pair<int, int>> path;
```

```
solve(startX, startY, 1, N, visited, path);
return 0;
}
```

### • Mã nguồn Java

```
import java.util.ArrayList;
import java.util.List;
public class KnightTour {
    2 } ;
    private static final int[] yMove = \{1, 2, 2, 1, -1, -2, -2, 
-1};
   private int N; // Kích thước bàn cờ
   private int[] x; // Mảng lưu trữ tọa độ x của đường đi
   private int[] y; // Mảng lưu trữ tọa độ y của đường đi
   private int movei; // Đếm số nước đi
   public KnightTour(int N) {
       this.N = N;
       x = new int[N * N];
       y = new int[N * N];
       movei = 1; // Bắt đầu từ nước đi thứ 1
    }
   public void solveKT(int x, int y, int movei) {
       this.x[movei] = x;
       this.y[movei] = y;
       if (movei == N * N) {
           // Đã tìm được đường đi
           printSolution();
           return;
        }
       for (int k = 0; k < 8; k++) {
           int nextX = x + xMove[k];
           int nextY = y + yMove[k];
           if (isSafe(nextX, nextY)) {
               solveKT(nextX, nextY, movei + 1);
        }
       // Backtrack
       movei--;
    }
   private boolean isSafe(int x, int y) {
       return x >= 0 && x < N && y >= 0 && y < N &&
              (x[0] == -1 || (x[movei] != x || y[movei] != y));
    }
   private void printSolution() {
       for (int i = 0; i < N * N; i++) {
```

```
System.out.print("(" + x[i] + ", " + y[i] + ") ");
}
System.out.println();
}

public static void main(String[] args) {
  int N = 8; // Kích thước bàn cờ
  KnightTour knight = new KnightTour(N);
  // Bắt đầu từ ô (0, 0)
  knight.solveKT(0, 0, 1);
}
```

### Mã nguồn Python

```
def solve_knight_tour(N, x, y, movei, visited):
  Giải bài toán mã đi tuần bằng đệ quy
 Args:
   N: Kích thước bàn cờ
   x: Tọa độ x hiện tại
   y: Tọa độ y hiện tại
   movei: Số nước đi hiện tại
   visited: Mảng đánh dấu các ô đã đi qua
  if movei == N * N:
    # Đã tìm được một đường đi
    print_solution(visited)
   return True
  # Thử tất cả các nước đi có thể
  for k in range(8):
   next_x = x + x_move[k]
    next_y = y + y_move[k]
    if is_safe(next_x, next_y, N, visited):
     visited[next_x][next_y] = True
      if solve_knight_tour(N, next_x, next_y, movei + 1, visited):
      visited[next_x][next_y] = False # Backtracking
 return False
def is_safe(x, y, N, visited):
  """Kiểm tra xem một ô có hợp lệ để di chuyển đến hay không"""
 return 0 \ll x \ll N and 0 \ll y \ll N and not visited[x][y]
def print_solution(visited):
 """In ra đường đi"""
 for i in range(N):
    for j in range(N):
     print(visited[i][j], end=" ")
   print()
# Các hướng di chuyển của quân mã
```

```
x_move = [2, 1, -1, -2, -2, -1, 1, 2]
y_move = [1, 2, 2, 1, -1, -2, -2, -1]

# Kích thước bàn cờ
N = 8

# Khởi tạo mảng đánh dấu các ô đã đi qua
visited = [[False for _ in range(N)] for _ in range(N)]

# Bắt đầu từ ô (0, 0)
if solve_knight_tour(N, 0, 0, 1, visited):
    print("Đã tìm thấy một đường đi")
else:
    print("Không tìm thấy đường đi")
```

### - Giải thích

- · Bắt đầu từ một ô bất kỳ trên bàn cờ.
- Xét tất cả các ô mà quân mã có thể di chuyển đến từ ô hiện tại.
- Đánh dấu ô hiện tại là đã đi qua.
- · Gọi đệ quy để tìm đường đi từ các ô tiếp theo.
- Nếu không tìm thấy đường đi, quay lại và thử một nước đi khác.
- Nếu đã đi qua tất cả các ô trên bàn cờ thì đã tìm được một đường đi.
- Nếu không tìm được nước đi hợp lệ nào từ ô hiện tại thì quay lại.

### - Lưu ý:

- Hiệu suất: Bài toán mã đi tuần là một bài toán NP-hard, do đó thời gian tính toán có thể rất lâu đối với các bàn cờ lớn.
- Cắt tỉa: Có thể sử dụng các kỹ thuật cắt tỉa để giảm không gian tìm kiếm và tăng hiệu suất.
- Các biến thể: Có nhiều biến thể của bài toán mã đi tuần, ví dụ như tìm tất cả các đường đi, tìm đường đi ngắn nhất, v.v.

### - Ứng dụng:

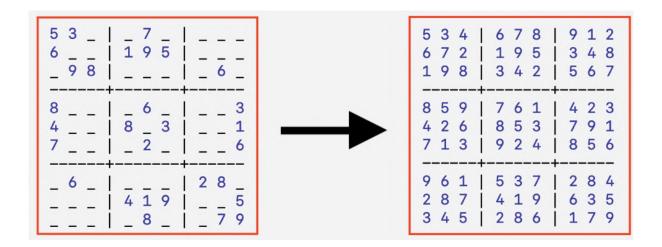
- Tìm kiếm đường đi: Bài toán có ứng dụng trong các bài toán tìm đường đi tối ưu trong đồ thi.
- Tối ưu hóa: Các thuật toán tìm kiếm đường đi của quân mã có thể được áp dụng để giải quyết các bài toán tối ưu hóa khác.

# 14. THUẬT TOÁN GIẢI SUDOKU

### - Hiểu bài toán:

Sudoku là một trò chơi đố số dựa trên việc điền các số từ 1 đến 9 vào một bảng vuông 9x9, chia thành 9 ô vuông con 3x3, sao cho mỗi hàng, mỗi cột và mỗi ô vuông con chỉ chứa một số từ 1 đến 9 duy nhất.

Thuật toán giải Sudoku thường sử dụng kỹ thuật quay lui (backtracking) để tìm kiếm tất cả các khả năng điền số vào các ô trống.



### - Ý tưởng chính của thuật toán:

Tương tự với bài toán quân mã đi tuần, bài toán Sudoku là phù hợp một cách tự nhiên với thuật toán backtracking (quay lui):

- 1. Khởi tạo: Bắt đầu từ ô trống đầu tiên.
- 2. Thử điền số: Thử điền lần lượt các số từ 1 đến 9 vào ô đó.
- 3. Kiểm tra hợp lệ: Kiểm tra xem số vừa điền có vi phạm quy tắc của Sudoku hay không (tức là có xuất hiện trong hàng, cột hoặc ô vuông con tương ứng hay không).
- 4. Đê quy: Nếu số vừa điền hợp lệ, gọi đê quy để điền số cho ô tiếp theo.
- 5. Quay lui: Nếu không tìm thấy số nào hợp lệ cho ô hiện tại, quay lại ô trước đó và thử điền số khác.

#### - Triển khai:

#### • Mã nguồn C++

```
#include <iostream>
#include <vector>
using namespace std;
bool isSafe(vector<vector<int>>& board, int row, int col, int num)
    // Kiểm tra hàng, cột và ô vuông con
    for (int i = 0; i < 9; i++) {
        if (board[row][i] == num | | board[i][col] == num) {
            return false;
    int boxRow = row - row % 3;
    int boxCol = col - col % 3;
    for (int i = boxRow; i < boxRow + 3; i++) {
        for (int j = boxCol; j < boxCol + 3; j++) {
            if (board[i][j] == num) {
                return false;
    }
    return true;
bool solveSudoku(vector<vector<int>>& board) {
    for (int row = 0; row < 9; row++) {
        for (int col = 0; col < 9; col++) {
            if (board[row][col] == 0) {
                for (int num = 1; num <= 9; num++) {
                    if (isSafe(board, row, col, num)) {
                        board[row][col] = num;
                        if (solveSudoku(board)) {
                            return true;
                        board[row][col] = 0; // Quay lui
                return false;
            }
    return true;
}
```

#### • Mã nguồn Java

```
public class SudokuSolver {
    private int[][] board;
    public SudokuSolver(int[][] board) {
        this.board = board;
    public boolean solveSudoku() {
        int row = 0, col = 0;
        if (!findUnassignedLocation(row, col)) {
           // Tất cả các ô đã được điền
            return true;
        }
        for (int num = 1; num <= 9; num++) {
            if (isSafe(row, col, num)) {
                board[row][col] = num;
                if (solveSudoku()) {
                   return true;
                }
                // Quay lui
                board[row][col] = 0;
        return false;
    }
    private boolean findUnassignedLocation(int row, int col) {
        for (row = 0; row < 9; row++) {
            for (col = 0; col < 9; col++) {
                if (board[row][col] == 0) {
                    return true;
        return false;
    }
    private boolean isSafe(int row, int col, int num) {
        // Kiểm tra hàng
        for (int i = 0; i < 9; i++) {
           if (board[row][i] == num) {
                return false;
        }
```

```
// Kiểm tra cột
        for (int i = 0; i < 9; i++) {
            if (board[i][col] == num) {
                return false;
        }
        // Kiểm tra ô 3x3
        int startRow = row - row % 3;
        int startCol = col - col % 3;
        for (int i = startRow; i < startRow + 3; i++) {</pre>
            for (int j = startCol; j < startCol + 3; j++) {</pre>
                if (board[i][j] == num) {
                   return false;
       return true;
   public void printSudoku() {
       for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                System.out.print(board[i][j] + " ");
           System.out.println();
        }
   public static void main(String[] args) {
        int[][] board = {
           // ... (Bạn nhập ma trận Sudoku tại đây)
        };
        SudokuSolver solver = new SudokuSolver(board);
        if (solver.solveSudoku()) {
           solver.printSudoku();
        } else {
           System.out.println("Không có lời giải");
   }
}
```

```
def solve sudoku(board):
    Giải một bàn cờ Sudoku
    Arqs:
        board: Một danh sách các danh sách, đại diện cho bàn cờ
Sudoku
    Returns:
        True nếu tìm thấy lời giải, False nếu không tìm thấy
    # Tìm ô trống đầu tiên
    row, col = find_empty_location(board)
    # Nếu không còn ô trống nào, tức là đã tìm thấy lời giải
    if row == -1:
       return True
    # Thử điền các số từ 1 đến 9 vào ô trống
    for num in range(1, 10):
        if is_safe(board, row, col, num):
            board[row][col] = num
            # Gọi đệ quy để giải tiếp
            if solve_sudoku(board):
                return True
            # Nếu không tìm thấy lời giải, quay lui và thử số khác
            board[row][col] = 0
    return False
def find_empty_location(board):
    Tìm vị trí của ô trống đầu tiên
    Returns:
      Một tuple (row, col) đại diện cho vị trí ô trống, hoặc (-1,
-1) nếu không tìm thấy
    for row in range(9):
        for col in range(9):
            if board[row][col] == 0:
                return row, col
    return -1, -1
def is_safe(board, row, col, num):
    Kiểm tra xem việc đặt số num vào ô (row, col) có hợp lệ hay
không
```

```
Args:
       board: Bàn cờ Sudoku
        row: Hàng
        col: Cột
        num: Số cần kiểm tra
    Returns:
       True nếu hợp lệ, False nếu không
    # Kiểm tra hàng
    for i in range(9):
        if board[row][i] == num:
           return False
    # Kiểm tra cột
    for i in range(9):
        if board[i][col] == num:
           return False
    # Kiểm tra ô 3x3
   box_row = row // 3
    box_col = col // 3
    for i in range(box_row * 3, (box_row + 1) * 3):
        for j in range(box_col * 3, (box_col + 1) * 3):
            if board[i][j] == num:
               return False
   return True
# Ví dụ sử dụng:
board = [
    [3, 0, 6, 5, 0, 8, 4, 0, 0],
    [5, 2, 0, 0, 0, 0, 0, 0, 0],
    # ... (các hàng còn lại)
if solve_sudoku(board):
   for row in board:
       print(row)
else:
   print ("Không tìm thấy lời giải")
```

#### - Giải thích:

- Hàm isSafe: Kiểm tra xem việc điền số num vào ô tại hàng row và cột col có hợp lệ hay không.
- · Hàm solveSudoku:
  - Tìm ô trống đầu tiên.
  - Thử điền lần lượt các số từ 1 đến 9 vào ô đó.
  - · Nếu số hợp lệ, gọi đệ quy để giải tiếp.
  - Nếu không tìm thấy số nào hợp lệ, quay lui và thử số khác.

#### - Lưu ý:

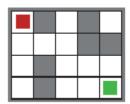
- Hiệu suất: Thuật toán này có thể rất chậm đối với các bàn cờ khó. Có thể sử dụng các kỹ thuật tối ưu hóa như heuristic để cải thiên hiệu suất.
- Nhiều giải pháp: Một số bàn cờ Sudoku có nhiều hơn một lời giải.
- Bàn cờ không có lời giải: Nếu không tìm thấy lời giải nào, có nghĩa là bàn cờ đã cho không hợp lệ.

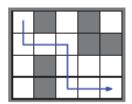
# 15. BÀI TOÁN VƯỢT QUA MÊ CUNG

#### - Mô tả:

Bài toán Mê Cung yêu cầu tìm đường đi từ điểm bắt đầu đến điểm kết thúc trong một mê cung. Mê cung được biểu diễn dưới dạng một ma trận, trong đó các ô có thể là đường đi hoặc chướng ngại vật. Chỉ có thể di chuyển theo phương ngang hoặc thẳng đứng, không thể di chuyển chéo.

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$





Thuật toán phổ biến nhất để giải quyết bài toán này là thuật toán tìm kiếm theo chiều sâu (Depth-First Search - DFS) hoặc thuật toán tìm kiếm theo chiều rộng (Breadth-First Search - BFS). Cả hai thuật toán đều có ưu và nhược điểm riêng, tùy vào yêu cầu của bài toán mà ta chọn thuật toán phù hợp.

#### a. Thuật toán DFS

- Ý tưởng: Khám phá sâu vào một nhánh của cây tìm kiếm trước khi chuyển sang nhánh khác.
- Ưu điểm: Đơn giản, dễ cài đặt.
- Nhược điểm: Có thể mắc kẹt trong các nhánh vô hạn và không tìm được đường đi ngắn nhất.

#### b. Thuật toán BFS

- Ý tưởng: Khám phá tất cả các nút ở một mức độ trước khi chuyển sang mức độ tiếp theo.
- Ưu điểm: Luôn tìm được đường đi ngắn nhất (nếu có đường đi).
- · Nhươc điểm: Tốn nhiều bô nhớ hơn DFS.

#### - Triển khai:

#### • Mã nguồn C++

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
const int ROWS = 5;
const int COLS = 5;
// Kiểm tra xem một ô có hợp lệ để di chuyển đến hay không
bool isValid(int row, int col, vector<vector<bool>>& visited) {
   return (row >= 0) && (row < ROWS) &&
           (col >= 0) && (col < COLS) &&
           visited[row][col] == false;
}
// Hàm DFS để tìm đường đi
void DFS(vector<vector<bool>>& maze, int row, int col, vec-
tor<vector<bool>>& visited, vector<pair<int, int>>& path) {
    // Đánh dấu ô hiện tại là đã đi qua
    visited[row][col] = true;
    path.push_back({row, col});
    // Kiểm tra nếu đã đến đích (bạn có thể thay đổi điều kiện
đích theo yêu cầu)
    if (row == ROWS - 1 \&\& col == COLS - 1) {
        // In ra đường đi (nếu cần)
        for (auto& p : path) {
            cout << "(" << p.first << ", " << p.second << ") ";
        cout << endl;</pre>
        return;
    }
    // Khám phá các ô lân cận
    int rowNbr[] = \{-1, 1, 0, 0\};
    int colNbr[] = \{0, 0, -1, 1\};
    for (int k = 0; k < 4; k++) {
       if (isValid(row + rowNbr[k], col + colNbr[k], visited) &&
maze[row + rowNbr[k]][col + colNbr[k]]) {
            DFS(maze, row + rowNbr[k], col + colNbr[k], visited,
path);
    // Quay lui
    path.pop_back();
int main() {
    vector<vector<bool>> maze = {
        {1, 0, 1, 1, 1},
        {1, 1, 1, 0, 1},
        \{0, 1, 0, 1, 1\},\
```

#### • Mã nguồn Java

```
import java.util.ArrayList;
import java.util.List;
public class MazeSolver {
    private static final int[][] DIRECTIONS = \{\{-1, 0\}, \{1, 0\}, 
\{0, -1\}, \{0, 1\}\};
    public static boolean solveMaze(int[][] maze, int startRow,
int startCol, int endRow, int endCol) {
        int rows = maze.length;
        int cols = maze[0].length;
        boolean[][] visited = new boolean[rows][cols];
       List<int[]> path = new ArrayList<>();
      return dfs(maze, startRow, startCol, endRow, endCol, visit-
ed, path);
   private static boolean dfs(int[][] maze, int row, int col, int
endRow, int endCol, boolean[][] visited, List<int[]> path) {
        // Kiểm tra xem ô hiện tại có hợp lệ không
        if (row < 0 | row >= maze.length | col < 0 | col >=
maze[0].length || maze[row][col] == 0 || visited[row][col]) {
           return false;
        }
        visited[row][col] = true;
        path.add(new int[]{row, col});
        // Kiểm tra nếu đã đến đích
        if (row == endRow && col == endCol) {
            // In ra đường đi (nếu cần)
            for (int[] p : path) {
                System.out.print("(" + p[0] + ", " + p[1] + ") ");
            System.out.println();
            return true;
        }
        // Khám phá các ô lân cận
        for (int[] dir : DIRECTIONS) {
            int newRow = row + dir[0];
```

```
int newCol = col + dir[1];
           if (dfs(maze, newRow, newCol, endRow, endCol, visited,
path)) {
                return true;
            }
        // Quay lui
        path.remove(path.size() - 1);
        return false;
    }
    public static void main(String[] args) {
        int[][] maze = {
                {1, 0, 1, 1, 1},
                {1, 1, 1, 0, 1},
                {0, 1, 0, 1, 1},
                {1, 1, 1, 1, 1},
                {1, 1, 1, 0, 1}
        };
        int startRow = 0, startCol = 0, endRow = 4, endCol = 4;
        if (solveMaze(maze, startRow, startCol, endRow, endCol))
{
            System.out.println("Đường đi tìm thấy!");
        } else {
            System.out.println("Không tìm thấy đường đi!");
    }

    Mã nguồn Python

def solve_maze(maze, start_row, start_col, end_row, end_col):
    rows = len(maze)
    cols = len(maze[0])
```

```
visited = [[False] * cols for _ in range(rows)]
    path = []
    def dfs(row, col):
        # Kiểm tra ô hiện tại có hợp lệ không
         if row < 0 or row >= rows or col < 0 or col >= cols or
maze[row][col] == 0 or visited[row][col]:
            return False
        visited[row][col] = True
        path.append((row, col))
        # Kiểm tra nếu đã đến đích
        if row == end_row and col == end_col:
            print("Đường đi tìm thấy:", path)
            return True
        # Khám phá các ô lân cận
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for dr, dc in directions:
            if dfs(row + dr, col + dc):
                return True
        # Quay lui
        path.pop()
        return False
    return dfs(start_row, start_col)
# Ví dụ sử dụng
maze = [
    [1, 0, 1, 1, 1],
    [1, 1, 1, 0, 1],
    [0, 1, 0, 1, 1],
    [1, 1, 1, 1, 1],
    [1, 1, 1, 0, 1]
start_row, start_col = 0, 0
end_row, end_col = 4, 4
if solve_maze(maze, start_row, start_col, end_row, end_col):
    print("Đường đi tìm thấy!")
   print("Không tìm thấy đường đi!")
```

#### - Giải thích chi tiết:

- Hàm isValid: Kiểm tra xem một ô có hợp lệ để di chuyển đến hay không.
- Hàm DFS:
- Đánh dấu ô hiện tại là đã đi qua.
- Thêm tọa độ ô vào path để lưu lại đường đi.
- Kiểm tra điều kiện đích.
- · Khám phá các ô lân cân bằng cách đê quy gọi hàm DFS.
- · Quay lui khi không tìm thấy đường đi.
- path: Một vector để lưu trữ đường đi, giúp in ra kết quả.

#### - Các cải tiến:

- Có thể tạo một kiểu dữ liệu để để lưu trữ tọa độ của một ô, giúp code gọn gàng
- In đường đi: In ra đường đi tìm được.
- Điều kiện đích: Bạn có thể tùy chỉnh điều kiện đích theo yêu cầu của bài toán.
- Tối ưu hóa: Có thể tối ưu hóa bằng cách sử dụng các kỹ thuật như cắt tỉa tìm kiếm, đánh dấu các ô đã bi loại bỏ.

#### - Các điểm cần lưu ý:

- Mảng visited: Dùng để đánh dấu các ô đã đi qua, tránh lặp vô hạn.
- Điều kiện dùng: Khi gặp tường hoặc ô đã đi qua, hoặc khi tìm thấy đích.
- Khám phá các ô lân cận: Thực hiện đệ quy để khám phá các ô lên, xuống, trái, phải.
- Mở rông:
- BFS: Để tìm đường đi ngắn nhất, bạn có thể sử dụng thuật toán BFS.
- A:\* Để tìm đường đi ngắn nhất hiệu quả hơn trong các mê cung lớn, bạn có thể sử dung thuật toán A\*.
- Mê cung 3 chiều: Mở rộng bài toán sang không gian 3 chiều.
- Mê cung có trọng số: Mỗi ô trong mê cung có một trọng số thể hiện chi phí di chuyển qua ô đó.

# 16. BÀI TOÁN PHÂN CHIA TẬP HỢP

#### - Mô tả:

Bài toán Phân chia Tập hợp (Partition Problem) yêu cầu bạn xác định xem một danh sách các số nguyên có thể được chia thành hai danh sách con có tổng bằng nhau hay không. Đây là một thuật toán thuộc dạng NP-complete.

#### - Ví du:

- Cho các số: [1, 5, 11, 5]
- Kết quả: True, vì có thể chia thành 2 nhóm là [1, 5, 5] và [11], cả hai đều có tổng bằng 11.

#### - Ý tưởng:

Để giải quyết bài toán này, chúng ta thường sử dụng kỹ thuật đệ quy. Ý tưởng cơ bản là:

- Trường hợp cơ sở:
  - Nếu tập hợp rỗng hoặc chỉ còn một phần tử, ta đã có kết quả phân chia.
- · Trường hợp đê quy:
  - · Chọn một phần tử bất kỳ trong tập hợp.
  - Thêm phần tử này vào một trong các tập con đang xây dựng.
  - Gọi đệ quy để phân chia phần còn lại của tập hợp.

•Thử tất cả các khả năng chọn phần tử và thêm vào các tập con khác nhau. Quá trình này tạo ra một cây tìm kiếm, nếu tồn tại một nhánh trong đó thỏa mãn thì ta đã tìm ra lời giải.

#### - Triển khai:

#### • Mã nguồn C++

```
#include <iostream>
#include <vector>
using namespace std;
bool canPartition(vector<int>& nums) {
    int sum = 0;
    for (int num : nums) {
      sum += num;
    // Nếu tổng không chia hết cho 2, không thể chia đều
    if (sum % 2 != 0) {
       return false;
    int target = sum / 2;
    return canPartitionRecursive(nums, target, 0, 0);
}
bool canPartitionRecursive(vector<int>& nums, int target, int
index, int currentSum) {
    if (currentSum == target) {
       return true;
    if (index >= nums.size() || currentSum > target) {
       return false;
    }
    // Chọn hoặc không chọn phần tử hiện tại
   return canPartitionRecursive(nums, target, index + 1, current-
Sum + nums[index]) ||
         canPartitionRecursive(nums, target, index + 1, current-
Sum);
int main() {
    vector<int> nums = \{1, 5, 11, 5\};
    if (canPartition(nums)) {
       cout << "Có thể chia tập hợp thành hai phần có tổng bằng
nhau" << endl;
    } else {
         cout << "Không thể chia tập hợp thành hai phần có tổng
bằng nhau" << endl;
   return 0;
}
```

#### • Mã nguồn Java

```
import java.util.Arrays;
public class PartitionEqualSubsetSum {
    public boolean canPartition(int[] nums) {
        int sum = Arrays.stream(nums).sum();
        if (sum % 2 != 0) {
           return false; // Nếu tổng không chia hết cho 2, không
thể chia đều
        int target = sum / 2;
        return canPartitionRecursive(nums, target, 0, 0);
   private boolean canPartitionRecursive(int[] nums, int target,
int index, int currentSum) {
        if (currentSum == target) {
           return true; // Đã tìm được một cách chia
        if (index >= nums.length || currentSum > target) {
           return false; // Không thể tìm được cách chia
        }
        // Chọn hoặc không chọn phần tử hiện tại
       return canPartitionRecursive(nums, target, index + 1, cur-
rentSum + nums[index]) ||
              canPartitionRecursive(nums, target, index + 1, cur-
rentSum);
    }
    public static void main(String[] args) {
        int[] nums = {1, 5, 11, 5};
       PartitionEqualSubsetSum solution = new PartitionEqualSub-
setSum();
        if (solution.canPartition(nums)) {
           System.out.println("Có thể chia tập hợp thành hai phần
có tổng bằng nhau");
        } else {
            System.out.println("Không thể chia tập hợp thành hai
phần có tổng bằng nhau");
    }
```

```
def can partition(nums):
    Kiểm tra xem có thể chia một tập hợp số nguyên thành hai phần
có tổng bằng nhau hay không.
    Args:
       nums: Một danh sách các số nguyên.
    Returns:
        True nếu có thể chia, False nếu không thể.
    total_sum = sum(nums)
    # Nếu tổng không chia hết cho 2, không thể chia đều
    if total_sum % 2 != 0:
        return False
    target = total_sum // 2
    return can_partition_recursive(nums, target, 0, 0)
def can_partition_recursive(nums, target, index, current_sum):
    Hàm đệ quy để kiểm tra khả năng chia.
    Args:
        nums: Danh sách các số nguyên.
        target: Tổng mục tiêu của mỗi phần.
        index: Chỉ số phần tử hiện tại đang xét.
        current sum: Tổng hiện tại của phần đang xây dựng.
    Returns:
        True nếu tìm thấy cách chia, False nếu không.
    11 11 11
    if current_sum == target:
        return True # Đã tìm được một cách chia
    if index >= len(nums) or current_sum > target:
        return False # Không thể tìm được cách chia
    # Chọn hoặc không chọn phần tử hiện tại
    return can_partition_recursive(nums, target, index + 1, cur-
rent_sum + nums[index]) or \
           can_partition_recursive(nums, target, index + 1, cur-
rent_sum)
# Ví dụ sử dụng
nums = [1, 5, 11, 5]
if can_partition(nums):
    print ("Có thể chia tập hợp thành hai phần có tổng bằng nhau")
else:
    print ("Không thể chia tập hợp thành hai phần có tổng bằng
nhau")
```

#### - Ưu điểm của cách giải này:

- Dễ đọc: Mã nguồn rõ ràng, sử dụng các tên biến gợi nhớ.
- · Linh hoạt: Có thể dễ dàng mở rộng để giải quyết các bài toán phân chia tương tự.

#### - Nhược điểm:

• Hiệu suất thấp: Tương tự như các phiên bản C++ và Java, thuật toán này có thể rất chậm với các tập hợp lớn.

#### - Cải tiến:

- Memoization: Sử dụng một dictionary để lưu trữ kết quả các cuộc gọi đệ quy đã tính toán để tránh tính toán lai.
- Cắt tỉa tìm kiếm: Loại bỏ các nhánh tìm kiếm không hứa hẹn.
- Biến đổi đệ quy thành lặp để tránh tràn stack.
- Bài toán này là một dạng của bài toán Subset Sum và có thể được giải quyết bằng các thuật toán động quy (hiệu quả hơn).



# 17. TÌM PHẦN TỬ NHỎ THỨ NHÌ TRONG MẢNG

#### - Đề bài:

Cho một mảng số nguyên, hãy tìm phần tử nhỏ thứ nhì trong mảng.

#### - Ví dụ:

Mảng [3, 6, 2, 7, 4, 5, 8, 4], phần tử nhỏ thứ nhì trong mảng là 3. Hãy tìm phần tử nhỏ thứ nhì trong mảng.

#### - Giải pháp

Thuật toán này có thể giải quyết bằng cách duyệt mảng và luôn ghi tạm phần tử nhỏ nhất và nhỏ thứ nhì trong mảng.

#### - Triển khai:

#### • Mã nguồn C++

```
int findSecondSmallest(vector<int> nums) {
   if (nums.size() < 2) {
        throw invalid_argument("Màng phải có ít nhất 2 phần tử");
   }

   int smallest = nums[0];
   int secondSmallest = INT_MAX;

   for (int num : nums) {
        if (num < smallest) {
            secondSmallest = smallest;
            smallest = num;
        } else if (num < secondSmallest && num != smallest) {
            secondSmallest = num;
        }
    }

    return secondSmallest;
}</pre>
```

#### Mã nguồn Java

```
public static int findSecondSmallest(int[] nums) {
    if (nums.length < 2) {
        throw new IllegalArgumentException("Mång phải có ít nhất
2 phần tử");
    }

    int smallest = nums[0];
    int secondSmallest = Integer.MAX_VALUE;

    for (int num : nums) {
        if (num < smallest) {
            secondSmallest = smallest;
            smallest = num;
        } else if (num < secondSmallest && num != smallest) {
            secondSmallest = num;
        }
    }

    return secondSmallest;
}</pre>
```

#### Mã nguồn Python

```
def find_second_smallest(nums):
    if len(nums) < 2:
        raise ValueError("Mång phải có ít nhất 2 phần tử")

smallest = nums[0]
    second_smallest = float('inf')

for num in nums:
    if num < smallest:
        second_smallest = smallest
        smallest = num
    elif num < second_smallest and num != smallest:
        second_smallest = num

return second_smallest</pre>
```

#### - Lưu ý:

- Thuật toán trên giả định rằng mảng không có phần tử trùng lặp. Nếu có phần tử trùng lặp, bạn có thể điều chỉnh điều kiện so sánh để loại bỏ các phần tử trùng lặp.
- Thuật toán có thể được mở rộng để tìm phần tử thứ k nhỏ nhất (k-th smallest element).

# 18. THUẬT TOÁN TÌM PHẦN TỬ BỊ THIẾU TRONG MẢNG

#### - Bài toán:

Cho một mảng các số nguyên từ 1 đến n (không có phần tử trùng lặp), nhưng thiếu một số. Nhiệm vụ của chúng ta là tìm ra số đó.

#### - Phương án:

- 1. Tính tổng của tất cả các phần tử có trong mảng.
- 2. Tính tổng của các số từ 1 đến 100.
- 3. Phần tử bị thiếu chính là hiệu giữa tổng của các số từ 1 đến 100 và tổng của các phần tử trong mảng.

#### - Triển khai

#### • Mã nguồn C++

```
int findMissingNumber(vector<int> nums) {
   int n = nums.size();
   int expectedSum = n * (n + 1) / 2;
   int actualSum = 0;
   for (int num : nums) {
      actualSum += num;
   }
   return expectedSum - actualSum;
}
```

#### • Mã nguồn Java

```
public static int findMissingNumber(int[] nums) {
   int n = nums.length;
   int expectedSum = n * (n + 1) / 2;
   int actualSum = 0;
   for (int num : nums) {
      actualSum += num;
   }
   return expectedSum - actualSum;
}
```

```
def find_missing_number(nums):
    n = len(nums)
    expected_sum = n * (n + 1) // 2
    actual_sum = sum(nums)
    return expected_sum - actual_sum
```

# 19. LOẠI BỎ PHẦN TỬ TRÙNG LẶP

#### - Đề bài:

Cho một mảng số nguyên. Hãy trình bày thuật toán để loại bỏ các phần tử trùng lặp ở trong mảng đó.

#### - Phương án:

Bài toán này rất phù hợp để sử dụng HashSet/Set - cấu trúc dữ liệu vốn không cho phép phần tử trùng lặp.

#### - Thực hiện:

- Duyệt qua mảng, thêm từng phần tử vào HashSet/set.
- · Các phần tử trùng lặp sẽ bị bỏ qua khi thêm.
- Tạo một mảng mới từ HashSet/set.

#### - Triển khai:

• Mã nguồn C++

```
vector<int> removeDuplicates(vector<int> nums) {
   unordered_set<int> s;
   for (int num : nums) {
       s.insert(num);
   }
   return vector<int>(s.begin(), s.end());
}
```

#### • Mã nguồn Java

```
public static List<Integer> removeDuplicates(int[] nums) {
    Set<Integer> set = new HashSet<>();
    for (int num : nums) {
        set.add(num);
    }
    return new ArrayList<>(set);
}
```

```
def remove_duplicates(nums):
    return list(set(nums))
```

# 20. TÌM SỐ CHỈ XUẤT HIỆN 1 LẦN TRONG MẢNG

#### - Mô tả bài toán:

Cho một mảng các số nguyên, toàn bộ các số đều xuất hiện 2 lần, ngoại trừ một số duy nhất chỉ xuất hiện 1 lần. Hãy viết thuật toán để tìm ra số đó.

#### - Ví dụ:

- Đầu vào: [2, 3, 5, 4, 5, 3, 4]
- Đầu ra: 2 (Vì chỉ có số 2 xuất hiện một lần trong mảng)

#### - Cách tiếp cận:

Để giải quyết bài toán này, ta có thể sử dụng phương pháp XOR (Exclusive OR) để tìm số khác biệt trong mảng. Phép XOR có tính chất:

- a ^ a = 0 (một số XOR với chính nó bằng 0)
- a ^ 0 = a (một số XOR với 0 bằng chính nó)
- Phép XOR có tính giao hoán và kết hợp.

Với tính chất này, nếu chúng ta XOR tất cả các phần tử trong mảng, các số xuất hiện hai lần sẽ triệt tiêu lẫn nhau, và kết quả cuối cùng sẽ là số xuất hiện một lần.

#### - Triển khai

#### • Mã nguồn C++

```
int findSingleNumber(vector<int> nums) {
   int result = 0;
   for (int num : nums) {
      result ^= num;
   }
   return result;
}
```

#### Mã nguồn Java

```
public int singleNumber(int[] nums) {
   int result = 0;
   for (int num : nums) {
      result ^= num;
   }
   return result;
}
```

```
def single_number(nums):
    result = 0
    for num in nums:
        result ^= num
    return result
```

#### - Giải thích:

- Khi XOR tất cả các phần tử trong mảng, các cặp số xuất hiện 2 lần sẽ triệt tiêu nhau (vì a XOR a=0).
- Cuối cùng, chỉ còn lại số duy nhất xuất hiện một lần trong result.

#### Ví dụ:

Nếu mảng là [2, 2, 1], quá trình tính toán sẽ là:

- result =  $0 \land 2 = 2$
- result =  $2 \land 2 = 0$
- result = 0 ^ 1 = 1 Vây số duy nhất là 1.

#### - Ưu điểm của phương pháp này:

- Hiệu quả: Độ phức tạp thời gian là O(n), độ phức tạp không gian là O(1).
- Đơn giản: Thuật toán dễ hiểu và triển khai.

#### - Lưu ý:

- Thuật toán này chỉ áp dụng cho trường hợp đặc biệt khi tất cả các số xuất hiện đúng 2 lần ngoại trừ một số duy nhất.
- Nếu có nhiều hơn một số xuất hiện một lần hoặc các số xuất hiện với tần suất khác nhau, cần sử dụng các thuật toán phức tạp hơn như sử dụng hash table hoặc sắp xếp:
  - Sử dụng HashMap: Duyệt qua mảng, lưu số lần xuất hiện của mỗi số vào Hash-Map. Cuối cùng, tìm số có giá trị bằng 1.
  - Sắp xếp: Sắp xếp mảng, sau đó duyệt qua mảng so sánh các phần tử liền kề.

# 21. TÌM CẶP SỐ CÓ TỔNG BẰNG MỘT SỐ CHO TRƯỚC

#### - Đề bài:

Cho một mảng các số nguyên và một số nguyên target, hãy tìm tất cả các cặp số trong mảng có tổng bằng target.

#### - Ví du:

- Cho mảng số nguyên: [2, 4, 3, 5, 7, 8, 9], và số target = 10
- Kết quả có 3 cặp số: [(2, 8), (3, 7), (4, 6)]

#### - Giải pháp sử dung HashMap

Thay vì duyệt qua tất cả các khả năng (sử dụng phép lặp chẳng hạn), chúng ta duyệt qua tất cả các số và kiểm tra xem phần bù của số đó có tồn tại trong danh sách những số đã từng duyệt hay không. HashMap rất phù hợp để làm cấu trúc dữ liệu thực hiện thuật toán này.

Để vét cạn kể cả những cặp số giống nhau, chúng ta cũng sẽ lưu giữ số lần xuất hiện của một số, và giảm số lần xuất hiện của số đó mỗi khi tìm thấy một cặp số thỏa mãn.

#### - Triển khai

#### • Mã nguồn C++

```
vector<vector<int>> findPairs(vector<int>& nums, int target) {
   unordered_map<int, int> mp;
   vector<vector<int>> result;

   for (int i = 0; i < nums.size(); i++) {
      int complement = target - nums[i];
      if (mp.count(complement) && mp[complement] > 0) {
          result.push_back({complement, nums[i]});
          mp[complement]--;
      } else {
          mp[nums[i]]++;
      }
   }
}
return result;
}
```

#### • Mã nguồn Java

#### • Mã nguồn Python

```
def find_pairs(nums, target):
    count = {}
    result = []

for num in nums:
    complement = target - num
    if complement in count and count[complement] > 0:
        result.append([complement, num])
        count[complement] -= 1
    else:
        count[num] = count.get(num, 0) + 1
```

#### - Một giải pháp khác

Nếu mảng đã được sắp xếp từ trước, ta có thể sử dụng một giải pháp khác hiệu quả hơn, bằng cách sử dụng hai con trỏ left và right lần lượt trỏ đến đầu và cuối mảng:

- Nếu tổng của hai phần tử tại left và right nhỏ hơn target, tăng left.
- · Nếu tổng lớn hơn target, giảm right.
- Nếu bằng target, thêm cặp số vào kết quả và tăng cả left và right.

## 22. TÌM ĐIỂM CÂN BẰNG CỦA MẢNG

#### - Đề bài:

Cho một mảng bao gồm các phần tử số nguyên dương. Hãy tìm điểm cân bằng của mảng đó.

Điểm cân bằng của mảng là vị trí mà tổng của những phần tử ở trước nó bằng với tổng của những phần tử nằm sau nó. Nếu không tìm được điểm cân bằng thì hãy trả về -1.

#### - Ví dụ:

Mảng [1, 3, 5, 2, 2] có điểm cân bằng là 3 (bắt đầu đếm vị trí đầu tiên là 1), bởi vì tổng của (1 + 3) và tổng của (2 + 2) đều bằng 4.

#### - Ý tưởng:

- •Đặt điểm cân tại 0, khởi tạo tổng bên trái bằng 0, tính tổng các số trong mảng để được tổng bên phải
- Duyệt từng phần tử, cộng phần tử đó vào tổng bên trái, trừ phần tử đó từ tổng bên phải.
- Kiểm tra xem tổng bên trái có bằng tổng bên phải không. Nếu bằng, đã tìm thấy điểm cân bằng.

#### - Triển khai

#### • Mã nguồn C++

```
int findEquilibrium(vector<int> arr) {
  int n = arr.size();
  if (n == 0) {
    return -1; // Mång rỗng
  }

int totalSum = 0;
  for (int num : arr) {
    totalSum += num;
  }

int leftSum = 0;
  for (int i = 0; i < n; i++) {
    if (leftSum == totalSum - leftSum - arr[i]) {
      return i;
    }
    leftSum += arr[i];
  }

return -1; // Không tìm thấy điểm cân bằng
}</pre>
```

#### Mã nguồn Java

```
public static int equilibriumPoint(long arr[], int n) {
   if (n == 0) {
      return -1;
   }

   long totalSum = Arrays.stream(arr).sum();
   long leftSum = 0;
   for (int i = 0; i < n; i++) {
      if (leftSum == totalSum - leftSum - arr[i]) {
        return i + 1; // Vì chỉ số trong Java bắt đầu từ 0
      }
      leftSum += arr[i];
   }

   return -1;
}</pre>
```

#### • Mã nguồn Python

```
def equilibriumPoint(arr, n):
    if n == 0:
        return -1

    total_sum = sum(arr)
    left_sum = 0
    for i in range(n):
        if left_sum == total_sum - left_sum - arr[i]:
            return i + 1
        left_sum += arr[i]
```

#### - Lưu ý:

- Nếu mảng không có điểm cân bằng, hàm sẽ trả về -1.
- Có thể tối ưu hóa thêm bằng cách kiểm tra điều kiện leftSum == totalSum leftSum - arr[i] trước khi cộng arr[i] vào leftSum để tránh phép tính thừa.

# 23. TÌM CHIỀU DÀI CỦA DÃY CON TĂNG DẦN DÀI NHẤT TRONG MẢNG

#### - Đề bài:

Cho một mảng các số nguyên, hãy tìm chiều dài của dãy con tăng dần dài nhất (Longest Increasing Subsequence - LIS) trong mảng đó.

Một dãy con là một dãy được tạo thành từ các phần tử của mảng mà không cần phải liên tiếp nhưng phải tuân thủ trật tự.

#### - Ví du:

Cho mảng số nguyên: [10, 9, 2, 5, 3, 7, 101, 18].

>> Chiều dài của dãy con tăng dần dài nhất là 4.

Giải thích: Các dãy con dài nhất của mảng này là [2, 3, 7, 101], [2, 5, 7, 101], [2, 3, 7, 18] và [2, 5, 7, 18] đều có đô dài là 4.

#### - Ý tưởng:

- Tạo một mảng dp có cùng kích thước với mảng ban đầu.
- dp[i] sẽ lưu trữ chiều dài của dãy con tăng dần dài nhất kết thúc tại vị trí i trong mảng ban đầu.
- Duyệt qua mảng và cập nhật dp[i] dựa trên các giá trị trước đó.
- Phần tử lớn nhất trong mảng dp chính là chiều dài dãy con tăng dần dài nhất, và vi trí của phần tử đó trong dp chính là vi trí bắt đầu của dãy cần tìm.

#### - Triển khai

#### • Mã nguồn C++

#### • Mã nguồn Java

#### - Lưu ý:

- Có các thuật toán khác hiệu quả hơn như sử dụng tìm kiếm nhị phân để giảm độ phức tạp thời gian xuống O(n log n), nhưng chúng phức tạp hơn và thường không cần thiết cho các bài toán cơ bản.
- Thuật toán này tìm ra chiều dài của dãy con tăng dần dài nhất. Nếu muốn tìm ra chính dãy con đó, cần thêm một mảng để lưu trữ các chỉ số tương ứng.

# 24. TÌM PHẦN TỬ THỨ K LỚN NHẤT TRONG MỘT MẢNG.

#### - Đề bài:

Cho một mảng các số nguyên và một số nguyên k, hãy tìm phần tử đứng thứ k trong mảng về độ lớn.

#### - Ví du 1:

- Cho mảng [3, 2, 1, 5, 6, 4], và số k = 2
- Kết quả: 5 (Phần tử lớn thứ hai là 5)

#### - Ví dụ 2:

- Cho mảng [3, 2, 3, 1, 2, 4, 5, 5, 6], và số k = 4
- Kết quả: 4 (Phần tử lớn thứ tư là 4)

#### - Giải pháp sử dụng min-heap

- Ý tưởng: Tạo một min-heap chứa k phần tử lớn nhất trong mảng. Sau khi duyệt hết mảng, phần tử ở đỉnh heap chính là phần tử thứ k lớn nhất.
- Ưu điểm: Hiệu quả hơn với các giá trị k nhỏ so với sắp xếp toàn bộ mảng.
- Nhược điểm: Cần thêm không gian để lưu trữ heap.

#### - Triển khai

#### • Mã nguồn C++

```
int findKthLargest(vector<int>& nums, int k) {
    priority_queue<int, vector<int>, greater<int>> pq;
    for (int num : nums) {
        pq.push(num);
        if (pq.size() > k) {
            pq.pop();
        }
    }
    return pq.top();
}
```

#### • Mã nguồn Java

```
public int findKthLargest(int[] nums, int k) {
    PriorityQueue<Integer> pq = new PriorityQueue<>();
    for (int num : nums) {
        pq.offer(num);
        if (pq.size() > k) {
            pq.poll();
        }
    }
    return pq.peek();
}
```

#### • Mã nguồn Python

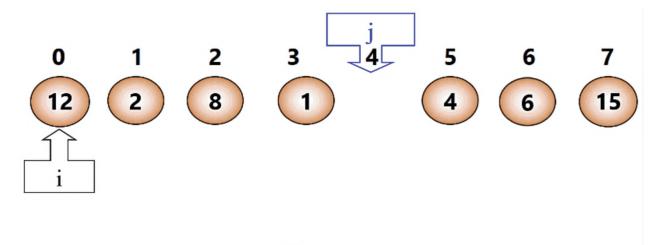
```
def findKthLargest(nums, k):
    heapq.heapify(nums)
    for _ in range(len(nums) - k):
        heapq.heappop(nums)
    return heapq.heappop(nums)
```

#### - Lưu ý:

Có thể sử dụng giải pháp khác, chẳng hạn như sắp xếp trước, như sẽ có ưu/nhược điểm nhất định về độ phức tạp.

# IV. THUẬT TOÁN SĂP XẾP

# 25. SẮP XẾP NỔI BỌT



5

# Nếu a[j]<a[j-1] thì đổi chỗ a[j], a[j-1]

#### - Ý tưởng của thuật toán sắp xếp nổi bọt

Thuật toán sắp xếp nổi bọt (Bubble Sort) là một trong những thuật toán sắp xếp đơn giản và dễ hiểu nhất. Ý tưởng chính của nó là lặp đi lặp lại so sánh từng cặp phần tử liền kề trong một danh sách, và hoán đổi vị trí của chúng nếu chúng không theo thứ tự đúng.

Cụ thể, thuật toán sẽ lặp qua danh sách nhiều lần. Trong mỗi lần lặp:

- · So sánh từng cặp phần tử liền kề.
- Nếu phần tử đứng trước lớn hơn phần tử đứng sau, hoán đổi vị trí của chúng.
- Sau mỗi lần lặp, phần tử lớn nhất sẽ "nổi lên" (như bọt) ở cuối danh sách. Quá trình này được lặp lại cho đến khi toàn bộ danh sách được sắp xếp theo thứ tư tăng dần hoặc giảm dần.

#### - Ví du

Với danh sách ban đầu [5, 2, 9, 1, 7]:

- Lần lặp 1: [2, 5, 1, 7, 9] (hoán đổi 5 và 2)
- Lần lặp 2: [2, 1, 5, 7, 9] (hoán đổi 5 và 1)
- Lần lặp 3: [1, 2, 5, 7, 9] (hoán đổi 2 và 1)
- Lần lặp 4: [1, 2, 5, 7, 9] (không cần hoán đổi)
- >> Như vậy, sau 4 lần lặp, danh sách đã được sắp xếp hoàn toàn.

#### - Triển khai

#### • Mã nguồn C++

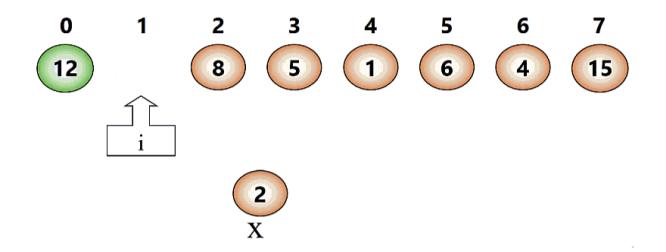
```
void bubbleSort(vector<int>& arr) {
   int n = arr.size();
   for (int i = 0; i < n - 1; i++) {
      for (int j = 0; j < n - i - 1; j++) {
        if (arr[j] > arr[j + 1]) {
            // Đổi chỗ arr[j] và arr[j+1]
            swap(arr[j], arr[j + 1]);
      }
   }
}
```

#### • Mã nguồn Java

```
public void bubbleSort(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n - 1; i++)
        for (int j = 0; j < n - i - 1; j++)
        if (arr[j] > arr[j + 1]) {
            // swap arr[j] and arr[j+1]
            int temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
        }
}
```

## 26. SẮP XẾP CHÈN

#### Chen a[1] vào (a[0], a[1])



#### - Ý tưởng:

Ý tưởng chính của thuật toán sắp xếp chèn là:

- · Chia mảng thành hai phần: phần đã sắp xếp và phần chưa sắp xếp.
- Lần lượt lấy từng phần tử từ phần chưa sắp xếp và chèn vào vị trí thích hợp trong phần đã sắp xếp.

Cụ thể, thuật toán sẽ lặp qua từng phần tử của mảng, bắt đầu từ phần tử thứ hai. Với mỗi phần tử, nó sẽ so sánh và chèn phần tử đó vào vị trí thích hợp trong phần đã sắp xếp (các phần tử từ đầu mảng đến phần tử trước đó).

#### - Quá trình diễn ra như sau:

- 1. Lấy phần tử đầu tiên làm "phần đã sắp xếp", phần còn lai là "phần chưa sắp xếp".
- 2. Lặp qua các phần tử trong "phần chưa sắp xếp":
  - · Lấy phần tử đầu tiên trong "phần chưa sắp xếp".
  - So sánh và chèn phần tử đó vào vi trí thích hợp trong "phần đã sắp xếp".
  - · Cập nhật "phần đã sắp xếp" và "phần chưa sắp xếp".
- 3. Lặp lại bước 2 cho đến khi "phần chưa sắp xếp" trống.

#### - Ví dụ

Cho mảng: [5, 2, 4, 6, 1, 3]

- Bước 1: Xét phần tử đầu tiên 5, vì đây là phần tử duy nhất nên không cần sắp xếp.
- Bước 2: Xét phần tử thứ hai 2, chèn 2 vào trước 5 => [2, 5, 4, 6, 1, 3]
- Bước 3: Xét phần tử thứ ba 4, chèn 4 vào giữa 2 và 5 => [2, 4, 5, 6, 1, 3]
- Bước 4: Xét phần tử thứ tư 6, giữ nguyên vị trí => [2, 4, 5, 6, 1, 3]
- Bước 5: Xét phần tử thứ năm 1, chèn 1 vào đầu mảng => [1, 2, 4, 5, 6, 3]
- Bước 6: Xét phần tử thứ sáu 3, chèn 3 vào giữa 2 và 4 => [1, 2, 3, 4, 5, 6]

Như vậy, mảng đã được sắp xếp theo thứ tự tăng dần.

#### - Triển khai

#### • Mã nguồn C++

```
void insertionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        /* Di chuyển các phần tử có giá trị lớn hơn key về bên
phải một vị trí */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

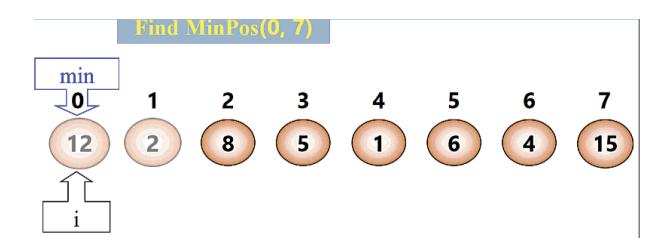
#### • Mã nguồn Java

```
public void sort(int array[]) {
    int n = array.length;
    for (int i = 1; i < n; ++i) {
        int key = array[i];
        int j = i - 1;

        /* Di chuyển các phần tử có giá trị lớn hơn key về bên
phải một vị trí */
        while (j >= 0 && array[j] > key) {
            array[j + 1] = array[j];
            j = j - 1;
        }
        array[j + 1] = key;
    }
}
```

```
def insertionSort(array):
    for i in range(1, len(array)):
        key = array[i]
        j = i - 1
        while j >= 0 and key < array[j]:
            array[j + 1] = array[j]
            j -= 1
        array[j + 1] = key</pre>
```

# 27. THUẬT TOÁN SẮP XẾP CHỌN



#### - Ý tưởng:

Ý tưởng của thuật toán sắp xếp chọn (selection sort) là:

- •Tìm phần tử nhỏ nhất trong mảng chưa được sắp xếp.
- Hoán đổi phần tử nhỏ nhất với phần tử đầu tiên của mảng chưa được sắp xếp.
- Di chuyển con trỏ đến phần tử tiếp theo của mảng chưa được sắp xếp.
- Lặp lại các bước 1-3 cho đến khi toàn bộ mảng được sắp xếp.

#### - Cụ thể, thuật toán sẽ thực hiện như sau:

- •Tìm phần tử nhỏ nhất trong mảng.
- Hoán đổi phần tử nhỏ nhất với phần tử đầu tiên của mảng.
- Lặp lại các bước trên với phần còn lại của mảng, bắt đầu từ phần tử thứ hai.
- Tiếp tục lặp lại cho đến khi toàn bộ mảng được sắp xếp.

#### - Ví dụ: Cho mảng [5, 2, 4, 6, 1, 3]:

- 1. Tìm phần tử nhỏ nhất trong mảng là 1, hoán đổi 1 với 5 => [1, 2, 4, 6, 5, 3].
- 2. Tìm phần tử nhỏ nhất trong mảng [2, 4, 6, 5, 3] là 2, hoán đổi 2 với 2 => [1, 2, 4, 6, 5, 3].
- 3. Tìm phần tử nhỏ nhất trong mảng [4, 6, 5, 3] là 3, hoán đổi 3 với 4 => [1, 2, 3, 6, 5, 4].
- 4. Tìm phần tử nhỏ nhất trong mảng [6, 5, 4] là 4, hoán đổi 4 với 6 => [1, 2, 3, 4, 5, 6].
- 5. Mảng đã được sắp xếp.

#### - Triển khai:

#### • Mã nguồn C++

#### • Mã nguồn Java

```
def selectionSort(array):
    for i in range(len(array)):
        min_idx = i
        for j in range(i+1, len(array)):
            if array[min_idx] > array[j]:
                  min_idx = j
        # Đổi chỗ phần tử nhỏ nhất với phần tử đầu tiên
        array[i], array[min_idx] = array[min_idx], array[i]
```

### - Câu hỏi:

Hãy minh họa các bước để sắp xếp mảng [4, 2, 6, 1, 7, 3, 5] bằng cách áp dụng thuật toán sắp xếp chọn.

Đáp án: Có 6 bước:

- 1. Tìm phần tử nhỏ nhất trong mảng, đó là 1, hoán vị với phần tử ở vị trí đầu tiên. Kết quả: [1, 2, 6, 4, 7, 3, 5]
- 2. Tìm phần tử nhỏ nhất trong mảng còn lại, đó là 2, không cần hoán vị. Kết quả: [1, 2, 6, 4, 7, 3, 5]
- 3. Tìm phần tử nhỏ nhất trong mảng còn lại, đó là 3, hoán vị với phần tử ở vị trí thứ 3. Kết quả: [1, 2, 3, 4, 7, 6, 5]
- 4. Tìm phần tử nhỏ nhất trong mảng còn lại, đó là 4, không cần hoán vị. Kết quả: [1, 2, 3, 4, 7, 6, 5]
- 5. Tìm phần tử nhỏ nhất trong mảng còn lại, đó là 5, hoán vị với phần tử ở vị trí thứ 5. Kết quả: [1, 2, 3, 4, 5, 6, 7]
- 6. Tìm phần tử nhỏ nhất trong mảng còn lại, đó là 6, hoán vị với phần tử ở vị trí thứ 6. Kết quả: [1, 2, 3, 4, 5, 6, 7]

# 28. THUẬT TOÁN TIMSORT

Timsort là một thuật toán sắp xếp hiệu quả cao, được sử dụng làm thuật toán sắp xếp mặc định trong nhiều ngôn ngữ lập trình hiện đại (chẳng hạn như Python). Timsort kết hợp ưu điểm của cả Insertion Sort và Merge Sort để đạt được hiệu suất cao trong nhiều trường hợp thực tế.

## - Ý tưởng chính:

- Phân chia mảng: Mảng đầu vào được chia thành các đoạn con có độ dài tối ưu, thường gọi là run.
- Sắp xếp các run: Mỗi run được sắp xếp bằng Insertion Sort, một thuật toán hiệu quả với các mảng nhỏ.
- Hợp nhất các run: Các run đã được sắp xếp được hợp nhất lại với nhau bằng Merge Sort, một thuật toán ổn định và hiệu quả với các mảng lớn.

### - Triển khai:

```
// ... (Cần các hàm hỗ trợ: insertionSort, merge, tìm độ dài run
tối ưu)

void timsort(vector<int>& arr) {
   int n = arr.size();
   for (int i = 0; i < n; i += MIN_RUN) {
      insertionSort(arr, i, min(i + MIN_RUN - 1, n - 1));
   }

for (int size = MIN_RUN; size < n; size = 2 * size) {
   for (int left = 0; left < n; left += 2 * size) {
      int mid = min(left + size - 1, n - 1);
      int right = min(left + 2 * size - 1, n - 1);
      merge(arr, left, mid, right);
   }
}
</pre>
```

### • Mã nguồn Java

```
// ... (Các hàm hỗ trợ: insertionSort, merge, tìm độ dài run tối
uu)

public void timsort(int[] arr) {
   int n = arr.length;
   for (int i = 0; i < n; i += MIN_RUN) {
      insertionSort(arr, i, Math.min(i + MIN_RUN - 1, n - 1));
   }

   for (int size = MIN_RUN; size < n; size = 2 * size) {
      for (int left = 0; left < n; left += 2 * size) {
        int mid = Math.min(left + size - 1, n - 1);
        int right = Math.min(left + 2 * size - 1, n - 1);
        merge(arr, left, mid, right);
    }
}
</pre>
```

# • Mã nguồn Python

```
# ... (Các hàm hỗ trợ: insertionSort, merge, tìm độ dài run tối
uu)

def timsort(arr):
    n = len(arr)
    for i in range(0, n, MIN_RUN):
        insertion_sort(arr, i, min(i + MIN_RUN - 1, n - 1))

for size in range(MIN_RUN, n, 2 * size):
    for left in range(0, n, 2 * size):
        mid = min(left + size - 1, n - 1)
        right = min(left + 2 * size - 1, n - 1)
        merge(arr, left, mid, right)
```

### - Điểm mạnh của Timsort:

- Hiệu quả cao: Kết hợp ưu điểm của Insertion Sort và Merge Sort.
- Őn định: Giữ nguyên thứ tự tương đối của các phần tử bằng nhau.
- Tối ưu hóa: Độ dài run tối ưu được chọn dựa trên đặc tính của dữ liệu đầu vào.
- Sử dụng rộng rãi: Được sử dụng làm thuật toán sắp xếp mặc định trong nhiều ngôn ngữ lập trình.

### - Các hàm hỗ trợ:

- insertionSort: Sắp xếp một đoạn con bằng Insertion Sort.
- merge: Hợp nhất hai đoạn con đã sắp xếp bằng Merge Sort.
- Tìm độ dài run tối ưu: Xác định độ dài run phù hợp dựa trên đặc tính của dữ liệu.

### - Lưu ý:

- MIN\_RUN: Là một hằng số quyết định độ dài của các run, thường được chọn dựa trên kích thước bộ nhớ cache. Có thể đơn giản hóa bằng cách đặt một hằng số cố định, chẳng hạn 32, 64...
- Độ phức tạp: Độ phức tạp thời gian trung bình của Timsort là O(n log n).

# 29. GỘP 2 MẢNG ĐÃ ĐƯỢC SẮP XẾP

#### - Đề bài:

Cho hai mảng số nguyên đã được sắp xếp theo trật tự tăng dần. Hãy viết thuật toán để gộp hai mảng này lại thành một mảng mới cũng được sắp xếp với trật tự tăng dần.

### - Ví du:

```
Mång 1: arr1[] = [1 3 5 7]
Mång 2: arr2[] = [0 2 6 8 9]
Mång sau khi gộp: merged_arr[] = [0 1 2 3 5 6 7 8 9]
```

## - Giải pháp

Ý tưởng chính là so sánh từng phần tử của hai mảng đã sắp xếp, chọn phần tử nhỏ hơn đưa vào mảng kết quả, và lặp lại quá trình này cho đến khi cả hai mảng đều được duyệt hết.

### - Triển khai:

```
void merge(vector<int>& arr1, int m, vector<int>& arr2, int n,
vector<int>& result) {
    int i = 0, j = 0, k = 0;
    // Trong khi cả hai mảng đều còn phần tử
    while (i < m \&\& j < n) {
        if (arr1[i] <= arr2[j]) {</pre>
            result[k] = arr1[i];
            i++;
        } else {
            result[k] = arr2[j];
            j++;
        k++;
    }
    // Copy các phần tử còn lại của mảng 1 (nếu có)
    while (i < m) {
       result[k] = arr1[i];
        i++;
        k++;
    }
    // Copy các phần tử còn lại của mảng 2 (nếu có)
    while (j < n) {
        result[k] = arr2[j];
        j++;
        k++;
    }
}
```

## • Mã nguồn Java

```
public static void merge(int[] arr1, int m, int[] arr2, int n,
int[] result) {
    int i = 0, j = 0, k = 0;
    while (i < m \&\& j < n) {
        if (arr1[i] <= arr2[j]) {</pre>
            result[k] = arr1[i];
            i++;
        } else {
           result[k] = arr2[j];
            j++;
        k++;
    while (i < m) {
        result[k] = arr1[i];
        i++;
       k++;
    }
    while (j < n) {
       result[k] = arr2[j];
        j++;
        k++;
    }
}
```

# • Mã nguồn Python

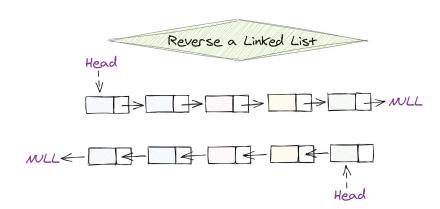
```
def merge(arr1, m, arr2, n, result):
    i = j = k = 0
    while i < m and j < n:
        if arr1[i] <= arr2[j]:</pre>
           result[k] = arr1[i]
            i += 1
        else:
           result[k] = arr2[j]
            j += 1
        k += 1
    while i < m:
        result[k] = arr1[i]
        i += 1
        k += 1
    while j < n:
        result[k] = arr2[j]
        j += 1
        k += 1
```



# 30. ĐẢO NGƯỢC CÁC PHẦN TỬ CỦA LINKED LIST

### - Đề bài:

Đảo ngược thứ tự một linked list là một bài toán cơ bản trong cấu trúc dữ liệu và thuật toán. Bài toán này yêu cầu ta sắp xếp lại các node trong linked list sao cho node cuối cùng trở thành node đầu tiên và ngược lại.



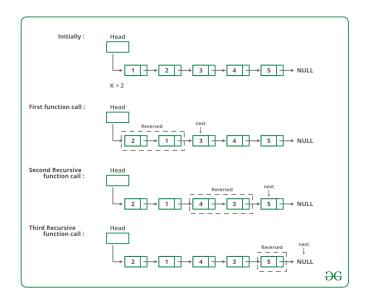
## - Giải thuật chung

Có nhiều cách để giải quyết bài toán này, nhưng cách phổ biến nhất là sử dụng 3 con trỏ:

- 1. current: Con trỏ trỏ đến node hiện tại đang xét.
- 2. prev: Con trỏ trỏ đến node trước đó của node hiện tại.
- 3. next: Con trỏ tạm thời để lưu trữ node tiếp theo của node hiện tại.

### - Các bước:

- 1. Khởi tạo prev và next bằng null.
- 2. Lặp qua linked list:
  - · Lưu node tiếp theo của current vào next.
  - Chỉnh next của current trỏ về prev.
  - · Câp nhât prev và current cho vòng lặp tiếp theo.



### - Triển khai

## • Mã nguồn C++

```
class Node {
   constructor(data) {
      this.data = data;
      this.next = null;
   }
}

function reverseList(head) {
   let prev = null;
   let current = head;
   let next = null;

   while (current !== null) {
      next = current.next;
      current.next = prev;
      prev = current;
      current = next;
   }

   return prev;
}
```

# Mã nguồn Java

```
class Node {
   int data;
   Node next;
   Node(int data) {
       this.data = data;
       this.next = null;
    }
}
public class ReverseLinkedList {
    public Node reverseList(Node head) {
        Node prev = null;
        Node current = head;
        Node next = null;
        while (current != null) {
           next = current.next;
           current.next = prev;
           prev = current;
            current = next;
       return prev;
```

## • Mã nguồn Python

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

def reverse_list(head):
    prev = None
    current = head
    while current is not None:
        next = current.next
        current.next = prev
        prev = current
        current = next
    head = prev
    return head
```

#### - Giải thích

- · Vòng lặp while: Tiếp tục cho đến khi current trở thành null (đến cuối danh sách).
- Lưu node tiếp theo: Lưu next của current vào next để không bị mất liên kết.
- Đảo ngược liên kết: Chỉnh next của current trỏ về prev, tức là đảo ngược chiều liên kết.
- Cập nhật con trỏ: Cập nhật prev và current cho vòng lặp tiếp theo.

## - Độ phức tạp

- Thời gian: O(n), vì chúng ta duyệt qua danh sách một lần.
- Không gian: O(1), vì chúng ta chỉ sử dụng một lượng không gian cố định cho các biến.

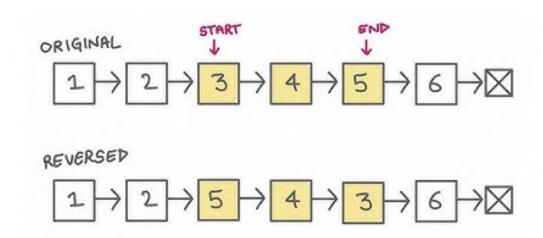
### - Ứng dụng

- Stack: Có thể sử dụng stack để thực hiện đảo ngược linked list, nhưng cách làm này thường kém hiệu quả hơn so với cách sử dụng 3 con trỏ.
- Các bài toán liên quan đến linked list: Đảo ngược linked list là một bước cơ bản trong nhiều thuật toán liên quan đến linked list, như kiểm tra tính đối xứng, sắp xếp, v.v.

### - Lưu ý:

- Linked list đơn: Các ví dụ trên áp dụng cho linked list đơn. Với linked list đôi, cần cập nhất thêm con trỏ prev của các node.
- · Linked list vòng: Cần xử lý trường hợp đặc biệt khi linked list là vòng.

# 31. ĐẢO NGƯỢC MỘT SLICE BẤT KỲ TRONG LINKED LIST



### - Hiểu về bài toán:

- Slice: Một đoạn con của linked list, được xác định bởi một node bắt đầu và một node kết thúc.
- Đảo ngược: Thay đổi thứ tự các node trong slice đó, sao cho node cuối cùng của slice trở thành node đầu tiên và ngược lại.

## - Giải thuật:

Thực chất, chúng ta chỉ cần tái sử dụng thuật toán đảo ngược linked list đã giới thiệu ở post trước, áp dụng riêng cho một lát cắt của linked list, sau đó nối các đoạn linked list lại với nhau.

- 1. Xác định node bắt đầu và kết thúc của slice: Sử dụng hai biến start và end để đánh dấu.
- 2. Đảo ngược slice: Áp dụng thuật toán đảo ngược linked list thông thường cho phần từ start đến end.
- 3. Kết nối lại: Kết nối lại phần đã đảo ngược với phần còn lại của linked list.

### - Triển khai

```
Node* reverseBetween(Node* head, int m, int n) {
    if (head == nullptr || m == n) {
       return head;
    // Tìm đến nút trước vị trí m
    Node* prev = nullptr;
   Node* current = head;
    for (int i = 0; i < m - 1 && current != nullptr; i++) {
      prev = current;
      current = current->next;
    }
   // Lưu lại nút đầu của phần cần đảo ngược và nút tiếp theo của
phần đó
    Node* start = current;
   Node* temp = nullptr;
    // Đảo ngược phần danh sách từ m đến n
    for (int i = 0; i < n - m; i++) {
       temp = current->next;
       current->next = prev;
       prev = current;
       current = temp;
    }
    // Kết nối lại các phần
    if (prev != nullptr) {
       prev->next = current;
    } else {
      head = prev;
    start->next = temp;
   return head;
}
```

## • Mã nguồn Java

```
public Node reverseBetween(Node head, int left, int right) {
  Node current = head;
 Node prev = null;
 // Di chuyển đến node bắt đầu (left - 1)
 for (int i = 0; i < left - 1; i++) {
    prev = current;
    current = current.next;
 // Node trước khi bắt đầu slice
 Node start = prev;
 // Node bắt đầu slice
 Node tail = current;
 // Node sau khi kết thúc slice
 Node next = null;
 // Đảo ngược slice
 for (int i = 0; i < right - left + 1; i++) {
    next = current.next;
    current.next = prev;
    prev = current;
    current = next;
 }
 // Kết nối lại
 if (start != null) {
    start.next = prev;
 } else {
    head = prev;
 tail.next = current;
 return head;
```

## • Mã nguồn Python

```
def reverseBetween(head, left, right):
    current = head
    prev = None
    # Di chuyển đến node bắt đầu (left - 1)
    for _ in range(left - 1):
       prev = current
        current = current.next
    # Node trước khi bắt đầu slice
    start = prev
    # Node bắt đầu slice
    tail = current
    # Node sau khi kết thúc slice
    next = None
    # Đảo ngược slice
    for _ in range(right - left + 1):
       next = current.next
       current.next = prev
       prev = current
       current = next
    # Kết nối lại
    if start:
       start.next = prev
    else:
       head = prev
    tail.next = current
    return head
```

#### - Giải thích:

- Di chuyển đến node bắt đầu: Duyệt linked list đến node trước node bắt đầu của slice.
- Đảo ngược slice: Áp dụng thuật toán đảo ngược linked list thông thường cho phần tử từ start đến end.
- Kết nối lại: Kết nối prev (node cuối cùng sau khi đảo ngược) với node trước start và tail (node cuối cùng của slice) với node sau end.

## - Độ phức tạp:

- Thời gian: O(n), trong đó n là số lượng node trong slice cần đảo ngược.
- Không gian: O(1), không sử dụng thêm không gian phụ đáng kể.

### - Lưu ý:

- Các trường hợp đặc biệt: Cần xử lý các trường hợp khi left bằng 1 (đảo ngược từ đầu danh sách) hoặc right bằng độ dài danh sách (đảo ngược đến cuối danh sách).
- Linked list đôi: Với linked list đôi, cần cập nhật cả con trỏ prev và next của các node trong slice.

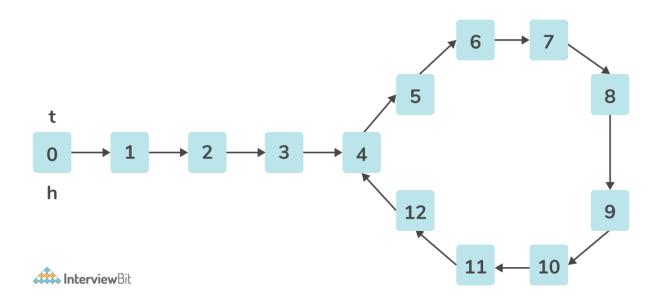
## - Ứng dụng:

- Sắp xếp linked list: Có thể sử dụng đảo ngược slice để thực hiện các thuật toán sắp xếp như insertion sort, quick sort.
- Các bài toán liên quan đến linked list: Đảo ngược slice là một kỹ thuật hữu ích trong nhiều bài toán liên quan đến linked list.

# 32. PHÁT HIỆN LINKED LIST VÒNG

## - Hiểu về linked list vòng

Một linked list được gọi là vòng nếu có ít nhất một node xuất hiện hai lần trong danh sách, tạo thành một vòng lặp



### - Thuật toán Floyd's Cycle-Finding Algorithm (Hai con trỏ):

Đây là một thuật toán hiệu quả để phát hiện vòng trong linked list.

### - Ý tưởng:

- Sử dụng hai con trỏ: slow và fast.
- slow di chuyển một bước mỗi lần.
- fast di chuyển hai bước mỗi lần.
- Nếu có vòng, hai con trỏ sẽ gặp nhau tại một điểm nào đó trong vòng.

### - Triển khai

## • Mã nguồn C++

```
bool hasCycle(Node* head) {
   if (head == nullptr || head->next == nullptr) {
      return false; // Danh sách rỗng hoặc chỉ có một nút
   }

   Node *slow = head;
   Node *fast = head;

   while (fast != nullptr && fast->next != nullptr) {
      slow = slow->next;
      fast = fast->next->next;
      if (slow == fast) {
            return true; // Có vòng
      }
   }

   return false; // Không có vòng
}
```

## • Mã nguồn Java

```
public boolean hasCycle(Node head) {
   if (head == null || head.next == null) {
      return false;
   }

   Node slow = head;
   Node fast = head.next;

   while (slow != fast) {
      if (fast == null || fast.next == null) {
          return false;
      }
      slow = slow.next;
      fast = fast.next.next;
   }
   return true;
}
```

## • Mã nguồn Python

```
def hasCycle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True
    return False
```

### - Giải thích:

- Hai con trở: slow di chuyển một bước mỗi lần, trong khi fast di chuyển hai bước.
- Điều kiện dừng: Nếu không có vòng, fast sẽ sớm trở thành null hoặc fast.next sẽ là null. Nếu có vòng, slow và fast chắc chắn sẽ gặp nhau tại một điểm nào đó trong vòng.

## - Tại sao thuật toán này hoạt động?

- Hình dung: Hãy tưởng tượng hai người chạy trên một đường tròn. Người chạy nhanh hơn chắc chắn sẽ đuổi kịp người chạy chậm hơn tại một điểm nào đó trên đường tròn.
- Chứng minh: Có thể chứng minh bằng toán học rằng nếu có vòng, hai con trỏ sẽ gặp nhau.

## - Ứng dụng

- Kiểm tra tính hợp lệ của linked list: Đảm bảo linked list không bị hỏng do các lỗi lập trình.
- Tìm điểm bắt đầu của vòng: Sau khi phát hiện ra vòng, có thể sử dụng một con trỏ khác để tìm điểm bắt đầu của vòng.

## - Ưu điểm của thuật toán Floyd:

- Hiệu quả: Độ phức tạp thời gian là O(n) và không gian là O(1).
- Đơn giản: Dễ hiểu và triển khai.

## - Các thuật toán khác:

• Sử dụng hash set: Lưu trữ các node đã duyệt qua trong một hash set. Nếu gặp lại một node đã có trong hash set, có nghĩa là có vòng. Tuy nhiên, phương pháp này thường tiêu tốn nhiều không gian hơn.

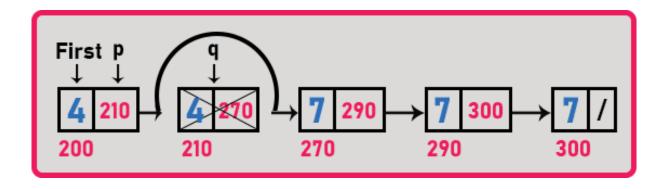
### - Lưu ý:

- Thuật toán này chỉ phát hiện được vòng đơn.
- •Để tìm điểm bắt đầu của vòng, có thể sử dụng một con trỏ thứ ba bắt đầu từ đầu danh sách và di chuyển một bước mỗi lần. Khi gặp lại slow (điểm gặp nhau của hai con trỏ ban đầu), con trỏ thứ ba sẽ đang ở vị trí bắt đầu của vòng.

### - Các bài toán liên quan đến vòng lặp:

Ví dụ, tìm độ dài của vòng lặp, tìm số lượng node trong vòng lặp.

# 33. XÓA TẤT CẢ CÁC NODE TRÙNG LẶP TRONG MỘT LINKED LIST



### - Hiểu bài toán:

- Node trùng lặp: Hai hoặc nhiều node có cùng giá trị dữ liệu.
- Mục tiêu: Xóa tất cả các node trùng lặp, chỉ giữ lại một node duy nhất cho mỗi giá tri.

### - Giải thuật:

- 1. Sử dụng một hash set:
  - Tạo một hash set để lưu trữ các giá trị đã duyệt qua.
  - Duyệt qua linked list:
  - Nếu giá trị của node hiện tại đã có trong hash set, xóa node đó.
  - Ngược lại, thêm giá trị của node vào hash set.

#### - Triển khai

```
Node* removeDuplicates(Node* head) {
    if (head == nullptr) {
        return nullptr;
    std::unordered_set<int> seen;
    Node* current = head;
    Node* previous = nullptr;
    while (current != nullptr) {
        if (seen.count(current->data) > 0) {
            // Xóa node hiện tại
            previous->next = current->next;
            delete current;
        } else {
            seen.insert(current->data);
            previous = current;
        current = current->next;
   return head;
}
```

## • Mã nguồn Java

```
public Node removeDuplicates(Node head) {
    if (head == null) {
       return null;
    }
    HashSet<Integer> seen = new HashSet<>();
    Node current = head;
    Node prev = null;
    while (current != null) {
        if (seen.contains(current.data)) {
            prev.next = current.next;
        } else {
            seen.add(current.data);
            prev = current;
        current = current.next;
    }
    return head;
```

## • Mã nguồn Python

```
def remove_duplicates(head):
    seen = set()
    current = head
    prev = None

while current:
    if current.data in seen:
        prev.next = current.next
    else:
        seen.add(current.data)
        prev = current
        current = current.next

return head
```

### - Độ phức tạp:

- Thời gian: O(n), trong đó n là số lượng node trong linked list.
- Không gian: O(n), để lưu trữ hash set.

### - Ưu điểm:

- Hiệu quả: Thời gian trung bình để tìm kiếm và chèn trong hash set là O(1).
- Đơn giản: Dễ hiểu và triển khai.

### - Lưu ý:

- Sắp xếp: Nếu linked list đã được sắp xếp, có thể sử dụng hai con trỏ để so sánh giá trị của các node liền kề và xóa các node trùng lặp mà không cần sử dụng hash
- Các trường hợp đặc biệt: Cần xử lý trường hợp danh sách rỗng hoặc chỉ có một node.

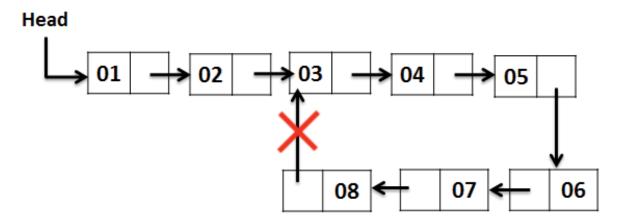
## - Ứng dụng:

- · Loại bỏ dữ liệu trùng lặp: Sử dụng trong nhiều bài toán xử lý dữ liệu.
- Tạo danh sách duy nhất: Đảm bảo mỗi phần tử chỉ xuất hiện một lần trong danh sách.

## - Các phương pháp khác:

• Sử dụng hai vòng lặp: Một vòng lặp để duyệt các node, vòng lặp bên trong để so sánh với các node phía sau. Tuy nhiên, phương pháp này thường kém hiệu quả hơn so với sử dụng hash set.

# 34. THUẬT TOÁN THÁO VÒNG CHO LINKED LIST



### - Hiểu về bài toán

- Linked list vòng: Một linked list có ít nhất một node xuất hiện hai lần, tạo thành một vòng lặp.
- Tháo vòng: Tìm và cắt liên kết để phá vỡ vòng lặp, biến linked list thành một danh sách tuyến tính.

### - Thuật toán Floyd's Cycle-Finding Algorithm

Chúng ta đã sử dụng thuật toán này để phát hiện vòng. Để tháo vòng, chúng ta cần tìm điểm bắt đầu của vòng và cắt liên kết tại đó.

#### - Các bước:

- 1. Phát hiện vòng: Sử dụng hai con trỏ slow và fast để phát hiện linked list vòng.
- 2. Tìm điểm bắt đầu vòng:
  - Khi hai con trỏ gặp nhau, di chuyển slow về đầu danh sách.
  - Tiếp tục di chuyển cả slow và fast với tốc độ 1 node/bước.
  - Điểm mà chúng gặp nhau lần nữa chính là điểm bắt đầu của vòng.
- 3. Cắt vòng:
  - · Lưu node trước điểm bắt đầu của vòng.
  - Đặt next của node trước đó bằng null.

### - Triển khai

## • Mã nguồn C++

```
Node* removeCycle(Node* head) {
    Node* slow = head, *fast = head;
    Node* meet = detectCycle(head);
    if (meet == nullptr) {
       return head; // Không có vòng
    // Tìm điểm bắt đầu của vòng
    slow = head;
    while (slow != meet) {
      slow = slow->next;
       meet = meet->next;
    }
    // Tìm node trước node bắt đầu vòng
    Node* prev = slow;
    while (prev->next != meet) {
      prev = prev->next;
    // Tháo vòng
    prev->next = nullptr;
   return head;
```

## • Mã nguồn Java

```
public void removeLoop() {
    if (head == null | head.next == null) {
       return; // Danh sách rỗng hoặc chỉ có 1 node
   Node slow = head, fast = head;
    while (fast != null && fast.next != null) {
       slow = slow.next;
       fast = fast.next.next;
        if (slow == fast) { // Tìm thấy vòng
           slow = head;
           while (slow != fast) {
               slow = slow.next;
               fast = fast.next;
            // fast là node bắt đầu vòng
           fast.next = null; // Cắt vòng
           return;
       }
  }
```

## • Mã nguồn Python

```
def remove loop(head):
    slow = fast = head
    # Tìm điểm giao nhau của vòng
    while fast and fast.next:
        slow = slow.next
       fast = fast.next.next
        if slow == fast:
           break
    # Nếu không có vòng, hàm sẽ kết thúc tại đây
    # Di chuyển slow về đầu danh sách
    slow = head
   # Di chuyển cả slow và fast cùng một bước cho đến khi gặp nhau
    while slow.next != fast.next:
        slow = slow.next
        fast = fast.next
    # Phá vỡ vòng
    fast.next = None
    return head
```

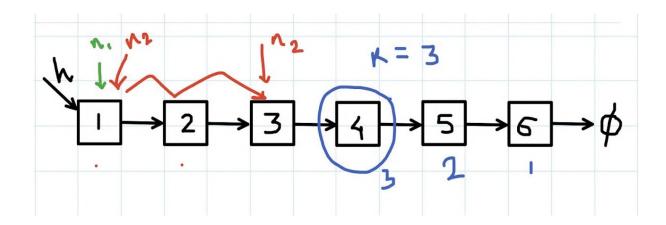
## - Lưu ý

- Thuật toán này giả định rằng vòng chỉ có một điểm bắt đầu.
- Nếu có nhiều vòng lặp phức tạp hơn, thuật toán có thể cần điều chỉnh.

### - Các thuật toán khác:

• Sử dụng hash set: Lưu trữ các node đã duyệt qua trong một hash set. Nếu gặp lại một node đã có trong hash set, có nghĩa là có vòng. Tuy nhiên, phương pháp này thường tiêu tốn nhiều không gian hơn.

# 35. THUẬT TOÁN TÌM PHẦN TỬ THỨ K TÍNH TỪ CUỐI CỦA LINKED LIST



#### - Hiểu bài toán

- Mục tiêu: Tìm và trả về giá trị của node ở vị trí thứ k tính từ cuối danh sách liên kết.
- Khó khăn: Không thể biết trước độ dài của danh sách.

## - Phương pháp phổ biến: Hai con trỏ

- Ý tưởng: Sử dụng hai con trỏ slow và fast.
- Cách thực hiện:
  - 1. Khởi tạo slow và fast cùng trỏ đến đầu danh sách.
  - 2. Cho fast tiến k bước.
  - 3. Di chuyển cả hai con trỏ với cùng tốc độ cho đến khi fast đến cuối danh sách.
  - 4. Khi đó, slow sẽ đang ở vị trí cần tìm.

### - Triển khai:

```
int findKthFromEnd(Node* head, int k) {
    if (head == nullptr) {
       return -1; // Danh sách rỗng
    }
   Node* slow = head;
   Node* fast = head;
    // Đưa fast tiến k bước
    for (int i = 0; i < k && fast != nullptr; i++) {</pre>
        fast = fast->next;
    }
    // Nếu k lớn hơn đô dài của danh sách, trả về -1
    if (fast == nullptr) {
       return -1;
    // Di chuyển cả slow và fast cùng một bước cho đến khi fast
đến cuối
   while (fast != nullptr) {
```

```
slow = slow->next;
fast = fast->next;
}

return slow->data;
}
```

## Mã nguồn Java

```
public int findKthFromEnd(Node head, int k) {
    Node slow = head, fast = head;
    // Cho fast đi trước k bước
    for (int i = 0; i < k && fast != null; i++) {
       fast = fast.next;
    }
    // Nếu k lớn hơn độ dài danh sách, trả về -1
    if (fast == null) {
       return -1;
    }
    // Di chuyển cả hai con trỏ cho đến khi fast đến cuối
    while (fast != null) {
       slow = slow.next;
       fast = fast.next;
    }
    // slow sẽ đang ở vị trí thứ k từ cuối
    return slow.data;
```

## • Mã nguồn Python

```
def findKthFromEnd(head, k):
    slow = fast = head

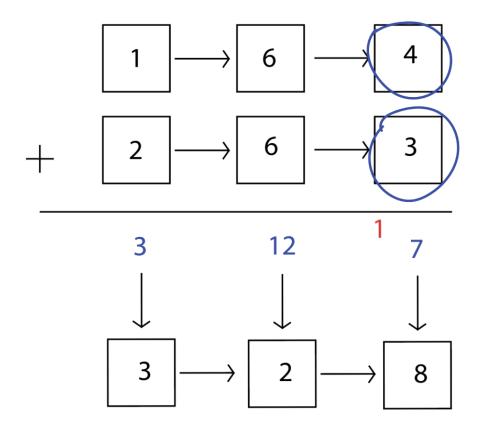
# Cho fast di truớc k bước
for _ in range(k):
    if not fast:
        return -1 # k lớn hơn độ dài danh sách
    fast = fast.next

# Di chuyển cả hai con trỏ cho đến khi fast đến cuối
while fast:
    slow = slow.next
    fast = fast.next
```

## - Ứng dụng thực tế

- Phân tích dữ liệu: Tìm kiếm các yếu tố gần nhất từ một tập dữ liệu lớn mà không cần phải duyệt qua toàn bộ dữ liệu.
- Lập trình hệ thống: Các hệ điều hành thường sử dụng phương pháp này để theo dõi và quản lý bộ nhớ, nơi mà các đối tượng không còn sử dụng.
- Thuật toán tìm kiếm: Trong các cơ sở dữ liệu, việc tìm kiếm các bản ghi hoặc phần tử thứ k từ một danh sách các bản ghi cũng có thể sử dụng phương pháp này.

# 36. BIỂU DIỄN PHÉP CỘNG HAI SỐ BẰNG LINKED LIST: MỘT CÁCH TIẾP CẬN THỦ VỊ



# - Ý tưởng:

- Mỗi node: Đai diên cho một chữ số của một số.
- Linked list: Đại diện cho toàn bộ số.
- Phép cộng: Thực hiện từ phải sang trái, giống như cách học sinh cấp 1 thường làm.

### - Ví du:

Số 123 sẽ được biểu diễn thành một linked list với các node chứa lần lượt các giá trị 3, 2, 1.

### - Thuật toán:

- 1. Khởi tạo hai linked list: Mỗi linked list đại diện cho một số.
- 2. Khởi tạo một linked list kết quả: Để lưu trữ kết quả phép cộng.
- 3. Cộng từng cặp node tương ứng:

- Bắt đầu từ node cuối cùng của hai linked list.
- · Cộng giá trị của hai node tương ứng và nhớ phần nhớ (carry).
- Tạo một node mới với giá trị là phần dư của phép cộng và nối vào đầu linked list kết quả.
- Tiếp tục với cặp node tiếp theo, cộng thêm phần nhớ nếu có.
- 4. Xử lý phần nhớ cuối cùng: Nếu sau khi cộng xong cặp node đầu tiên vẫn còn phần nhớ, tạo thêm một node mới với giá trị là phần nhớ và nối vào đầu linked list kết quả.

## - Triển khai:

## • Mã nguồn C++

```
Node* addTwoNumbers(Node* 11, Node* 12) {
    Node* dummyHead = new Node(0);
    Node* current = dummyHead;
    int carry = 0;

while (11 || 12 || carry) {
        int x = 11 ? 11->data : 0;
        int y = 12 ? 12->data : 0;
        int sum = x + y + carry;
        carry = sum / 10;
        current->next = new Node(sum % 10);
        current = current->next;
        11 = 11 ? 11->next : nullptr;
        12 = 12 ? 12->next : nullptr;
    }

    return dummyHead->next;
}
```

### Mã nguồn Java

```
public Node addTwoNumbers(Node 11, Node 12) {
    Node dummyHead = new Node(0);
    Node current = dummyHead;
    int carry = 0;
    while (11 != null || 12 != null) {
        int x = 11 != null ? 11.data : 0;
        int y = 12 != null ? 12.data : 0;
        int sum = x + y + carry;
        carry = sum / 10;
        current.next = new Node(sum % 10);
        current = current.next;
        if (l1 != null) l1 = l1.next;
        if (12 != null) 12 = 12.next;
    }
    if (carry > 0) {
        current.next = new Node(carry);
    return dummyHead.next;
```

## • Mã nguồn Python

```
def addTwoNumbers(self, l1, l2):
    dummy = ListNode()
    curr = dummy
    carry = 0
    while 11 or 12:
       x = 11.val if 11 else 0
       y = 12.val if 12 else 0
       sum = x + y + carry
       carry = sum // 10
       curr.next = ListNode(sum % 10)
       curr = curr.next
       11 = 11.next if 11 else None
       12 = 12.next if 12 else None
    if carry > 0:
       curr.next = ListNode(carry)
    return dummy.next
```

### - Giải thích:

- Dummy node: Được sử dụng để đơn giản hóa việc tạo node đầu tiên của linked list kết quả.
- Carry: Lưu trữ phần nhớ khi tổng của hai chữ số lớn hơn 10.
- Lặp: Tiếp tục cho đến khi cả hai linked list đều hết node và phần nhớ bằng 0.
- Tạo node mới: Tạo node mới với giá trị là phần dư của phép cộng và nối vào linked list kết quả.

## - Ưu điểm của cách tiếp cận này:

- Minh hoa rõ ràng: Giúp hiểu rõ hơn về cách thực hiện phép công theo côt.
- Khả năng mở rộng: Có thể dễ dàng mở rộng để thực hiện các phép toán khác trên linked list.
- Hiệu quả: Độ phức tạp thời gian là O(max(m, n)), trong đó m và n là độ dài của hai linked list.

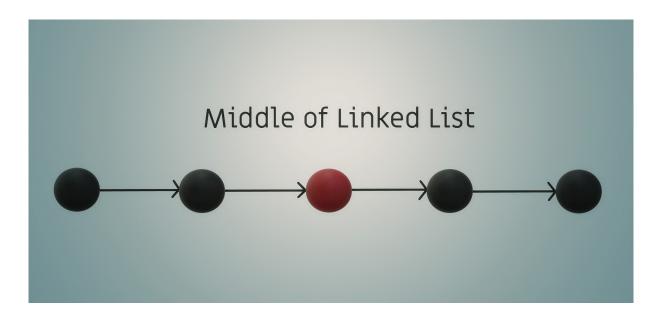
### - Lưu ý:

- Linked list đảo ngược: Trong ví dụ trên, các số được biểu diễn với chữ số hàng đơn vị ở đầu linked list. Nếu muốn biểu diễn theo thứ tự tự nhiên, cần đảo ngược linked list trước khi thực hiện phép cộng.
- Số âm: Để xử lý số âm, có thể sử dụng một bit để biểu diễn dấu.

## - Ứng dụng:

- Bài toán liên quan đến số lớn: Khi các số quá lớn để biểu diễn bằng kiểu số nguyên cơ bản.
- Thực hiện các phép toán số học: Ngoài phép cộng, có thể mở rộng để thực hiện phép trừ, nhân, chia.

# 37. TÌM VỊ TRÍ CHÍNH GIỮA CỦA LINKED LIST



### - Hiểu bài toán:

- Vị trí chính giữa: Node nằm ở giữa danh sách liên kết. Nếu danh sách có số lượng node chẵn, có thể chọn một trong hai node ở giữa.
- Mục tiêu: Viết thuật toán để tìm và trả về node ở vị trí chính giữa của linked list.

## - Giải pháp: Sử dụng hai con trỏ

- Ý tưởng:
- Sử dụng hai con trỏ slow và fast.
- slow di chuyển một bước mỗi lần.
- fast di chuyển hai bước mỗi lần.
- Khi fast đến cuối danh sách, slow sẽ đang ở vị trí chính giữa.

### - Triển khai:

```
Node* findMiddle(Node* head) {
   if (head == nullptr) {
      return nullptr; // Danh sách rỗng
   }

   Node* slow = head;
   Node* fast = head;

   while (fast != nullptr && fast->next != nullptr) {
      slow = slow->next;
      fast = fast->next->next;
   }

   return slow;
}
```

### Mã nguồn Java

```
public Node middleNode(Node head) {
   Node slow = head, fast = head;
   while (fast != null && fast.next != null) {
      slow = slow.next;
      fast = fast.next.next;
   }
   return slow;
}
```

## • Mã nguồn Python

```
def middleNode(self, head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow
```

#### - Giải thích:

- Hai con trỏ: slow di chuyển một bước mỗi lần, trong khi fast di chuyển hai bước.
- Điều kiện dừng: Khi fast hoặc fast.next bằng null (tức là fast đã đến cuối danh sách hoặc node cuối cùng), vòng lặp kết thúc.
- Trả về kết quả: Khi vòng lặp kết thúc, slow sẽ đang ở vị trí chính giữa của danh sách.

## - Tại sao thuật toán này hoạt động?

- Hình dung: Hãy tưởng tượng một cuộc đua giữa hai người, một người chạy nhanh gấp đôi người kia. Khi người chạy nhanh đến đích, người chạy chậm sẽ ở đúng giữa quãng đường.
- Chứng minh: Có thể chứng minh bằng toán học rằng khi fast đến cuối danh sách, slow sẽ ở vị trí chính giữa.

## - Ưu điểm của phương pháp này:

- Hiệu quả: Độ phức tạp thời gian là O(n), trong đó n là số lượng node trong danh sách.
- Đơn giản: Dễ hiểu và triển khai.
- Không yêu cầu biết trước độ dài danh sách: Đây là một ưu điểm lớn so với các phương pháp khác.

# - Ứng dụng:

- Chia danh sách: Có thể sử dụng để chia danh sách thành hai nửa.
- Các thuật toán trên danh sách liên kết: Nhiều thuật toán yêu cầu tìm node ở vị trí giữa.

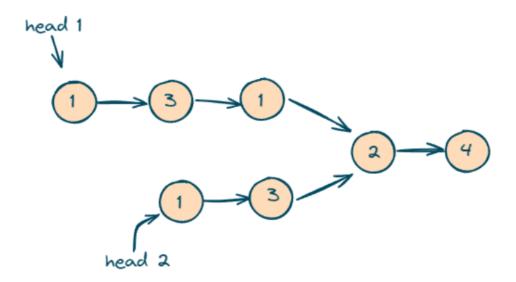
#### - Lưu ý:

- Danh sách rỗng: Nếu danh sách rỗng, cần xử lý trường hợp đặc biệt.
- Danh sách có một node: Node duy nhất đó cũng là node chính giữa.

### - Thuật toán mở rộng:

- Tìm node thứ k từ cuối danh sách.
- Kiểm tra xem một linked list có đối xứng hay không.

# 38. TÌM ĐIỂM GIAO NHAU GIỮA HAI LINKED LIST



### - Hiểu bài toán:

- Linked list: Là một cấu trúc dữ liệu gồm các node, mỗi node chứa một dữ liệu và một con trỏ trỏ đến node tiếp theo.
- Điểm giao: Là một node mà xuất hiện trong cả hai linked list.

## - Các cách tiếp cận:

Có nhiều cách để giải quyết bài toán này, nhưng chúng ta sẽ tập trung vào hai cách phổ biến nhất:

## + Sử dụng hai vòng lặp (Two-pointer technique):

- Ý tưởng:
  - Cho hai con trỏ di chuyển đồng thời qua hai linked list.
  - Khi một con trỏ đến cuối danh sách, nó sẽ chuyển sang đầu danh sách còn lại. Bằng cách này cả hai con trỏ sẽ có cùng khoảng cách tới điểm giao (nếu có).
  - Thực hiện lần duyệt tới cuối danh sách thêm một lần nữa. Nếu hai danh sách có điểm giao thì sẽ có một lúc nào đó hai con trở mang cùng một value.
  - Sau lần duyệt thứ hai, nếu con trỏ nào chạy tới cuối danh sách thì sẽ được cho trỏ tới NULL (nhằm kết thúc vòng lặp).
- Ưu điểm: Đơn giản, hiệu quả và không cần thêm bộ nhớ.

### + Sử dụng HashMap:

- •Ý tưởng: Duyệt qua một linked list và lưu các node của nó vào một HashMap. Sau đó, duyệt qua linked list còn lại và kiểm tra xem mỗi node có tồn tại trong Hash-Map hay không.
- Ưu điểm: Nhanh hơn trong trường hợp hai linked list chênh lệch chiều dài lớn.
- Nhược điểm: Tốn thêm bộ nhớ để lưu HashMap.

### - Triển khai:

## • Mã nguồn C++

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    // Sử dụng hai vòng lặp
    ListNode *pA = headA, *pB = headB;
    while (pA != pB) {
        pA = pA == nullptr ? headB : pA->next;
        pB = pB == nullptr ? headA : pB->next;
    }
    return pA;
}
```

## • Mã nguồn Java

```
public Node getIntersectionNode(Node headA, Node headB) {
    // Sử dụng hai vòng lặp
    Node pA = headA, pB = headB;
    while (pA != pB) {
        pA = pA == null ? headB : pA.next;
        pB = pB == null ? headA : pB.next;
    }
    return pA;
}
```

## • Mã nguồn Python

```
def getIntersectionNode(headA, headB):
    # Sử dụng hai vòng lặp
    pA = headA
    pB = headB
    while pA != pB:
        pA = headB if not pA else pA.next
        pB = headA if not pB else pB.next
    return pA
```

## - Độ phức tạp:

- Thời gian: O(m + n), trong đó m và n là chiều dài của hai linked list.
- Không gian: O(1), không sử dụng thêm không gian phụ đáng kể.

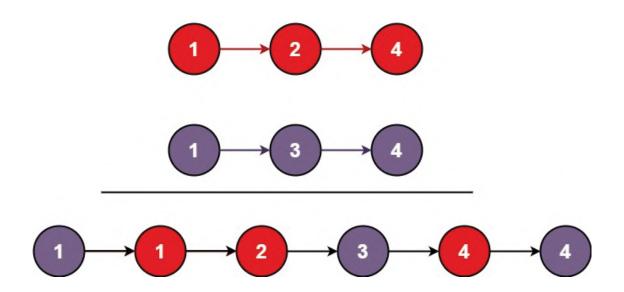
#### - Lưu ý:

- Bạn có thể tùy chỉnh code để sử dụng HashMap nếu muốn.
- Đừng quên xử lý trường hợp đặc biệt khi một trong hai linked list là null.
- Để kiểm tra kết quả, bạn có thể tạo hai linked list mẫu và cho chúng giao nhau tại một node bất kỳ.

### - Mở rộng:

• Tìm độ dài phần chung: Sau khi tìm được điểm giao, bạn có thể dễ dàng tính được độ dài của phần chung bằng cách duyệt tiếp từ điểm giao đến cuối.

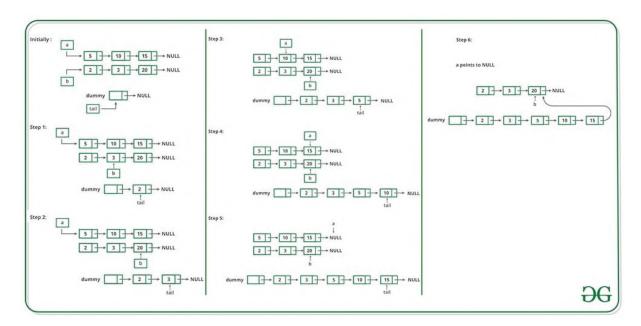
# 39. THUẬT TOÁN MERGE HAI LINKED LISTS ĐÃ ĐƯỢC SẮP XẾP



## - Giải thuật chung

- Tạo một danh sách liên kết mới: Đây sẽ là danh sách kết quả sau khi merge.
- So sánh phần tử đầu tiên của hai danh sách:
  - Đặt hai con trỏ vào hai phần tử đầu tiên của hai danh sách đầu vào
  - Bởi hai danh sách đã được sắp xếp trước, nên ta chỉ cần so sánh để tìm ra trong hai danh sách đầu vào thì phần tử đầu tiên của danh sách nào là nhỏ hơn.
  - Thêm phần tử nhỏ hơn vào danh sách liên kết mới và di chuyển con trỏ tương ứng sang node tiếp theo
- Lặp lại bước 2: Tiếp tục so sánh hai phần tử đang đứng dưới tham chiếu của hai con trỏ, tìm ra phần tử nhỏ hơn và thêm vào danh sách mới cho đến khi một trong hai danh sách trở thành rỗng.
- Thêm phần còn lại của danh sách chưa rỗng: Sau khi một danh sách trở thành rỗng, ta thêm phần còn lai của danh sách kia vào cuối danh sách kết quả.

### - Minh họa



### - Triển khai:

## • Mã nguồn C++

```
ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        // Tạo một node giả để làm đầu danh sách kết quả
        ListNode* dummy = new ListNode();
        ListNode* current = dummy;
        // Duyệt qua hai danh sách cho đến khi một trong hai danh
sách trở thành rỗng
        while (list1 && list2) {
            // So sánh giá trị của hai node hiện tại
            if (list1->val <= list2->val) {
               // Nếu node của list1 nhỏ hơn hoặc bằng, thêm vào
danh sách kết quả
                current->next = list1;
                list1 = list1->next;
            } else {
                // Ngược lại, thêm node của list2 vào danh sách
kết quả
                current->next = list2;
                list2 = list2->next;
            // Di chuyển con trỏ current đến node vừa thêm
            current = current->next;
        }
       // Thêm phần còn lại của danh sách chưa rỗng vào danh sách
kết quả
        current->next = list1 ? list1 : list2;
        // Trả về danh sách kết quả (bỏ qua node giả)
        return dummy->next;
```

### • Mã nguồn Java

```
public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
    // Tạo một node giả để làm đầu danh sách kết quả
   ListNode dummy = new ListNode();
   ListNode current = dummy;
   // Duyệt qua hai danh sách cho đến khi một trong hai danh sách
trở thành rỗng
    while (list1 != null && list2 != null) {
        // So sánh giá trị của hai node hiện tại
        if (list1.val <= list2.val) {</pre>
           // Nếu node của list1 nhỏ hơn hoặc bằng, thêm vào danh
sách kết quả
            current.next = list1;
            list1 = list1.next;
        } else {
            // Ngược lại, thêm node của list2 vào danh sách kết
quả
            current.next = list2;
            list2 = list2.next;
```

## • Mã nguồn Python

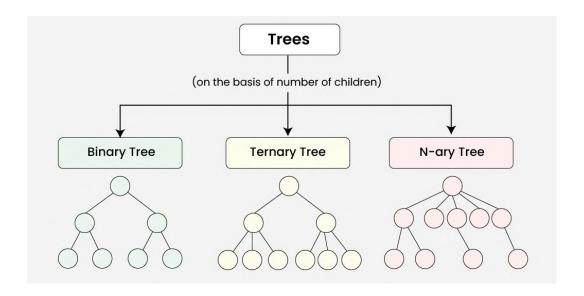
```
def mergeTwoLists(self, list1, list2):
    # Tạo một node giả để làm đầu danh sách kết quả
    dummy = ListNode()
    current = dummy
    # Duyệt qua hai danh sách cho đến khi một trong hai danh sách
trở thành rỗng
    while list1 and list2:
        # So sánh giá trị của hai node hiện tại
        if list1.val <= list2.val:</pre>
           // Nếu node của list1 nhỏ hơn hoặc bằng, thêm vào danh
sách kết quả
            current.next = list1
            list1 = list1.next
        else:
            // Ngược lại, thêm node của list2 vào danh sách kết
quả
            current.next = list2
            list2 = list2.next
        # Di chuyển con trỏ current đến node vừa thêm
        current = current.next
    # Thêm phần còn lại của danh sách chưa rỗng vào danh sách kết
quả
    current.next = list1 or list2
    # Trả về danh sách kết quả (bỏ qua node giả)
    return dummy.next
```

### - Điểm quan trong

- Dummy node: ta sử dụng một node giả để đơn giản hóa việc xử lý trường hợp danh sách kết quả rỗng ban đầu.
- Time complexity: O(m+n), với m và n là độ dài của hai danh sách.
- Space complexity: O(1), không sử dụng thêm không gian bộ nhớ đáng kể.



# 40. THUẬT TOÁN TẠO CÂY NHỊ PHÂN



- **Hiểu về bài toán:** Trước khi đi vào giải quyết, chúng ta cần hiểu rõ một số khái niệm cơ bản:
  - Cây nhị phân: Mỗi nút trong cây có tối đa hai con, gọi là nút con trái và nút con phải.
  - Mảng số: Một dãy các số có thứ tự.
  - Cây nhị phân chưa yêu cầu phải có quy luật về sự bố trí tương đối giữa nút con trái và nút con phải, chẳng hạn như ở trường hợp cây nhị phân tìm kiếm.
- Bài toán đặt ra: Cho một mảng số, ta muốn xây dựng một cây nhị phân sao cho các phần tử trong mảng tương ứng với các nút trong cây.

### - Cách tiếp cân:

Có nhiều cách để xây dựng cây nhị phân từ mảng, nhưng có hai cách phổ biến nhất, nhắm vào hai mục tiêu khác nhau:

- Xây dựng cây đầy đủ: Mỗi cấp của cây được điền đầy trước khi chuyển sang cấp tiếp theo.
- Xây dựng cây cân bằng: Cây được xây dựng sao cho độ cao của hai cây con của một nút bất kỳ khác nhau không quá 1.

Lưu ý: Để đơn giản, chúng ta sẽ xây dựng cây đầy đủ, từ một danh sách các số nguyên, trong các ví dụ dưới đây.

### - Triển khai:

```
Node* constructTree(int arr[], int start, int end) {
   if (start > end)
      return NULL;

int mid = (start + end) / 2;
   Node* root = newNode(arr[mid]);
   root->left = constructTree(arr, start, mid - 1);
   root->right = constructTree(arr, mid + 1, end);

return root;
}
```

### Mã nguồn Java

```
Node constructTreeUtil(int arr[], int start, int end) {
   if (start > end)
      return null;

   int mid = (start + end) / 2;
   Node node = new Node(arr[mid]);
   node.left = constructTreeUtil(arr, start, mid - 1);
   node.right = constructTreeUtil(arr, mid + 1, end);

   return node;
}

Node constructTree(int arr[]) {
   int n = arr.length;
   return constructTreeUtil(arr, 0, n - 1);
}
```

# • Mã nguồn Python

```
def constructTree(arr):
    if not arr:
        return None

mid = (len(arr)) // 2
    node = Node(arr[mid])
    node.left = constructTree(arr[:mid])
    node.right = constructTree(arr[mid+1:])

return node
```

### - Giải thích

- Cấu trúc Node: Mỗi nút trong cây có dữ liệu và các con trái, con phải.
- · Hàm khởi tạo tree sẽ:
- •Tìm phần tử giữa mảng để làm gốc.
- · Gọi đệ quy để xây dựng cây con trái và phải.
- · Hàm duyệt cây:
- Duyệt cây theo thể thức duyệt trước (pre-order) để kiểm tra kết quả.

#### - Lưu ý:

- Cây cân bằng: Để xây dựng cây cân bằng, có thể sử dụng các thuật toán như AVL tree (Adelson-Velsky and Landis tree cây nhị phân tìm kiếm) hoặc Red-Black tree.
- Các loại duyệt cây: Ngoài duyệt trước (duyệt theo thể thức pre-order), còn có duyệt giữa và duyệt sau.
- Úng dụng: Cây nhị phân được sử dụng rộng rãi trong nhiều thuật toán tìm kiếm, sắp xếp, và cấu trúc dữ liệu khác.

Với đoạn mã này, bạn có thể dễ dàng tạo ra một cây nhị phân từ một mảng số bất kỳ và thực hiện các phép toán khác trên cây.

# 41. THUẬT TOÁN DUYỆT CÂY NHỊ PHÂN

### - Hiểu bài toán:

Duyệt cây là quá trình đi qua tất cả các nút trong cây một lần. Đối với cây nhị phân, có ba cách duyệt cây phổ biến:

- Duyệt trước (Pre-order): Gốc -> Trái -> Phải
- Duyệt giữa (In-order): Trái -> Gốc -> Phải
- Duyệt sau (Post-order): Trái -> Phải -> Gốc

Mỗi cách duyệt sẽ cho ra một thứ tự khác nhau của các nút, và ứng dụng vào các bài toán khác nhau.

### - Triển khai:

```
void preorder(Node* root) {
    if (root == NULL)
       return;
    std::cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
void inorder(Node* root) {
    if (root == NULL)
       return;
    inorder(root->left);
    std::cout << root->data << " ";
    inorder(root->right);
void postorder(Node* root) {
    if (root == NULL)
       return;
   postorder(root->left);
   postorder(root->right);
    std::cout << root->data << " ";
```

## • Mã nguồn Java

```
void preorder(Node node) {
    if (node == null)
        return;
    System.out.print(node.data + " ");
    preorder(node.left);
    preorder(node.right);
void inorder(Node node) {
    if (node == null)
        return;
    inorder(node.left);
    System.out.print(node.data + " ");
    inorder(node.right);
void postorder(Node node) {
    if (node == null)
       return;
    postorder(node.left);
    postorder(node.right);
    System.out.print(node.data + " ");
}
```

## • Mã nguồn Python

```
def preorder(root):
    if root:
        print(root.data, end=' ')
        preorder(root.left)
        preorder(root.right)

def inorder(root):
    if root:
        inorder(root.left)
        print(root.data, end=' ')
        inorder(root):
    if root:
        postorder(root.left)
        postorder(root.left)
        postorder(root.left)
        postorder(root.left)
        postorder(root.left)
        postorder(root.data, end=' ')
```

#### - Giải thích

- Duyệt trước (Pre-order): In ra giá trị của nút gốc trước khi duyệt các cây con.
- Duyệt giữa (In-order): Duyệt cây con bên trái, sau đó in ra giá trị của nút gốc, cuối cùng duyệt cây con bên phải. Cách duyệt này sẽ cho ra dãy các nút theo thứ tự tăng dần.
- Duyệt sau (Post-order): Duyệt các cây con trước, sau đó mới in ra giá trị của nút gốc.

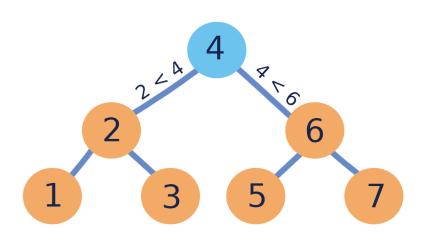
## - Ứng dụng

- Duyệt trước: Sử dụng để tạo một biểu diễn tiền tố của một biểu thức toán học.
- Duyệt giữa: Sử dụng để sắp xếp các phần tử trong cây.
- Duyệt sau: Sử dụng để giải phóng bộ nhớ của cây.

## - Lưu ý

- · Việc lựa chọn cách duyệt phụ thuộc vào bài toán cụ thể.
- Có nhiều cách khác để duyệt cây nhị phân, chẳng hạn như sử dụng stack hoặc queue.
- Đối với cây nhị phân tìm kiếm, duyệt giữa sẽ cho ra dãy các nút theo thứ tự tăng dần.

## 42. THUẬT TOÁN TẠO CÂY NHỊ PHÂN TÌM KIẾM



## In Order Traversal: 1 2 3 4 5 6 7

## - Hiểu về Cây nhị phân tìm kiếm:

Cây nhị phân tìm kiếm là một cấu trúc dữ liệu cây trong đó:

- Mỗi nút chứa một khóa (key).
- Các khóa trong cây con bên trái của một nút luôn nhỏ hơn khóa của nút đó.
- Các khóa trong cây con bên phải của một nút luôn lớn hơn khóa của nút đó.

Với cấu trúc dữ liệu như trên, cây nhị phân tìm kiếm có thể dễ dàng được duyệt theo thứ tự tăng dần, có thể thêm hoặc gỡ bất kỳ nút nào trên cây mà không làm hỏng thứ tự đó, cũng hỗ trợ tìm kiếm bất kỳ phần tử nào trên cây một cách đặc biệt nhanh.

## - Thuật toán Tạo cây BST

Ý tưởng chính:

- · Khởi tạo một nút mới với giá trị cần chèn.
- So sánh giá trị mới với giá trị của nút gốc:
  - · Nếu nhỏ hơn: Đệ quy chèn vào cây con bên trái.
  - Nếu lớn hơn: Đệ quy chèn vào cây con bên phải.
  - Nếu bằng: Có thể chọn cách xử lý khác nhau (ví dụ: không chèn, chèn nhiều nút có cùng giá trị).

#### - Triển khai:

## • Mã nguồn C++

```
Node* insert(Node* node, int data) {
   if (node == NULL) return newNode(data);

if (data < node->data)
      node->left = insert(node->left, data);
   else if (data > node->data)
      node->right = insert(node->right, data);

return node;
}
```

### • Mã nguồn Java

```
void insert(int data) {
    root = insertRec(root, data);
}

Node insertRec(Node root, int data) {
    if (root == null) {
        root = new Node(data);
        return root;
    }

    if (data < root.data)
        root.left = insertRec(root.left, data);
    else if (data > root.data)
        root.right = insertRec(root.right, data);

    return root;
}
```

```
def insert(self, root, node):
    if root is None:
        root = node
    else:
        if root.data < node.data:
            if root.right is None:
                root.right = node
        else:
                self.insert(root.right, node)
    else:
        if root.left is None:
                root.left = node
        else:
                self.insert(root.left, node)
    return root</pre>
```

#### - Giải thích

- Cấu trúc Node: Mỗi nút có 3 phần: dữ liệu, con trái và con phải.
- · Hàm insert:
  - Điều kiện dừng: Nếu nút hiện tại là NULL, tạo nút mới và trả về.
  - So sánh: So sánh giá trị cần chèn với giá trị của nút hiện tại.
  - Đệ quy: Gọi đệ quy cho cây con bên trái hoặc bên phải tùy thuộc vào kết quả so sánh.

## - Ứng dụng

Cây BST được sử dụng rộng rãi trong các thuật toán tìm kiếm, sắp xếp, và lưu trữ dữ liêu. Một số ứng dụng phổ biến:

- Tìm kiếm: Tìm kiếm một phần tử trong cây có độ phức tạp trung bình là O(log n).
- Sắp xếp: Duyệt cây theo thứ tự inorder sẽ cho ra danh sách các phần tử đã được sắp xếp.
- Thực hiện các phép toán tập hợp: Tìm giao, hợp, hiệu của hai tập hợp.
- Lưu trữ dữ liệu: Sử dụng trong các cơ sở dữ liệu, cấu trúc dữ liệu khác.

#### - Lưu ý:

- Để đảm bảo hiệu suất tốt, nên cân bằng cây BST.
- Có nhiều thuật toán cân bằng cây BST khác nhau như AVL tree, Red-Black tree.

## 43. THUẬT TOÁN TÌM KIẾM PHẦN TỬ TRONG CÂY NHỊ PHÂN TÌM KIẾM

## - Hiểu về thuật toán

Thuật toán tìm kiếm trong cây nhị phân tìm kiếm, về bản chất, là tìm kiếm phần tử từ một danh sách đã được sắp xếp. Theo đó mỗi một lần đệ quy, phép tìm kiếm có thể loại bỏ đi một nửa số lượng phần tử cần phải duyệt, chính điều này tạo ra hiệu quả tìm kiếm vượt trội so với tìm kiếm tuyến tính thông thường.

Và với cấu trúc dữ liệu đặc biệt của mình, các cây nhị phân tìm kiếm có thể dễ dàng bảo toàn tính chất "được sắp xếp" của mình. Đó là lý do tại sao cây nhị phân dạng này lại được gọi là cây "tìm kiếm".

Ý tưởng chính đằng sau phép tìm kiếm là so sánh giá trị cần tìm với giá trị tại nút gốc:

- · Nếu bằng: Đã tìm thấy phần tử.
- Nếu nhỏ hơn: Tiếp tục tìm kiếm trong cây con bên trái.
- Nếu lớn hơn: Tiếp tục tìm kiếm trong cây con bên phải.

#### - Triển khai:

## • Mã nguồn C++

```
Node* search(Node* root, int key) {
   if (root == NULL || root->data == key)
        return root;

if (key < root->data)
        return search(root->left, key);

return search(root->right, key);
}
```

## • Mã nguồn Java

```
Node search(Node root, int key) {
   if (root==null || root.data==key)
      return root;

if (key < root.data)
      return search(root.left, key);

return search(root.right, key);
}</pre>
```

```
def search(root, key):
    if root is None or root.data == key:
        return root

if key < root.data:
        return search(root.left, key)

return search(root.right, key)</pre>
```

#### - Giải thích

- Điều kiện dừng: Nếu cây rỗng hoặc giá trị tại nút gốc bằng giá trị cần tìm thì trả về nút đó.
- So sánh: So sánh giá trị cần tìm với giá trị tại nút gốc.
- Đệ quy: Gọi đệ quy cho cây con bên trái hoặc bên phải tùy thuộc vào kết quả so sánh.

## - Độ phức tạp

Trong trường hợp cây cân bằng, độ phức tạp trung bình của thuật toán tìm kiếm là O(log n), trong đó n là số lượng nút trong cây. Tuy nhiên, trong trường hợp xấu nhất (cây bị lệch hản về một bên), độ phức tạp có thể lên đến O(n).

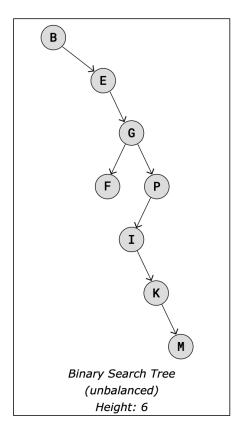
## - Ứng dụng

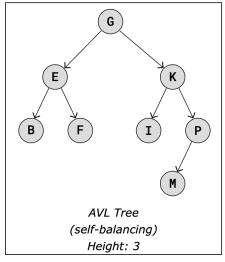
- Tìm kiếm nhanh: Tìm kiếm một phần tử trong một tập hợp lớn các phần tử đã được sắp xếp.
- Cơ sở dữ liêu: Sử dung để truy vấn dữ liêu.
- Cấu trúc dữ liệu khác: Làm nền tảng cho các cấu trúc dữ liệu phức tạp hơn.

#### - Lưu ý

- Để đảm bảo hiệu suất tìm kiếm, nên thực hiện cân bằng cây.
- Có thể mở rộng thuật toán để tìm kiếm phần tử lớn nhất/nhỏ nhất, tìm kiếm khoảng, v.v.

## 44. THUẬT TOÁN CÂN BẰNG CÂY NHỊ PHÂN TÌM KIẾM





## - Tại sao cần cân bằng cây nhị phân?

Một cây nhị phân tìm kiếm khi không được cân bằng có thể trở nên lệch về một bên, dẫn đến hiệu suất tìm kiếm, chèn, xóa giảm sút, thậm chí tiệm cận với O(n) trong trường hợp xấu nhất. Để đảm bảo hiệu suất tìm kiếm luôn ở mức tối ưu O(log n), chúng ta cần cân bằng cây.

## - Các thuật toán cân bằng phổ biến

Có nhiều thuật toán cân bằng cây nhị phân tìm kiếm, trong đó hai thuật toán phổ biến nhất là:

- AVL tree: Mỗi nút trong cây AVL có một yếu tố cân bằng (balance factor) thể hiện sự chênh lệch chiều cao giữa hai cây con. Khi chèn hoặc xóa nút, cây sẽ được tự động cân bằng lại để đảm bảo rằng yếu tố cân bằng của mọi nút nằm trong khoảng [-1, 1].
- Red-black tree: Mỗi nút trong cây Red-black tree có một màu (đỏ hoặc đen) và tuân theo một số quy tắc nhất định. Các quy tắc này đảm bảo rằng đường đi dài nhất từ gốc đến bất kỳ lá nào cũng không quá gấp đôi đường đi ngắn nhất.

#### - Cài đăt thuật toán AVL

Thuật toán AVL hoạt động dựa trên một thuộc tính của cây là height (chiều cao). Về cơ bản, chiều cao của một nút = max(chiều cao của cây con trái, chiều cao của cây con phải) + 1. Như vậy một nút không có nút con thì sẽ có chiều cao là 1.

```
struct Node {
   int data;
   Node *left, *right;
   int height;
};
```

Khi phát hiện có sự chênh lệch chiều cao giữa cây con bên trái và bên phải, thuật toán AVL sử dụng thao tác "xoay" (rotate) để cân bằng cây. Phép xoay được thiết kế để không làm hỏng cây - cây vẫn là cây nhị phân tìm kiếm sau khi xoay.

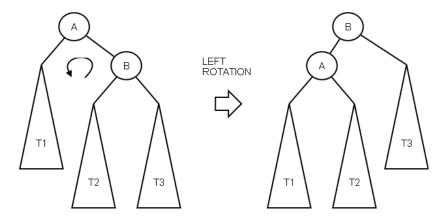
Sự cân bằng được tính bằng cách tính chênh lệch giữa chiều cao của cây con bên trái và bên phải, chênh lệch này được gọi là balance factor. Cây được gọi là cân bằng nếu balance factor nằm trong khoảng từ -1 đến +1. Số 0 là cân bằng, 1 và -1 thể hiện rằng cây con trái (hay phải) cao hơn bên còn lại, nhưng chỉ hơn 1 đơn vị.

```
int getBalance(Node* N) {
   if (N == NULL)
      return 0;
   return height(N->left) - height(N->right);
}
```

Để phòng ngừa trường hợp N chỉ có một nút con, ta coi một nút rỗng có chiều cao bằng 0:

```
int height(Node* N) {
   if (N == NULL)
      return 0;
   return N->height;
}
```

Về cơ bản, AVL sẽ xoay trái khi cây lệch về bên phải:



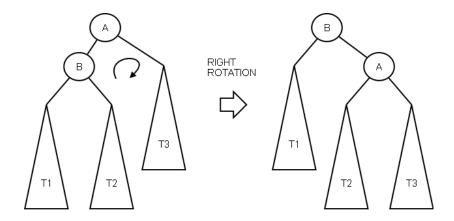
```
Node* leftRotate(Node* x) {
   Node* y = x->right;
   Node* T2 = y->left;

   // Thực hiện xoay
   y->left = x;
   x->right = T2;

   // Cập nhật chiều cao
   x->height = max(height(x->left), height(x->right)) + 1;
   y->height = max(height(y->left), height(y->right)) + 1;

   // Trả về nút gốc mới
   return y;
}
```

... và xoay phải khi cây lệch về bên trái:



```
Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;

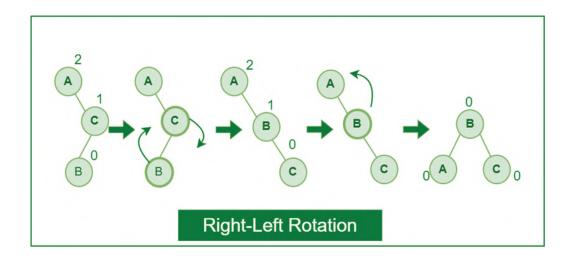
    // Thực hiện xoay
    x->right = y;
    y->left = T2;

    // Cập nhật chiều cao
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

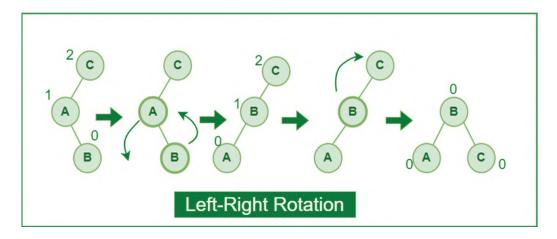
    // Trả về nút gốc mới
    return x;
}
```

Tuy nhiên, có những trường hợp mà sau khi thực hiện một trong hai phép xoay trên, cây sẽ tiếp tục rơi vào trạng thái mất cân bằng. Khi đó ta cần thực hiện phép xoay kép, cụ thể như dưới đây.

Cây lệch về bên phải, nhưng cây con bên phải lại lệch về bên trái, ta đảo ngược điều này bằng cách xoay phải trên cây con phải. Sau đó xoay trái trên nút gốc như bình thường:



Cây lệch về bên trái, nhưng cây con bên trái lại lệch về bên phải. Ta đảo ngược điều này bằng cách xoay trái trên cây con phải. Sau đó thực hiện xoay phải trên nút gốc:



Với các yếu tố cơ bản như trên, thuật toán để cân bằng cây dựa trên chiều cao sẽ như sau:

```
Node* balance(Node* node) {
   int balanceFactor = getBalance(node);

if (balanceFactor > 1) {
    if (getBalance(node->left) < 0)
        node->left = leftRotate(node->left);
    return rightRotate(node);
} else if (balanceFactor < -1) {
    if (getBalance(node->right) > 0)
        node->right = rightRotate(node->right);
    return leftRotate(node);
}

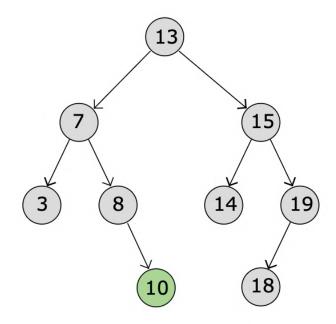
/* Cây đã cân bằng */
   return node;
}
```

Mỗi khi có thay đổi (thêm, xóa nút) trên cây, thông thường ta sẽ sử dụng thuật toán duyệt cây (sử dụng đệ quy) và gọi hàm balance ở trên mỗi khi tìm thấy một nút nào đó mất cân bằng.

## - Lưu ý

- Độ phức tạp: Độ phức tạp của các phép toán trên cây AVL và Red-Black tree thường là O(log n).
- Các loại cây cân bằng: Ngoài AVL tree và Red-Black tree, còn có các loại cây cân bằng khác như AA tree, Splay tree, ... Mỗi loại sẽ thể hiện có ưu nhược điểm khác nhau trong các trường hợp khác nhau.

## 45. THUẬT TOÁN THÊM PHẦN TỬ VÀO CÂY NHỊ PHÂN TÌM KIẾM



## - Cây nhị phân tìm kiếm (BST):

Là một cấu trúc dữ liệu hiệu quả cho việc tìm kiếm, chèn và xóa. Mỗi nút trong cây đều có một khóa (key), và các khóa trong cây con trái luôn nhỏ hơn khóa của nút cha, trong khi các khóa trong cây con phải luôn lớn hơn. Khi thực hiện các thay đổi trên cây (chẳng hạn như khi chèn, hay khi thực hiện cân bằng), ta cần phải giữ cho cây luôn tuân thủ trạng thái này.

## - Thuật toán thêm phần tử:

- 1. Khởi tạo cây: Nếu cây rỗng, tạo một nút mới và gán giá trị cần chèn vào nút đó.
- 2. So sánh giá trị: So sánh giá trị cần chèn với giá trị của nút hiện tại:
  - Nhỏ hơn: Di chuyển xuống cây con trái và lặp lại bước 2.
  - Lớn hơn: Di chuyển xuống cây con phải và lặp lại bước 2.
  - Bằng nhau: Giá trị đã tồn tại trong cây, không cần chèn thêm.
- 3. Tìm vị trí chèn: Khi đến một nút lá (không có con), tạo một nút mới tại vị trí đó và gán giá trị cần chèn.

#### - Triển khai:

#### • Mã nguồn C++

```
Node* insert(Node* root, int data) {
   if (root == NULL) {
      return newNode(data);
   }

   if (data < root->data) {
      root->left = insert(root->left, data);
   } else if (data > root->data) {
      root->right = insert(root->right, data);
   }

   return root;
}
```

## • Mã nguồn Java

```
Node insert(Node root, int data) {
   if (root == null) {
      return new Node(data);
   }

   if (data < root.data) {
      root.left = insert(root.left, data);
   } else if (data > root.data) {
      root.right = insert(root.right, data);
   }

   return root;
}
```

## • Mã nguồn Python

```
def insert(root, data):
    if root is None:
        return Node(data)

if data < root.data:
        root.left = insert(root.left, data)

elif data > root.data:
        root.right = insert(root.right, data)
return root
```

#### - Giải thích code:

- •Trường hợp cây rỗng: Lấy nút cần chèn làm nút gốc.
- Giá trị cần chèn nhỏ hơn giá trị tại nút gốc: Gọi đệ quy để chèn vào cây con trái.
- Giá trị cần chèn lớn hơn giá trị tại nút gốc: Gọi đệ quy để chèn vào cây con phải.

### - Lưu ý:

Nếu muốn giữ cây ở trạng thái cân bằng, ta cũng sẽ gọi thuật toán cân bằng cây mỗi khi thực hiện chèn phần tử.



## 46. HAI CHUỗI CÙNG CÓ CÁC KÝ TỰ GIỐNG NHAU

#### - Đề bài:

Cho hai chuỗi string1 và string2. Hãy kiểm tra xem string1 và string2 có cùng chứa các ký tự giống như nhau hay không (trật tự của các ký tự có thể khác nhau, không phân biệt chữ hoa và chữa thường).

#### - Thuât toán:

Ý tưởng chính của thuật toán này là sử dụng bảng băm để lưu trữ tần suất xuất hiện của mỗi ký tự, sau đó đối chiếu với chuỗi ký tự thứ hai để kiểm tra xem chúng có chứa cùng những ký tự đó hay không.

- 1. Loại bỏ khoảng trắng và chuyển hai chuỗi ký tự sang chữ thường để so sánh không phân biệt chữ hoa chữ thường.
- 2. Kiểm tra xem hai chuỗi ký tự có cùng độ dài không. Nếu không, chúng không thể là anagram nên trả về kết quả false.
- 3. Tạo ra một bảng băm (dictionary) để lưu trữ tần suất xuất hiện của mỗi ký tự trong chuỗi ký tự thứ nhất.
- 4. Duyệt qua chuỗi ký tự thứ nhất, tăng giá trị tương ứng với mỗi ký tự trong bảng băm.
- 5. Duyệt qua chuỗi ký tự thứ hai, giảm giá trị tương ứng với mỗi ký tự trong bảng băm. Nếu gặp ký tự không có trong bảng băm, trả về kết quả false vì chúng không thể là anagram.
- 6. Kiểm tra tất cả các giá trị trong bảng băm có bằng 0 không. Nếu có, trả về kết quả true vì chúng là anagram. Nếu không, trả về kết quả false.

### - Triển khai:

#### • Mã nguồn C++

```
bool checkAnagram(string s, string t) {
    if (s.length() != t.length()) {
        return false;
    }

    unordered_map<char, int> count;

    for (char c : s) {
        count[tolower(c)]++;
    }

    for (char c : t) {
        count[tolower(c)]--;
        if (count[tolower(c)] < 0) {
            return false;
        }
    }

    return true;
}</pre>
```

## • Mã nguồn Java

```
public static boolean isAnagram(String s, String t) {
    if (s.length() != t.length()) {
       return false;
    }
    Map<Character, Integer> count = new HashMap<>();
    for (char c : s.toCharArray()) {
             count.put(Character.toLowerCase(c), count.getOrDe-
fault(Character.toLowerCase(c), 0) + 1);
    for (char c : t.toCharArray()) {
       int countC = count.getOrDefault(Character.toLowerCase(c),
0);
        if (countC == 0) {
           return false;
        count.put(Character.toLowerCase(c), countC - 1);
    }
    return true;
}
```

## • Mã nguồn Python

```
def is_anagram(s, t):
    if len(s) != len(t):
        return False

count = defaultdict(int)

for char in s:
        count[char.lower()] += 1

for char in t:
        count[char.lower()] -= 1
        if count[char.lower()] < 0:
        return False

return True</pre>
```

## 47. TÌM ĐỘ DÀI ĐOẠN CHUỐI CON CHUNG DÀI NHẤT

#### - Đề bài:

Cho 2 chuỗi string1 và string2. Hãy tìm độ dài của đoạn chuỗi chung dài nhất giữa hai chuỗi này.

### - Ví dụ:

Hai chuỗi string1 = "ABCDGH" và string2 = "ACDGHR" có độ dài của đoạn chuỗi chung dài nhất là 4.

Giải thích: Đoạn chuỗi chung dài nhất là "CDGH", có độ dài là 4.

## - Giải thuật:

Sử dụng quy hoạch động để giải quyết bài toán này. Ý tưởng chính là xây dựng một ma trận để lưu trữ độ dài chuỗi con chung dài nhất của các tiền tố của hai chuỗi.

#### - Triển khai:

## • Mã nguồn C++

```
int longestCommonSubsequence(string text1, string text2) {
   int m = text1.length();
   int n = text2.length();

   // Tạo ma trận dp để lưu trữ độ dài chuỗi con chung dài nhất
   vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

   for (int i = 1; i <= m; ++i) {
      for (int j = 1; j <= n; ++j) {
        if (text1[i - 1] == text2[j - 1]) {
            dp[i][j] = dp[i - 1][j - 1] + 1;
        } else {
            dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }
    }
   return dp[m][n];
}</pre>
```

## • Mã nguồn Java

```
public static int longestCommonSubsequence(String text1, String
text2) {
   int m = text1.length();
   int n = text2.length();

   int[][] dp = new int[m + 1][n + 1];

   for (int i = 1; i <= m; i++) {
      for (int j = 1; j <= n; j++) {
        if (text1.charAt(i - 1) == text2.charAt(j - 1)) {
            dp[i][j] = dp[i - 1][j - 1] + 1;
        } else {
            dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
      }
}</pre>
```

```
}
}
return dp[m][n];
}
```

```
def longest_common_subsequence(text1, text2):
    m = len(text1)
    n = len(text2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

for i in range(1, m + 1):
    for j in range(1, n + 1):
        if text1[i - 1] == text2[j - 1]:
            dp[i][j] = dp[i - 1][j - 1] + 1
        else:
            dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

return dp[m][n]
```

#### - Giải thích:

- 1. Ma trận dp: Mỗi phần tử dp[i][j] lưu trữ độ dài chuỗi con chung dài nhất của các tiền tố text1[0..i-1] và text2[0..j-1].
- 2. Khởi tạo: Hàng và cột đầu tiên của ma trận được khởi tạo bằng 0 vì chuỗi rỗng không có chuỗi con chung.
- 3. Điền giá trị:
  - Trường hợp bằng nhau: Nếu text1[i-1] bằng text2[j-1], thì độ dài chuỗi con chung dài nhất tăng thêm 1 so với trường hợp trước đó (dp[i-1][j-1]).
  - Trường hợp khác nhau: Chọn giá trị lớn nhất giữa độ dài chuỗi con chung dài nhất của các tiền tố text1[0..i-1] và text2[0..j-1], hoặc text1[0..i] và text2[0..j-1].
- 4. Kết quả: Giá trị cuối cùng dp[m][n] chính là độ dài chuỗi con chung dài nhất của hai chuỗi.

#### - Độ phức tạp:

- Thời gian: O(m\*n), trong đó m và n là đô dài của hai chuỗi.
- Không gian: O(m\*n) để lưu trữ ma trân dp.

#### - Lưu ý:

- Tối ưu hóa: Có thể sử dụng kỹ thuật backtracking để tìm ra các chuỗi con chung dài nhất cụ thể, không chỉ độ dài.
- Úng dụng: Bài toán này có nhiều ứng dụng thực tế như so sánh trình tự gen, tìm kiếm các đoạn văn bản trùng lặp, ...

## 48. KIỂM TRA MỘT CHUỗI CÓ PHẢI LÀ CHUỖI CON CỦA MỘT CHUỗI KHÁC HAY KHÔNG

#### - Mô tả bài toán:

Cho hai chuỗi string1 và string2, hãy kiểm tra xem string1 có phải là chuỗi con của string2 hay không. Một chuỗi con là một chuỗi có thể được tạo ra từ một chuỗi khác bằng cách xóa đi một số hoặc không có ký tự nào mà không làm thay đổi thứ tự của các ký tự còn lại.

### - Ví dụ 1:

```
Cho hai chuỗi: str1 = "abc", str2 = "aebdc"
Kết quả: : true (Vì "abc" là chuỗi con của "aebdc")
```

#### - Ví dụ 2:

Cho hai chuỗi: string1 = "abc", string2 = "acbd" Kết quả: false (Vì "abc" không phải là chuỗi con của "acbd")

- Thuật toán cơ bản: Sử dụng hai vòng lặp lồng nhau
  - · Vòng lặp ngoài: Duyệt qua từng ký tự của chuỗi lớn.
  - Vòng lặp trong: Từ vị trí hiện tại của chuỗi lớn, so sánh từng ký tự với chuỗi con. Nếu tất cả các ký tự đều khớp, thì chuỗi con đã được tìm thấy.

#### - Triển khai:

• Mã nguồn C++

#### Mã nguồn Java

```
public boolean isSubstring(String text, String pattern) {
   int n = text.length();
   int m = pattern.length();

   for (int i = 0; i <= n - m; i++) {
      int j = 0;
      while (j < m && text.charAt(i + j) == pattern.charAt(j))
   {
         j++;
      }
      if (j == m) {</pre>
```

```
return true;
}
}
return false;
}
```

```
def is_substring(text, pattern):
    n = len(text)
    m = len(pattern)

for i in range(n - m + 1):
    j = 0
    while j < m and text[i + j] == pattern[j]:
        j += 1
    if j == m:
        return True
    return False</pre>
```

#### - Giải thích:

- Vòng lặp ngoài: Duyệt qua từng vị trí bắt đầu có thể của chuỗi con trong chuỗi chính.
- Vòng lặp trong: So sánh từng ký tự của chuỗi con với các ký tự tương ứng trong chuỗi chính. Nếu tất cả các ký tư đều khớp, trả về true.
- Điều kiện dừng: Nếu không tìm thấy sự khớp hoặc đã duyệt hết chuỗi chính, trả về false.

## - Độ phức tạp:

- Thời gian: O(mn), trong đó m là độ dài của chuỗi con và n là độ dài của chuỗi chính.
- Không gian: O(1), không sử dụng thêm không gian phụ đáng kể.

#### - Lưu ý:

- Thuật toán này khá cơ bản. Có nhiều thuật toán hiệu quả hơn như thuật toán KMP, Boyer-Moore để giải quyết bài toán này, đặc biệt khi thực hiện nhiều lần tìm kiếm trên cùng một chuỗi.
- Các thư viện: Hầu hết các ngôn ngữ lập trình đều có các thư viện hỗ trợ tìm kiếm chuỗi, bạn có thể sử dụng các hàm có sẵn để đơn giản hóa quá trình.

#### - Mở rông:

- •Tìm tất cả các vị trí xuất hiện của chuỗi con: Thay vì chỉ trả về true hoặc false, bạn có thể lưu lại các vị trí bắt đầu tìm thấy chuỗi con.
- Tìm chuỗi con dài nhất: Bạn có thể sửa đổi thuật toán để tìm chuỗi con chung dài nhất giữa hai chuỗi.

## 49. THUẬT TOÁN KIỂM TRA CHUỗI ĐỐI XỨNG

Chuỗi đối xứng là chuỗi đọc xuôi hay ngược đều giống nhau, ví dụ: "abba", "racecar".

#### - Thuât toán:

- 1. Khởi tạo hai chỉ số:
  - · left: Chỉ vào ký tự đầu tiên của chuỗi.
  - right: Chỉ vào ký tự cuối cùng của chuỗi.
- 2. So sánh ký tự:
  - · So sánh ký tự tại vị trí left và right.
  - Nếu hai ký tự bằng nhau, tăng left và giảm right.
  - · Nếu không bằng nhau, trả về false.
- 3. Lặp lại bước 2: Tiếp tục so sánh cho đến khi left lớn hơn hoặc bằng right.
- 4. Trả về kết quả: Nếu tất cả các cặp ký tự đều bằng nhau, trả về true, ngược lại trả về false.

### - Triển khai:

## • Mã nguồn C++

```
bool isPalindrome(string s) {
   int left = 0;
   int right = s.length() - 1;

   while (left < right) {
      if (tolower(s[left]) != tolower(s[right])) {
         return false;
      }
      left++;
      right--;
   }
   return true;
}</pre>
```

## • Mã nguồn Java

```
public boolean isPalindrome(String str) {
   int left = 0;
   int right = str.length() - 1;

   while (left < right) {
       if (Character.toLowerCase(str.charAt(left)) != Character.toLowerCase(str.charAt(right))) {
            return false;
       }
       left++;
       right--;
    }
    return true;
}</pre>
```

```
def is_palindrome(s):
    left, right = 0, len(s) - 1
    while left < right:
        if s[left].lower() != s[right].lower():
            return False
        left += 1
        right -= 1
    return True</pre>
```

#### - Giải thích:

- · Hàm isPalindrome:
- · Nhân vào một chuỗi s.
- Khởi tạo hai chỉ số left và right lần lượt ở đầu và cuối chuỗi.
- Sử dụng vòng lặp while để so sánh từng cặp ký tự từ hai đầu vào trong.
- Hàm tolower được sử dụng để chuyển đổi các ký tự thành chữ thường, giúp so sánh không phân biệt chữ hoa và chữ thường.
- Điều kiện dừng: Khi left lớn hơn hoặc bằng right, nghĩa là tất cả các cặp ký tự đã được so sánh.
- Kết quả: Nếu tất cả các cặp ký tự đều bằng nhau, chuỗi là đối xứng, ngược lại không phải.

## - Ưu điểm của thuật toán:

- Hiệu quả: Độ phức tạp thời gian là O(n/2) ≈ O(n), trong đó n là độ dài của chuỗi.
- Dễ hiểu: Thuật toán dễ hiểu và triển khai.
- Linh hoạt: Có thể dễ dàng mở rộng để xử lý các trường hợp phức tạp hơn, như kiểm tra đối xứng bỏ qua dấu cách hoặc ký tự đặc biệt.

#### - Các cải tiến:

- Bỏ qua khoảng trắng và ký tự đặc biệt: Có thể thêm logic để bỏ qua các ký tự không phải chữ cái hoặc số trước khi so sánh.
- Kiểm tra đối xứng cho các kiểu dữ liệu khác: Có thể áp dụng thuật toán tương tự cho các kiểu dữ liệu khác như mảng, danh sách liên kết.

## - Ứng dụng:

Thuật toán này là một trong những thuật toán cơ bản và được sử dụng rộng rãi trong xử lý chuỗi.

- · Kiểm tra tính đối xứng của các từ, câu.
- Xử lý các bài toán liên quan đến palindrome trong lập trình.
- Kiểm tra tính hợp lệ của các cấu trúc dữ liệu.

## 50. TÌM CHUỗI CON ĐỐI XỨNG DÀI NHẤT

#### - Đề bài:

Cho một chuỗi, hãy tìm chuỗi con đối xứng (palindromic substring) dài nhất. Một chuỗi đối xứng là chuỗi mà khi đọc từ trái sang phải cũng như từ phải sang trái đều giống nhau.

#### - Ví du:

- Đầu vào: "babad"
- Đầu ra: "bab" hoặc "aba"
- Đầu vào: "cbbd"
- Đầu ra: "bb"

## - Ý tưởng sử dụng quy hoạch động:

- Ma trận dp: Tạo một ma trận dp[i][j] để lưu trữ thông tin về việc đoạn con từ vị trí i đến vị trí j có phải là chuỗi đối xứng hay không.
- Điều kiên cơ sở:
- Các phần tử trên đường chéo chính (i = j) đều là chuỗi đối xứng (độ dài 1).
- Nếu hai ký tự ở vị trí i và j bằng nhau và đoạn con từ i+1 đến j-1 là chuỗi đối xứng thì đoan con từ i đến j cũng là chuỗi đối xứng.
- Công thức truy hồi: dp[i][j] = (s[i] == s[j]) && (i + 1 >= j || dp[i + 1][j 1])

#### - Triển khai:

• Mã nguồn C++

```
string longestPalindrome(string s) {
   int n = s.length();
   bool dp[n][n];
   int start = 0, end = 0;

   for (int i = 0; i < n; ++i) {
        for (int j = 0; j <= i; ++j) {
            dp[i][j] = (s[i] == s[j]) && (i - j < 2 || dp[i - 1][j + 1]);

        if (dp[i][j] && end - start < i - j) {
            start = j;
            end = i;
            }
        }
    }
   return s.substr(start, end - start + 1);
}</pre>
```

## • Mã nguồn Java

```
public String longestPalindrome(String s) {
    int n = s.length();
    boolean[][] dp = new boolean[n][n];
    int start = 0, end = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= i; j++) {
            dp[i][j] = (s.charAt(i) == s.charAt(j)) && (i - j < 2)
|| dp[i - 1][j + 1]);
            if (dp[i][j] && end - start < i - j) {
                start = j;
                end = i;
            }
        }
    }
    return s.substring(start, end + 1);
}
```

## Mã nguồn Python

```
def longest_palindrome(s):
    n = len(s)
    dp = [[False] * n for _ in range(n)]
    start = 0
    end = 0

    for i in range(n):
        for j in range(i + 1):
            dp[i][j] = (s[i] == s[j]) and (i - j < 2 or dp[i - 1][j + 1])

        if dp[i][j] and end - start < i - j:
            start = j
            end = i

    return s[start:end + 1]</pre>
```

#### - Giải thích:

- Ma trận dp: Mỗi phần tử dp[i][j] cho biết đoạn con từ i đến j có phải là chuỗi đối xứng hay không.
- Vòng lặp: Duyệt qua tất cả các đoạn con có thể của chuỗi.
- Điều kiện: Nếu hai ký tự đầu cuối của đoạn con bằng nhau và đoạn con bên trong cũng là chuỗi đối xứng thì đoạn con hiện tại là chuỗi đối xứng.
- Cập nhật kết quả: Nếu tìm được chuỗi đối xứng dài hơn, cập nhật start và end.
- Trả về kết quả: Trả về chuỗi đối xứng dài nhất tìm được.

## - Độ phức tạp:

- •Thời gian: O(n^2)
- Không gian: O(n^2)

#### - Ưu điểm:

- · Hiệu quả: Giải quyết bài toán một cách hệ thống và hiệu quả.
- Dễ hiểu: Logic rõ ràng, dễ theo dõi.

## - Nhược điểm:

• Sử dụng nhiều bộ nhớ: Cần một ma trận kích thước n x n.

## - Lưu ý:

- Cổ thể tối ưu hóa bằng cách sử dụng kỹ thuật Manacher để giảm độ phức tạp không gian xuống O(n).
- •Thuật toán này có thể được mở rộng để tìm tất cả các chuỗi đối xứng trong một chuỗi.

## - Ứng dụng:

- Xử lý ngôn ngữ tự nhiên
- Sinh học tính toán
- Mã hóa

# LỜI KẾT

Chúng ta đã cùng nhau đi qua một hành trình đầy thú vị và bổ ích với 50 thuật toán quan trọng trong lĩnh vực lập trình. Mỗi thuật toán được trình bày trong cẩm nang này đều là một mảnh ghép quan trọng trong bộ kỹ năng của một lập trình viên xuất sắc, đóng vai trò thiết yếu trong việc giải quyết các vấn đề thực tế trong lập trình.

Hãy nhớ rằng, việc học và áp dụng thuật toán không phải là điểm đến cuối cùng, mà là một quá trình liên tục. Công nghệ luôn thay đổi, và với nó, những thuật toán mới, những cách tiếp cận mới sẽ không ngừng xuất hiện. Vì vậy, hãy giữ cho mình tinh thần học hỏi không ngừng, luôn tò mò và sẵn sàng đón nhận những thách thức mới.

Sức mạnh thực sự của một lập trình viên không chỉ nằm ở việc ghi nhớ thuật toán, mà còn ở khả năng phân tích vấn đề, lựa chọn công cụ phù hợp và áp dụng chúng một cách sáng tạo. Hãy tiếp tục rèn luyện, thực hành và không ngừng nâng cao kỹ năng của mình.

Chúc bạn thành công trên con đường sự nghiệp và hy vọng rằng cuốn cẩm nang này sẽ là người bạn đồng hành đáng tin cậy trong hành trình đó.

#### CODEGYM Raising the bar

THUẬT TOÁN CHO LẬP TRÌNH VIÊN