Chuyên đề A2: Đệ quy quay lui

Ban Chuyên môn Tin học – Tổ chức The Gifted Battlefield nhiệm kỳ 2024-2025

Xuất bản vào Ngày 5 tháng 2 năm 2025

Mục lục

1 Đệ quy **2**
1.1 Định nghĩa . . . 2
1.2 Cài đặt minh họa (tính giai thừa) . . . 2

2 Đệ quy quay lui (Backtracking) **2**
2.1 Định nghĩa . . . 2
2.2 Cách thức hoạt động . . . 2
2.3 Bài tập minh họa: Bài toán N-Queen . . . 2

3 Sinh hoán vị (Permutations) **4**
3.1 Bài toán . . . 4
3.2 Đệ quy quay lui . . . 4
3.3 Mở rộng (Sinh hoán vị bằng hàm `next_permutation`) . . . 5

4 Sinh tập con (Subsets) **6**
4.1 Đệ quy quay lui . . . 6
4.2 Mở rộng (Sinh tập con bằng bitmask) . . . 7

5 Cải tiến Đệ quy quay lui **8**
5.1 Phương pháp Nhánh cận (Branch and Bound) . . . 8
5.2 Phương pháp Cắt tỉa (Pruning) . . . 9
5.3 Bài tập minh họa: Grid Paths (CSES) . . . 9
5.4 Bài tập minh họa: Bài toán cái túi (Knapsack Problem) . . . 11

6 Chia đôi tập (Meet in the Middle) **13**
6.1 Định nghĩa . . . 13
6.2 Bài tập minh họa: Meet in the Middle (CSES) . . . 13

7 Bài tập ứng dụng **14**
7.1 Bài tập: Hoán vị lệch . . . 14
7.2 Bài tập: Dãy con lớn nhất . . . 15
7.3 Gợi ý bài tập . . . 17

1 Đệ quy

1.1 Định nghĩa

Đệ quy là một kỹ thuật trong lập trình, trong đó một hàm tự gọi lại chính nó để giải quyết một bài toán. Kỹ thuật này thường được sử dụng khi bài toán có thể được chia nhỏ thành các bài toán con giống hệt bài toán ban đầu.

Ví dụ đơn giản về đệ quy là tính giai thừa của một số, trong đó giai thừa của $n!$ được tính qua công thức:

$$n! = \begin{cases} 1 & n = 0 \\ (n-1)! \cdot n & n > 0 \end{cases}$$

1.2 Cài đặt minh họa (tính giai thừa)

```
int factorial(int x)
{
    if (x == 1) // base case
        return 1;
    return x * factorial(x - 1);
}
```

2 Đệ quy quay lui (Backtracking)

2.1 Định nghĩa

Đệ quy quay lui là một phương pháp giải quyết bài toán bằng cách thử tất cả các trường hợp của một vấn đề, sau đó quay lại (backtrack) khi gặp phải một giải pháp không hợp lệ hoặc không hiệu quả. Đệ quy quay lui thường được sử dụng cho các bài toán tìm kiếm và tối ưu, như bài toán sắp xếp, bài toán **N-Queen**, hay bài toán tìm đường trong mê cung.

2.2 Cách thức hoạt động

1. **Khởi tạo**: Chọn lựa một bước hoặc một phần tử để bắt đầu.
2. **Đệ quy**: Tiến hành thử một bước hoặc trạng thái tiếp theo, nếu nó hợp lý và không dẫn đến mâu thuẫn, tiếp tục thử với các bước tiếp theo.
3. **Kiểm tra điều kiện dừng**: Nếu bài toán đã hoàn thành (hoặc không thể hoàn thành), ta trả về kết quả.
4. **Quay lui**: Nếu một bước dẫn đến không hợp lệ (hoặc không mang lại kết quả khả thi), ta quay lại bước trước đó và thử một lựa chọn khác.

2.3 Bài tập minh họa: Bài toán N-Queen

Bài toán **N-Queen** là một ví dụ điển hình của Đệ quy quay lui. Bài toán yêu cầu đếm số cách đặt n (với $1 \le n \le 9$) quân hậu trên bàn cờ $n \times n$ sao cho không quân hậu nào có thể tấn công quân hậu khác.

Ta có thể sử dụng kỹ thuật quay lui để thử từng vị trí đặt quân hậu và quay lại nếu phát hiện một vị trí không hợp lệ.

Cài đặt

```cpp
bool usedCol[20], usedDiag1[20], usedDiag2[20];
int n, ans;

void backtrack (int row) {
    if (row == n + 1) return ans++, void();

    // Thử các lựa chọn trên dòng `row`
    for (int col = 1; col <= n; col++) {
        if (usedCol[col] || usedDiag1[row + col] || usedDiag2[n - col + row]) continue;
        usedCol[col] = usedDiag1[row + col] = usedDiag2[n - col + row] = true;
        backtrack(row + 1);
        usedCol[col] = usedDiag1[row + col] = usedDiag2[n - col + row] = false;
    }
}
```

Giải thích

Do ta đang backtracking theo dòng nên chắc chắn các quân hậu sẽ khác dòng. Vậy ta chỉ cần xét các cột và 2 đường chéo:

- **Cột**: Dễ dàng lưu bằng một mảng biến.
- **Đường chéo**: Hai điểm nằm trên cùng một đường chéo có:
  - `row` + `col` giống nhau (cho các đường chéo hướng từ trái dưới sang phải trên), hoặc
  - n − `col` + `row` giống nhau (cho các đường chéo hướng từ trái trên sang phải dưới).

Từ đó, ta sử dụng các mảng đánh dấu tương ứng.

Column

1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4

Diagonal 1

2	3	4	5
3	4	5	6
4	5	6	7
5	6	7	8

Diagonal 2

4	3	2	1
5	4	3	2
6	5	4	3
7	6	5	4

Hình 2.1. Đánh số cho cột và đường chéo

Dễ thấy, độ phức tạp thời gian cho thuật toán sẽ là $\mathcal{O}(n! \cdot n)$ do ta có $n - i + 1$ cách chọn cột ở dòng thứ i và ta cần thử hết n phương án chọn cột ở mỗi lượt gọi hàm `backtrack`.

3 Sinh hoán vị (Permutations)

3.1 Bài toán

Cho dãy A gồm các số nguyên từ 1 đến n ($1 \leq n \leq 10$). Hãy liệt kê các hoán vị của n theo thứ tự bất kì.

Ví dụ, với $n = 4$, ta có các hoán vị:

1, 2, 3, 4	1, 2, 4, 3	1, 3, 2, 4	1, 3, 4, 2
1, 4, 2, 3	1, 4, 3, 2	2, 1, 3, 4	2, 1, 4, 3
2, 3, 1, 4	2, 3, 4, 1	2, 4, 1, 3	2, 4, 3, 1
3, 1, 2, 4	3, 1, 4, 2	3, 2, 1, 4	3, 2, 4, 1
3, 4, 1, 2	3, 4, 2, 1	4, 1, 2, 3	4, 1, 3, 2
4, 2, 1, 3	4, 2, 3, 1	4, 3, 1, 2	4, 3, 2, 1

3.2 Đệ quy quay lui

Ý tưởng

Chọn ra một giá trị i (từ 1 đến n) chưa được sử dụng và lưu vào một mảng, cho đến khi nào mảng đó được lấp đầy thì in xuất ra toàn bộ giá trị trong mảng đó (tức là 1 hoán vị). Sau khi xuất xong thì quay lại bước trước đó và đánh dấu giá trị là chưa sử dụng. Dùng mảng đánh dấu để không dùng lại phần tử một cách trùng lặp.

Cài đặt

```cpp
int n;
vector<bool> vis;
vector<int> a;

void xuat() {
    for(int i = 1; i <= n; i++)
        cout << a[i] << " ";
    cout << endl;
}

void hoanvi (int x) {
    for(int i = 1; i <= n; i++) {
        if (!vis[i]) {
            vis[i] = true;
            a[x] = i;

            if(x == n) xuat();
            else hoanvi(x+1);

            vis[i] = false;

        }
    }
}
```

Giải thích

- Mảng a[i] dùng để lưu một giá trị của hoán vị nằm ở vị trí i.
- Mảng vis[i] dùng để kiểm tra giá trị i đã được sử dụng hay chưa, để tránh việc phần tử được dùng lại một cách trùng lặp.

3.3 Mở rộng (Sinh hoán vị bằng hàm `next_permutation`)

Định nghĩa

Định nghĩa 3.1 Thứ tự từ điển

Để so sánh hai dãy số, hai xâu ký tự,... khác nhau, chúng ta thường dùng thứ tự từ điển. Đúng như cái tên của nó, cách so sánh này được dùng để quy định các từ ngữ xuất hiện trong từ điển, giúp người dùng có thể dễ dàng tra cứu từ hơn.

Nói một cách toán học, dãy $a_0, a_1, a_2, \ldots$ có thứ tự từ điển nhỏ hơn dãy $b_0, b_1, b_2, \ldots$ nếu tồn tại một số nguyên k sao cho a và b có **tiền tố chung** độ dài k và phần tử thứ k của a **bé hơn** phần tử tương ứng của b, hay:

$$\begin{cases} a_0 = b_0 \\ a_1 = b_1 \\ \vdots \\ a_{k-1} = b_{k-1} \\ a_k < b_k \end{cases}$$

Trong trường hợp a, b là hai dãy không có cùng độ dài, người ta thường thêm các "ký tự rỗng" và cuối dãy ngắn hơn và quy ước ký tự rỗng này có thứ tự bé nhất.

Hàm `next_permutation` là hàm dùng để sắp xếp lại các giá trị của hoán vị hiện tại để tạo ra một hoán vị mới tiếp theo theo thứ tự từ điển. Cho dãy $A = [1, 2, 3]$ là một hoán vị của các số từ 1 đến 3. Khi áp dụng `next_permutation`, nó sẽ thay đổi A thành $[1, 3, 2]$.

Lưu ý
Khi sinh hoán vị bằng `next_permutation`, các hoán vị sẽ được sinh theo thứ tự từ điển.

Cách hoạt động

Với một mảng A là hoán vị của các số tự nhiên từ 1 đến n (với n là kích thước mảng) không phải mảng giảm dần, hàm `next_permutation` cho biết mảng có thứ tự từ điển nhỏ nhất trong số các mảng có **thứ tự từ điển lớn hơn** A.

Thuật toán 3.1 Hàm `next_permutation`

Bước 1: Tìm vị trí đầu tiên khi duyệt từ phải sang trái sao cho giá trị của nó bé hơn giá trị phần tử bên phải. Nói cách khác ta tìm i lớn nhất thỏa điều kiện $A_i < A_{i+1}$. Với trường hợp mảng giảm dần (không tồn tại bất kì i thỏa $A_i < A_{i+1}$), vì đây đã là hoán vị có thứ tự từ điển lớn nhất nên ta cũng không cần thực hiện `next_permutation`.

Bước 2: Tìm $p \in [i+1; n]$ sao cho $A_p > A_i$ và p là nhỏ nhất. Hoán đổi hai phần tử A_i và A_p.

Bước 3: Sắp xếp lại các phần tử từ vị trí $i+1$ đến n theo thứ tự tăng dần. Tuy nhiên, nhận xét rằng lúc này, các phần tử có vị trí trong khoảng $[i+1; n]$ tạo thành một dãy giảm dần nên ta chỉ việc **đảo ngược thứ tự** của chúng.

Ví dụ: Với dãy $A = [1, 2, 5, 4, 3]$, ta có $i = 2$ và $p = 5$. Sau bước 2, mảng A trở thành $[1, 3, 5, 4, 2]$ và sau bước 3 thì mảng trở thành $[1, 3, 2, 4, 5]$.

Cài đặt

Đối với ngôn ngữ lập trình `C++`, ta có thể sử dụng hàm cài đặt sẵn `next_permutation` có độ phức tạp tuyến tính. Ta có thể sử dụng hàm này để sinh hoán vị như sau:

```cpp
vector<int> a = {1, 2, 3, 4, 5, 6};
do {
    // xử lý hoán vị hiện tại
} while (next_permutation(a.begin(), a.end()));
```

4 Sinh tập con (Subsets)

Bài toán

Cho số nguyên n $(1 \le n \le 20)$. In ra tất cả các dãy con khác rỗng của dãy $0, 1, 2, \cdots, n-1$.

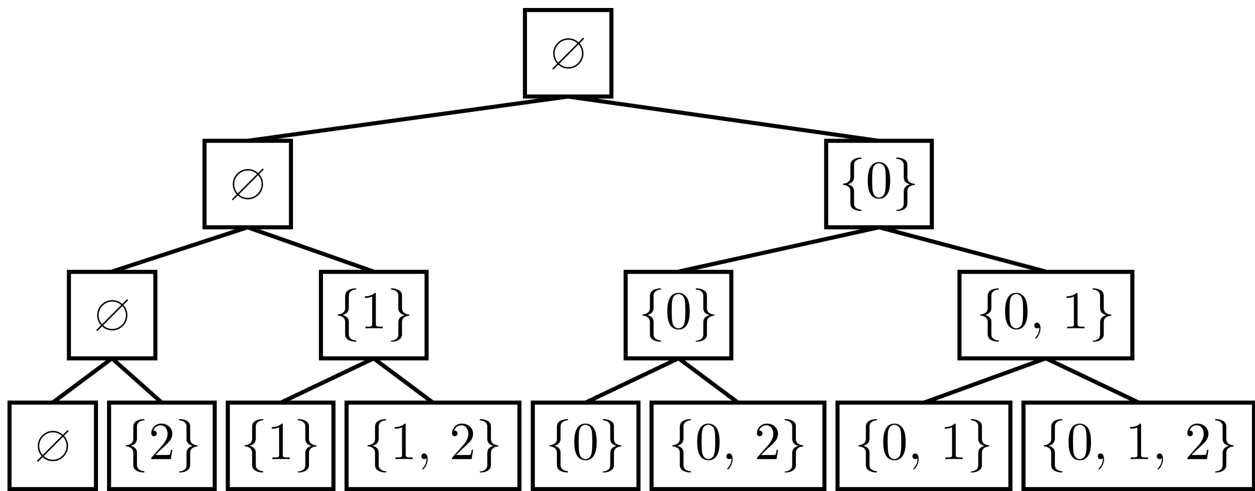
4.1 Đệ quy quay lui

Ý tưởng

Trước tiên, ta cần mô hình hóa lại bài toán như sau: Mỗi phần tử của dãy có hai trạng thái là *chọn* hoặc *không chọn*. Hãy liệt kê tất cả các tổ hợp *chọn/không chọn* của dãy. Dễ thấy, ta có 2^n tổ hợp như vậy.

Áp dụng phương pháp Đệ quy quay lui, ta chọn trạng thái cho các phần tử từ trái qua phải và duy trì những lựa chọn của các phần tử đã được chọn trạng thái, gọi là một *cấu hình*. Với một cấu hình đã chọn cho i phần tử đầu tiên (với $i < n$), ta chọn tiếp trạng thái cho phần tử thứ $i+1$ rồi gọi `backtrack` hai lần ứng với mỗi lựa chọn.

Để làm điều đó, ta dùng một vector gọi là `subset` để lưu các phần tử ở cấu hình hiện tại. Với hàm `backtrack`(x) (với x vị trí của phần tử hiện tại của hàm), đẩy giá trị của x vào `subset` và gọi tiếp hàm `backtrack`$(x+1)$, xóa nó ra khỏi `subset` và gọi tiếp hàm `backtrack`$(x+1)$. Thao tác này như là việc bỏ qua giá trị x hiện tại và xét tiếp đến các cấu hình sau.



Hình 4.1. Cây mô tả quá trình Đệ quy quay lui

Ở hình minh họa trên, tại tầng thứ x của cây, ta sẽ chọn trạng thái cho phần tử x trong tập hợp với cây con trái tương ứng việc không chọn x và cây con phải tương ứng việc chọn x.

- **Độ phức tạp**: Một tập hợp gồm n phần tử có 2^n tập con. Ta duyệt qua 2^n tập con và in ra các phần tử trong tập con đó. Vậy độ phức tạp của cách trên là $\mathcal{O}(n \times 2^n)$.

Cài đặt

```
const int N = 20;
vector<int> subset;
int a[N+5], n;

void backtrack(int x) {
    if(x == n) {
        for(int v : subset)
            cout << v << " ";
        cout << "\n";
    }
    else {
        subset.push_back(x);
        backtrack(x+1);
        subset.pop_back();
        backtrack(x+1);
    }
}
```

Để bắt đầu sinh tập con, ta gọi `backtrack(0)`.

4.2 Mở rộng (Sinh tập con bằng bitmask)

Ý tưởng

Sinh tập con bằng bitmask nghĩa là biểu diễn tập hợp dưới dạng một xâu nhị phân và xâu này được gọi là bitmask. Giả sử ta có một tập hợp $A = \{a_1, a_2, a_3, \cdots, a_n\}$, ta chọn ra một tập con $B = \{a_{x_1}, a_{x_2}, a_{x_3}, ...a_{x_k}\}$. Khi đó ta có thể biểu diễn các phần tử được chọn trong tập hợp B bằng một xâu nhị phân có độ dài là n, bit thứ i bật nếu a_i thuộc tập hợp B được chọn.

Ví dụ 4.1

Cho tập hợp $A = \{1, 2, 3, 4\}$, tập con $B = \{1, 3, 4\}$ (B được chọn từ các vị trí $0, 2, 3$ của A).

Vậy bitmask biểu diễn tập con B được chọn từ A là `1101`. Lưu ý bit thứ i được đếm từ phải sang và được đánh số từ 0.

Vậy để sinh tất cả các tập con n phần tử, ta duyệt qua mọi bitmask từ $0 \to 2^n$ và ở mỗi `bitmask` ta truy cập đến bit thứ i của `bitmask` đó và xem liệu nó có được chọn hay không. Ta có thể kiểm tra bit thứ i của một `bitmask` bất kì có được bật hay không bằng đoạn code:

```
if (mask & (1 << i)) {
    // Xử lý ...
}
```

Bạn đọc có thể tự tìm hiểu thêm về bitmask và các phép biến đổi trên hệ cơ số 2 để hiểu rõ hơn về phương pháp này.

- **Độ phức tạp:** Tương tự như Đệ quy quay lui, ta duyệt qua 2^n tập con, in ra tất cả phần tử trong mỗi tập con. Vậy độ phức tạp của cách trên là $\mathcal{O}(2^n \cdot n)$.

Cài đặt

```
for (int mask = 1; mask < (1 << n); mask++) {// mask bắt đầu từ 0 do không xét dãy rỗng 0 = 000...0
    for (int i = 0; i < n; i++) {
        if (mask & (1 << i))
            cout << i << " ";
    }
    cout << endl;
}
```

5 Cải tiến Đệ quy quay lui

Như đã phân tích, bản chất thuật toán Đệ quy quay lui là ta xét hết mọi trường hợp có thể xảy ra, nên thường sẽ có độ phức tạp khá lớn. Tuy những kĩ thuật cải tiến thuật toán này không làm giảm độ phức tạp về mặt lý thuyết, nhưng chúng sẽ làm giảm thời gian chạy một cách đáng kể.

5.1 Phương pháp Nhánh cận (Branch and Bound)

Đối với các dạng bài toán tối ưu (tìm phương án cực đại/cực tiểu hóa một giá trị), đôi khi ta có thể tính trước xem cấu hình mà ta đang xây dựng có khả năng tạo ra phương án tối ưu hơn hay không. Khi đã biết chắc cấu hình này không còn khả năng tạo ra phương án tối ưu hơn, ta có thể dừng vòng đệ quy hiện tại rồi thực hiện quay luôn mà không cần đợi đến điều kiện dừng.

Đối với bài toán cực tiểu hóa, giả sử đáp án hiện tại ta đang có sau những phương án đã được duyệt là $ans_{\texttt{current}}$ (đã biết) và đáp án cuối cùng của bài toán là ans (chưa biết). Do ans phải là giá trị nhỏ nhất trong các phương án, ta biết:

$$ans_{\texttt{current}} \geq ans$$

Dễ thấy, ta sẽ cố gắng tìm một phương án cho ra $ans_{\texttt{new}} < ans_{\texttt{current}}$.

Giả sử ta đã xây dựng được cấu hình $(x_1, x_2, \cdots, x_i)$ và cần xây dựng tiếp cho các vị trí từ $i+1$ đến n để có được một phương án hợp lệ. Tuy nhiên, ta biết rằng mọi trạng thái X xây dựng từ cấu hình $(x_1, x_2, \cdots, x_i)$ đều cho ra giá trị lớn hơn hoặc bằng $ans_{\texttt{current}}$, khi đó, ta có thể dừng lại và quay lui về cấu hình $(x_1, x_2, \cdots, x_{i-1})$ chứ không xây dựng tiếp trên cấu hình hiện tại một cách phung phí.

Định nghĩa 5.1 Phương pháp Nhánh cận

Như vậy, **phương pháp Nhánh cận** là phương pháp tối ưu Đệ quy quay lui cho các bài toán tối ưu. Ý tưởng của phương pháp này là ta sẽ dừng vòng đệ quy khi biết chắc chắn rằng từ cấu hình hiện tại, ta không thể đưa ra một phương án nào đủ tốt bằng phương án tối ưu nhất cho đến thời điểm hiện tại.

5.2 Phương pháp Cắt tỉa (Pruning)

Định nghĩa 5.2 Phương pháp Cắt tỉa

Phương pháp Cắt tỉa có thể được sử dụng cho cả các bài toán đếm chứ không chỉ là bài toán tối ưu. Ý tưởng của phương pháp này là ta sẽ dừng vòng đệ quy khi biết chắc chắn rằng từ cấu hình hiện tại, không còn cách nào để tạo ra một phương án hợp lệ thỏa đề bài

Tuy không giảm độ phức tạp thời gian về mặt lý thuyết, nhưng đối với một số bài toán, việc áp dụng hai phương pháp nêu trên có thể tạo ra sự khác biệt vô cùng lớn.

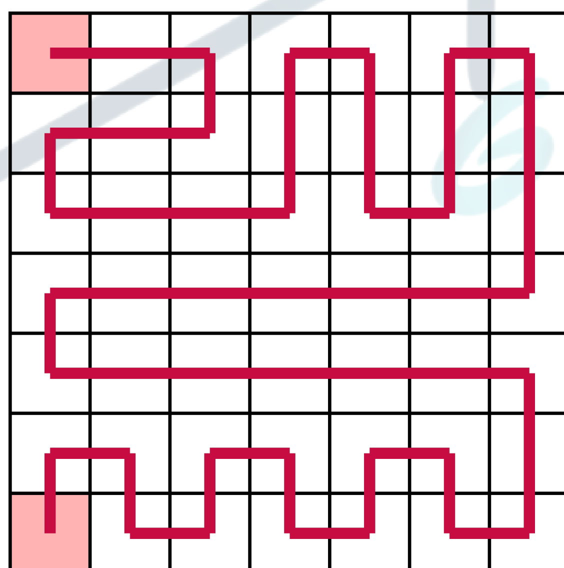
5.3 Bài tập minh họa: Grid Paths (CSES)

Đề bài

- Link đề gốc: `https://cses.fi/problemset/task/1625`

Chúng ta sẽ nghiên cứu một phiên bản đơn giản hơn của bài toán Grid Paths (CSES) như sau: Cho một bảng vuông có kích thước 7×7. Đếm số cách di chuyển từ **góc trên trái** đến **góc dưới trái** sao cho mỗi ô được thăm đúng 1 lần. Ta có thể di chuyển theo 1 trong 4 hướng: trên, dưới, trái, phải.

Ví dụ, hình bên dưới mô tả một phương án di chuyển hợp lệ:



Hình 5.1. Một phương án di chuyển hợp lệ

Thuật toán cơ bản

Sử dụng thuật toán Đệ quy quay lui để sinh tất cả các đường đi có thể có. Để đảm bảo điều kiện mỗi ô được thăm đúng 1 lần, ta dùng một mảng đánh dấu 2 chiều. Ta tìm được phương án hợp lệ khi thăm được góc dưới trái của bảng sau khi đã thăm đủ 49 ô.

Thuật toán trên chạy trong khoảng 360 giây. Đây rõ ràng là một thuật toán quá chậm. Để tăng tốc cho chương trình, ta cần áp dụng 3 cải tiến theo phương pháp Cắt tỉa được trình bày sau đây.

Cải tiến 1

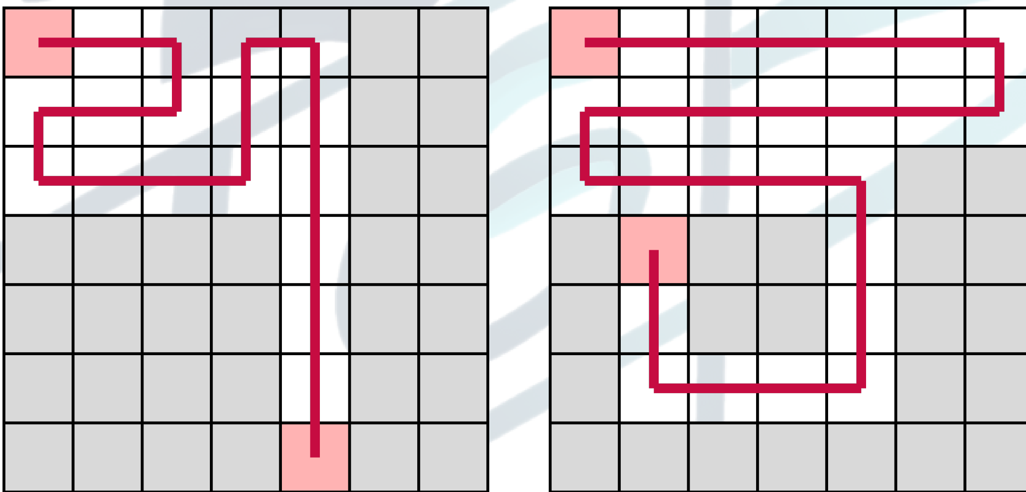
Rõ ràng, khi đã thăm góc dưới trái mà chưa thăm đủ 49 ô, thì ở phần còn lại của đường đi dù có thế nào thì ta cũng không thể tạo ra một phương án hợp lệ nữa. Do đó, ta dừng đệ quy.

Sau cải tiến 1, thuật toán chạy trong khoảng 87 giây.

Cải tiến 2

Ở những chỗ bắt buộc phải rẽ trái/phải (chạm tường, chạm đường đi cũ) mà cả hai phía trái phải đều có ô chưa thăm, tức là ta đã tách bảng ra thành 2 thành phần riêng biệt chưa được thăm. Trong trường hợp này, ta cũng không thể tạo ra phương án hợp lệ bất kể phần còn lại có di chuyển như thế nào. Một lần nữa, ta dừng đệ quy.

Ví dụ, ở hai trường hợp sau, ta chắc chắn không thể xây dựng tiếp phương án di chuyển hợp lệ:



Hình 5.2. Hai trường hợp không thể xây tiếp phương án hợp lệ

Cụ thể hơn, điều kiện để dừng đệ quy khi này sẽ là:

- Khi cả hai ô ở trên/dưới đã được thăm (hoặc nằm ngoài bảng) nhưng cả hai ô ở trái/phải chưa được thăm, hoặc
- Khi cả hai ô ở trái/phải đã được thăm (hoặc nằm ngoài bảng) nhưng cả hai ô ở trên/dưới chưa được thăm.

Sau cải tiến 2, thuật toán chạy trong khoảng 0.4 giây.

Cải tiến 3

Ngoài ra, việc di chuyển ở quá xa ô $(7, 1)$ (góc dưới trái) khi còn lại quá ít bước cũng là điều không hợp lý. Do đó, ta có thể cắt bỏ các trường hợp mà khoảng cách Manhattan giữa ô hiện tại và ô $(7, 1)$ lớn hơn hẳn số bước đi còn lại.

Sau cải tiến 3, thuật toán chạy trong 0.3 giây.

Có thể thấy, mặc dù có độ phức tạp $\mathcal{O}(4^{7\cdot 7})$ về mặt lý thuyết. Nhưng sau 3 cải tiến, thuật toán của chúng ta đã chạy nhanh hơn gấp 1000 lần và đã có thể chạy kịp thời gian đối với một bài lập trình thi đấu.

Cài đặt

Để thuận tiện hơn cho phần cài đặt, ta có thể đánh dấu phần viền của bảng là các ô đã thăm. Như vậy, trong lúc quay lui, ta sẽ không cần xét riêng trường hợp di chuyển ra khỏi bảng. Ngoài ra, ta đánh dấu đã thăm ô $(1, 1)$ để hàm quay lui không chọn lại ô bắt đầu để di chuyển.

```
for (int i = 0; i <= 8; i++) vis[i][0] = vis[i][8] = 1;
for (int j = 0; j <= 8; j++) vis[0][j] = vis[8][j] = 1;
vis[1][1] = 1;
```

Sau đó, ta cài đặt hàm Đệ quy quay lui kết hợp với các tối ưu như mô tả ở trên:

```
const int boardSize = 7 * 7;
int direct[50], ans;
bool vis[9][9];

const int dr[4] = {-1, 0, 1, 0};
const int dc[4] = {0, -1, 0, 1};

void backtrack (int step, int x, int y) {
    if (step == boardSize)
        return ans += (x == 7 && y == 1), void();
    // Cải tiến 1
    if (x == 7 && y == 1) return;
    // Cải tiến 2
    if (vis[x - 1][y] && vis[x + 1][y] && !vis[x][y - 1] && !vis[x][y + 1]) return;
    if (!vis[x - 1][y] && !vis[x + 1][y] && vis[x][y - 1] && vis[x][y + 1]) return;
    // Cải tiến 3
    if (abs(x - 7) + abs(y - 1) > boardSize - step) return;

    for (int k = 0; k < 4; k++) {
        int x2 = x + dr[k], y2 = y + dc[k];
        if (!vis[x2][y2]) {
            vis[x2][y2] = 1;
            backtrack(step + 1, x2, y2);
            vis[x2][y2] = 0;
        }
    }
}
```

Để giải quyết bài Grid Paths (CSES) ta xử lý xâu và thực hiện di chuyển đúng hướng yêu cầu ở các chỗ có ký tự khác ?. Ý tưởng chính của thuật toán cũng không khác mô tả ở trên nhiều.

5.4 Bài tập minh họa: Bài toán cái túi (Knapsack Problem)

Đề bài

Cho n món đồ có giá trị lần lượt là $v_1, v_2, v_3, \cdots, v_n$ và trọng lượng $w_1, w_2, w_3, \cdots, w_n$. Do chưa biết nên chọn những món đồ nào và cái túi có thể chịu được trọng lượng tối đa là bao nhiêu, hãy tính tổng lượng nhỏ nhất nếu chọn các món đồ có tổng giá trị tối thiểu là V.

- $1 \leq n \leq 30$.
- $1 \leq v_i, w_i \leq 10^{16}$.
- $1 \leq V \leq \sum v_i$.

Đối với bài toán này, một phương án chọn sẽ được biểu diễn bởi dãy $(x_1, x_2, \cdots, x_n)$ với $x_i \in \{0; 1\}$ tương ứng việc chọn hay không chọn món đồ thứ i, tương tự thuật toán sinh tập con đã được trình bày ở phần trước. Với thuật toán Đệ quy quay lui thông thường, rõ ràng độ phức tạp là $\mathcal{O}(2^n)$ do ta phải duyệt hết 2^n phương án.

Cải tiến 1

Áp dụng phương pháp nhánh cận, nếu đang xây dựng cấu hình $(x_1, x_2, \cdots, x_i)$ có tổng trọng lượng là $w_{\texttt{current}}$, đáp án hiện tại là $ans_{\texttt{current}}$ và $w_{\texttt{current}} \geq ans_{\texttt{current}}$. Ta thấy rằng, cho dù có chọn các món đồ còn lại như thế nào thì $w_{\texttt{current}}$ cũng không thể nhỏ hơn $ans_{\texttt{current}}$ (hiển nhiên vì w_i dương).

Do đó, ta có thể dừng vòng đệ quy hiện tại nếu $w_{\texttt{current}} \geq ans_{\texttt{current}}$.

Cải tiến 2

Áp dụng phương pháp cắt tỉa, nếu đang xây dựng cấu hình $(x_1, x_2, \cdots, x_i)$ mà số lượng món đồ đã được chọn là quá ít, đến mức cho dù có lấy hết $n - i$ món đồ còn lại, ta vẫn không thể tạo ra tổng giá trị $\geq V$, ta có thể dừng vòng đệ quy hiện tại vì cấu hình này không cho ra bất kì một phương án hợp lệ nào.

Thuật toán

Như vậy, ta sẽ áp dụng thuật toán Đệ quy quay lui kết hợp thêm hai điều kiện dừng đệ quy như sau:

- Dừng đệ quy khi $\sum_{1 \leq j \leq i} w_j \cdot x_j \geq ans_{\texttt{current}}$, tức là ta không thể tìm ra phương án tốt hơn với cấu hình $(x_1, x_2, \cdots, x_i)$ hiện tại.
- Dừng đệ quy khi $\sum_{1 \leq j \leq i} v_j \cdot x_j + \sum_{i < j \leq n} v_j < V$, tức là không thể tìm ra phương án hợp lệ với cấu hình $(x_1, x_2, \cdots, x_i)$ hiện tại.

Bên cạnh đó, ta có thể trộn thứ tự của các món đồ trước khi thực hiện đệ quy để giúp tốc độ chạy của hàm được ổn định hơn.

Như đã nói, mặc dù không thay đổi độ phức tạp nhưng hai phương pháp Nhánh cận và Cắt tỉa đã giảm số lần gọi đệ quy một cách đáng kể, và tối ưu một thuật toán vốn phải mất từ 8 đến 10 giây để thực hiện xuống còn dưới 1 giây.

Cài đặt

```
long long v[N], w[N], suffixSum[N];
long long currentWeight, currentValue, n, V, ans = LLONG_MAX;

void backtrack (int i) {
    if (currentWeight >= ans || currentValue + suffixSum[i] < V) return;
    if (i == n) return ans = currentWeight, void();

    // chọn món đồ thứ i
    currentWeight += w[i], currentValue += v[i];
    backtrack(i + 1);
```

```
11
12     // bỏ chọn món đồ thứ i
13     currentWeight -= w[i], currentValue -= v[i];
14     backtrack(i + 1);
15 }
```

6 Chia đôi tập (Meet in the Middle)

6.1 Định nghĩa

Meet in the Middle là một kĩ thuật còn được gọi là chia đôi tập. Ý tưởng của kĩ thuật này là chia đôi bài toán thành hai phần rồi giải riêng từng bài toán, xong rồi kết hợp kết quả bài toán của hai tập hợp một cách hiệu quả. Thông thường, kỹ thuật này được áp dụng trong các bài toán Đệ quy quay lui, nhưng dữ liệu lại quá lớn để có thể xử lý trực tiếp.

6.2 Bài tập minh họa: Meet in the Middle (CSES)

Đề bài

- Link đề gốc: `https://cses.fi/problemset/task/1628`

Cho một dãy a gồm n phần tử, đếm số lượng tập con có tổng là x.

Giới hạn:

- $1 \le n \le 40$
- $1 \le a_i \le 10^9$
- $1 \le x \le 10^9$

Ý tưởng

Có thể thấy với $n = 40$, việc sinh tất cả các tập con sẽ bị quá thời gian. Tuy nhiên, nếu ta sửa lại giới hạn thành $n \le 20$ thì ta có thể sử dụng cách sinh các tập con để giải bài toán được ngay.

Ta sẽ cha đôi mảng thành 2 phần:

- Phần 1: Bao gồm $\lfloor \frac{n}{2} \rfloor$ đầu tiên.
- Phần 2: Bao gồm các phần tử còn lại.

Với mỗi phần, ta sẽ sinh một mảng bao gồm tổng có thể có thể tạo được từ các tập con trong mỗi phần riêng biệt. Độ phức tạp của phần này là $\mathcal{O}(2^{n/2+1})$.

> **Lưu ý**
> Giữa $2^{n/2}$ và 2^n có khoảng cách lớn, ta có thể dễ thấy điều này hơn khi viết lại $2^{n/2}$ thành $\sqrt{2^n}$

Ta có duyệt qua mỗi tổng của tập con ở phần đầu, rồi với từng tổng này ta duyệt mỗi tổng của các tập con ở phần sau, nếu như vậy thì độ phức tạp sẽ quay trở lại thành $\mathcal{O}(2^{n/2} \cdot 2^{n/2}) = \mathcal{O}(2^n)$, đây là cách không tối ưu.

Thay vì đó, ta có thể sort lại một trong hai mảng chứa các tổng của tập con, xong rồi có thể sử dụng binary search để tìm được kết quả bài toán độ phức tạp cuối cùng của bài toán là $\mathcal{O}(2^{n/2} \cdot \log 2^{n/2})$, hoặc có thể kết hợp bằng cách sort hai mảng và dùng hai con trỏ trong $\mathcal{O}(2^{n/2})$.

■ Cài đặt

```cpp
#include <bits/stdc++.h>
using namespace std;

#define int long long

vector<long long> gen(vector<int> &v) { // Sinh tất các tổng của các tập con của một phần
    int n = (int)v.size();
    vector<int> res;
    for(int b = 0; b < (1 << n); b++) {
        int cur = 0;
        for(int i = 0; i < n; i++) if(b & (1 << i)) cur += v[i];
        res.push_back(cur);
    }
    return res;
}

signed main() {
    int n; cin >> n;
    int x; cin >> x;
    vector<int> a(n); for(int &i : a) cin >> i;

    // Tách thành hai phần
    vector<int> v1; for(int i = 0; i < n / 2; i++) v1.push_back(a[i]);
    vector<int> v2; for(int i = n / 2; i < n; i++) v2.push_back(a[i]);

    vector<int> f1 = gen(v1), f2 = gen(v2);
    sort(f2.begin(), f2.end());

    int res = 0;

    for(int num : f1) {
        res += upper_bound(f2.begin(), f2.end(), x - num)
             - lower_bound(f2.begin(), f2.end(), x - num);
    }

    cout << res << '\n';
}
```

7 Bài tập ứng dụng

7.1 Bài tập: Hoán vị lệch

Cho số nguyên dương N ($N \leq 10$), hãy in ra tất cả các hoán vị p có độ dài N sao cho $p_i \neq i$, $\forall i \in [1; n]$, theo thứ tự từ điển. Ví dụ, với $N = 3$, ta có hai hoán vị là $2, 3, 1$ và $3, 1, 2$.

■ Lời giải

Thực hiện Đệ quy quay lui tương tự với cách sinh hoán vị. Tuy nhiên, khi chọn giá trị phần tử ở vị trí i, ta không được chọn i làm giá trị của phần tử này.

Cài đặt

Ta cài đặt hàm `backtrack` như sau:

```
int permutation[15], n;
bool used[15];

void backtrack (int i) {
    if (i == n + 1) {
        for (int j = 1; j <= n; j++) cout << permutation[j] << " ";
        cout << "\n";
        return;
    }
    for (int cur = 1; cur <= n; cur++) {
        if (used[cur] || cur == i) continue;
        permutation[i] = cur, used[cur] = 1;
        backtrack(i + 1);
        used[cur] = 0;
    }
}
```

Sau đó, ta bắt đầu Đệ quy quay lui bằng cách gọi `backtrack(1)`.

7.2 Bài tập: Dãy con lớn nhất

Đề bài

- Link đề gốc: `https://codeforces.com/problemset/problem/888/E`

Cho dãy a gồm n số nguyên và một số m. Chọn ra một dãy con (có thể rỗng, không nhất thiết liên tiếp) của a sao cho tổng các phần tử được chọn modulo m đạt giá trị lớn nhất. Nói cách khác, chọn ra một dãy b $(1 < b_1 < b_2 < ... < b_k \le n, 1 \le k \le n)$ để cực đại hóa giá trị:

$$\left(\sum_{i=1}^{k} a_{b_i}\right) \bmod m$$

Yêu cầu: In ra giá trị $(\sum_{i=1}^{k} a_{b_i}) \bmod m$ lớn nhất.

Giới hạn

- Subtask 1: $1 \le n \le 20$; $1 \le m \le 10^9$; $1 \le a_i \le 10^9$.
- Subtask 2: $1 \le n \le 35$; $1 \le m \le 10^9$; $1 \le a_i \le 10^9$.

Lời giải subtask 1

Ta có thể đơn thuần sinh hết tất cả tập con và xét từng tập con rồi tìm đáp án.

Cài đặt subtask 1

```
#include<bits/stdc++.h>
using namespace std;

```

```
int main() {
    int n, m; cin >> n >> m;
    vector<int> a(n); for(int &i : a) cin >> i;
    int res = 0;
    for(int b = 1; b < (1 << n); b++) {
        int cur = 0;
        for(int i = 0; i < n; i++) {
            if(b & (1 << i)) cur += a[i], cur %= m;
        }
        res = max(res, cur);
    }
    cout << res << '\n';
}
```

Lời giải subtask 2

Sử dụng kĩ thuật Chia đôi tập, ta chia mảng thành 2 phần, mỗi phần sinh tất cả các tổng của tập con. Ta gọi 2 mảng lần lượt chứa 2 tổng sinh được từ hai phần này lần lượt là A và B.

Trước hết, ta sẽ sort mảng B không giảm dần. Với mỗi tổng sinh được trong phần 1, hay là từng phần tử trong mảng A, xét phần tử hiện tại là x:

- Có thể kết hợp tổng này với một tổng lớn nhất có thể sao cho khi cộng với tổng này thì 2 phần cộng lại với nhau chứa vượt qua m, hay nói cách khác ta phải tìm j lớn nhất sao cho $x + B_j < m$, điều này sẽ đảm bảo được tổng sau khi mod vẫn giữ nguyên giá trị cũ.
- Trường hợp khác, ta kết hợp với số lớn nhất ở mảng B.
- Không kết hợp với số khác.

Ngoài ra, ta còn phải xử lý các trường hợp biên như $n = 1$, các trường hợp này sẽ được giải thích và nhắc đến trong code sau.

Cài đặt subtask 2

```
vector<int> gen(vector<int> &a, int m) {
    int n = a.size();
    vector<int> res;
    for(int b = 1; b < (1 << n); b++) {
        int sum = 0;
        for(int i = 0; i < n; i++)
            if(b & (1 << i)) sum += a[i], sum %= m;
        res.push_back(sum);
    }
    return res;
}

void solve()
{
    int n, m; cin >> n >> m;
    vector<int> a(n); for(int &i : a) cin >> i;
    if(n == 1) {
        // Dòng 32 sẽ trả về RTE nếu không xử lý trường hợp này
        cout << a[0] % m << '\n';
```

```cpp
        return;
    }
    vector<int> f1; for(int i = 0; i < n / 2; i++) f1.push_back(a[i]);
    vector<int> f2; for(int i = n / 2; i < n; i++) f2.push_back(a[i]);

    vector<int> r1 = gen(f1, m); // mảng A
    sort(r1.begin(), r1.end());
    vector<int> r2 = gen(f2, m); // mảng B
    sort(r2.begin(), r2.end());
    int ans = (r1[0] + r2[0]) % m;

    for(int x : r2) ans = max(ans, x % m); // Tìm số lớn nhất trong mảng B
    for(int x : r1) {
        ans = max(ans, x % m);
        int l = 0, r = r2.size() - 1;
        int res = -1;
        while(l <= r) {
            int mid = (l + r) / 2;
            if(r2[mid] + x < m) res = mid, l = mid + 1;
            else r = mid - 1;
        }
        if(res != -1) ans = max(ans, r2[res] + x);
        ans = max(ans, (x + r2.back()) % m);
    }
    cout << ans << '\n';
}
```

7.3 Gợi ý bài tập

Ngoài các bài tập được nêu trên, chúng mình khuyến khích bạn đọc luyện tập thêm các dạng bài liên quan để có thể hiểu sâu bản chất của các thuật toán cũng như rèn luyện khả năng vận dụng vào các tình huống khác nhau khi giải bài tập.

- MarisaOJ: Scheduling
- MarisaOJ: Word search
- Free Contest: Shelf 2
- VNOJ: 34 đồng xu
- VNOJ: Tổng vector

Ngoài ra, nếu muốn thử sức với những bài tập nâng cao hơn, bạn đọc cũng có thể tham khảo Educational Backtracking Contest của cộng đồng VNOI.