

Đệ quy và Thuật toán quay lui

Đệ quy và Thuật toán quay lui

Người viết:

- Nguyễn Đức Kiên, Trường Đại học Công nghệ, ĐHQGHN.

Reviewer:

- Nguyễn Minh Nhật, Trường THPT chuyên KHTN - ĐHKHTN - ĐHQGHN.
- Cao Thanh Hậu, Trường Đại học Khoa học Tự nhiên, ĐHQG-HCM.
- Nguyễn Hoàng Vũ, Trường Đại học Công nghệ, ĐHQGHN.

Mở đầu



Búp bê Matryoshka (ảnh trên Google Images)

Trong cuộc sống, chúng ta đôi khi bắt gặp những hình ảnh về một vật mà chứa bên trong nó là một vật khác giống hệt nó, như búp bê Matryoska, cửa sổ OBS khi bạn cố dùng nó để quay màn hình của chính nó, sách giáo khoa Toán lớp 3 cũ, hoặc [link này](#), ... Tương tự như vậy, trong khoa học máy tính và lập trình, chúng ta xây dựng khái niệm về đệ quy.

Đệ quy và giải thuật đệ quy

Khái niệm



Ta gọi một đối tượng là **đệ quy** (recursion) nếu nó được định nghĩa qua chính nó hoặc một đối tượng cùng dạng với chính nó bằng quy nạp.

Ví dụ:

- Với $n!$ thì ta có $n! = (n - 1)! \times n$
- Gọi $\gcd(a, b)$ là ước chung lớn nhất của a và b ($a \geq b$) thì ta có $\gcd(a, b) = \gcd(b, a \bmod b)$ với \bmod là phép lấy phần dư

Nếu một bài toán P có lời giải được thực hiện bằng một bài toán con P' có dạng giống P thì đó là một giải thuật đệ quy. Ở đây, P' cần là một bài toán đơn giản hơn P (có kích cỡ dữ liệu nhỏ hơn, hoặc độ phức tạp nhỏ hơn, ...), và đương nhiên không cần đến P để giải nó.

Về cơ bản thì ta có thể gọi một hàm là đệ quy nếu hàm đó tự gọi chính nó, với các biến đầu vào có thể khác.

Một bài toán đệ quy có lời giải gồm 2 phần:

- **Phần neo/trường hợp cơ sở (anchor/base case):** Đây là phần có thể giải trực tiếp mà không cần phải dựa vào một bài toán con nào, và cũng chính là điểm dừng của lời giải đệ quy. Phần này thường là các trường hợp cụ thể, như $x == 0$, $x == n$, ...
- **Phần đệ quy:** Đây chính là phần mà bạn phải gọi ra bài toán con và giải nó, cũng chính là gọi hàm đệ quy. Phần này sẽ được thực hiện đến khi nào bài toán đưa được về trường hợp cơ sở.

Nếu ta đem so sánh với con Matryoska, thì trường hợp cơ sở là con bé nhất ở trong cùng, còn đệ quy chính là thực hiện việc mở một con to liên tục đến lúc nào không thể mở được nữa.

Lý thuyết suông thì quá khó hiểu, hãy cùng xem một số ví dụ:

Tính giai thừa

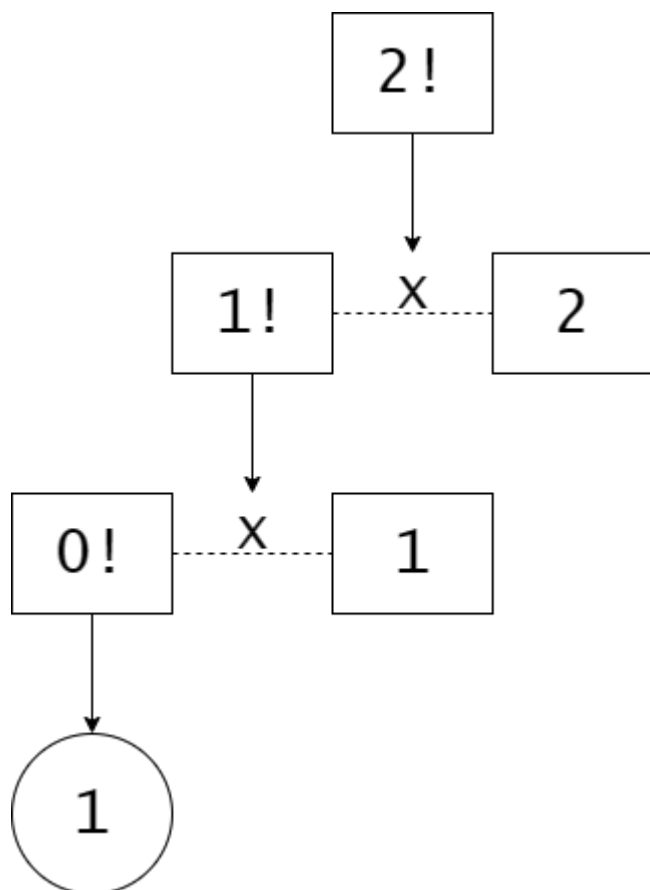
Bài toán: Cho số tự nhiên n ($n \leq 15$). Tính $n!$

Phân tích:

Ở phần trên, chúng ta đã tìm được công thức đệ quy của bài toán này: $n! = (n - 1)! \times n$. Khi lập trình đệ quy, cơ bản thuật toán sẽ hoạt động như sau với $n = 2$: để tính $2!$, chúng ta phải tính nó qua $1!$; để tính $1!$ chúng ta phải tính $0!$. Lúc này, ta có hai lựa chọn:

- Tiếp tục với $0! = (-1)! \times 0$
- Thay $0! = 1$ vào và tính tiếp

Lựa chọn thứ nhất có vẻ không khả thi, phần vì $(-1)!$ không xác định, mặt khác nếu thay tiếp thì chúng ta cũng chẳng biết dừng ở đâu cả. Với $0! = 1$, ta có thể thay vào để tính $1!$ rồi. Có $1!$ ta lại thay nó vào tiếp để tính $2!$, và chúng ta có thứ chúng ta cần.



Phân tích thì dài dòng vậy thôi, còn cài đặt thì rất đơn giản:

```

1 | void factorial(int n)
2 | {
3 |     if (n == 0) return 1;    //trường hợp cơ sở
4 |     return factorial(n - 1) * n;    //phần đệ quy
5 | }
  
```

Nếu bạn chưa quen với cú pháp đệ quy như vậy thì có thể hiểu hàm trên tương đương với hàm `factorial_2()` trong đoạn code sau với $n = 2$:

```

1 | //n = 0
2 | void factorial_0()
3 | {
4 |     return 1;
5 | }
6 |
7 | //n = 1
8 | void factorial_1()
9 | {
10 |     return factorial_0() * 1;
11 | }
12 |
13 | //n = 2
14 | void factorial_2()
  
```

```

15 | {
16 |     return factorial_1() * 2;
17 | }
```

Tính số Fibonacci

Dãy Fibonacci là dãy số được định nghĩa theo công thức truy hồi sau:

$$f_n = \begin{cases} 0 & \text{với } n = 0 \\ 1 & \text{với } n = 1 \\ f_{n-2} + f_{n-1} & \text{với } n > 1 \end{cases}$$

Vấn đề hơn thì trong dãy này, mỗi số hạng bằng tổng của hai số hạng liền trước nó. Các giá trị f_0, f_1 có thể khác một chút tùy tài liệu.

Bài toán: Tìm số Fibonacci thứ n ($n \leq 20$).

Dựa vào công thức truy hồi đã cho và lập luận kiểu "để tính f này thì ta cần có f kia" như trên, chúng ta có thể cài đặt như sau:

```

1 | int fibo(int n)
2 | {
3 |     if (n == 0) return 0;    //trường hợp cơ sở
4 |     if (n == 1) return 1;    //trường hợp cơ sở
5 |     return fibo(n - 2) + fibo(n - 1);    //phần đệ quy
6 | }
```

Cần chú ý rằng ở chương trình này cần có tới 2 trường hợp cơ sở, vì đó cũng là hai trường hợp không thể áp dụng công thức truy hồi.

Mở rộng: Hãy thử sử dụng phương pháp này để cài đặt một chương trình đệ quy tính ƯCLN dựa vào công thức ở ví dụ phía trên.

Đương nhiên, không phải bài toán đệ quy nào cũng để chúng ta nhìn thấy một công thức đệ quy đơn giản như vậy. Thậm chí, đôi khi chúng ta còn chẳng có một công thức cụ thể, mà chỉ đơn thuần là công việc được thực hiện sau đó có điểm tương đồng với phần trước thôi. Lúc này, ta cần giải đáp những câu hỏi:

- Bài toán có thể được giải qua những bài toán con nào tương tự không? Nếu được, đó là gì?
- Tới trạng thái nào, chúng ta sẽ dừng lại?

Thuật toán quay lui

Khái niệm

Thuật toán **quay lui** (backtracking) dùng để giải bài toán liệt kê các cấu hình. Mỗi cấu hình được xây dựng bằng cách xây dựng từng phần tử, mỗi phần tử được chọn bằng cách thử tất cả các

khả năng.

Tóm gọn lại, chúng ta đang xây dựng một danh sách gồm tất cả các tập hợp (hay dãy, ...), mà mỗi phần tử được xét tất cả các trường hợp có thể của nó. Phương pháp này cũng gọi là **duyệt vét cạn**.

Để cho khỏi "lú", trong bài viết này chúng ta thống nhất sẽ dùng cụm từ **danh sách các tập hợp/dãy/xâu**.

Ví dụ: khi tìm danh sách các dãy nhị phân (các dãy gồm toàn các ký tự 0, 1 như dãy 0001011) độ dài 3, ta sẽ:

- ▶ Xét mọi trường hợp ký tự thứ nhất. Ta được 0 hoặc 1.
- ▶ Với mỗi trường hợp của ký tự thứ nhất, xét tiếp mọi trường hợp ký tự thứ hai. Ta được 00, 01 từ 0 ở bước trước và 10, 11 từ 1 ở bước trước.
- ▶ Tương tự, với mỗi trường hợp của ký tự thứ hai, ta xét nốt mọi trường hợp ở ký tự thứ ba. Các dãy nhận được là 000, 001, 010, 011, 100, 101, 110, 111

Nếu bạn vẫn chưa hiểu các dãy này được tạo dựng theo thứ tự như thế nào, hãy xem sơ đồ ở phần dưới.

Trên phương diện quy nạp, nếu cần dựng danh sách các tập hợp mà mỗi tập có dạng $\{x_1, x_2, \dots, x_n\}$, ta xét mọi giá trị của x_1 , rồi sau đó duyệt tiếp $\{x_2, x_3, \dots, x_n\}$, tiếp tục xét mọi giá trị x_2 , rồi lại duyệt $\{x_3, x_4, \dots, x_n\}$, ..., cho đến khi nào tất cả các giá trị đều đã xác định. Lúc này, ta lưu tập vừa tạo lại vào danh sách và tiếp tục chuyển sang tập khác từ các giá trị khác của các x_i

```

1 void backtrack(int pos)
2 {
3     // Trường hợp cơ sở
4     if (<pos là vị trí cuối cùng>)
5     {
6         <output/lưu lại tập hợp đã dựng nếu thoả mãn>
7         return;
8     }
9
10    //Phần đệ quy
11    for (<tất cả giá trị i có thể ở vị trí pos>)
12    {
13        <thêm giá trị i vào tập đang xét>
14        backtrack(pos + 1);
15        <xóa bỏ giá trị i khỏi tập đang xét>
16    }
17 }
```

Việc thêm giá trị mới vào tập đang xét rồi cuối cùng xóa bỏ nó ra khỏi tập giải thích cho tên gọi "quay lui" của thuật toán. Đó là việc khôi phục lại trạng thái cũ của tập hợp sau khi kết thúc việc gọi đệ quy.

Bạn đọc có thể thử vẽ sơ đồ tính toán giống như bài giai thừa ở trên và quen sát xem các hàm đệ quy được gọi ra theo thứ tự như thế nào.

Bạn vừa tiếp thu một lượng khá lớn kiến thức, và có thể sẽ gặp chút vấn đề. Không sao, chúng ta hãy đi vào ví dụ để hiểu hơn.

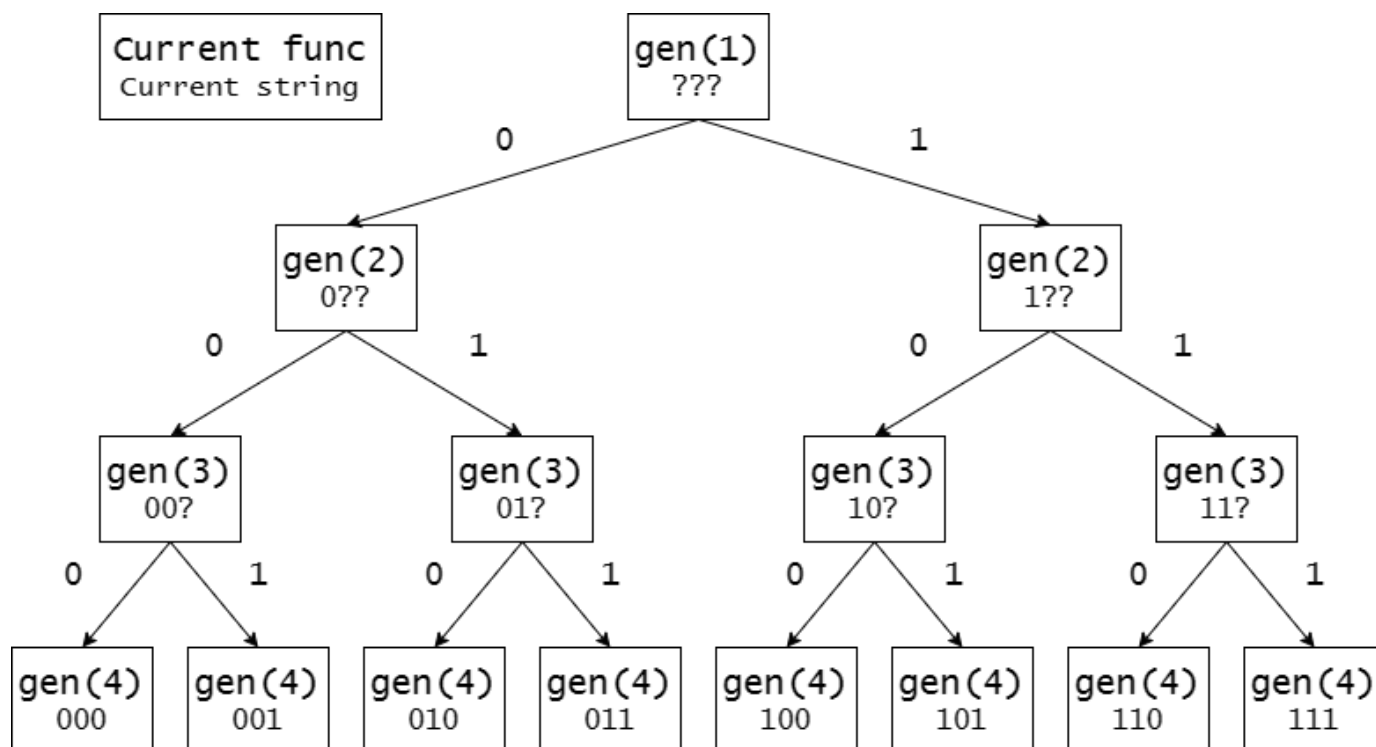
Sinh các dãy nhị phân

Bài toán: Liệt kê tất cả các dãy nhị phân độ dài n , là dãy có tất cả n ký tự và gồm toàn các ký tự 0 và 1.

Ví dụ, với $n = 3$ ta có các dãy 000, 001, 010, 011, 100, 101, 110, 111.

Phân tích:

Ở ví dụ phía trên, chúng ta đã nói về việc xét mọi trường hợp để xây dựng các dãy này như thế nào. Khi cài đặt đệ quy, sử dụng tư duy quy nạp "xây tập sau từ tập trước", thuật toán sẽ hoạt động như sau:



Tại hàm `gen(1)`, ta xét từng giá trị của ký tự hiện tại, sau đó gọi `gen(2)` với từng ký tự đó. Tương tự như vậy, ta gọi `gen(3)` từ các ký tự ở `gen(2)` và rồi `gen(4)`. Tới `gen(4)`, ta đã duyệt hết các vị trí và không thể thử thêm nữa, nên có thể in ra xâu.

```

1  int n;
2  string curString;
3
4  void genString(int pos)
5  {
6      if (pos > n)
7      {
8          cout << curString << "\n";
9          return;
10     }
11     for (char i = '0'; i <= '1'; i++)
12     {
13         curString.push_back(i);    //thêm ký tự mới vào dãy
14         genString(pos + 1);
15         curString.pop_back();      //bỏ ký tự này đi
16     }
17 }
18

```

```

19 | int main()
20 | {
21 |     cin >> n;
22 |     curString = "";
23 |     genString(1);
24 |
25 |     return 0;
26 | }

```

Chú ý rằng, cách sinh này cũng chưa phải là tốt nhất nếu xét về độ dài của code. Sử dụng các phép toán trên bit của C++ sẽ giúp liệt kê tất cả các dãy trên với một đoạn code đơn giản hơn nhiều mà thời gian chạy vẫn không chậm hơn (tất nhiên là không cần sử dụng đệ quy).

Sinh tổ hợp (tập hợp con)

Bài toán: Cho tập $S = \{1, 2, 3, \dots, n\}$. In ra tất cả các tập con có chính xác k phần tử của S . Hai tập con là hoán vị của nhau chỉ tính là một.

Phân tích:

Có một số ý tưởng cho bài này, như biểu diễn tập hợp bằng một dãy nhị phân rồi tìm các dãy có đúng k ký tự 1, hay lần lượt xây dựng các số trong dãy sao cho số sau lớn hơn số trước đến khi đủ k . Mình sẽ trình bày hướng thứ hai.

Để tránh trùng lặp, ta luôn luôn dựng các tập con P là các dãy thoả mãn $P_i > P_{i-1}$, hay mọi phần tử đều lớn hơn hẳn phần tử được dựng trước đó. Giả sử ta đã xây dựng được dãy đến vị trí thứ i , và P_i là giá trị cuối cùng được thêm vào. Tại vị trí thứ $i + 1$, do có $P_{i+1} > P_i$, nên ta chỉ thử các số từ $P_i + 1$ đến n .

Phần đệ quy sẽ kết thúc khi tập con đã có đủ k phần tử.

Sử dụng ý tưởng trên ta cài đặt như sau:

```

int n, k;
vector <int> curSubset;

//Hàm đệ quy
void printSubset()
{
    for (int i : curSubset) cout << i << " ";
    cout << "\n";
}

void genSubset(int pos)
{
    int lastNum = (curSubset.empty() ? 0 : curSubset.back()); //số cuối cùng
    for (int i = lastNum + 1; i <= n; i++)
    {
        curSubset.push_back(i);
        if (curSubset.size() == k) printSubset();
        else genSubset(pos + 1);
        curSubset.pop_back();
    }
}

```

```

20     }
21 }
22
23
24 int main()
25 {
26     cin >> n >> k;
27     curSubset.clear();
28     genSubset(1);
29
30     return 0;
31 }

```

Mở rộng: Vấn đề bài trên nhưng giờ bỏ đi điều kiện "Hai tập con là hoán vị của nhau chỉ tính là một." thì chúng ta sẽ làm như thế nào? (gợi ý: lúc này thay vì có mọi số lớn hơn số liền trước, ta chỉ cần các số trong tập hợp khác nhau là đủ)

Còn về hướng biểu diễn dãy nhị phân, bạn đọc hãy thử tự suy nghĩ và cài đặt. Trong lập trình thi đấu, khi phải duyệt mọi tập con, cách này dễ đọc và hiệu quả hơn hẳn. Nhưng đây là bài giới thiệu về đệ quy nên là...

Bài toán phân tích số

Bài toán: Ở một quốc gia có n loại tiền gồm các mệnh giá a_1, a_2, \dots, a_n ($n \leq 10$). Có những cách nào để lấy các tờ tiền sao cho tổng mệnh giá của chúng là S ? Biết rằng mỗi mệnh giá tiền có thể được lấy nhiều lần và hai cách lấy là hoán vị của nhau chỉ tính là một.

Ví dụ: với 3 loại tiền mệnh giá 10, 20, 50, có 10 cách lấy tiền để có tổng là 100, bao gồm 10 tờ 10, hoặc 2 tờ 50, hoặc 3 tờ 10, 1 tờ 20 và 1 tờ 50, ...

Một cách rất tự nhiên, chúng ta sẽ tiếp tục làm tương tự như bài trước: lưu các tờ tiền đã có vào một tập hợp, sau đó lấy tiền sao cho tờ sau có mệnh giá không nhỏ hơn tờ trước. Hàm đệ quy như thế sẽ có dạng

```
genMoneySet(int pos)
```

vậy thì khi nào chúng ta dừng lại? Đó là khi tổng số tiền chúng ta đã lấy được đạt mức yêu cầu, hoặc lớn hơn. Khi đó, kết quả hợp lệ sẽ là trường hợp số tiền đạt mức yêu cầu.

Trong quá trình cài đặt, song song với việc duy trì một tập hợp tiền đang xây dựng `curMoneySet`, chúng ta sẽ cần lưu thêm một giá trị tổng `curMoneySum` để đơn giản tính toán.

```

1  int n, a[15];
2  long long S, curMoneySum;
3  vector <int> curMoneySet;
4
5  void printMoneySet()
6  {
7      for (auto i : curMoneySet) cout << a[i] << " ";
8      cout << "\n";
9  }
10
11 //Hàm đệ quy

```




```

12 void genMoneySet(int pos)
13 {
14     int lastIndex = (curMoneySet.empty() ? 1 : curMoneySet.back());
15     for (int i = lastIndex; i <= n; i++)
16     {
17         //Lấy thêm 1 tờ tiền mới vào tập hợp
18         curMoneySet.push_back(i);
19         curMoneySum += a[i];
20
21         //Gọi đệ quy
22         if (curMoneySum >= S)
23         {
24             if (curMoneySum == S) printMoneySet();
25         }
26         else genMoneySet(pos + 1);
27
28         //Bỏ tờ tiền này ra khỏi tập hợp
29         curMoneySet.pop_back();
30         curMoneySum -= a[i];
31     }
32 }
33
34 int main()
35 {
36     cin >> n >> S;
37     for (int i = 1; i <= n; i++) cin >> a[i];
38     curMoneySet.clear();
39     curMoneySum = 0;
40     genMoneySet(1);
41
42     return 0;
43 }

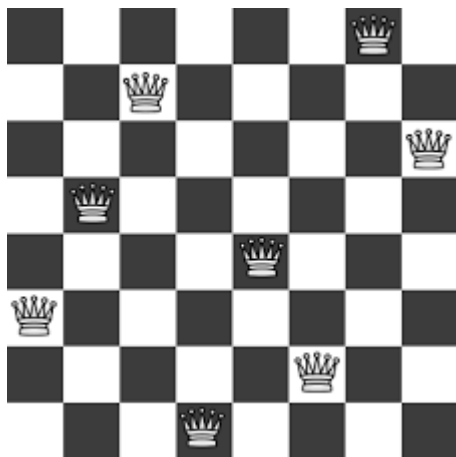
```

Nếu bạn đọc để ý kỹ thì chúng ta không sử dụng tham số `pos` trong hàm `genMoneySet` vào mục đích gì cả. Có thể bỏ tham số này đi, và chúng ta có một hàm đệ quy không tham số. Tham số này ở đây chỉ giúp chúng ta hiểu hàm này hơn thôi.

Bài toán xếp hậu

Xếp hậu  là bài toán rất kinh điển, có lẽ nếu bạn học đệ quy quay lui ở đâu thì cũng sẽ gặp.

Bài toán: Tìm tất cả các cách xếp n ($n \leq 12$) quân Hậu lên một bàn cờ $n \times n$ sao cho không có hai quân Hậu nào có thể ăn được nhau. Nếu có hai cách là hoán vị của nhau (về vị trí) thì chỉ tính là một, ví dụ hai tập hợp $\{(1, 2), (3, 4), (5, 6)\}$ và $\{(1, 2), (5, 6), (3, 4)\}$ chỉ lấy 1. Hai quân Hậu được gọi là có thể ăn được nhau nếu chúng nằm cùng hàng, cột hoặc đường chéo của bàn cờ.



(Hình ảnh tìm trên Google Images)

Phân tích:

Giả sử quân Hậu thứ i nằm ở hàng x_i và cột y_i . Cách gọi này hơi ngược một chút so với hệ toạ độ Decartes thông thường nhưng nó sẽ rất hợp cho những bài toán liên quan đến bảng.

Khi nào thì hai quân Hậu A và B ăn nhau?

- Khi A và B nằm cùng hàng, tức là $x_A = x_B$.
- Khi A và B nằm cùng cột, tức là $y_A = y_B$.
- Khi A và B nằm trên cùng đường chéo. Lúc này có hai trường hợp:
 - $x_A + y_A = x_B + y_B$
 - $x_A - y_A = x_B - y_B$

Đối với đường chéo, bạn đọc có thể tự kiểm chứng thấy rằng, hiệu hoặc tổng của chỉ số hàng và chỉ số cột luôn luôn là một số không đổi đối với hai phần tử cùng đường chéo, lần lượt ứng với đường chéo chính (hướng tây bắc - đông nam) và đường chéo phụ (hướng đông bắc - tây nam).

Vậy thì, việc của chúng ta bây giờ chỉ là sinh ra những bộ toạ độ đôi một thoả mãn các điều kiện trên thôi. Chúng ta sẽ sinh các bộ toạ độ này lần lượt theo từng hàng, và đảm bảo rằng quân Hậu sau sẽ không cùng cột và cùng đường chéo với quân Hậu trước. Để khỏi phải **for** lại từ đầu tập đã sinh để kiểm tra trùng lặp, chúng ta sẽ duy trì một số mảng đánh dấu cột, đường chéo phụ, đường chéo chính: `isInCol[]`, `isInDiag1[]`, `isInDiag2[]`

- `isInCol[k]` nhận giá trị **true** nếu đã có một quân Hậu đã nằm ở cột k .
- `isInDiag1[k]` nhận giá trị **true** nếu đã có một quân Hậu đã nằm ở đường chéo phụ có tổng toạ độ x và y là k
- `isInDiag2[k]` nhận giá trị **true** nếu đã có một quân Hậu đã nằm ở đường chéo chính có hiệu toạ độ x và y là k

Một vòng đệ quy sẽ kết thúc nếu ta sinh thành công n quân Hậu. Lúc này, ta chỉ việc in kết quả, và đi tiếp tới các trường hợp khác.

```

1 | int n;
2 |
3 | //mảng đánh dấu cột, đường chéo phụ và đường chéo chính
```

```
4  bool isInCol[13], isInDiag1[26], isInDiag2[26];
5
6  //gọi 2 tập riêng chi hàng và cột
7  //tập X có thể bỏ qua do các quân Hậu được sinh lần lượt theo từng hàng
8  vector <int> curQueensSetX, curQueensSetY;
9
10 //In kết quả dạng (X, Y)
11 void printQueensSet()
12 {
13     for (int i = 0; i < n; i++)
14     {
15         cout << "(" << curQueensSetX[i] << ", " << curQueensSetY[i] << ")";
16         if (i < n - 1) cout << ", ";
17     }
18     cout << "\n";
19 }
20
21 //Hàm đệ quy
22 void genQueensSet(int curRow)
23 {
24     for (int curCol = 1; curCol <= n; curCol++)
25     {
26         //Xác định đường chéo phụ và chính hiện tại
27         int curDiag1 = curRow + curCol;
28         int curDiag2 = curRow - curCol + 13;    //+13 để tránh chỉ số âm
29
30         //Kiểm tra toạ độ mới xem có thoả mãn không
31         if (isInCol[curCol] == true) continue;
32         if (isInDiag1[curDiag1] == true) continue;
33         if (isInDiag2[curDiag2] == true) continue;
34
35         //Thêm nó vào tập hợp hiện tại nếu thoả mãn
36         curQueensSetX.push_back(curRow);
37         curQueensSetY.push_back(curCol);
38         isInCol[curCol] = true;
39         isInDiag1[curDiag1] = true;
40         isInDiag2[curDiag2] = true;
41
42         //Gọi đệ quy thêm quân tiếp theo hoặc in kết quả
43         if (curQueensSetX.size() == n) printQueensSet();
44         else genQueensSet(curRow + 1);
45
46         //Xoá quân vừa thêm vào khỏi tập hợp
47         curQueensSetX.pop_back();
48         curQueensSetY.pop_back();
49         isInCol[curCol] = false;
50         isInDiag1[curDiag1] = false;
51         isInDiag2[curDiag2] = false;
52     }
53 }
54
55
```

```

56 int main()
57 {
58     cin >> n;
59
60     memset(isInCol, 0, sizeof(isInCol));
61     memset(isInDiag1, 0, sizeof(isInDiag1));
62     memset(isInDiag2, 0, sizeof(isInDiag2));
63
64     genQueensSet(1);
65
66     return 0;
67 }

```

Nếu bạn thực hiện thuật toán một cách chính xác, với $n = 8$ bạn sẽ thu được 92 cách xếp thoả mãn. 92 cách có vẻ nhiều, nhưng nếu không code, liệu bạn có xếp được không? 😊).

Kỹ thuật Nhánh cận

Trong một số trường hợp, thay vì yêu cầu liệt kê tất cả các cách chọn thoả mãn, ta sẽ phải tìm xem cách nào là **cách chọn tốt nhất**. Khi đó, việc dùng phương pháp **nhánh cận** (branch and bound) sẽ giúp chúng ta giải những bài toán dạng như vậy.

Thực chất, đây vẫn là thuật toán quay lui, nhưng thay vì in ra hoặc lưu lại tất cả kết quả trong mỗi lần tính toán, ta cập nhật lại trạng thái tốt nhất. Để thuật toán tối ưu hơn, nếu tại một bước, bất kỳ bước nào tiếp theo cũng không thể làm cho kết quả tốt hơn kết quả hiện có, ta có thể bỏ qua nó luôn.

Phương pháp nhánh cận thường được sử dụng trong các bài toán mà tại một vòng đệ quy, mọi cách đi tiếp đều không thể đưa ra một trường hợp thoả mãn. Từ đó, ta loại bỏ những công đoạn không cần thiết.

Quay trở lại bài toán phân tích số ở trên. Lần này, ta sẽ thêm vào đề bài một điều kiện:
"Số tờ tiền được xếp ra là nhỏ nhất. Nếu có nhiều cách xếp thoả mãn, chọn cách bất kỳ."

Vẫn với ý tưởng đệ quy như trên, chúng ta hoàn toàn có thể liệt kê tất cả cách xếp rồi lấy cách tốt nhất. Tuy nhiên, rõ ràng tại một số cách, số tiền còn lại khi duyệt tới những tờ giữa đã hơi "cấn" rồi. Ví dụ đã có một cách xếp $2 \times 20 + 1 \times 10 + 1 \times 50 = 100$, trong một bước khác mới xét tới 5×10 thôi đã chẳng còn ý nghĩa gì. Những cách đó có thể bỏ đi để chương trình chạy nhanh hơn.

```

int n, a[15];
long long S, curMoneySum;
vector <int> curMoneySet, bestSet;

void genMoneySet(int pos)
{
    int lastIndex = (curMoneySet.empty() ? 1 : curMoneySet.back());
    for (int i = lastIndex; i <= n; i++)
    {
        curMoneySet.push_back(i);
        curMoneySum += a[i];
    }
}

```

```

13         if (curMoneySum >= S)
14         {
15             if (curMoneySum == S)
16             {
17                 bestSet.clear();
18                 for (int i : curMoneySet) bestSet.push_back(i);
19             }
20         }
21         else if (bestSet.empty() || curMoneySet.size() < bestSet.size()) //lưu lại
22             genMoneySet(pos + 1);
23
24         curMoneySet.pop_back();
25         curMoneySum -= a[i];
26     }
27 }
28
29 int main()
30 {
31     cin >> n >> S;
32     for (int i = 1; i <= n; i++) cin >> a[i];
33     curMoneySet.clear();
34     curMoneySum = 0;
35     bestSet.clear();
36     genMoneySet(1);
37
38     for (int i : bestSet) cout << a[i] << " ";
39     cout << "\n";
40
41     return 0;
42 }

```

Chú ý thêm

Vì sao lại dùng đệ quy?

Ưu điểm mà chúng ta thấy ngay được của việc sử dụng đệ quy là viết code ngắn gọn hơn. Lấy ví dụ, khi tính số Fibonacci mà không sử dụng đệ quy, ta sẽ phải tạo hai biến nhớ cho số gần thứ nhì và gần nhất, cộng chúng lại, lưu vào biến mới rồi cập nhật hai biến nhớ; hoặc có thể sử dụng mảng rồi cập nhật lại sao mỗi lần tính. Chúng đều làm cho đoạn code trở nên dài hơn một chút so với việc dùng đệ quy ở trên. Nhưng ở những bài toán lớn hơn, ví dụ như những bài toán sinh dãy ở trên, việc không sử dụng đệ quy sẽ làm bài lời giải của chúng ta cồng



Một ưu điểm khác của đệ quy giúp giải dễ dàng các bài toán có dạng một phần nhỏ hơn của công việc cộng thêm một vài lệnh khác, ví dụ như các bài toán duyệt cây và đồ thị.

Tất nhiên, đệ quy không phải công cụ toàn năng. Đệ quy làm cho thuật toán trở nên khó hiểu hơn khi đọc trực tiếp, đặc biệt là với những thuật toán dài. Đệ quy cũng sử dụng thời gian và bộ nhớ hơn so với phương pháp

duyet trực tiếp, do bộ nhớ cần phải lưu trữ lại stack các hàm đệ quy.

Một số ứng dụng của đệ quy

Ngoài các bài toán sinh hoặc duyệt vét cạn, đệ quy còn được sử dụng phổ biến trong các bài toán duyệt cây, duyệt đồ thị và quy hoạch động. Rất nhiều bài toán "chia để trị" khác cũng sử dụng đệ quy, điển hình là thuật toán QuickSort.

Độ phức tạp của đệ quy


Một hàm đệ quy có dạng như sau:

```
1 void recursive(int x)
2 {
3     if (x > n) return;
4     for (int i = 1; i <= m; i++)
5         recursive(x + 1);
6 }
```

Hàm trên được gọi đệ quy n lần, mỗi lần phải thực hiện m lần vòng lặp nên độ phức tạp sẽ là $O(m^n)$.






Có thể thấy, các thuật toán đệ quy có thể có độ phức tạp rất lớn, nhiều khi lên tới hàm mũ, tuy vậy lại có lúc nhỏ cỡ \log như hàm tính ƯCLN. Do vậy, việc xác định số lần bị gọi của hàm đệ quy rất quan trọng.

Những bài toán yêu cầu duyệt vét cạn như ở trên thường đòi hỏi phải duyệt trên mọi trạng thái chưa biết, và vì thế dữ liệu đầu vào rất nhỏ.

Có một cách hiệu quả để xác định độ phức tạp của các hàm đệ quy là [định lý thợ \(Master Theorem\)](#)  (tên gọi tiếng Việt có thể khác tùy tài liệu). Chúng ta sẽ không đi sâu vào nó trong bài viết này.

Ngoài ra, trong một số bài toán yêu cầu tính toán ta cũng có thể lưu lại kết quả trả về của một vài vòng đệ quy để không phải duyệt lại những phần đã duyệt rồi. Phương pháp này gọi là **đệ quy có nhớ**.

Luyện tập

- ▶ [Một số bài tập trên VNOI](#) 
- ▶ [MNS](#) 
- ▶ [Bridge Crossing](#) 
- ▶ [Weird Rooks](#) 
- ▶ Giải Sudoku: Hãy thử vào [sudoku.com](#) , tìm một bảng Sudoku bất kì rồi thử viết một chương trình sử dụng đệ quy quay lui để giải nó.

Tài liệu tham khảo

- ▶ Lê Minh Hoàng (2003), *Giải thuật và lập trình*

Được cung cấp bởi [Wiki.js](#)