**Sử dụng Kỹ thuật Đệ quy - Recursion**

***Nội dung:***

* Đệ quy là gì?
* Khi nào dùng kỹ thuật đệ quy
* Kỹ thuật xây dựng hành vi đệ quy
* Nhận xét về hàm đệ quy
* Minh hoạ:
* Khử đệ quy
* Hồi quy - Backtracking
* Bài tập.

1. **Đệ quy là gì?**

Hàm đệ quy là hàm chứa trong thân lời gọi chính nó.

1. **Khi nào dùng kỹ thuật đệ quy**

* Muốn thay thế một diễn đạt lặp 🡪 Đệ quy là phương tiện diễn đạt vòng lặp.
* Diễn đạt cách thực hiện trong tự nhiên : Làm việc 1; làm việc 2; …. và cứ thế.

1. **Phân loại các dạng đệ quy**

**Góc nhìn về nơi gọi đệ quy trong hàm**:

Đệ quy đầu (head, left, nonTail), đệ quy đuôi (tail, right recursion)

**Góc nhìn về số lần và cách gọi đệ quy**

Đệ quy lồng nhau:

Đệ quy tuyến tính (linear- gọi 1 lần – hàm tìm giai thừa), nhị phân ( gọi 2 lần –hàm tìm trị Fibonacci), Đệ quy gián tiếp/ hỗ tương -correlative recursion, indirect

Các hàm đệ quy gọi qua lại nhau.

Đệ quy quá mức (excessive): Thân hàm gọi đệ quy nhiều lần ( từ 2 lần trở lên – bài toán Fibonacci)

1. **Kỹ thuật xây dựng hành vi đệ quy**

**Thí dụ : Tìm n! = 1\*2\*3\*…\*n = n(1\*2\*3\*…\*(n-1))🡪 Tác vụ int factorial (int n)**

**n! = 1 ; n<2**

**n! = n \* (n-1)!**

* **Bước 1**: Xác định tình huống nền – chặn ( ground case, anchor) ở đó không gọi hàm đệ quy: **if (n==1) return 1;**
* **Bước 2**: Xác định quy luật gọi chính tác vụ này để đô ohu71c tạp của tác vụ chuyển sần về tình hướng bị chặn.

**if (n>1) return n\* factorial(n-1) ;**

1. **Nhận xét về hàm đệ quy**

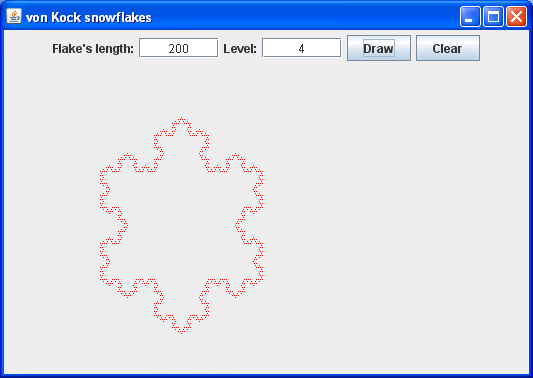
* Dễ hiện thực vì rất nhiều điều trong toán học hoặc tự nhiên được diễn đạt dạng đệ quy.
* Thực thi chậm (vì gọi hàm nhiều lần) và tốn bộ nhớ (vì có nhiều bản hàm phải được chạy tuần tự 🡪 có nhiều vùng nhớ chứa biến của các bản hàm trong stack- **activation record**) 🡪 Có thể cạn bộ nhớ.

1. **Minh hoạ**

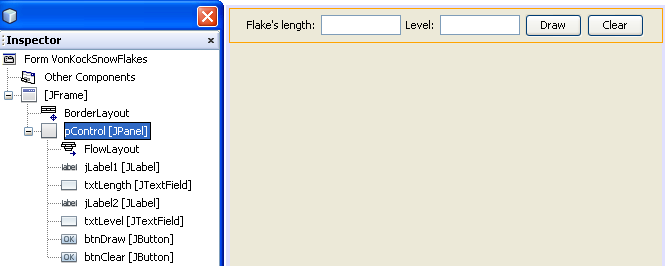
**Vẽ hình bánh tuyết von Kock ( von Kock snowflakes)**

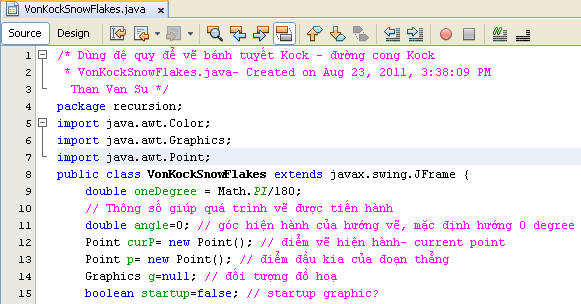
Trong minh hoạ này những điều học được:

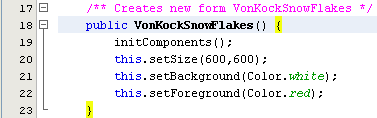
* Cách tạo đồ hoạ cơ bản trong Java
* Dùng kỹ thuật đệ quy để tạo sản phẩm đồ hoạ.

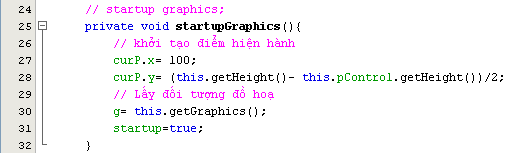


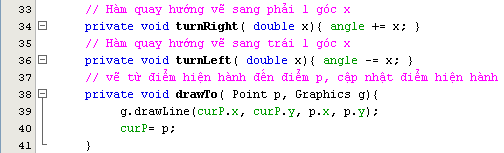
Thiết kế GUI

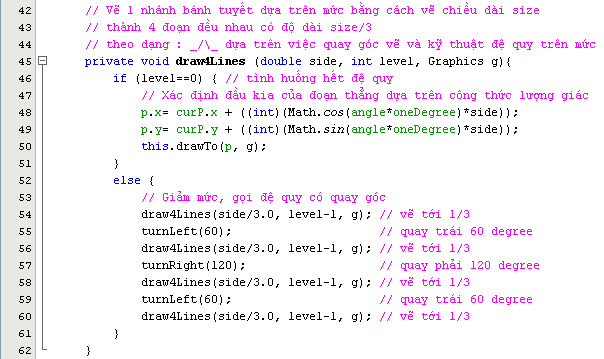
****

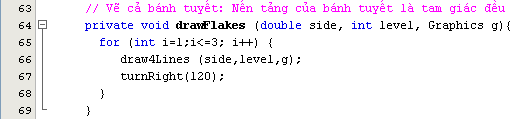
****

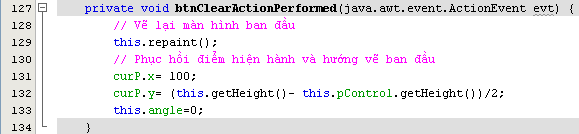
****

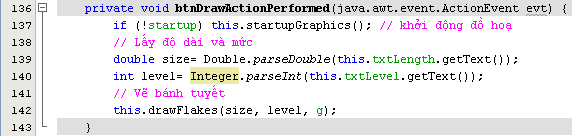
****

****

****

****

****

****

1. **Khử đệ quy**

**Đệ quy là một cách diễn đât vòng lặp 🡪 Dùng vòng lặp để khử đệ quy khi biết được cơ chế tính toán.**

*Thí dụ: Hàm tính giai thừa của một số nguyên – viết dạng đệ quy và không đệ quy*

|  |  |
| --- | --- |
| **int** factorial1 ( int n) {  **if** (n<2) return 1;  **return** n\* fatorial1(n-1);  } | **int** factorial1 ( int n) {  **int** result =1;  **for** (**int** i=2; i<=n;i++) result \*= i;  **return** result;  } |

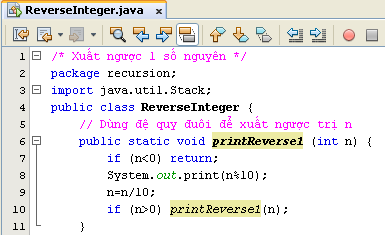
*Thí dụ: Hàm tính trị thứ n của dãy Fibonacci*

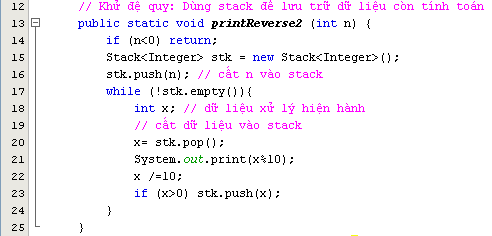
|  |  |
| --- | --- |
| **int** fibo1 ( int n) {  **if** (n<3) return 1;  **return** fnibo1(n-2) + fibo(n-1);  } | **int** fibo2 ( int n) {  **int** t1=1, t2=1, result =1;  **for** (**int** i=3; i<=n;i++) {  result = t1 + t2;  t2= result;  t1= t2;  }  **return** result;  } |

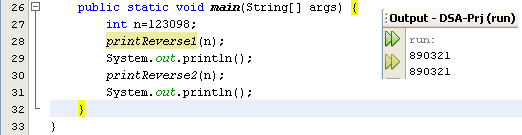
**Hàm đệ quy 🡪 cần cấp bộ nhớ để lưu trữ 🡪 dùng system stack**

* **Khử đệ quy: Dùng stack tự tạo để tạm lưu những các dữ liệu. 🡺 Chúng tương đương về bản chất.**

Thí dụ: Hàm xuất ngược các ký sô của 1 số nguyên dương







1. **Hồi quy – Backtracking**

**Bài toán:** Tập các biến **xi**, mỗi biến có thể có miền trị riêng **Di** có thể kèm theo tập điều kiện **C**.

**Lời giải – cấu hình (solution, configuration)** là một trạng thái của bài toán ở đó các biến đã được gán các trị thoả điều kiện tập C.

**Không gian bài toán:** problem space, state space: Tập các khả năng gán trị cho biến dù trị của biến có thoả mãn C hay không.

**Giải bài toán** được xem là quá trình gán trị cho các biến sao cho thoả C.

Với một bài toán cho trước, việc tìm được một cấu hình có thể phải dò tìm từ nhiều cách (hướng giải quyết) khác nhau (các cách gán trị khác nhau). Từ một hướng giải, hy vọng phát hiện một/vài cấu hình lời giải của bài toán. Trong tình huống xấu nhất, chúng ta phải xét hết các khả năng (xét hết các hướng giải – vét cạn – exhaustive searching) có thể có để giải bài toán.

Hồi quy là kỹ thuật cho phép ta quay lui trở về bước trước đó nếu như bước hiện hành thất bại để tìm ra lời giải.

*Áp dụng:*

|  |  |
| --- | --- |
| Bài toán | Cách áp dụng |
| Bài toán sinh dữ liệu như bài toán tìm các hoán vị, bài toán tìm tập con của một tập cho trước. | Vét cạn mọi khả năng |
| Tìm lời giải ban đầu cho các bài toán NP-complete như bài toán lập lịch để sau đó tinh chỉnh lời giải | Ngưng ngay sau khi phát hiện một lời giải |

*Nhận xét:*

* Nếu bài toán thực sự không có lời giải thì giải thuật quay lui cũng không tìm thấy lời giải vì đã vét cạn các khả năng.
* Giải thuật tốt nhưng có thể không hiệu quả vì có thể phải vét cạn mọi khả năng.
* Giải thuật backtrack là giải thuật tìm kiếm hệ thống (systematic searching) và chắc chắn tìm ra lời giải nếu bài bài toán thực sự có lời giải.

**Giải thuật Backtrack cơ bản dạng vét cạn – Quay lui 1 bước**

Đầu vào: Tập biến **xi**, tập miền trị **Di***, ­* điều kiện **C** (nếu có)

Đầu ra: Thành công/ thất bại, cấu hình **v0, v1, v2, …..,vn-1** với **vi** thuộc tập **Bi**

Ý tưởng: Thử gán trị cho biến thứ i. Nếu gán được thì tiếp tục gán biến thứ i+1. Nếu gán không được biến thứ i+1 thì quay lại lấy trị khác cho biến thứ i.

**success = false;**

**try ( i, xi, Di, C)**

**Begin**

**for v in Di** // xét từng trị trong miền trị của biến thứ i

**if** ( **v** satifies **C** when it is assigned to **xi** ) {

**xi = v;**

**if ( i = n-1 )** **{** // đã có 1 cấu hình, xử lý cấu hình này

**process (x0, x1, x2, …..,xn-1);**

**success= true;**

**}**

**else try ( i+1, xi, Di, C);**  // đệ quy tìm trị cho biến kế tiếp

}

**End**

**Giải thuật Backtrack dạng ngưng ngay khi phát hiện một lời giải**

Đầu vào: Biến chung **proceed** giúp ngắt tất cả các quá trình gọi đệ quy.

Tập biến **xi**, tập miền trị **Di***, ­* điều kiện **C** (nếu có)

Đầu ra: Thành công/ thất bại, cấu hình **v0, v1, v2, …..,vn-1** với vi thuộc tập **Bi**

**proceed= true;**

**success = false;**

**try ( i, xi, Di, C)**

**Begin**

**for v in Di** // xét từng trị trong miền trị của biến thứ i

**if** (**proceed** && **v** satifies **C** when it is assigned to **xi** ) {

**xi = v;**

**if ( i = n-1 )** **{** // đã có 1 cấu hình, xử lý cấu hình này

**process (x0, x1, x2, …..,xn-1);**

**success= true;**

**proceed= false;**

**}**

**else try ( i+1, x0, x1, x2, …..,xn-1 , C);**  // đệ quy tìm trị cho biến kế tiếp

}

**End**

**Việc quay lui ở chỗ nào trong giải thuật backtrack**

Giả sử hàm **try (5, xi, Di, C)** đang thực đang thực thi để gán trị cho biến **x5**. Giả sử trị **v2** trong miền **D5** hợp với **x5**, hàm **try (6, xi, Di, C)** được gọi trong hàm **try (5, xi, Di, C)** để gán trị cho biến **x6**. Già sử việc gán trị cho biến **x6** thất bại, việc quay lui sẽ quay lại tìm trị khác cho biến **x5**.

Chú ý rằng hàm **try (5, xi, Di, C)** tại lúc này chưa thực thi xong ( xem lại vòng lặp for) nên biến kết tiếp **v3** trong miền **D5** sẽ được xem xét ( quay lui về biến **x5**)

**Sử dụng giải thuật Backtrack để giải bài toán**

Đầu vào, Tập biến **xi**, tập miền trị **Di***, ­* điều kiện **C** (nếu có)

Đầu ra: Thành công/ thất bại

**solveProblem(x0, Di, C)**

**Begin**

**success= try ( 0, xi, Di, C, success);** // gán biến đi từ biến đầu tiên

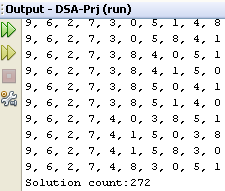
**if (success==false) message (“Failed”);**

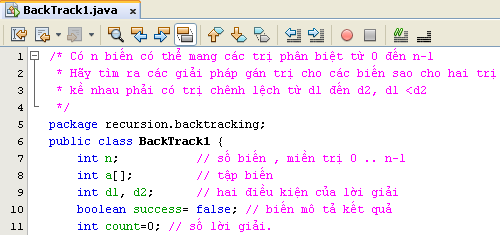
**End**

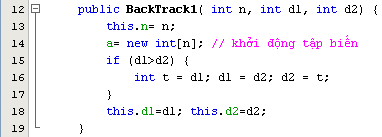
**Minh hoạ**

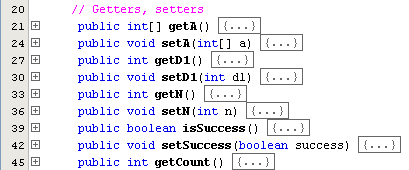
Bái toán: Có n biến mang các trị phân biệt từ 0.. n-1. Hai trị kế nhau phải chệnh lệch trong khoảng d1 đến d2 Viết chương trình xuất các lời giải có thể có. Test với n-10.

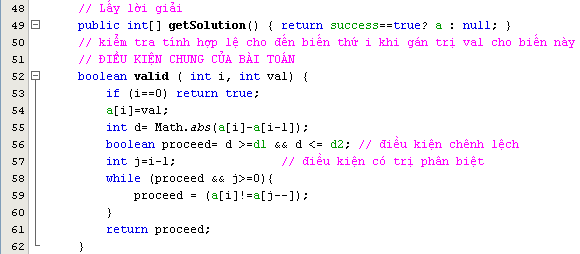
*Kết quả với n=10, d1= 3, d2=5*

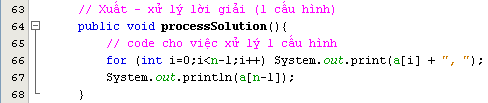


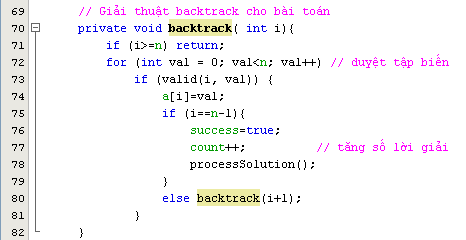


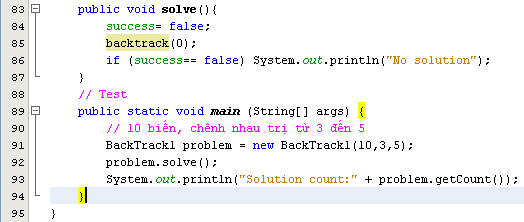






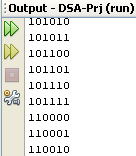
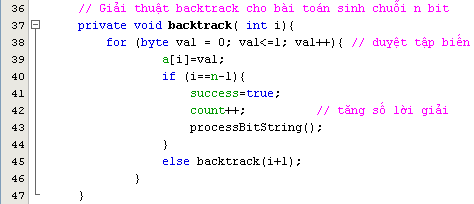




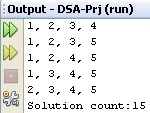
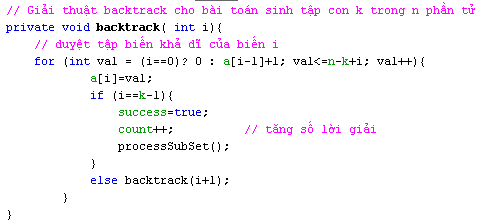


**Nhận xét**

* Nếu tập trị cho các biến chỉ là {0, 1} và không có điều kiện thì ta có lời giải sinh chuỗi bit (bạn tự code)



* Nếu thay tập trị được xét như sau sẽ giải bài toán sinh tập con k phần tử từ tập cha có n phần tử { 0, 1, ,,,, n-1}- Bạn tự code.

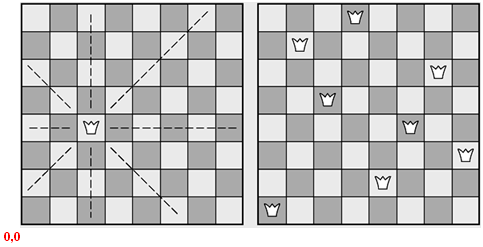


**Áp dụng Backtracking cho bài toán 8 hậu – 8-queens Problem**

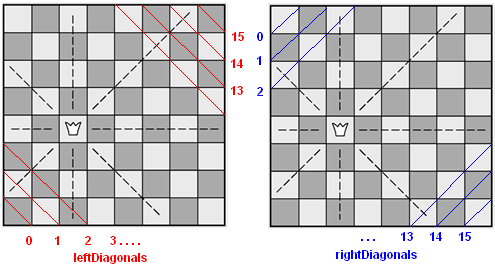
Người ta muốn đặt 8 quân hậu trên bàn cờ quốc tế 8x8 sao cho chúng không thể ăn nhau. Chú ý quân hậu có thể ăn thẳng, ăn ngang, ăn chéo quân đối diện nó ( bàn cờ n ô có thể xếp n quân hậu).

Nhận xét:

* Khi đặt 1 quân vào một vị trí thì không thể đặt một quân khác vào hàng/cột này (luật ăn thẳng). Như thế nếu xếp các quân theo hàng tăng dần từ thi không cần kiểm tra xung đột vầ hàng, chỉ cần kiểm tra xung đột về cột 🡪 Cần mảng các cột **columns** giúp đánh dấu cột thứ i đang sẵn sáng hay là đã bị khoá 🡪 **boolean[] columns**, cần mảng **int[] rows** để chứa vị trí của quân hậu được đặt. Phần tử **row[i]=a** mang ý nghĩa quân hậu thứ i được đặt tại vi trí **(i,a)**. Các chỉ số i sẽ đánh chỉ số hàng từ dưới lên trên, cột từ trái sang phải.



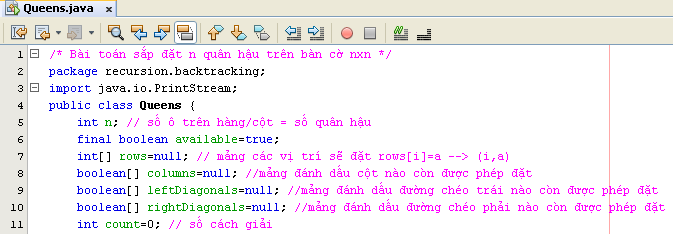
* Khi đặt 1 quân vào một vị trí thì không thể đặt một quân vào một trong hai đường chéo tương ứng (luật ăn chéo). Bàn cờ có n ô thì có 2n-1 đường chéo trái và 2n-1 đường chéo phải.Nếu chúng ta lưu thông tin trạng thái của các đường chéo trái, phải sẽ giúp nhanh chóng khoá các đường chéo.

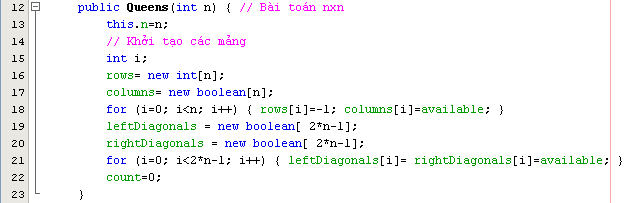


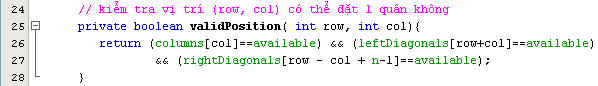
Khi 1 quân được đặt vào vị trí dòng r cột c thì

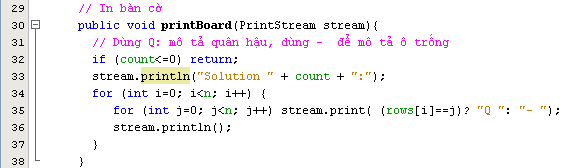
leftDiagonals[ r+c] bị khoá ( r=c=0 🡺 r+c=0)

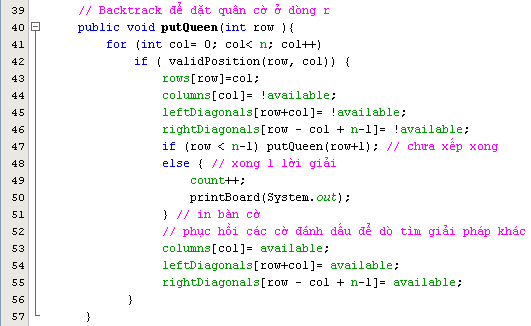
rightDiagonals[r-c+n-1] bị khoá (r=c=0, n=8 🡺 r-c + n-1=7)

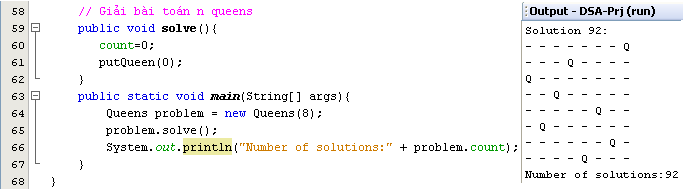












1. **Bài tập.**

Hiện thực lại các bài mẫu này.