

CHƯƠNG 2. ĐỆ QUI

Chương 2 trình bày các khái niệm về định nghĩa đệ qui, chương trình đệ qui. Ngoài việc trình bày các ưu điểm của chương trình đệ qui, các tình huống không nên sử dụng đệ qui cũng được đề cập cùng với các ví dụ minh họa.

Chương này cũng đề cập và phân tích một số thuật toán đệ qui tiêu biểu và kinh điển như bài toán tháp Hà nội, các thuật toán quay lui .v.v

Để học tốt chương này, sinh viên cần nắm vững phần lý thuyết. Sau đó, nghiên cứu kỹ các phân tích thuật toán và thực hiện chạy thử chương trình. Có thể thay đổi một số điểm trong chương trình và chạy thử để nắm kỹ hơn về thuật toán. Ngoài ra, sinh viên cũng có thể tìm các bài toán tương tự để phân tích và giải quyết bằng chương trình.

2.1 KHÁI NIỆM

Đệ qui là một khái niệm cơ bản trong toán học và khoa học máy tính. Một đối tượng được gọi là đệ qui nếu nó hoặc một phần của nó được định nghĩa thông qua khái niệm về chính nó. Một số ví dụ điển hình về việc định nghĩa bằng đệ qui là:

1- Định nghĩa số tự nhiên:

- 0 là số tự nhiên.
- Nếu k là số tự nhiên thì $k+1$ cũng là số tự nhiên.

Như vậy, bắt đầu từ phát biểu “0 là số tự nhiên”, ta suy ra $0+1=1$ là số tự nhiên.

Tiếp theo $1+1=2$ là số tự nhiên, v.v.

2- Định nghĩa xâu ký tự bằng đệ qui:

- Xâu rỗng là 1 xâu ký tự.
- Một chữ cái bất kỳ ghép với 1 xâu sẽ tạo thành 1 xâu mới.

Từ phát biểu “Xâu rỗng là 1 xâu ký tự”, ta ghép bất kỳ 1 chữ cái nào với xâu rỗng đều tạo thành xâu ký tự. Như vậy, chữ cái bất kỳ có thể coi là xâu ký tự.

Tiếp tục ghép 1 chữ cái bất kỳ với 1 chữ cái bất kỳ cũng tạo thành 1 xâu ký tự, v.v.

3- Định nghĩa hàm giai thừa, $n!$.

- Khi $n=0$, định nghĩa $0!=1$
- Khi $n>0$, định nghĩa $n!=(n-1)! \times n$

Như vậy, khi $n=1$, ta có $1!=0! \times 1 = 1 \times 1 = 1$. Khi $n=2$, ta có $2!=1! \times 2 = 1 \times 2 = 2$, v.v.

Trong lĩnh vực lập trình, một chương trình máy tính gọi là đệ qui nếu trong chương trình có lời gọi chính nó. Điều này, thoạt tiên, nghe có vẻ hơi vô lý. Một chương trình không thể gọi mãi chính

nó, vì như vậy sẽ tạo ra một vòng lặp vô hạn. Trên thực tế, một chương trình đệ qui trước khi gọi chính nó bao giờ cũng có một thao tác kiểm tra điều kiện dừng. Nếu điều kiện dừng thỏa mãn, chương trình sẽ không gọi chính nó nữa, và quá trình đệ qui chấm dứt. Trong các ví dụ ở trên, ta đều thấy có các điểm dừng. Chẳng hạn, trong ví dụ thứ nhất, nếu $k = 0$ thì có thể suy ngay k là số tự nhiên, không cần tham chiếu xem $k-1$ có là số tự nhiên hay không.

Nhìn chung, các chương trình đệ qui đều có các đặc điểm sau:

- Chương trình này có thể gọi chính nó.
- Khi chương trình gọi chính nó, mục đích là để giải quyết 1 vấn đề tương tự, nhưng nhỏ hơn.
- Vấn đề nhỏ hơn này, cho tới 1 lúc nào đó, sẽ đơn giản tới mức chương trình có thể tự giải quyết được mà không cần gọi tới chính nó nữa.

Khi chương trình gọi tới chính nó, các tham số, hoặc khoảng tham số, thường trở nên nhỏ hơn, để phản ánh 1 thực tế là vấn đề đã trở nên nhỏ hơn, dễ hơn. Khi tham số giảm tới mức cực tiểu, một điều kiện so sánh được kiểm tra và chương trình kết thúc, chấm dứt việc gọi tới chính nó.

Ưu điểm của chương trình đệ qui cũng như định nghĩa bằng đệ qui là có thể thực hiện một số lượng lớn các thao tác tính toán thông qua 1 đoạn chương trình ngắn gọn (thậm chí không có vòng lặp, hoặc không tường minh để có thể thực hiện bằng các vòng lặp) hay có thể định nghĩa một tập hợp vô hạn các đối tượng thông qua một số hữu hạn lời phát biểu. Thông thường, một chương trình được viết dưới dạng đệ qui khi vấn đề cần xử lý có thể được giải quyết bằng đệ qui. Tức là vấn đề cần giải quyết có thể đưa được về vấn đề tương tự, nhưng đơn giản hơn. Vấn đề này lại được đưa về vấn đề tương tự nhưng đơn giản hơn nữa .v.v, cho đến khi đơn giản tới mức có thể trực tiếp giải quyết được ngay mà không cần đưa về vấn đề đơn giản hơn nữa.

2.1.1 Điều kiện để có thể viết một chương trình đệ qui

Như đã nói ở trên, để chương trình có thể viết dưới dạng đệ qui thì vấn đề cần xử lý phải được giải quyết 1 cách đệ qui. Ngoài ra, ngôn ngữ dùng để viết chương trình phải hỗ trợ đệ qui. Để có thể viết chương trình đệ qui chỉ cần sử dụng ngôn ngữ lập trình có hỗ trợ hàm hoặc thủ tục, nhờ đó một thủ tục hoặc hàm có thể có lời gọi đến chính thủ tục hoặc hàm đó. Các ngôn ngữ lập trình thông dụng hiện nay đều hỗ trợ kỹ thuật này, do vậy vấn đề công cụ để tạo các chương trình đệ qui không phải là vấn đề cần phải xem xét. Tuy nhiên, cũng nên lưu ý rằng khi một thủ tục đệ qui gọi đến chính nó, một bản sao của tập các đối tượng được sử dụng trong thủ tục này như các biến, hằng, các thủ tục con, .v.v. cũng được tạo ra. Do vậy, nên hạn chế việc khai báo và sử dụng các đối tượng này trong thủ tục đệ qui nếu không cần thiết nhằm tránh lãng phí bộ nhớ, đặc biệt đối với các lời gọi đệ qui được gọi đi gọi lại nhiều lần. Các đối tượng cục bộ của 1 thủ tục

đệ qui khi được tạo ra nhiều lần, mặc dù có cùng tên, nhưng do khác phạm vi nên không ảnh hưởng gì đến chương trình. Các đối tượng đó sẽ được giải phóng khi thủ tục chứa nó kết thúc.

Nếu trong một thủ tục có lời gọi đến chính nó thì ta gọi đó là đệ qui trực tiếp. Còn trong trường hợp một thủ tục có một lời gọi thủ tục khác, thủ tục này lại gọi đến thủ tục ban đầu thì được gọi là đệ qui gián tiếp. Như vậy, trong chương trình khi nhìn vào có thể không thấy ngay sự đệ qui, nhưng khi xem xét kỹ hơn thì sẽ nhận ra.

2.1.2 Khi nào không nên sử dụng đệ qui

Trong nhiều trường hợp, một chương trình có thể viết dưới dạng đệ qui. Tuy nhiên, đệ qui không hẳn đã là giải pháp tốt nhất cho vấn đề. Nhìn chung, khi chương trình có thể viết dưới dạng lặp hoặc các cấu trúc lệnh khác thì không nên sử dụng đệ qui.

Lý do thứ nhất là, như đã nói ở trên, khi một thủ tục đệ qui gọi chính nó, tập các đối tượng được sử dụng trong thủ tục này như các biến, hằng, cấu trúc .v.v sẽ được tạo ra. Ngoài ra, việc chuyển giao điều khiển từ các thủ tục cũng cần lưu trữ các thông số dùng cho việc trả lại điều khiển cho thủ tục ban đầu.

Lý do thứ hai là việc sử dụng đệ qui đôi khi tạo ra các tính toán thừa, không cần thiết do tính chất tự động gọi thực hiện thủ tục khi chưa gặp điều kiện dừng của đệ qui. Để minh họa cho điều này, chúng ta sẽ xem xét một ví dụ, trong đó cả đệ qui và lặp đều có thể được sử dụng. Tuy nhiên, ta sẽ phân tích để thấy sử dụng đệ qui trong trường hợp này gây lãng phí bộ nhớ và các tính toán không cần thiết như thế nào.

Xét bài toán tính các phần tử của dãy Fibonacci. Dãy Fibonacci được định nghĩa như sau:

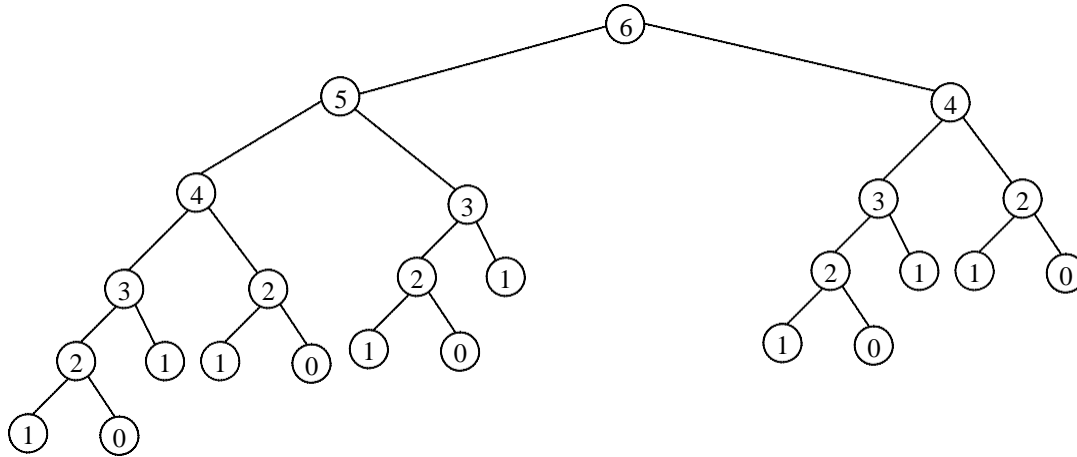
- $F(0) = 0$
- $F(1) = 1$
- Với $n > 1$ thì $F(n) = F(n-1) + F(n-2)$

Rõ ràng là nhìn vào một định nghĩa đệ qui như trên, chương trình tính phần tử của dãy Fibonacci có vẻ như rất phù hợp với thuật toán đệ qui. Phương thức đệ qui để tính dãy này có thể được viết như sau:

```
int Fibonacci(int i) {
    if (i==0) return
    0; if (i==1)
    return 1;
    return Fibonacci(i-1) + Fibonacci (i-2)
}
```

Kết quả thực hiện chương trình không có gì sai. Tuy nhiên, chú ý rằng một lời gọi đệ qui Fibonacci (n) sẽ dẫn tới 2 lời gọi đệ qui khác ứng với n-1 và n-2. Hai lời gọi

này lại gây ra 4 lời gọi nữa .v.v, cứ như vậy số lời gọi đệ qui sẽ tăng theo cấp số mũ. Điều này rõ ràng là không hiệu quả vì trong số các lời gọi đệ qui đó có rất nhiều lời gọi trùng nhau. Ví dụ lời gọi đệ qui Fibonacci (6) sẽ dẫn đến 2 lời gọi Fibonacci (5) và Fibonacci (4). Lời gọi Fibonacci (5) sẽ gọi Fibonacci (4) và Fibonacci (3). Ngay chỗ này, ta đã thấy có 2 lời gọi Fibonacci (4) được thực hiện. Hình 2.1 cho thấy số các lời gọi được thực hiện khi gọi thủ tục Fibonacci (6).



Hình 2.1 Các lời gọi đệ qui được thực hiện khi gọi thủ tục Fibonacci (6)

Trong hình vẽ trên, ta thấy để tính được phần tử thứ 6 thì cần có tới 25 lời gọi ! Sau đây, ta sẽ xem xét việc sử dụng vòng lặp để tính giá trị các phần tử của dãy Fibonacci như thế nào.

Đầu tiên, ta khai báo một mảng F các số tự nhiên để chứa các số Fibonacci. Vòng lặp để tính và gán các số này vào mảng rất đơn giản như sau:

```

F[0]=0;
F[1]=1;
for (i=2; i<n-1; i++)
    F[i] = F[i-1] + F[i-2];

```

Rõ ràng là với vòng lặp này, mỗi số Fibonacci (n) chỉ được tính 1 lần thay vì được tính toán chồng chéo như ở trên.

Tóm lại, nên tránh sử dụng đệ qui nếu có một giải pháp khác cho bài toán. Mặc dù vậy, một số bài toán tỏ ra rất phù hợp với phương pháp đệ qui. Việc sử dụng đệ qui để giải quyết các bài toán này hiệu quả và rất dễ hiểu. Trên thực tế, tất cả các giải thuật đệ qui đều có thể được đưa về dạng lặp (còn gọi là “khử” đệ qui). Tuy nhiên, điều này có thể làm cho chương trình trở nên phức tạp, nhất là khi phải thực hiện các thao tác điều khiển stack đệ qui (bạn đọc có thể tìm hiểu thêm kỹ thuật khử đệ qui ở các tài liệu tham khảo khác), dẫn đến việc chương trình trở nên rất khó hiểu. Phần tiếp theo sẽ trình bày một số thuật toán đệ qui điển hình.

2.2 THIẾT KẾ GIẢI THUẬT ĐỆ QUI

2.2.1 Chương trình tính hàm $n!$

Theo định nghĩa đã trình bày ở phần trước, $n! = 1$ nếu $n=0$, ngược lại, $n! = (n-1)! * n$.

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

Trong chương trình trên, điểm dừng của thuật toán đệ qui là khi $n=0$. Khi đó, giá trị của hàm `giaithua(0)` có thể tính được ngay lập tức mà không cần gọi hàm đệ qui khác. Nếu điều kiện dừng không thỏa mãn, sẽ có một lời gọi đệ qui hàm giai thừa với tham số là $n-1$, nhỏ hơn tham số ban đầu 1 đơn vị (tức là bài toán tính $n!$ đã được qui về bài toán đơn giản hơn là tính $(n-1)!$).

2.2.2 Thuật toán Euclid tính ước số chung lớn nhất của 2 số nguyên dương

Ước số chung lớn nhất (USCLN) của 2 số nguyên dương m, n là 1 số k lớn nhất sao cho m và n đều chia hết cho k . Một phương pháp đơn giản nhất để tìm USCLN của m và n là duyệt từ số nhỏ hơn trong 2 số m, n cho đến 1, ngay khi gặp số nào đó mà m và n đều chia hết cho nó thì đó chính là USCLN của m, n . Tuy nhiên, phương pháp này không phải là cách tìm USCLN hiệu quả. Cách đây hơn 2000 năm, Euclid đã phát minh ra một giải thuật tìm USCLN của 2 số nguyên dương m, n rất hiệu quả. Ý tưởng cơ bản của thuật toán này cũng tương tự như ý tưởng đệ qui, tức là đưa bài toán về 1 bài toán đơn giản hơn. Cụ thể, giả sử m lớn hơn n , khi đó việc tính USCLN của m và n sẽ được đưa về bài toán tính USCLN của $m \bmod n$ và n vì $\text{USCLN}(m, n) = \text{USCLN}(m \bmod n, n)$.

Thuật toán được cài đặt như sau:

```
int USCLN(int m, int  
    n){ if (n==0)  
    return m;  
    else return USCLN(n, m % n);  
}
```

Điểm dừng của thuật toán là khi $n=0$. Khi đó đương nhiên là USCLN của m và 0 chính là m , vì 0 chia hết cho mọi số. Khi n khác 0, lời gọi đệ qui $\text{USCLN}(n, m \% n)$ được thực hiện. Chú ý rằng ta giả sử $m \geq n$ trong thủ tục tính USCLN, do đó, khi gọi đệ qui ta gọi $\text{USCLN}(n, m \% n)$ để đảm bảo thứ tự các tham số vì n bao giờ cũng lớn hơn phần dư của phép m cho n . Sau mỗi lần gọi đệ qui, các tham số của thủ tục sẽ nhỏ dần đi, và sau 1 số hữu hạn lời gọi tham số nhỏ hơn sẽ bằng 0. Đó chính là điểm dừng của thuật toán.

Ví dụ, để tính USCLN của 108 và 45, ta gọi thủ tục $\text{USCLN}(108, 45)$. Khi đó, các

thủ tục sau sẽ lần lượt được gọi:

USCLN(108, 45) 108 chia 45 dư 18, do đó tiếp theo gọi

USCLN(45, 18) 45 chia 18 dư 9, do đó tiếp theo gọi

USCLN(18, 9) 18 chia 9 dư 0, do đó tiếp theo gọi

USCLN(9, 0) tham số thứ 2 = 0, do đó kết quả là tham số thứ nhất, tức là 9.

Như vậy, ta tìm được USCLN của 108 và 45 là 9 chỉ sau 4 lần gọi thủ tục.

2.2.3 Các giải thuật đệ qui dạng chia để trị (divide and conquer)

Ý tưởng cơ bản của các thuật toán dạng chia để trị là phân chia bài toán ban đầu thành 2 hoặc nhiều bài toán con có dạng tương tự và lần lượt giải quyết từng bài toán con này. Các bài toán con này được coi là dạng đơn giản hơn của bài toán ban đầu, do vậy có thể sử dụng các lời gọi đệ qui để giải quyết. Thông thường, các thuật toán chia để trị chia bộ dữ liệu đầu vào thành 2 phần riêng rẽ, sau đó gọi 2 thủ tục đệ qui để với các bộ dữ liệu đầu vào là các phần vừa được chia.

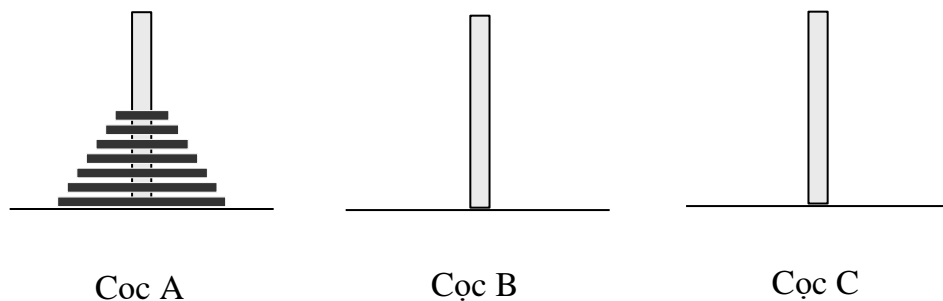
Một ví dụ điển hình của giải thuật chia để trị là Quicksort, một giải thuật sắp xếp nhanh. Ý tưởng cơ bản của giải thuật này như sau:

Giải sử ta cần sắp xếp 1 dãy các số theo chiều tăng dần. Tiến hành chia dãy đó thành 2 nửa sao cho các số trong nửa đầu đều nhỏ hơn các số trong nửa sau. Sau đó, tiến hành thực hiện sắp xếp trên mỗi nửa này. Rõ ràng là sau khi mỗi nửa đã được sắp, ta tiến hành ghép chúng lại thì sẽ có toàn bộ dãy được sắp. Chi tiết về giải thuật Quicksort sẽ được trình bày trong chương 7 - Sắp xếp và tìm kiếm.

Tiếp theo, chúng ta sẽ xem xét một bài toán cũng rất điển hình cho lớp bài toán được giải bằng giải thuật đệ qui chia để trị.

Bài toán tháp Hà nội

Có 3 chiếc cọc và một bộ n chiếc đĩa. Các đĩa này có kích thước khác nhau và mỗi đĩa đều có 1 lỗ ở giữa để có thể xuyên chúng vào các cọc. Ban đầu, tất cả các đĩa đều nằm trên 1 cọc, trong đó, đĩa nhỏ hơn bao giờ cũng nằm trên đĩa lớn hơn.



Hình 2.2 Bài toán tháp Hà nội

Yêu cầu của bài toán là chuyển bộ n đĩa từ cọc ban đầu A sang cọc đích C (có thể sử dụng cọc trung gian B), với các điều kiện:

- Mỗi lần chuyển 1 đĩa.
- Trong mọi trường hợp, đĩa có kích thước nhỏ hơn bao giờ cũng phải nằm trên đĩa có kích thước lớn hơn.

Với $n=1$, có thể thực hiện yêu cầu bài toán bằng cách chuyển trực tiếp đĩa 1 từ cọc A sang cọc

C.

Với $n=2$, có thể thực hiện như sau:

- Chuyển đĩa nhỏ từ cọc A sang cọc trung gian B.
- Chuyển đĩa lớn từ cọc A sang cọc đích C.
- Cuối cùng, chuyển đĩa nhỏ từ cọc trung gian B sang cọc đích C.

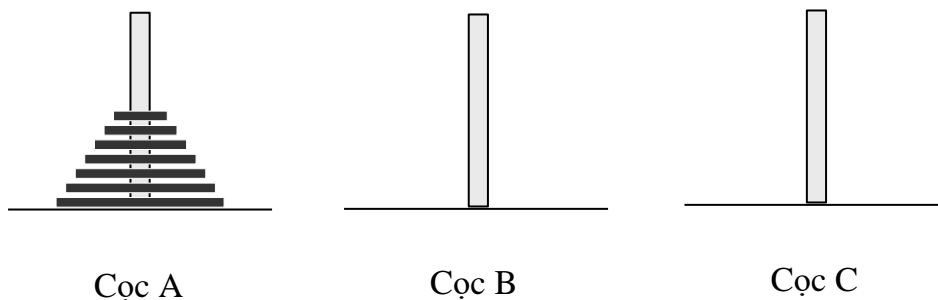
Như vậy, cả 2 đĩa đã được chuyển sang cọc đích C và không có tình huống nào đĩa lớn nằm trên đĩa nhỏ.

Với $n > 2$, giả sử ta đã có cách chuyển $n-1$ đĩa, ta thực hiện như sau:

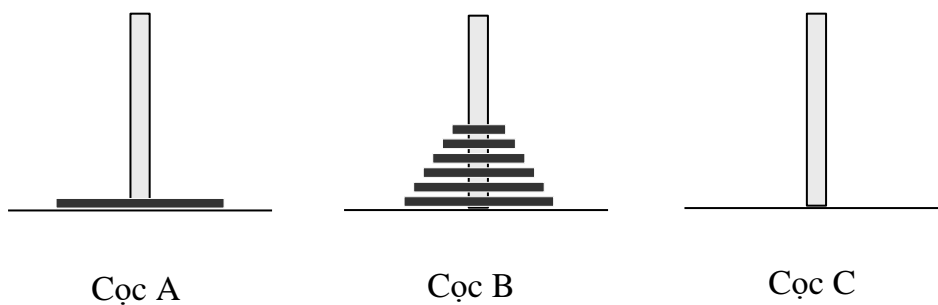
- Lấy cọc đích C làm cọc trung gian để chuyển $n-1$ đĩa bên trên sang cọc trung gian B.
- Chuyển cọc dưới cùng (cọc thứ n) sang cọc đích C.
- Lấy cọc ban đầu A làm cọc trung gian để chuyển $n-1$ đĩa từ cọc trung gian B sang cọc đích C.

Có thể minh họa quá trình chuyển này như

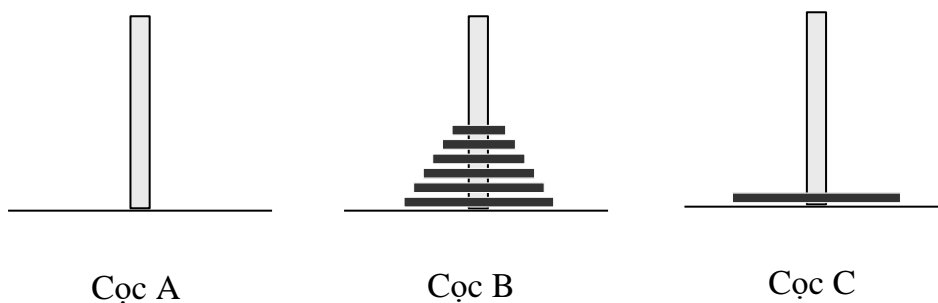
sau: Trạng thái ban đầu:



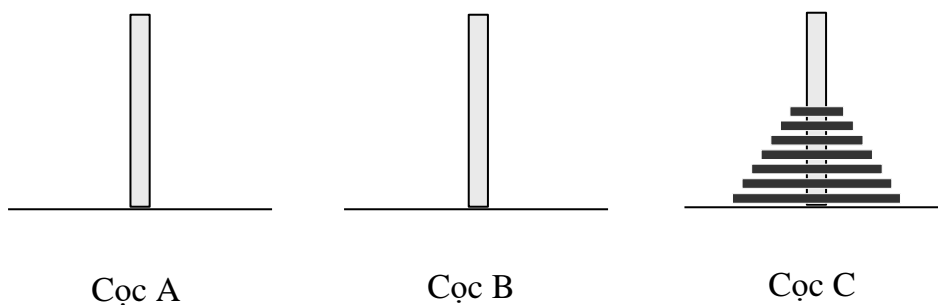
Bước 1: Chuyển $n-1$ đĩa bên trên từ cọc A sang cọc B, sử dụng cọc C làm cọc trung gian.



Bước 2: Chuyển đĩa dưới cùng từ cọc A thẳng sang cọc C.



Bước 3: Chuyển $n-2$ đĩa từ cọc B sang cọc C sử dụng cọc A làm cọc trung gian.



Như vậy, ta thấy toàn bộ n đĩa đã được chuyển từ cọc A sang cọc C và không vi phạm bất cứ điều kiện nào của bài toán.

Ở đây, ta thấy rằng bài toán chuyển n cọc đã được chuyển về bài toán đơn giản hơn là chuyển n-1 cọc. Điểm dừng của thuật toán đệ qui là khi n=1 và ta chuyển thẳng cọc này từ cọc ban đầu sang cọc đích.

Tính chất chia để trị của thuật toán này thể hiện ở chỗ: Bài toán chuyển n đĩa được chia làm 2 bài toán nhỏ hơn là chuyển n-1 đĩa. Lần thứ nhất chuyển n-1 đĩa từ cọc a sang cọc trung gian b, và lần thứ 2 chuyển n-1 đĩa từ cọc trung gian b sang cọc đích c.

Cài đặt đệ qui cho thuật toán như sau:

- Hàm chuyen(int n, int a, int c) thực hiện việc chuyển đĩa thứ n từ cọc a sang cọc c.
- Hàm thaphanoi(int n, int a, int c, int b) là hàm đệ qui thực hiện việc chuyển n đĩa từ cọc a sang cọc c, sử dụng cọc trung gian là cọc b.

Chương trình như sau:

```
void chuyen(int n, char a, char c){
    printf("Chuyen dia thu %d tu coc %c sang coc %c\n",n,a,c); return;
}

void thaphanoi(int n, char a, char c,
char b){ if (n==1) chuyen(1, a,
c);
else{
    thaphanoi(n-1, a, b, c);
    chuyen(n, a, c);
    thaphanoi(n-1, b, c,a);
}
return;
}
```

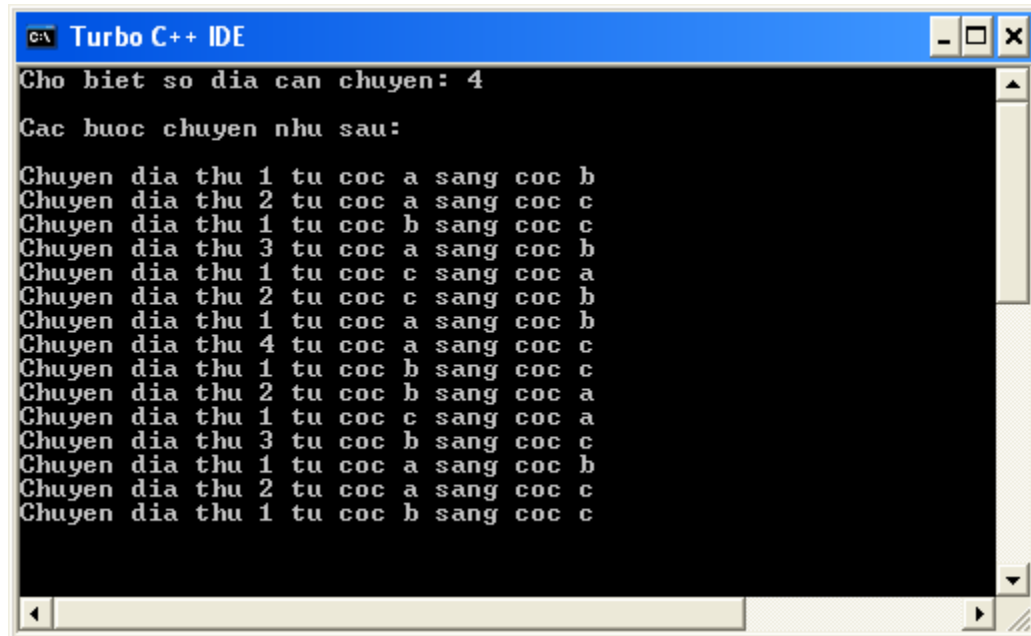
Hàm chuyen thực hiện thao tác in ra 1 dòng cho biết chuyển đĩa thứ mấy từ cọc nào sang cọc nào.

Hàm thaphanoi kiểm tra nếu số đĩa bằng 1 thì thực hiện chuyển trực tiếp đĩa từ cọc a sang cọc

c. Nếu số đĩa lớn hơn 1, có 3 lệnh được thực hiện:

- 1- Lệnh gọi đệ qui thapanoi(n-1, a, b, c) để chuyển n-1 đĩa từ cọc a sang cọc b, sử dụng cọc c làm cọc trung gian.
- 2- Thực hiện chuyển đĩa thứ n từ cọc a sang cọc c.
- 3- Lệnh gọi đệ qui thapanoi(n-1, b, c, a) để chuyển n-1 đĩa từ cọc b sang cọc c, sử dụng cọc a làm cọc trung gian.

Khi chạy chương trình với số đĩa là 4, ta có kết quả như sau:



```
C:\ Turbo C++ IDE
Cho biet so dia can chuyen: 4
Cac buoc chuyen nhu sau:
Chuyen dia thu 1 tu coc a sang coc b
Chuyen dia thu 2 tu coc a sang coc c
Chuyen dia thu 1 tu coc b sang coc c
Chuyen dia thu 3 tu coc a sang coc b
Chuyen dia thu 1 tu coc c sang coc a
Chuyen dia thu 2 tu coc c sang coc b
Chuyen dia thu 1 tu coc a sang coc b
Chuyen dia thu 4 tu coc a sang coc c
Chuyen dia thu 1 tu coc b sang coc c
Chuyen dia thu 2 tu coc b sang coc a
Chuyen dia thu 1 tu coc c sang coc a
Chuyen dia thu 3 tu coc b sang coc c
Chuyen dia thu 1 tu coc a sang coc b
Chuyen dia thu 2 tu coc a sang coc c
Chuyen dia thu 1 tu coc b sang coc c
```

Hình 2.3 Kết quả chạy chương trình tháp Hà nội với 4 đĩa

Độ phức tạp của thuật toán là $2^n - 1$. Nghĩa là để chuyển n cọc thì mất $2^n - 1$ thao tác chuyển. Ta sẽ chứng minh điều này bằng phương pháp qui nạp toán học:

Với $n=1$ thì số lần chuyển là $1 = 2^1 - 1$.

Giả sử giả thiết đúng với $n-1$, tức là để chuyển n-1 đĩa cần thực hiện $2^{n-1} - 1$ thao tác chuyển. Ta sẽ chứng minh rằng để chuyển n đĩa cần $2^n - 1$ thao tác chuyển.

Thật vậy, theo phương pháp chuyển của giải thuật thì có 3 bước. Bước 1 chuyển n-1 đĩa từ cọc a sang cọc b mất $2^{n-1} - 1$ thao tác. Bước 2 chuyển 1 đĩa từ cọc a sang cọc c mất 1 thao tác. Bước 3 chuyển n-1 đĩa từ cọc b sang cọc c mất $2^{n-1} - 1$ thao tác. Tổng cộng ta mất $(2^{n-1} - 1) + (2^{n-1} - 1) + 1 = 2 * 2^{n-1} - 1 = 2^n - 1$ thao tác chuyển. Đó là điều cần chứng minh.

Như vậy, thuật toán có cấp độ tăng rất lớn. Nói về cấp độ tăng này, có một truyền thuyết vui về bài toán tháp Hà nội như sau: Ngày tận thế sẽ đến khi các nhà sư ở một

ngôi chùa thực hiện xong việc chuyển 40 chiếc đĩa theo quy tắc như bài toán vừa trình bày. Với độ phức tạp của bài toán vừa tính được, nếu giả sử mỗi lần chuyển 1 đĩa từ cọc này sang cọc khác mất 1 giây thì với $2^{40}-1$ lần chuyển, các nhà sư này phải mất ít nhất 34.800 năm thì mới có thể chuyển xong toàn bộ số đĩa này !

Dưới đây là toàn bộ mã nguồn chương trình tháp Hà nội viết bằng C:

```
#include<stdio.h>
#include<conio.h>

void chuyen(int n, char a, char c);
void thapanoi(int n, char a, char c, char b);
void chuyen(int n, char a, char c){
    printf("Chuyen dia thu %d tu coc %c sang coc %c\n", n, a, c);
    return;
}

void thapanoi(int n, char a, char c,
char b){ if (n==1) chuyen(1, a,
c);
else{
    thapanoi(n-1, a, b, c);
    chuyen(n, a, c);
    thapanoi(n-1, b, c,a);
}
return;
}

void main(){
    int sodia;
    clrscr();
    printf("Cho biet so dia can chuyen: ");
    scanf("%d",&sodia);
    printf("\nCac buoc chuyen nhu
sau:\n\n"); thapanoi(sodia, 'a', 'c',
```

```

        'b') ;
    getch() ;
    return;
}

```

2.2.4 Thuật toán quay lui (backtracking algorithms)

Như chúng ta đã biết, các thuật toán được xây dựng để giải quyết vấn đề thường đưa ra 1 quy tắc tính toán nào đó. Tuy nhiên, có những vấn đề không tuân theo 1 quy tắc, và khi đó ta phải dùng phương pháp thử - sai (trial-and-error) để giải quyết. Theo phương pháp này, quá trình thử - sai được xem xét trên các bài toán đơn giản hơn (thường chỉ là 1 phần của bài toán ban đầu). Các bài toán này thường được mô tả dưới dạng đệ qui và thường liên quan đến việc giải quyết một số hữu hạn các bài toán con.

Để hiểu rõ hơn thuật toán này, chúng ta sẽ xem xét 1 ví dụ điển hình cho thuật toán quay lui, đó là bài toán Mã đi tuần.

Cho bàn cờ có kích thước $n \times n$ (có n^2 ô). Một quân mã được đặt tại ô ban đầu có tọa độ x_0, y_0 và được phép dịch chuyển theo luật cờ thông thường. Bài toán đặt ra là từ ô ban đầu, tìm một chuỗi các nước đi của quân mã, sao cho quân mã này đi qua tất cả các ô của bàn cờ, mỗi ô đúng 1 lần.

Như đã nói ở trên, quá trình thử - sai ban đầu được xem xét ở mức đơn giản hơn. Cụ thể, trong bài toán này, thay vì xem xét việc tìm kiếm chuỗi nước đi phủ khắp bàn cờ, ta xem xét vấn đề đơn giản hơn là tìm kiếm nước đi tiếp theo của quân mã, hoặc kết luận rằng không còn nước đi kế tiếp thỏa mãn. Tại mỗi bước, nếu có thể tìm kiếm được 1 nước đi kế tiếp, ta tiến hành ghi lại nước đi này cùng với chuỗi các nước đi trước đó và tiếp tục quá trình tìm kiếm nước đi. Nếu tại bước nào đó, không thể tìm nước đi kế tiếp thỏa mãn yêu cầu của bài toán, ta quay trở lại bước trước, hủy bỏ nước đi đã lưu lại trước đó và thử sang 1 nước đi mới. Quá trình có thể phải thử rồi quay lại nhiều lần, cho tới khi tìm ra giải pháp hoặc đã thử hết các phương án mà không tìm ra giải pháp.

Quá trình trên có thể được mô tả bằng hàm sau:

```

void ThuNuocTiepTheo;
{
    Khởi tạo danh sách các nước đi kế
    tiếp; do{
        Lựa chọn 1 nước đi kế tiếp từ danh
        sách; if Chấp nhận được
        {
            Ghi lại nước đi;

```

```

        if Bàn cờ còn ô trống
        {
            ThuNuocTiepTheo;
            if Nước đi không thành công
                Hủy bỏ nước đi đã lưu ở bước trước
        }
    }
}while (nước đi không thành công) && (vẫn còn nước đi)
}

```

Để thể hiện hàm 1 cách cụ thể hơn qua ngôn ngữ C, trước hết ta phải định nghĩa các cấu trúc dữ liệu và các biến dùng cho quá trình xử lý.

Đầu tiên, ta sử dụng 1 mảng 2 chiều để mô tả

bàn cờ: `int Banco[n][n];`

Các phần tử của mảng này có kiểu dữ liệu số nguyên. Mỗi phần tử của mảng đại diện cho 1 ô của bàn cờ. Chỉ số của phần tử tương ứng với tọa độ của ô, chẳng hạn phần tử `Banco[0][0]` tương ứng với ô (0,0) của bàn cờ. Giá trị của phần tử cho biết ô đó đã được quân mã đi qua hay chưa. Nếu giá trị ô = 0 tức là quân mã chưa đi qua, ngược lại ô đã được quân mã đi qua.

`Banco[x][y] = 0`: ô (x,y) chưa được quân mã đi qua

`Banco[x][y] = i`: ô (x,y) đã được quân mã đi qua tại nước thứ i.

Tiếp theo, ta cần phải thiết lập thêm 1 số tham số. Để xác định danh sách các nước đi kế tiếp, ta cần chỉ ra tọa độ hiện tại của quân mã, từ đó theo luật cờ thông thường ta xác định các ô quân mã có thể đi tới. Như vậy, cần có 2 biến x, y để biểu thị tọa độ hiện tại của quân mã. Để cho biết nước đi có thành công hay không, ta cần dùng 1 biến kiểu boolean.

Nước đi kế tiếp chấp nhận được nếu nó chưa được quân mã đi qua, tức là nếu ô (u,v) được chọn là nước đi kế tiếp thì `Banco[u][v] = 0` là điều kiện để chấp nhận. Ngoài ra, hiển nhiên là ô đó phải nằm trong bàn cờ nên $0 \leq u, v < n$.

Việc ghi lại nước đi tức là đánh dấu rằng ô đó đã được quân mã đi qua. Tuy nhiên, ta cũng cần biết là quân mã đi qua ô đó tại nước đi thứ mấy. Như vậy, ta cần 1 biến i để cho biết hiện tại đang thử ở nước đi thứ mấy, và ghi lại nước đi thành công bằng cách gán giá trị `Banco[u][v]=i`.

Do i tăng lên theo từng bước thử, nên ta có thể kiểm tra xem bàn cờ còn ô trống không bằng cách kiểm tra xem i đã bằng n^2 chưa. Nếu $i < n^2$ tức là bàn cờ vẫn còn ô trống.

Để biết nước đi có thành công hay không, ta có thể kiểm tra biến boolean như đã


nói ở trên. Khi nước đi không thành công, ta tiến hành hủy nước đi đã lưu ở bước trước bằng cách cho giá trị $Banco[u][v] = 0$.

Như vậy, ta có thể mô tả cụ thể hơn hàm ở trên như sau:

```
void ThuNuocTiepTheo(int i, int x, int y, int *q)
{
    int u, v, *q1;
    Khởi tạo danh sách các nước đi kế
    tiếp; do{
        *q1=0;
        Chọn nước đi (u,v) trong danh sách nước đi kế tiếp;
        if ((0 <= u) && (u<n) && (0 <= v) && (v<n) &&
            (Banco[u][v]==0))
        {
            Banco[u][v]=i
            ; if (i<n*n)
            {
                ThuNuocTiepTheo(i+1, u, v,
                q1) if (*q1==0)
                Banco[u][v]=0;
            } else *q1=1;
        }
    }while ((*q1==0) && (Vẫn còn nước đi))
    *q=*q1;
}
```

Trong đoạn chương trình trên vẫn còn 1 thao tác chưa được thể hiện bằng ngôn ngữ lập trình, đó là thao tác khởi tạo và chọn nước đi kế tiếp. Bây giờ, ta sẽ xem xét xem từ ô (x,y), quân mã có thể đi tới các ô nào, và cách tính vị trí tương đối của các ô đó so với ô (x,y) ra sao.

Theo luật cờ thông thường, quân mã từ ô (x,y) có thể đi tới 8 ô trên bàn cờ như trong hình vẽ:

	3		2	
4				1
				
5				8
	6		7	

x

y

Hình 2.4 Các nước đi của quân mã

Ta thấy rằng 8 ô mà quân mã có thể đi tới từ ô (x,y) có thể tính tương đối so với (x,y) là: (x+2, y-1); (x+1, y-2); (x-1, y-2); (x-2, y-1); (x-2, y+1); (x-1, y+2); (x+1, y+2); (x+2, y+1)

Nếu gọi dx, dy là các giá trị mà x, y lần lượt phải cộng vào để tạo thành ô mà quân mã có thể đi tới, thì ta có thể gán cho dx, dy mảng các giá trị như sau:

dx = {2, 1, -1, -2, -2, -1, 1, 2}

dy = {-1, -2, -2, -1, 1, 2, 2, 1}

Như vậy, danh sách các nước đi kế tiếp (u, v) có thể được tạo ra

như sau: $u = x + dx[i]$

$v = y +$

$dy[i] \quad i = ..8$

Chú ý rằng, với các nước đi như trên thì (u, v) có thể là ô nằm ngoài bàn cờ. Tuy nhiên, như đã nói ở trên, ta đã có điều kiện $0 \leq u, v < n$, do vậy luôn đảm bảo ô (u, v) được chọn là hợp lệ.

Cuối cùng, hàm *ThuNuocTiepTheo* có thể được viết lại hoàn toàn bằng ngôn ngữ C như sau:

```
void ThuNuocTiepTheo(int i, int x, int y, int *q)
{
    int k, u, v, *q1;
    k=0;
    do{
        *q1=0;
        u=x+dx[k];
        v=y+dy[k];
```

```

if ((0 <= u) && (u < n) && (0 <= v) && (v < n) &&
(Banco[u][v]==0))
{
    Banco[u][v]=i
    ; if (i<n*n)
    {
        ThuNuocTiepTheo(i+1, u, v,
        q1) if (*q1==0)
        Banco[u][v]=0;
    } else *q1=1;
}
k=k+1;

```



```

    }while ((*q1==0) && (k<8));
    *q=*q1;
}

```

Như vậy, có thể thấy đặc điểm của thuật toán là giải pháp cho toàn bộ vấn đề được thực hiện dần từng bước, và tại mỗi bước có ghi lại kết quả để sau này có thể quay lại và hủy kết quả đó nếu phát hiện ra rằng hướng giải quyết theo bước đó đi vào ngõ cụt và không đem lại giải pháp tổng thể cho vấn đề. Do đó, thuật toán được gọi là *thuật toán quay lui*.

Dưới đây là mã nguồn của toàn bộ chương trình Mã đi tuần viết bằng ngôn ngữ C:

```

#include<stdio.h>
#include<conio.h>
#define maxn 10
void ThuNuocTiepTheo(int i, int x, int y,
int *q); void InBanco(int n);
void XoaBanco(int n);

int Banco[maxn][maxn];
int dx[8]={2,1,-1,-2,-2,-1,1,2};
int dy[8]={-1,-2,-2,-1,1,2,2,1};
int n=8;

void ThuNuocTiepTheo(int i, int x, int y, int *q)
{
    int k, u, v,
    *q1; k=0;
    do{
        *q1=0;

        u=x+dx[k];
        v=y+dy[k];
        if ((0 <= u) && (u<n) && (0 <= v) && (v<n) &&
        && (Banco[u][v]==0))
        {
            Banco[u][v]
            =i; if

```

```

        (i<n*n)
        {
            ThuNuocTiepTheo(i+1, u, v,
                q1); if ((*q1)==0)
                Banco[u][v]=0;
            }else (*q1)=1;
        }
        k++;
        ;
    }while (((*q1)==0) && (k<8));
    *q=*q1;
}

void InBanco(int n){
    int i, j;
    for (i=0;i<=n-1;i++){
        for (j=0;j<=n-1;j++){
            if (Banco[i][j]<10) printf("%d
                                ",Banco[i][j]); else
                printf("%d      ",Banco[i][j]);
            printf("\n\n");
        }
    }

}

void XoaBanco(int n){
    int i, j;
    for (i=0;i<=n-1;i++)
        for (j=0;j<=n-1;j++) Banco[i][j]=0;
}

void main(){
    int *q=0;
    clrscr();
    printf("Cho kich thuoc ban co: ");
    scanf(" %d",&n);
    XoaBanco(n);
}

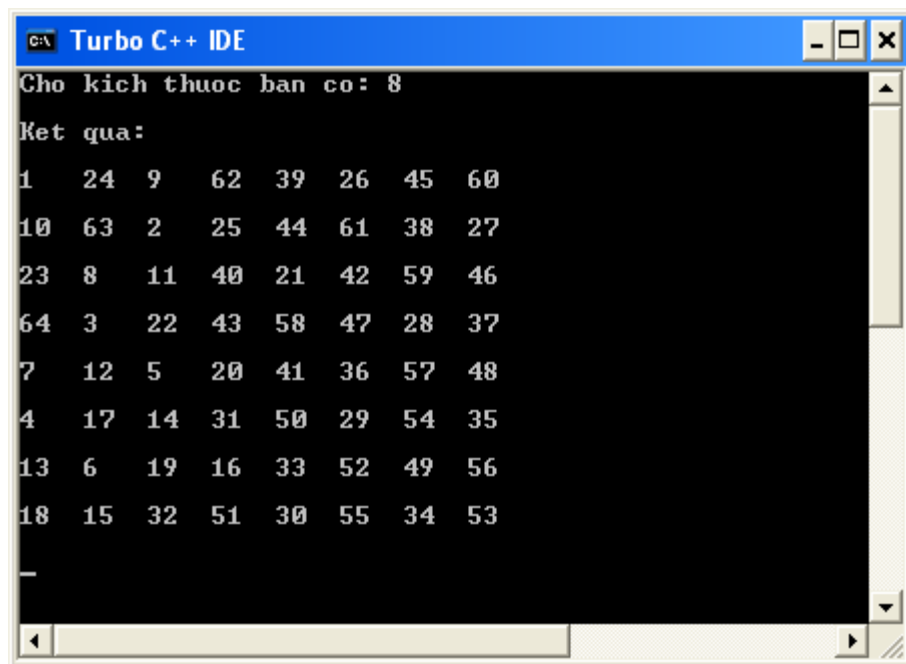
```

```

    Banco[0][0]=1;
    ThuNuocTiepTheo(2,0,0,q);
    printf("\n Ket qua: \n\n");
    InBanco(n);
    getch();
    return;
}

```

Và kết quả chạy chương trình với bàn cờ 8x8 và ô bắt đầu là ô (0,0):



Hình 2.5 Kết quả chạy chương trình mã đi tuần

Bài toán 8 quân hậu

Bài toán 8 quân hậu là 1 ví dụ rất nổi tiếng về việc sử dụng phương pháp thử - sai và thuật toán quay lui. Đặc điểm của các bài toán dạng này là không thể dùng các biện pháp phân tích để giải được mà phải cần đến các phương pháp tính toán thử công, với sự kiên trì và độ chính xác cao. Do đó, các thuật toán kiểu này phù hợp với việc sử dụng máy tính vì máy tính có khả năng tính toán nhanh và chính xác hơn nhiều so với con người.

Bài toán 8 quân hậu được phát biểu ngắn gọn như sau: Tìm cách đặt 8 quân hậu trên 1 bàn cờ sao cho không có 2 quân hậu nào có thể ăn được nhau.

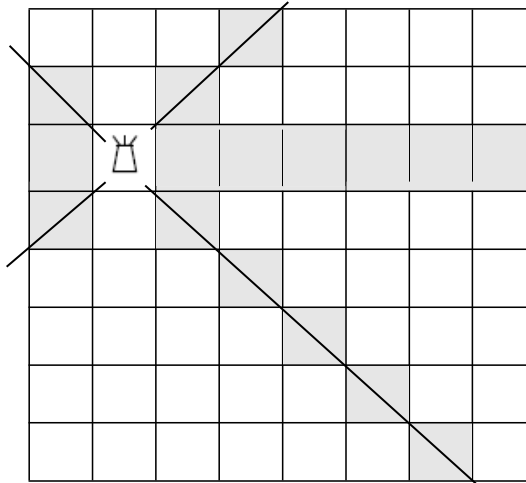
Tương tự như phân tích ở bài Mã đi tuần, ta có hàm DatHau để tìm vị trí đặt quân hậu tiếp theo như sau:

```

void DatHau(int i)
{
    Khởi tạo danh sách các vị trí có thể đặt quân hậu tiếp
    theo; do{
        Lựa chọn vị trí đặt quân hậu tiếp
        theo; if Vị trí đặt là an toàn
        {
            Đặt
            hậu; if
            i<8
            {
                DatHau(i+1);
                if Không thành công
                    Bỏ hậu đã đặt ra khỏi vị trí
            }
        }
    }while (Không thành công) && (Vẫn còn lựa chọn)
}

```

Tiếp theo, ta xem xét các cấu trúc dữ liệu và biến sẽ được dùng để thực hiện các công việc trong hàm. Theo luật cờ thông thường thì quân hậu có thể ăn tất cả các quân nằm trên cùng hàng, cùng cột, hoặc đường chéo. Do vậy, ta có thể suy ra rằng mỗi cột của bàn cờ chỉ có thể chứa 1 và chỉ 1 quân hậu, và từ đó ta có quy định là quân hậu thứ i phải đặt ở cột thứ i. Như vậy, ta sẽ dùng biến i để biểu thị chỉ số cột, và quá trình lựa chọn vị trí đặt quân hậu sẽ chọn 1 trong 8 vị trí trong cột cho biến chỉ số hàng j.



Hình 2.6 Các nước chiếu của quân hậu

Trong bài toán Mã đi tuần, ta sử dụng một mảng 2 chiều $Banco(i, j)$ để biểu thị bàn cờ. Tuy nhiên, trong bài toán này nếu tiếp tục dùng cấu trúc dữ liệu đó sẽ dẫn tới một số phức tạp trong việc kiểm tra vị trí đặt quân hậu có an toàn hay không, bởi vì ta cần phải kiểm tra hàng và các đường chéo đi qua ô quân hậu sẽ được đặt (không cần kiểm tra cột vì theo quy định ban đầu, có đúng 1 quân hậu được đặt trên mỗi cột). Đối với mỗi ô trong cột, sẽ có 1 hàng và 2 đường chéo đi qua nó là đường chéo trái và đường chéo phải.

Ta sẽ dùng 3 mảng kiểu boolean để biểu thị cho các hàng, các đường chéo trái, và các đường chéo phải (có tất cả 15 đường chéo trái và 15 đường chéo phải).

`int a[8];`

`int b[15], c[15];`

Trong đó:

`a[j] = 0`: Hàng j chưa bị chiếm bởi quân hậu nào.

`b[k] = 0`: Đường chéo trái k chưa bị chiếm bởi quân hậu

nào. `c[k] = 0`: Đường chéo phải k chưa bị chiếm bởi quân hậu nào.

Chú ý rằng các ô (i, j) cùng nằm trên 1 đường chéo trái thì có cùng giá trị $i + j$, và cùng nằm trên đường chéo phải thì có cùng giá trị $i - j$. Nếu đánh số các đường chéo trái và phải từ 0 đến 14, thì ô (i, j) sẽ nằm trên đường chéo trái $(i + j)$ và nằm trên đường chéo phải $(i - j + 7)$.

Do vậy, để kiểm tra xem ô (i, j) có an toàn không, ta chỉ cần kiểm tra xem hàng j và các đường chéo $(i + j)$, $(i - j + 7)$ đã bị chiếm chưa, tức là kiểm tra `a[i]`, `b[i + j]`, và `c[i - j + 7]`.

Ngoài ra, ta cần có 1 mảng x để lưu giữ chỉ số hàng của quân hậu

trong cột i . $\text{int } x[8]$;

Với thao tác đặt hậu vào vị trí hàng j trên cột i , ta cần thực hiện các công việc: $x[i] = j$; $a[j] = 1$; $b[i + j] = 1$; $c[i - j + 7] = 1$;

Với thao tác bỏ hậu ra khỏi hàng j trong cột i , ta cần thực hiện các công việc: $a[j] = 0$; $b[i + j] = 0$; $c[i - j + 7] = 0$;

Còn điều kiện để kiểm tra xem vị trí tại hàng j trong cột i có an toàn không là: $(a[j] == 0) \ \&\& \ (b[i + j] == 0) \ \&\& \ (c[i - j + 7] == 0)$

Như vậy, hàm *DatHau* sẽ được thể hiện cụ thể bằng ngôn ngữ C như sau:

```
void DatHau(int i, int *q)
{
    int j; j=0;
    do{
        *q=0;
        if ((a[j] == 0) && (b[i + j] == 0) && (c[i - j + 7] == 0))
        {
            x[i] = j;
            a[j] = 1;
            b[i + j] = 1;
            c[i - j + 7] = 1;
            if (i<7)
            {
                DatHau(i+1, q);

                if (*q==0)
                    j++;
            }
        }
        else (*q)=1;
        a[j] = 0; b[i + j] = 0; c[i - j + 7] = 0;
    }while ((*q==0) && (j<8))
}
```

2.3 TÓM TẮT CHƯƠNG 2

Các kiến thức cần nhớ trong chương 2:

- Định nghĩa bằng đệ qui: Một đối tượng được gọi là đệ qui nếu nó hoặc một phần của nó được định nghĩa thông qua khái niệm về chính nó.
- Chương trình đệ qui: Một chương trình máy tính gọi là đệ qui nếu trong chương trình có lời gọi chính nó (có kiểm tra điều kiện dừng).
- Để viết một chương trình dạng đệ qui thì vấn đề cần xử lý phải được giải quyết 1 cách đệ qui. Ngoài ra, ngôn ngữ dùng để viết chương trình phải hỗ trợ đệ qui (có hỗ trợ hàm và thủ tục).
- Nếu chương trình có thể viết dưới dạng lặp hoặc các cấu trúc lệnh khác thì không nên sử dụng đệ qui.
- Các thuật toán đệ qui dạng “chia để trị” là các thuật toán phân chia bài toán ban đầu thành 2 hoặc nhiều bài toán con có dạng tương tự và lần lượt giải quyết từng bài toán con này. Các bài toán con này được coi là dạng đơn giản hơn của bài toán ban đầu, do vậy có thể sử dụng các lời gọi đệ qui để giải quyết.
- Thuật toán quay lui dùng để giải quyết các bài toán không tuân theo 1 quy tắc, và khi đó ta phải dùng phương pháp thử - sai (trial-and-error) để giải quyết. Theo phương pháp này, quá trình thử - sai được xem xét trên các bài toán đơn giản hơn (thường chỉ là 1 phần của bài toán ban đầu). Các bài toán này thường được mô tả dưới dạng đệ qui và thường liên quan đến việc giải quyết một số hữu hạn các bài toán con.

2.4 CÂU HỎI VÀ BÀI TẬP

1. Hãy trình bày một số ví dụ về định nghĩa theo kiểu đệ qui.
2. Một chương trình đệ qui khi gọi chính nó thì bài toán khi đó có kích thước như thế nào so với bài toán ban đầu? Để chương trình đệ qui không bị lặp vô hạn thì cần phải làm gì?
3. Hãy cho biết tại sao khi chương trình có thể viết dưới dạng lặp hoặc cấu trúc khác thì không nên sử dụng đệ qui?
4. Viết chương trình đệ qui tính tổng các số lẻ trong khoảng từ 1 đến $2n+1$.
5. Hãy cho biết các bước thực hiện chuyển đĩa trong bài toán tháp Hà nội với số lượng đĩa là 5.
6. Hoàn thiện mã nguồn cho bài toán 8 quân hậu và chạy thử cho ra kết quả.