ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Chương 6:
Đệ quy và khử đệ quy

Nội dung

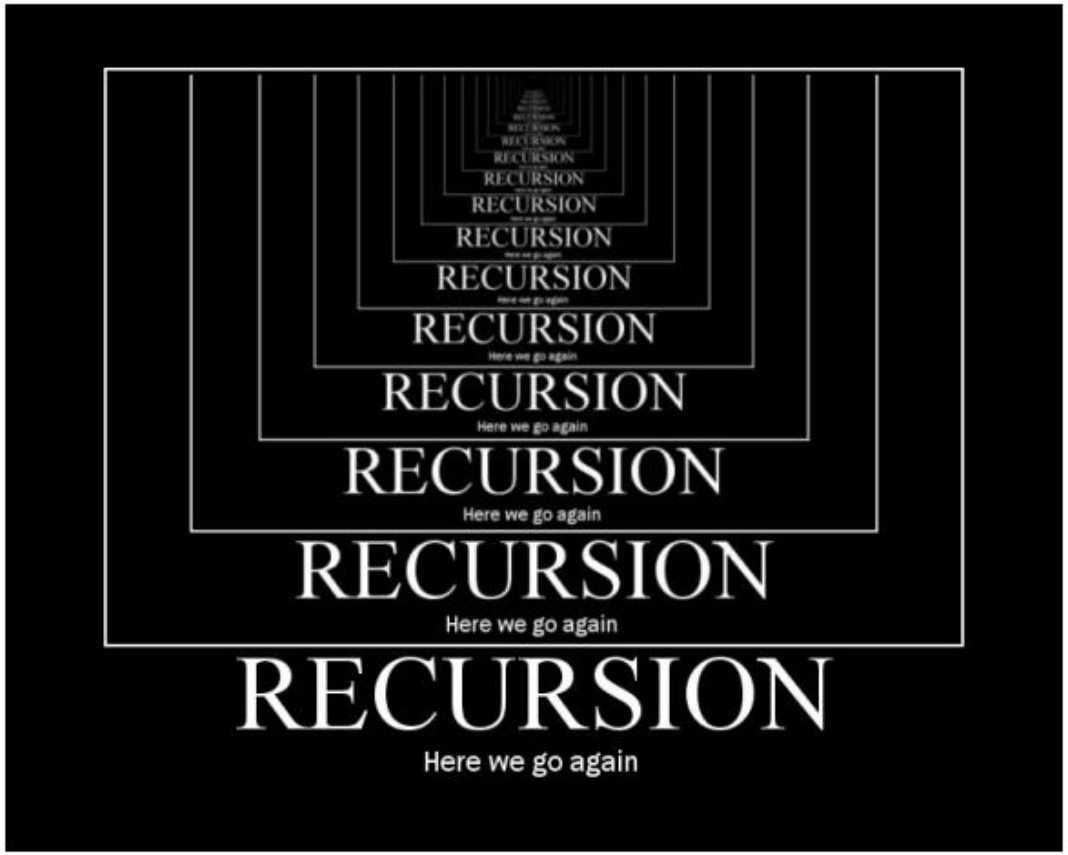
1. Nhắc lại khái niệm đệ quy
2. Phân loại đệ quy
3. Đệ quy có nhớ và đệ quy quay lui
4. Khử đệ quy

Nhắc lại khái niệm đệ quy

Khái niệm đệ quy

Là một kỹ thuật giải quyết vấn đề trong đó các vấn đề được giải quyết bằng cách chia nhỏ chúng thành các vấn đề nhỏ hơn có cùng dạng.

"A problem solving technique in which problems are solved by reducing them into smaller problems of the same form."



Ví dụ: Có bao nhiêu sinh viên ngồi sau bạn?

Có tất cả bao nhiêu bạn sinh viên ngồi ngay phía sau bạn theo "hàng dọc" trong lớp?

1. Bạn chỉ nhìn được người ngay phía trước và phía sau bạn.

Vì vậy, bạn không thể chỉ đơn giản xoay người lại và đếm.

2. Bạn được phép hỏi những người ngay trước hoặc sau bạn.

Liệu có thể giải quyết vấn đề này bằng đệ quy?

Ví dụ: Có bao nhiêu sinh viên ngồi sau bạn?

1. Người đầu tiên nhìn phía sau xem có người nào không. Nếu không, người này trả lời "0".
2. Nếu có người ngồi sau, lặp lại bước 1 và chờ câu trả lời.
3. Khi một người nhận câu trả lời, họ sẽ cộng thêm 1 vào câu trả lời của người ngồi sau và trả lời kết quả cho người hỏi họ.

Ví dụ: Có bao nhiêu sinh viên ngồi sau bạn?

```
int numStudentsBehind(Student curr) {
    if (noOneBehind(curr)) {
        return 0;
    } else {
        Student personBehind = curr.getBehind();
        return numStudentsBehind(personBehind) + 1;
    }
}
```

Recursive call!

Đệ quy

Cấu trúc các hàm đệ quy thường có dạng như sau:

```
recursiveFunction() {
    if (trường hợp cơ bản) {
        Tính toán lời giải trực tiếp không dùng đệ quy
    } else {
        Chia vấn đề thành nhiều vấn đề con cùng dạng
        Gọi đệ quy recursiveFunction() giải từng vấn đề con
        Kết hợp kết quả của các vấn đề con
    }
}
```

Đệ quy

Mọi giải thuật đệ quy đều cần ít nhất hai trường hợp:

- Trường hợp cơ bản (base case): là trường hợp đơn giản có thể tính toán câu trả lời trực tiếp. Các lời gọi đệ quy sẽ giảm dần tới trường hợp này.

- Trường hợp đệ quy (recursive case): là trường hợp phức tạp hơn của vấn đề mà không thể đưa ra câu trả lời trực tiếp được, nhưng có thể mô tả nó thông qua các trường hợp nhỏ hơn của cùng vấn đề.

Đệ quy

```
int numStudentsBehind(Student curr) {
    if (noOneBehind(curr)) {
        return 0;
    } else {
        Student personBehind = curr.getBehind();
        return numStudentsBehind(personBehind) + 1;
    }
}
```

Trường hợp cơ bản (base case)

Đệ quy

```
int numStudentsBehind(Student curr) {
    if (noOneBehind(curr)) {
        return 0;
    } else {
        Student personBehind = curr.getBehind();
        return numStudentsBehind(personBehind) + 1;
    }
}
```

Trường hợp đệ quy (recursive case)

Ba điều kiện cần của giải thuật đệ quy

- Mọi dữ liệu vào hợp lệ (valid input) phải tương ứng với một trường hợp nào đó trong code
- Phải có trường hợp cơ bản (base case) không thực hiện lời gọi đệ quy nào
- Khi thực hiện lời gọi đệ quy, lời gọi này cần gọi tới trường hợp đơn giản hơn của vấn đề và dần hướng tới trường hợp cơ bản.

Ví dụ: Tính hàm mũ

- Viết hàm đệ quy nhận một số x và số mũ n, trả về kết quả x^n

$$x^0 = 1$$
$$x^n = x \cdot x^{n-1}$$

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

Ví dụ: Tính hàm mũ

- Mỗi lời gọi trước sẽ đợi cho tới khi các lời gọi sau kết thúc và trả kết quả

```
cout << power(5, 3) << endl;
```

```
// first call: power (5, 3)
in// second call: power (5, 2)
   in// third call: power (5, 1)
      int// fourth call: power (5, 0)
      int power(int x, int exp) {
          if (exp == 0) {
              return 1;
          } else {
              return x * power(x, exp - 1);
          }
      }
   }
}
```

This call returns 1

Ví dụ: Tính hàm mũ

- Mỗi lời gọi trước sẽ đợi cho tới khi các lời gọi sau kết thúc và trả kết quả

```
cout << power(5, 3) << endl;
```

```
// first call: power (5, 3)
in// second call: power (5, 2)
   in// third call: power (5, 1)
   int power(int x, int exp) {
       if (exp == 0) {
           return 1;
       } else {
           return x * power(x, exp - 1);
       }
   }
}
   }
```

equals 1 from call

this entire statement returns 5 * 1

Ví dụ: Tính hàm mũ

- Mỗi lời gọi trước sẽ đợi cho tới khi các lời gọi sau kết thúc và trả kết quả

```
cout << power(5, 3) << endl;
```

```
// first call: power (5, 3)
in// second call: power (5, 2)
   int power(int x, int exp) {
       if (exp == 0) {
           return 1;
       } else {
           return x * power(x, exp - 1);
       }
   }
}
```

equals 5 from call

this entire statement returns 5 * 5

Ví dụ: Tính hàm mũ

- Mỗi lời gọi trước sẽ đợi cho tới khi các lời gọi sau kết thúc và trả kết quả

```
cout << power(5, 3) << endl;
```

```
// first call: power (5, 3)
int power(int x, int exp) {
    if (exp == 0) {
        return 1;
    } else {
        return x * power(x, exp - 1);
    }
}
```

equals 25 from call

this entire statement returns 5 * 25

Đây là lời gọi hàm ban đầu, trả về kết quả 125, tức là 5^3

Ví dụ: Tính hàm mũ nhanh hơn

```
int power(int x, int exp) {
    if(exp == 0) {
        // base case
        return 1;
    } else {
        if (exp % 2 == 1) {
        // if exp is odd
            return x * power(x, exp - 1);
        } else {
            // else, if exp is even
            int y = power(x, exp / 2);
            return y * y;
        }
    }
}
```

Độ phức tạp: O(logn)

Ví dụ: trace hàm đệ quy

```
int mystery(int n) {
    if (n < 10) {
        return n;
    } else {
        int a = n/10;
        int b = n % 10;
        return mystery(a + b);
    }
}
```

Hỏi kết quả của mystery(648)?

Ví dụ: trace hàm đệ quy

```
int mystery(int n) { // n = 648
    if (n < 10) {
        return n;
    } else {
        int a = n/10; // a = 64
        int b = n % 10; // b = 8
        return mystery(a + b); // mystery(72);
    }
}
```

Ví dụ: trace hàm đệ quy

```
int mystery(int n) { // n = 648
    int mystery(int n) { // n = 72
        if (n < 10) {
            return n;
        } else {
            int a = n/10; // a = 7
            int b = n % 10; // b = 2
            return mystery(a + b); // mystery(9);
        }
    }
}
```

Ví dụ: trace hàm đệ quy

```
int mystery(int n) { // n = 648
    int mystery(int n) { // n = 72
        int mystery(int n) { // n = 9
            if (n < 10) {
                return n; // return 9;
            } else {
                int a = n/10;
                int b = n % 10;
                return mystery(a + b);
            }
        }
    }
```

Ví dụ: trace hàm đệ quy

```
int mystery(int n) { // n = 648
    int mystery(int n) { // n = 72
        if (n < 10) {
            return n;
        } else {
            int a = n/10; // a = 7
            int b = n % 10; // b = 2
            return mystery(a + b); // mystery(9);
        }
    }
}
```

return 9;

Ví dụ: trace hàm đệ quy

```
int mystery(int n) { // n = 648
    if (n < 10) {
        return n;
    } else {
        int a = n/10; // a = 64
        int b = n % 10; // b = 8
        return mystery(a + b); // mystery(72);
    }
}
```

return 9;

Phân loại đệ quy

Phân loại đệ quy

- Đệ quy trực tiếp
  - Đệ quy tuyến tính
  - Đê qui nhị phân
  - Đệ quy phi tuyến
- Đệ quy gián tiếp
  - Đệ quy hỗ tương

Đệ quy tuyến tính

```
KieuDuLieu TenHam(Thamso){
    if(Dieu Kien Dung){
        ...;
        return Gia tri tra ve;
    }
    ...;
    TenHam(Thamso)
    ...;
}
```

- Là đệ quy có dạng

```
P( ) {
   If (B) thực hiện S;
   else { thực hiện S* ; gọi P }
}
```

với S , S* là các thao tác không đệ quy.

- Ví dụ tính giai thừa

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

Ví dụ đệ quy tuyến tính: Palindrome

- Viết hàm đệ quy isPalindrome nhận một xâu string và trả về true nếu xâu đó đọc ngược hay đọc xuôi đều như nhau

isPalindrome("madam") → true

isPalindrome("racecar") → true

isPalindrome("step on no pets") → true

isPalindrome("Java") → false

isPalindrome("byebye") → false

Ví dụ đệ quy tuyến tính: Palindrome

```
// Trivially true for empty or 1-letter strings.
bool isPalindrome(const string& s) {
    if (s.length() < 2) { // base case
        return true;
    } else { // recursive case
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        string middle = s.substr(1, s.length() - 2);
        return isPalindrome(middle);
    }
}
```

Đệ quy nhị phân

```
KieuDuLieu TenHam(Thamso){
    if(Dieu Kien Dung){
        ...;
        return Gia tri tra ve;
    }
    ...;
    TenHam(Thamso);
    ...;
    TenHam(Thamso);
    ...;
}
```

- Là đệ quy có dạng

```
P ( ) {
    If (B) thực hiện S;
    else {
            thực hiện S*;
            gọi P ; gọi P;
    }
}
```

với `S, S*` là các thao tác không đệ quy.

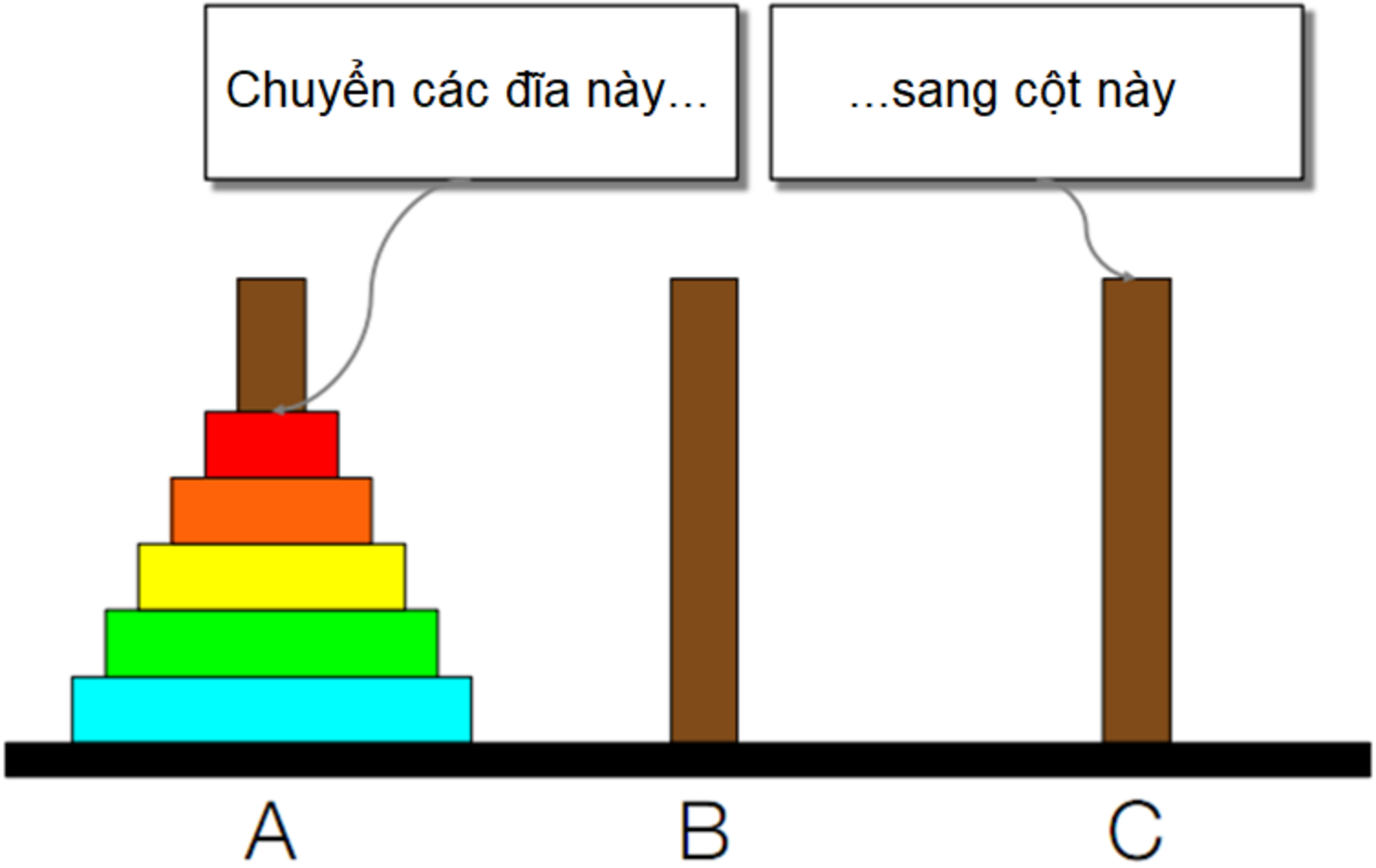
- Ví dụ tính số fibonacci

```
int fibo(int n) {
    if (n < 2) return 1;
    else return (fibo(n -1) + fibo(n -2));
}
```

Ví dụ đệ quy nhị phân: Tháp Hà Nội

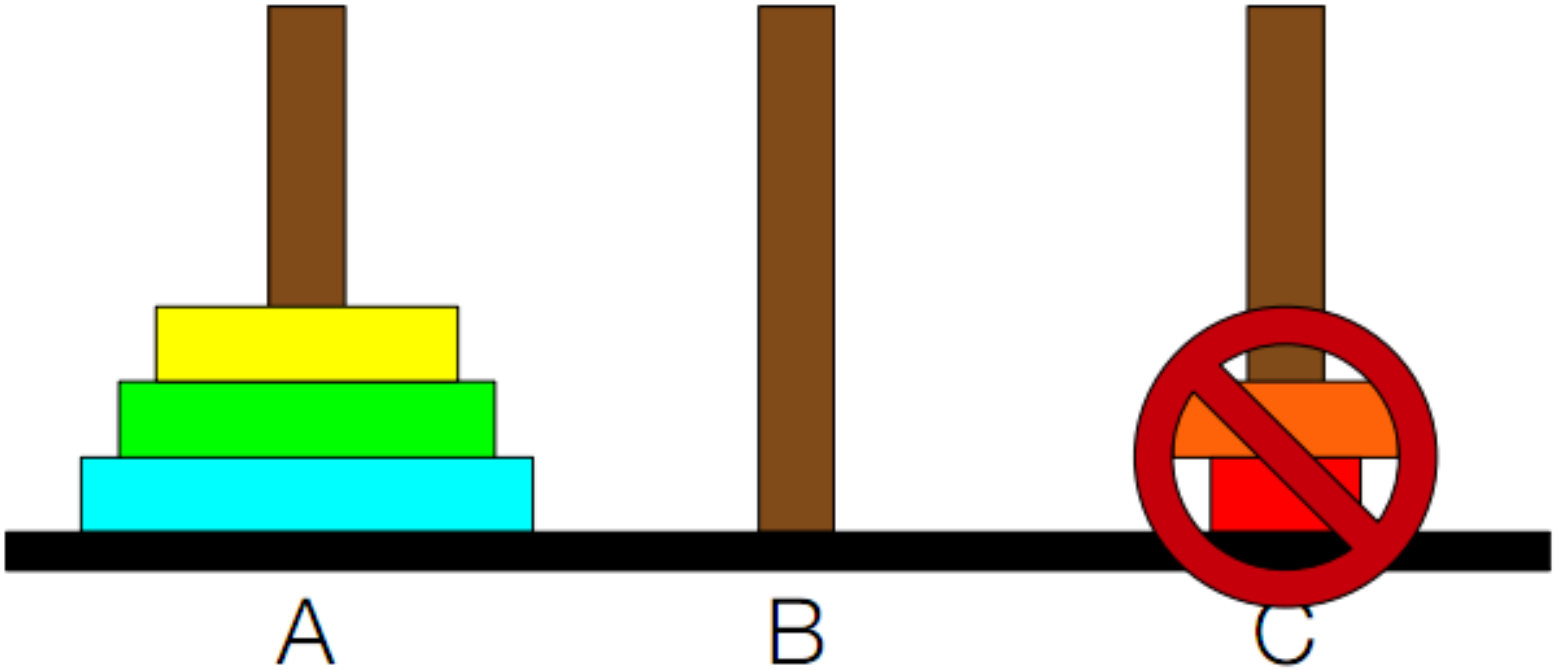
- Có 3 cọc A, B, C. Trên cọc A có một chồng gồm n cái đĩa đường kính giảm dần từ dưới lên trên. Cần phải chuyển chồng đĩa từ cọc A sang cọc C tuân thủ qui tắc: mỗi lần chỉ chuyển 1 đĩa và chỉ được xếp đĩa có đường kính nhỏ hơn lên trên đĩa có đường kính lớn hơn. Trong quá trình chuyển được phép dùng cọc B làm cọc trung gian.
- Bài toán đặt ra là: Hãy viết chương trình để hoàn thành nhiệm vụ đặt ra trong trò chơi tháp Hà nội với số lần di chuyển đĩa ít nhất cần thực hiện.

Ví dụ đệ quy nhị phân: Tháp Hà Nội



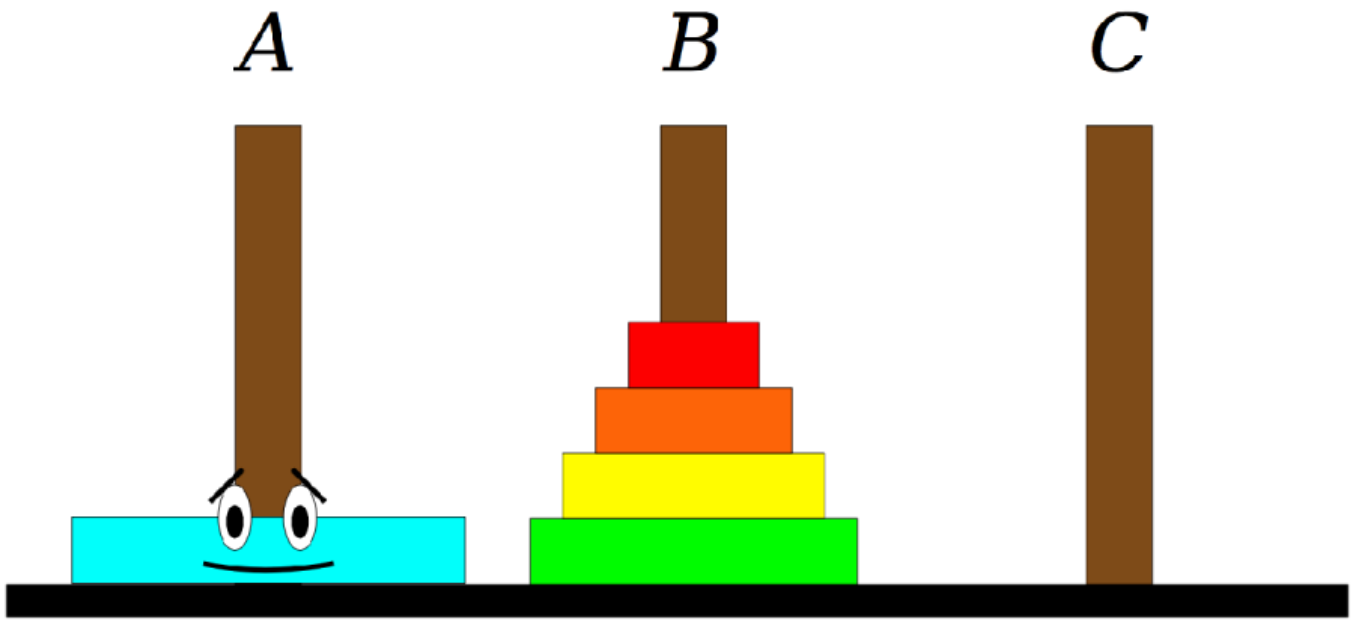
Ví dụ đệ quy nhị phân: Tháp Hà Nội

- Không được xếp đĩa có đường kính lớn hơn lên trên đĩa có đường kính nhỏ hơn



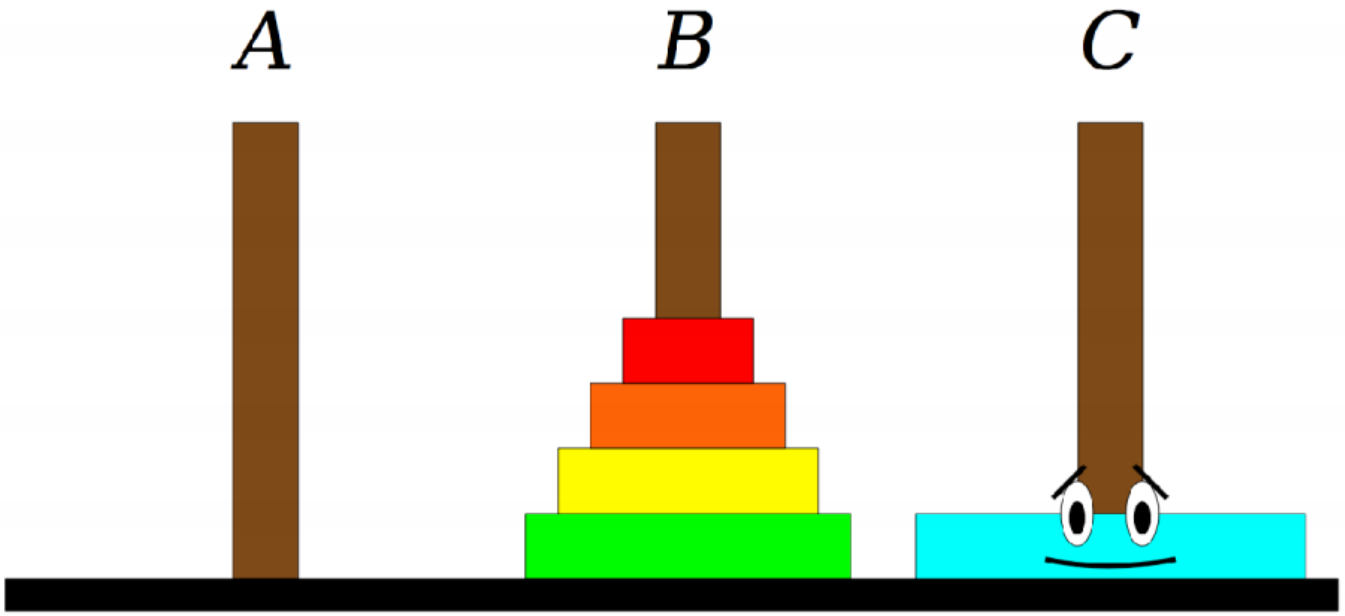
Ví dụ đệ quy nhị phân: Tháp Hà Nội

- Bước 1: Chuyển n – 1 đĩa từ cọc A sang cọc B dùng cọc C làm trung gian



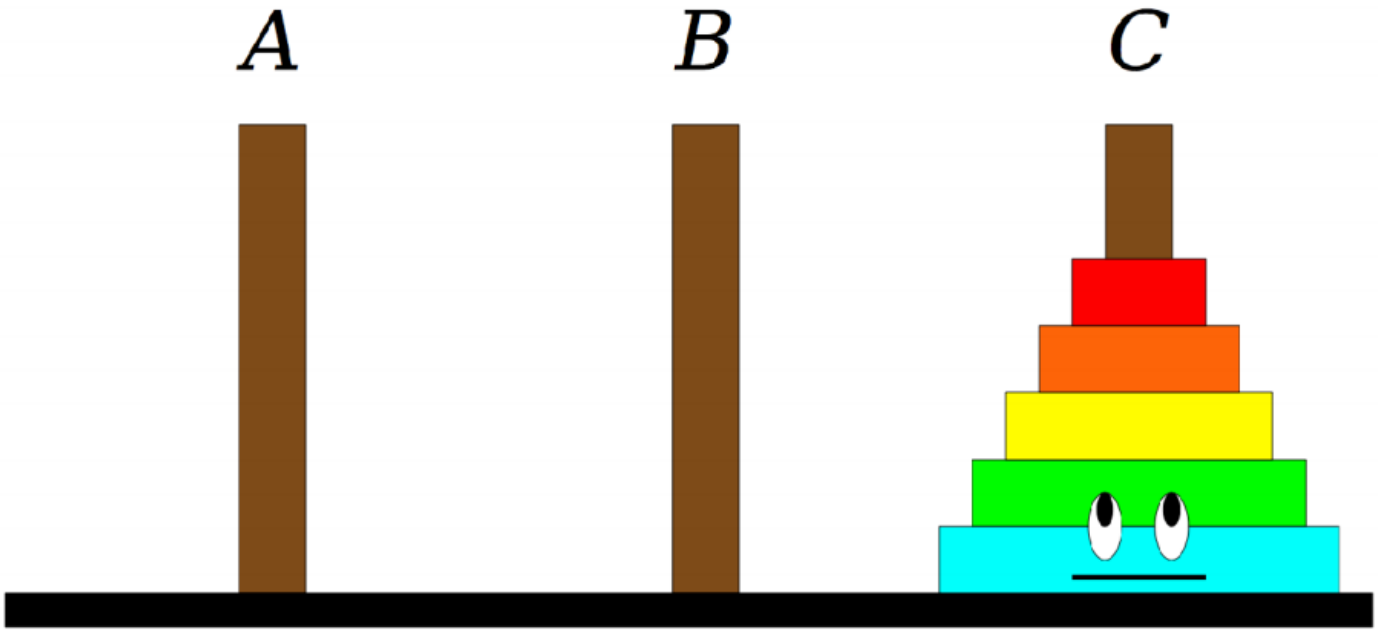
Ví dụ đệ quy nhị phân: Tháp Hà Nội

- Bước 1: Chuyển n – 1 đĩa từ cọc A sang cọc B dùng cọc C làm trung gian
- Bước 2: Chuyển đĩa to nhất từ cọc A sang cọc C



Ví dụ đệ quy nhị phân: Tháp Hà Nội

- Bước 1: Chuyển n – 1 đĩa từ cọc A sang cọc B dùng cọc C làm trung gian
- Bước 2: Chuyển đĩa to nhất từ cọc A sang cọc C
- Bước 3: Chuyển n – 1 đĩa từ cọc B sang cọc C dùng cọc A làm trung gian



Ví dụ đệ quy nhị phân: Tháp Hà Nội

```
void THN(int n, char from, char to, char temp){
    if (n > 0) {
        THN(n - 1, from, temp, to);
        move(from, to);
        THN(n - 1, temp, to, from);
    }
}
```

Đệ quy phi tuyến

```
KieuDuLieu TenHam(Thamso){
    if(Dieu Kien Dung)
    {
        ...;
        return Gia tri tra ve;
    }
    ...;
    vonglap(dieu kien lap){
        ...TenHam(Thamso)...;
    }
    return Gia tri tra ve;
}
```

- Là đệ quy mà lời gọi đệ quy được thực hiện bên trong vòng lặp

```
P ( ) {
    for (<giá tri đầu> to <giátrịcuối>) {
      thực hiện S ;
      if (điều kiện dừng) then thực hiện S*;
      else gọi P;
    }
}
```

với `S, S*` là các thao tác không đệ quy.

Đệ quy phi tuyến

- Ví dụ: Cho dãy { An } xác định theo công thức truy hồi:

$$A_0 = 1$$

$$A_n = n^2 A_0 + (n-1)^2 A_1 + \cdots + 2^2 A_{n-2} + 1^2 A_{n-1}$$

- Lời giải:

```
int A(int n) {
        if (n == 0) return 1 ;
        else {
                int temp = 0 ;
                for (int i = 0; i < n; i++)
                temp = temp + (n - i) * (n - i) * A(i);
                return temp;
        }
}
```

Đệ quy tương hỗ

- Là một loại đệ quy gián tiếp
- Trong đệ quy tương hỗ có 2 hàm, và trong thân của hàm này có lời gọi của hàm kia, điều kiện dừng và giá tri trả về của cả hai hàm có thể giống nhau hoặc khác nhau

```
KieuDuLieu TenHamX(Thamso){
    if(Dieu Kien Dung) {
        ...;
        return Gia tri tra ve;
    }
    ...;
    return TenHamX(Thamso) <Lien ket
hai ham> TenHamY(Thamso);
}

KieuDuLieu TenHamY(Thamso){
    if(Dieu Kien Dung){
        ...;
        return Gia tri tra ve;
    }
    ...;
    return TenHamY(Thamso)<Lien ket
hai ham> TenHamX(Thamso);
}
```

Đệ quy tương hỗ

- Ví dụ:

X(n) = 1, 2, 3, 5, 11, 41...

Y(n) = 1, 1, 2, 6, 30, 330…

```
void main() {
    int n;
    printf("\n Nhap n = ");
    scanf("%d",&n);
    printf( "\n  X = %d " ,X(n));
    printf( "\n Y = %d " , Y(n));
}
long Y(int n); //prototype cua ham y
long X(int n) {
    if(n == 0) return 1;
    else return X(n-1) + Y(n-1);
}
long Y(int n) {
    if(n == 0) return 1;
    else return X(n-1) * Y(n-1);
}
```

Đệ quy có nhớ và đệ quy quay lui

Đệ quy có nhớ

- Trong nhiều trường hợp, ví dụ các bài toán giải bằng quy hoạch động theo kiểu top-down, giải thuật đệ quy thường gọi tới cùng một trường hợp giống nhau nhiều lần.
- Khi đó, mỗi khi giải được một vấn đề con ta nên lưu lại lời giải và tái sử dụng kết quả khi vấn đề con đó được gọi tới trong các lần tiếp theo.
- Phương pháp này gọi là đệ quy có nhớ (memorization)

- Ví dụ: Tính số thứ n dãy Fibonacci

```
int fibo(int n) {
        if (n < 2) return 1;
        else return (fibo(n -1) + fibo(n -2));
}
```

Độ phức tạp: $O(2^n)$

Đệ quy có nhớ

```
int mem[1000];

int fibo(int n) {
    if (n <= 2) return 1;
    if (mem[n] != -1) return mem[n];
    int res = fibo(n - 2) + fibo(n - 1);
    mem[n] = res;
    return res;
}

int main(){
    memset(mem, - 1, sizeof(mem));
    cout << fibo(10);
    return 0;
}
```

Độ phức tạp: O(n)

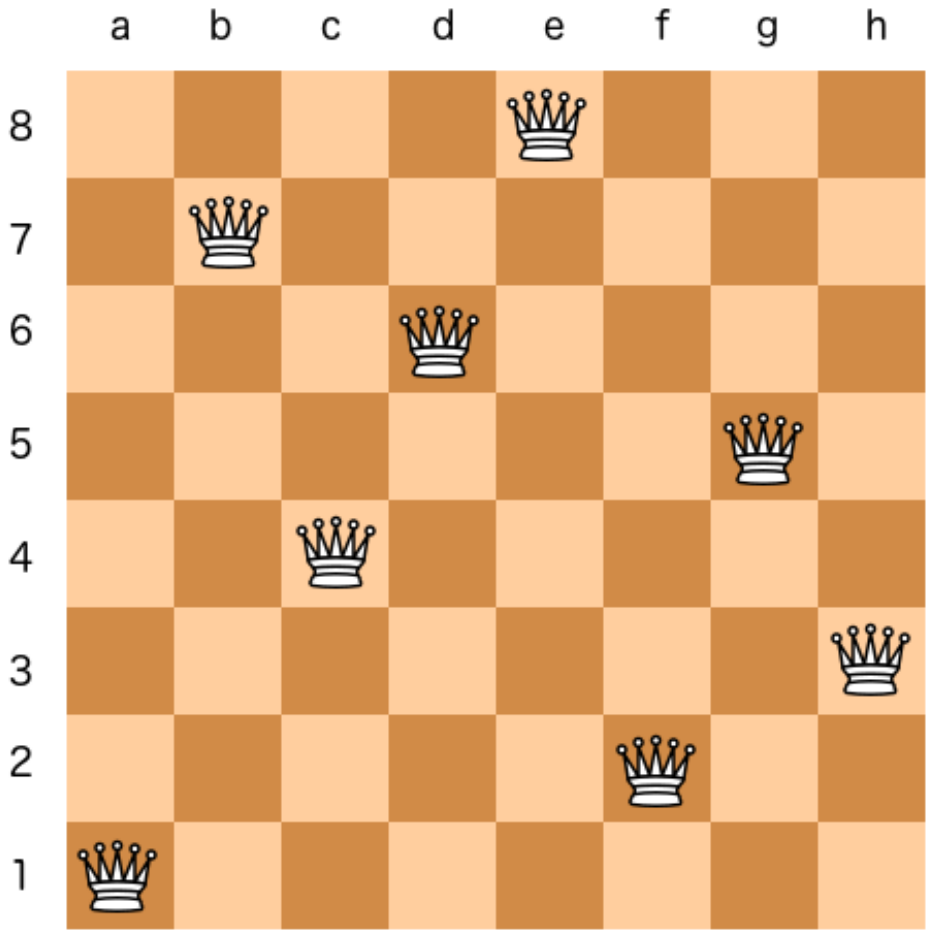
Đệ quy quay lui

- Quay lui là kỹ thuật giải quyết vấn đề bắt đầu từ lời giải rỗng và xây dựng dần lời giải bộ phận (partial solution) để ngày càng tiến gần tới lời giải bài toán.
  - Nếu một lời giải bộ phận không thể tiếp tục phát triển, ta sẽ bỏ nó
  - Và quay sang xét tiếp các ứng cử viên khác

```
void Try(int i) {
    foreach (ung vien duoc chap nhan C) {
        <update cac bien trang thai>
        <ghi nhan x[i] moi theo C>
        if (i == n) <ghi nhan mot loi giai>
        else Try(i + 1);
        <tra cac bien ve trang thai cu>
    }
}
```

Đệ quy quay lui

- Ví dụ bài toán n-Queens



```cpp
#include <bits/stdc++.h>
using namespace std;
const int NMAX = 20;
int a[NMAX], n;

void print_sol(){
    for(int i = 1; i <= n; i++)
        cout << a[i] << (i == n ? '\n' : ' ');
}

bool isCandidate(int j, int k){
    for(int i = 1; i < k; i++)
        if ((j == a[i]) || (fabs(j - a[i]) == (k - i)))
            return false;
    return true;
}

void TRY(int k){
    for(int j = 1; j <= n; j++){
        if (isCandidate(j, k)){
            a[k] = j;
            if (k == n) print_sol();
            else TRY(k + 1);
        }
    }
}

int main(){
    n = 4;
    TRY(1);
    return 0;
}
```

Khử đệ quy

Ưu nhược điểm đệ quy

- **Ưu điểm:**
  - Trong nhiều trường hợp, giải bằng đệ quy trực quan hơn vì nó mô phỏng (mimic) chúng ta giải quyết vấn đề
  - Lập trình đơn giản, dễ hiểu, dễ bảo trì
  - Một số CTDL như cây dễ dàng duyệt hơn bằng đệ quy
- **Nhược điểm:**
  - Tốn không gian nhớ và thời gian xử lý
  - Đệ quy chạy chậm và có thể gây tràn bộ nhớ stack

Khử đệ quy

- Mọi lời giải đệ quy đều có thể viết dưới dạng kiểu vòng lặp: đó chính là cách CPU cuối cùng sẽ làm để thực hiện lời gọi đệ quy. Bản chất lời gọi đệ quy là là đặt các lệnh gọi hàm trong một ngăn xếp stack.
- Việc thay thế một lời giải đệ quy bằng một lời giải không đệ quy tương đương gọi là **khử đệ quy**
- Khử để quy không phải bao giờ cũng đơn giản. Khử đệ quy có thể cần nhiều thao tác và làm cho mã chương trình khó bảo trì hơn
- Khi tối ưu chương trình, chỉ khử đệ quy khi profiling hoặc có bằng chứng rõ ràng chỉ ra rằng việc đó là cần thiết

Khử đệ quy bằng vòng lặp

- Công thức truy hồi:

f(n) = C nếu n = n_0 (C là một hằng số)

f(n) = g(n,f(n -1)) nếu n > n_0

- Giải thuật đệ quy tính f(n)

```
f(n) ≡ if(n == no) return C;
else return (g(n,f(n -1));
```

- Giải thuật lặp tính f(n)

```
K = no; F:= C;
F = f(no);
while(k < n){
     k++;
     F = g(k,F);
}
return F;
```

Khử đệ quy bằng vòng lặp

- Khử đệ quy hàm tính giai thừa

```
int fact( int n ) {
    int k = 0;
    int F = 1;
    while (k < n) F = ++k * F;
    return F;
}
```

- Tính tổng n số đan dấu: $S_n = 1 - 3 + 5 - \cdots + (-1)^{n+1}(2n-1)$

$S_n = 1$ nếu $n = 1$

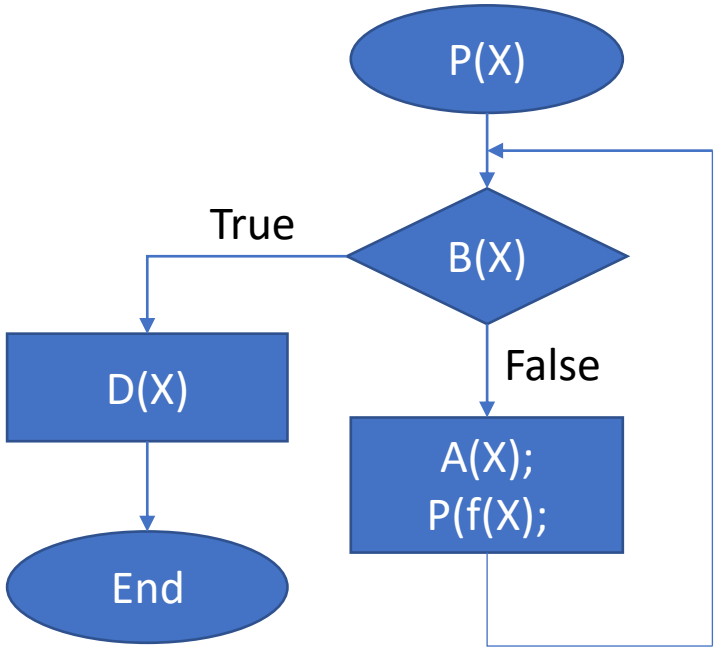
$S_n = S_{n-1} + (-1)^{n+1}(2n-1)$ nếu $n > 1$

```
int S(int n) {
    int k = 1, temp = 1 ;
    while (k < n) {
        k++;
        if (k%2 == 1)  temp += 2*k -1;
        else temp -= 2*k + 1;
    }
    return temp;
}
```

Khử đệ quy đuôi

- Xét thủ tục P dạng

```
P(X) ≡ if  B(X) then D(X)
       else {
            A(X) ;
            P(f(X)) ;
       }
```



- Trong đó:

X là tập biến (một hoặc một bộ nhiều biến)

P(X) là thủ tục đệ quy phụ thuộc X

A(X); D(X) là các thao tác không đệ quy

f(X) là hàm biến đổi X

Khử đệ quy đuôi

- Xét quá trình thực hiện P(X) :

gọi P_0 là lần gọi P thứ 0 (đầu tiên) P(X)

P_1 là lần gọi P thứ 1 (lần 2) P(f(X))

P_i là lần gọi P thứ i (lần i + 1) P(f(f(...f(X)...)

($P(f_i(X))$ hợp i lần hàm f)

- Xét lời gọi P_i, nếu $B(f_i(X))$

(false) { A và gọi P_{i+1}}

(true) { D }

- Giả sử P được gọi đúng n +1 lần . Khi đó ở trong lần gọi cuối cùng (thứ n) P_n thì $B(f_n(X))$ = true, lệnh D được thực hiện và chấm dứt thao tác gọi thủ tục P

Khử đệ quy đuôi

- Giải thuật vòng lặp:

```
while (!B(X)) {
      A(X);
      X = f(X);
}
D(X);
```

Ví dụ: Tìm USCLN

- Giải thuật đệ quy:

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

- X là (m , n)
- P(X) là USCLN(m ,n)
- B(X) là n == 0
- D(X) là lệnh return m
- A(X) là lệnh rỗng
- f(X) là f(m,n) = (n , m mod n)

- Giải thuật vòng lặp:

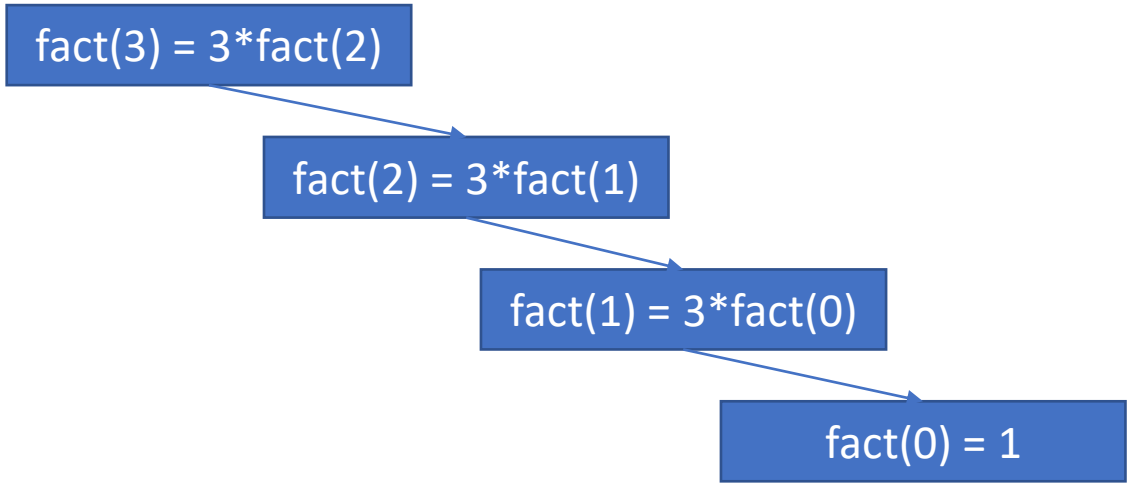
```
int USCLN(int m , int n ) {
    while(n != 0) { // !B(x)
        m = n; // X= f(X)
        n = m % n; // X= f(X)
    }
    return m; // D(x)
}
```

Khử đệ quy bằng stack

- Trạng thái của tiến trình xử lý một giải thuật: nội dung các biến và lệnh cần thực hiện kế tiếp.
- Với tiến trình xử lý một giải thuật đệ quy ở từng thời điểm thực hiện, cần lưu trữ cả các trạng thái xử lý đang còn dang dở
- Xét giải thuật tính giai thừa:

```
fact(n) ≡ if(n = 0) then return
else return (n * fact(n - 1));
```

- Sơ đồ thực hiện:

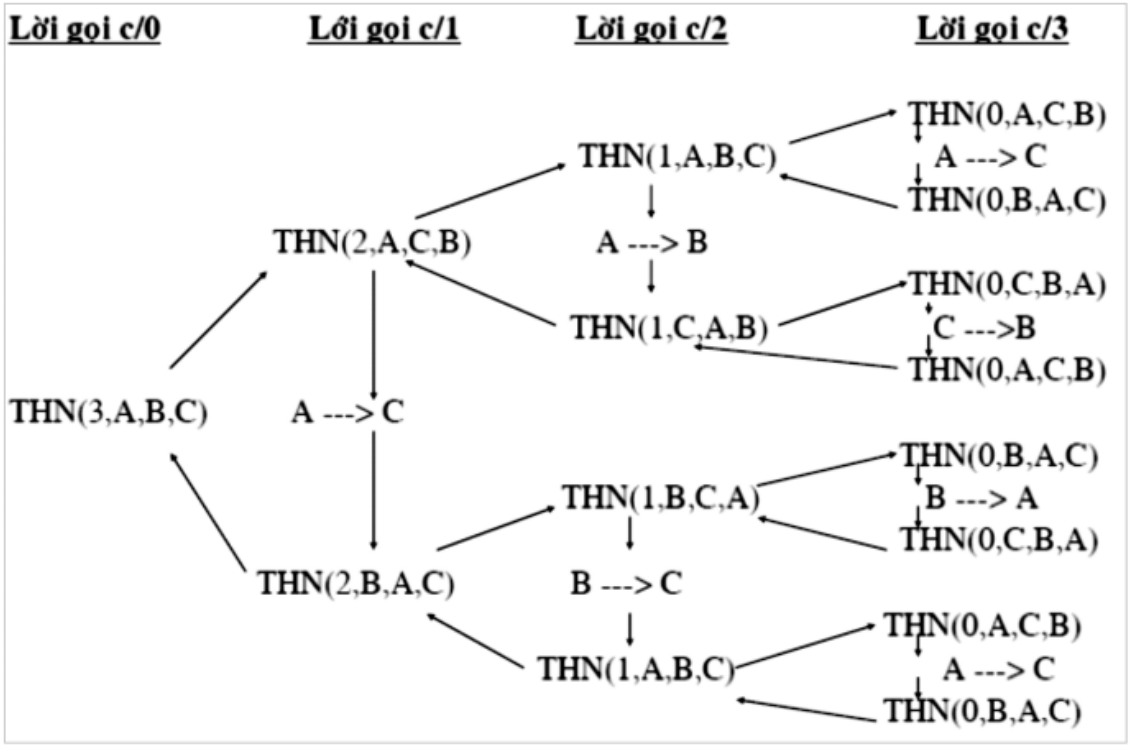


Khử đệ quy bằng stack

- Thủ tục đệ quy tháp Hà Nội

```
THN(int n, char A, char B, char C) ≡ {
if (n > 0 ) then { THN(n-1, A ,C ,B);
Move(A, C); THN(n-1,B,A,C);} }
```

- Sơ đồ thực hiện THN(3, A, B, C):



Khử đệ quy bằng stack

- Lời gọi đệ quy sinh ra lời gọi đệ quy mới cho đến khi gặp trường hợp cơ bản
- Ở mỗi lần gọi, phải lưu trữ thông tin trạng thái con dang dở của tiến trình xử lý ở thời điểm gọi. Số trạng thái này bằng số lần gọi chưa được hoàn tất.
- Khi thực hiện xong (hoàn tất) một lần gọi, cần khôi phục lại toàn bộ thông tin trạng thái trước khi gọi .
- Lệnh gọi cuối cùng (ứng với trương hợp cơ bản) sẽ được hoàn tất đầu tiên
- Cấu trúc dữ liệu cho phép lưu trữ dãy thông tin thỏa 3 yêu cầu trên là cấu trúc lưu trữ thỏa mãn LIFO (Last In First Out ~ Cấu trúc Stack)

Khử đệ quy tuyến tính bằng stack

- Xét đệ quy tuyến tính dạng sau:

```
P(X) ≡ if C(X) then D(X)
       else begin
            A(X) ;
            P(f(X)) ;
            B(X) ;
       end;
```

trong đó:

- X là một biến đơn hoặc nhiều biến
- C(X) là một biểu thức boolean của X
- A(X) , B(X) , D(X): không đệ quy
- f(X) là hàm của X

Khử đệ quy tuyến tính bằng stack

- Khử đệ quy thực hiện P(X) bằng stack:

```
P(X) ≡ {
    create_stack(S); // tạo stack S
    while(not(C(X)){
        A(X);
        push(S,X); // cất giá trị X vào stack S
        X := f(X);
    }
    D(X);
    while(not(empty(S))){
        pop(S,X); // lấy dữ liệu từ S
        B(X);
    }
}
```

Ví dụ: Chuyển từ cơ số thập phân sang nhị phân

- Đệ quy:

```
void binary(int m) {
    if (m > 0) {
        binary(m / 2);
        cout << m % 2;
    }
}
```

- X là m
- P(X) là binary(m)
- A(X) ; D(X) là lệnh rỗng
- B(X) là lệnh cout << m % 2 ;
- C(X) là (m <= 0) .
- f(X) là f(m) = m / 2

- Khử đệ quy:

```
void iter_binary(int m){
    stack<int> stk;
    while(m > 0){ // !(C(X))
        // A(X) empty
        stk.push(m); // push(S,X)
        m /= 2; // X = f(X)
    }
    // D(X) empty
    while(!stk.empty()){
        int u = stk.top();
        stk.pop();
        cout << u % 2; // B(X)
    }
}
```

Khử đệ quy nhị phân

- Xét đệ quy nhị phân dạng sau:

```
P(X) ≡ if C(X) then D(X)
       else begin
            A(X); P(f(X));
            B(X); P(g(X));
       end;
```

Trong đó:

X là một biến đơn hoặc nhiều biến

C(X) là một biểu thức boolean của X

A(X) , B(X) , D(X): không đệ quy

f(X), g(X) là các hàm của X

Khử đệ quy tuyến tính bằng stack

- Khử đệ quy thực hiện P(X) bằng stack:

```
P(X) ≡ {
     create_stack (S);
     push(S, (X,1));
     while (k != 1) {
          while (not C(X)){
               A(X);
               push (S, (X,2));
               X := f(X);
          }
          D(X) ;
          pop(S, (X,k));
          if (k != 1) {
               B(X);
               X := g(X);
          }
     }
}
```

Ví dụ: Tháp Hà Nội

- Đệ quy:

```
void move(char A, char C){
    cout << A << " -> " << C << endl;
}

void THN(int n, char A, char B, char C){
    if (n > 0) {
        THN(n - 1, A, C, B);
        move(A, C);
        THN(n - 1, B, A, C);
    }
}
```

- Biến X là bộ (n, A, B, C)
- C(X) là n <= 0
- D(X) và A(X) là rỗng
- B(X) = B(n,A,B,C) là move(A, C)
- f(X) = f(n,A,B,C) = (n-1,A,C,B)
- g(X) = g(n,A,B,C) = (n-1,B,A,C)

Ví dụ: Tháp Hà Nội

- Khử đệ quy:

```
struct state{
    int n, k;
    char A, B, C;
    state(int _n, char _A, char _B, char _C, int _k):
        n(_n), A(_A), B(_B), C(_C), k(_k) {}
};

void iter_THN(int n, char A, char B, char C){
    stack<state> stk;
    stk.push(state(n, A, B, C, 1));
    int k = 0;
    while (k != 1) {
        while (n != 1){
            stk.push(state(n, A, B, C, 2)); // push (S, (X, 2))
            n--;                            // X = f(X)
            std::swap(B, C);                // X = f(X)
        }
        move(A, C);                         // D(X)
        state s = stk.top();                // pop (S, (X, k))
        stk.pop();
        tie(n, A, B, C, k) = std::make_tuple(s.n, s.A, s.B, s.C, s.k);
        if (k != 1){
            move(A, C);                     // B(X)
            n--;                            // X = g(X)
            std::swap(A, B);                // X = g(X)
        }
    }
}
```

ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Xin cảm ơn!