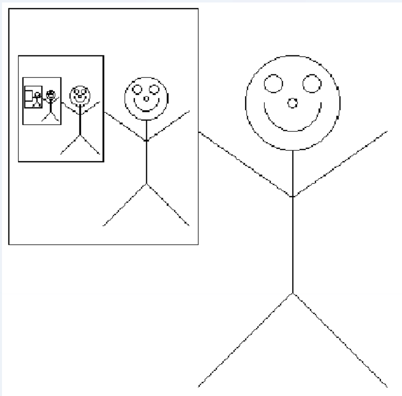
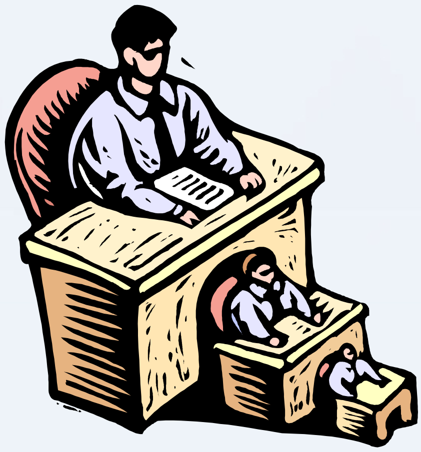
Bài 5. Đệ qui (Recursion)





Đệ qui trong thực tế (Recursion in practice)

- Hệ điều hành: Các thư mục
- Cú pháp của ngôn ngữ lập trình (Syntax of languages)
- Đồ họa máy tính (Computer Graphics)
- Tự nhiên: cây cối

Một cuộc hành trình 1000 bước và việc thực hiện hành trình bắt đầu ở bước thứ nhất.

- Làm thế nào thế nào để hoàn thành cuộc hành trình này?

- Thực hiện bước 1 và tạo ra cuộc hành trình mới có 999 bước.

Hàm (phương thức) đệ qui

- **Đệ qui**: Khi một hàm gọi đến chính nó
- Ví dụ tính giai thừa:
  $n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1) \cdot n$

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

- Hàm trong C++

```
// hàm đệ qui tính giai thừa
int recursiveFactorial(int n) {
   if (n == 0) return 1;                // trường hợp cơ sở
   else return n * recursiveFactorial(n- 1);
}
```

Đệ qui tuyến tính – Đệ qui 1 lần

- **Kiểm tra trường hợp cơ sở.**
  - Bắt đầu bằng việc kiểm tra các trường hợp cơ sở (ở đó phải có ít nhất một trường hợp). Đây chính là điều kiện để kết thúc đệ qui.
  - Các lời gọi đệ qui hàm phải thực sự hướng quá trình đệ qui về trường hợp cơ sở (để kết thúc đệ qui).
- **Đệ qui một lần.**
  - Thực hiện gọi đệ qui chỉ một lần trong hàm. (Có thể trong hàm có nhiều bước kiểm tra để quyết định lựa chọn lời gọi đệ qui, nhưng trong tất cả các trường hợp đó thì chỉ một trường hợp được gọi thực sự)
  - Khi định nghĩa hàm đệ qui thì mỗi lần gọi đệ qui trong hàm phải dẫn dần về trường hợp cơ sở.

Ví dụ 1:Cộng các phần tử của một mảng

Cho mảng *A có n phần tử*

4	3	6	2	5

Ví dụ đơn giản cho đệ qui tuyến tính

Algorithm LinearSum(*A, n*):

Input:

Một mảng A có kiểu nguyên và số nguyên n ≥ 1, A có ít nhất n phần tử

Output:

Tổng của n số nguyên đầu tiên trong A

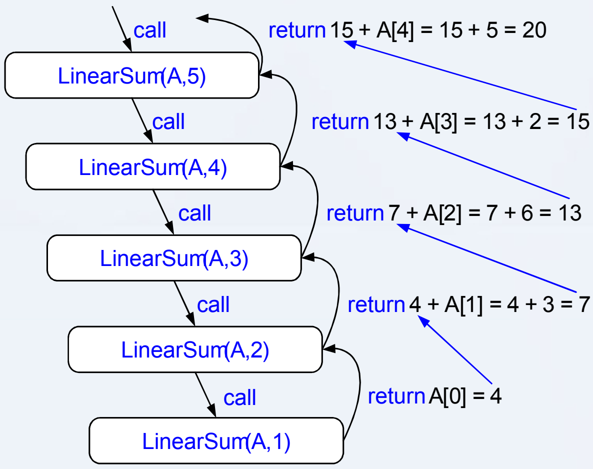
if *n* = 1 **then**

return *A*[0]

else

return LinearSum(*A, n* - 1) + *A*[*n* - 1]

Ví dụ vết đệ qui:



Ví dụ 2:Đảo ngược một mảng

Algorithm ReverseArray(*A, i, j*):

Input: Một mảng A và 2 chỉ số i, j nguyên không âm

Output: Đảo ngược mảng A từ chỉ số i đến j

if *i* < *j* **then**

Swap *A*[*i*] and *A*[*j*]

ReverseArray(*A, i* + 1, *j* - 1)

return

Định nghĩa các đối cho hàm đệ qui

- Việc tạo ra các đối cho các hàm đệ qui là rất quan trọng, nó làm cho việc xây dựng hàm đệ qui trở nên dễ dàng hơn.
- Trong một số trường hợp ta cần bổ sung thêm cho các hàm một số đối, khi đó dẫn tới hàm có thể gọi đệ qui.
- Ví dụ, chúng ta định nghĩa hàm đảo mảng như sau ***ReverseArray(A, i, j),***

  không định nghĩa *ReverseArray(A)*.

Cách tính số mũ

$$x^n x^m = x^{n+m}$$

Nếu *n* chẵn

$$x^n = x^{n/2} x^{n/2} = (x^{n/2})^2$$

Nếu *n lẻ*

$$x^n = x(x^{(n-1)/2})^2$$

Tính lũy thừa

- Hàm tính lũy thừa, $p(x,n)=x^n$, có thể định nghĩa đệ qui như sau:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{else} \end{cases}$$

- Với cách định nghĩa như trên dẫn đến hàm tính lũy thừa có thời gian chạy là O(n) (gọi đệ qui n lần).
- Tuy nhiên chúng ta có thể tính lũy thừa bằng cách khác tốt hơn cách trên.

Đệ qui bậc 2

- Chúng ta có thể đưa ra một thuật toán hiệu quả hơn với thuật toán đệ qui tuyến tính bằng việc sử dụng thuật toán đệ qui bậc 2.

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } n > 0 \text{ is odd} \\ p(x, n/2)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

$2^4 = 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$

$2^5 = 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32$

$2^6 = 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$

$2^7 = 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128.$

Hàm đệ qui bậc 2

Algorithm Power(*x*, *n*):

Input: một số x và số nguyên n ≥ 0

Output: Giá trị của x^n

if *n* = 0 **then**

  return 1

if *n là lẻ* **then**

  y = Power(*x*, (*n* - 1)/ 2)

  return *x* · *y* ·*y*

else

  y = Power(*x*, *n*/ 2)

  return *y* · *y*

Phân tích thuật toán đệ qui bậc 2

Algorithm Power(*x*, *n*):

Input: *một số x số nguyên n* ≥ 0

Output: *Giá trị* x^n

if *n* = 0 **then**

  return 1

if *n là lẻ* **then**

  y = Power(*x*, (*n* - 1)/ 2)

  return *x* · *y* · *y*

else

  y = Power(*x*, *n*/ 2)

  return *y* · *y*

Mỗi lần gọi đệ qui thì giá giá trị của n được chia đôi; do đó ta đã phải gọi đệ qui logn. Vậy thời gian thực hiện của thuật toán là O(logn).

Ở đây ta sử dụng biến y, nó rất quan trọng vì nó giúp ta tránh phải gọi đệ qui hai lần.

Logarit

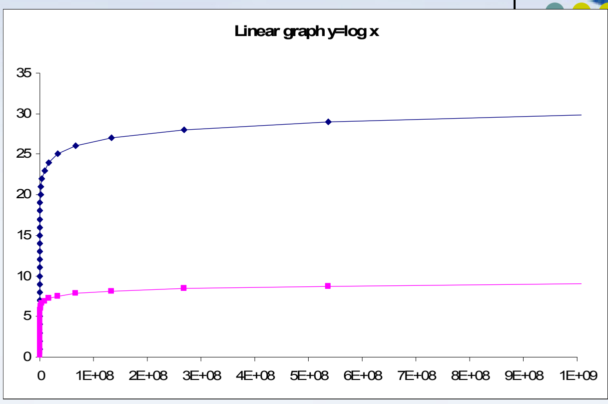
- Đây là một ví dụ rất tốt để nói đến log nói chung
- Phương pháp đệ qui bậc 2 có thời gian chạy là logn.
  - Cơ số của log ở trên là gì? 2.
  - Tại sao mỗi bước lại chia cho 2?
  - Nếu n=1000, Số bước là bao nhiêu? 10
  - Nếu chúng ta có thuật toán chạy trong thời gian là $\log_{10}$. Thuật toán này có thực sự khác với thuật toán trên hay không?

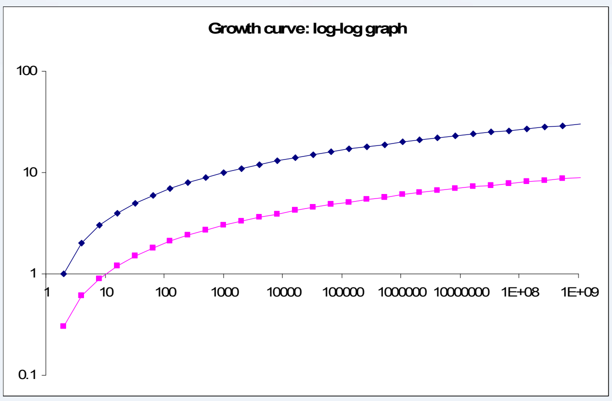
Với log cơ số nhỏ hay lớn chỉ khác nhau hằng số.Vì:

$$\log_a n = \log_b a * \log_b n$$
$$\log_{10} n = \log_2 10 * \log_2 n$$

Mối quan hệ giữa $\log_2$ and $\log_{10}$?

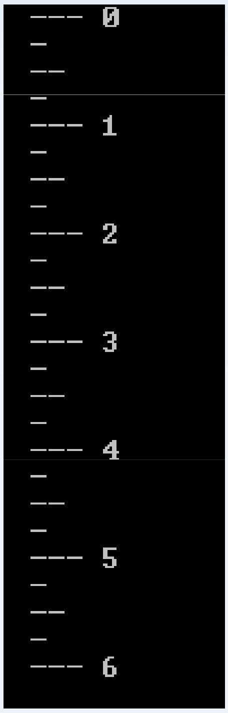
n	log(n,2)	log(n,10)
1	0	0
2	1	0.30
4	2	0.60
8	3	0.90
16	4	1.20
32	5	1.51
64	6	1.81
128	7	2.11
256	8	2.41
512	9	2.71
1024	10	3.01





Đệ qui nhị phân (Binary Recursion)

- **Hàm đệ qui nhị phân là hàm đệ qui mà trong nó gọi đệ qui hai lần.**
- Ví dụ: Hàm vẽ một cái thước kẻ.



Hàm đệ qui 2 lần để vẽ một cái thước kẻ

```
#include <conio.h>
#include <iostream.h>
using namespace std;

//Hàm vẽ một vạch trên thước
void drawonetick(int ticklength, int ticklabel=-1){
    cout<<"    ";
    for(int i=0;i<ticklength; i++)
          cout<<"-";
    if(ticklabel>=0)
            cout<<" "<<ticklabel;
    cout<<"\n";
}
```

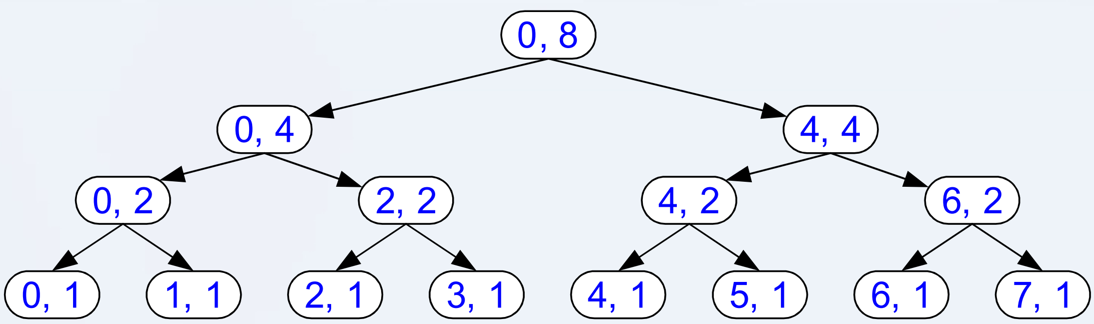
```
//Hàm vẽ một đơn vị của thước
void drawticks(int ticklength){
    if(ticklength>0){
        drawticks(ticklength-1);
        drawonetick(ticklength);
        drawticks(ticklength-1);
    }
}
//Hàm vẽ cả thước
void drawruler(int ninches, int majorlength){
    drawonetick(majorlength,0);
    for(int i=1; i<= ninches; i++){
        drawticks(majorlength-1);
      drawonetick(majorlength,i);
    }
}
```

```
void main(){
    drawruler(6,3);
    getch();
}
```


Một hàm đệ qui nhị phân khác

- **Bài toán: Cộng tất cả các số của môt mảng A các số nguyên:**
  Algorithm *BinarySum(A, i, n):*
      Input: *Mảng A và hai số nguyên i và n, trong đó n = 2 mũ k (k>0)*
      Output: *Tính tổng n số của mảng A có chỉ số bắt đầu từ i*
     if *n = 1* ***then***
        return *A[i]*
     return *BinarySum(A, i, n/ 2) + BinarySum(A, i + n/ 2, n/ 2)*

- **Ví dụ vết của thuật toán:**



Số tiếp theo là?

1

1

2

3

5

8

13

?

Tính số Fibonacci

- Các số Fibonacci được định nghĩa như sau :

  $F_0 = 1$

  $F_1 = 1$

  $F_i = F_{i-1} + F_{i-2}$ với $i > 1$.

- Thuật toán tìm số Fibonaci thứ k

  Algorithm BinaryFib(k):

  Input: **Số nguyên không âm** k

  Output: **Số Fibonaci thứ k là** F_k

  if $k \le 1$ **then**

      return *1*

  else

      return BinaryFib(k - 1) + BinaryFib(k - 2)

Phân tích thuật toán Fibonacci đệ qui nhị phân

- Gọi n_k là số lần gọi đệ qui của BinaryFib(k). Khi đó
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67.$
- **Chú ý:** Trừ 2 trường hợp đầu của n_k còn lại thì $n_k > 2^{k/2}$. Vậy nó là hàm mũ!

Thuật toán tính số Fibonacci tốt hơn

- **Thuật toán đệ qui một lần:**

  Algorithm LinearFibonacci(k):

  Input: Một số nguyên không âm k

  Output: Cặp hai số số Fibonacci (F_k, F_{k-1})

  if k = 1 **then**

  return (k, 1)

  else

  (i, j) = LinearFibonacci(k - 1)

  return (i +j, i)

- **Thời gian chạy là O(*k*).**

Bài tập

- Bài 1. Lập hàm đệ qui tính giá trị đa thức
- Bài 2. Lập hàm đệ qui tìm ước số chung của 2 số nguyên dương
- Bài 3. Lập hàm đệ qui tìm giá trị min của một dãy n số thực
- Bài 4. Viết hàm đệ qui tìm kiếm một chữ cái nào đó có trong một xâu ký tự hay không.

Hết