# Phân tích thuật toán

## Dãy số Fibonacci

Là dãy số với mỗi số là tổng của hai số trước đó

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ \ldots$$

Một cách hình thức, các số Fibonacci $F_n$ được định nghĩa bởi luật đơn giản:

$$F_n = \begin{cases} 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0. \end{cases}$$

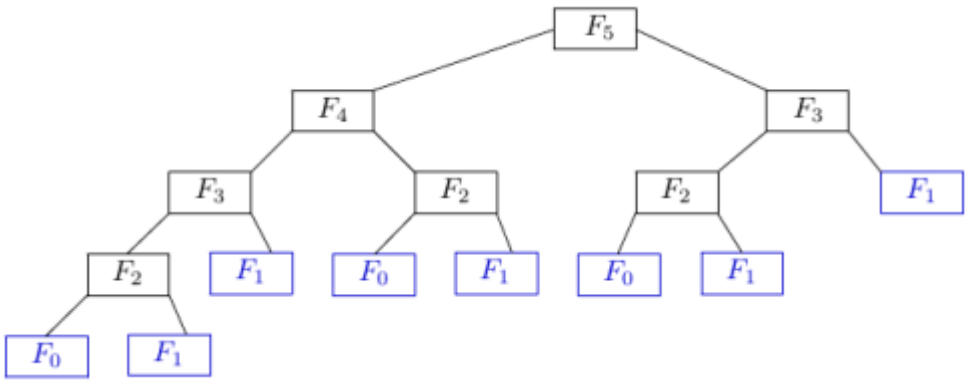
Các số Fibonacci tăng gần như lũy thừa của $2$. Ví dụ, $F_{30}$ lớn hơn một triệu, $F_{100}$ có khoảng $21$ chữ số.

**Câu hỏi:** Một cách tổng quát $F_n \approx 2^{0.694n}$. Tại sao?

Từ định nghĩa, ta dễ dàng đi tới cài đặt đệ quy như dưới đây:

```
long fib1 (int n){
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib1 (n-1) + fib1 (n-2);
}
```

Hình dưới đây mô tả các lời gọi đệ quy khi tính giá trị $F_5$.




**Câu hỏi:** Thuật toán này chạy trong thời gian bao lâu, tính thời gian như một hàm $T(n)$ theo $n$? Và liệu ta có thể tính nhanh hơn?

**Trả lời:** Thời gian chạy của thuật toán là

$$T(n) \geq F_n$$

Điều này có thể chứng minh dùng quy nạp. Thuật toán này thực sự tồi tệ vì $T(200) \geq F_{200} \geq 2^{138}$.

Lý do mà thuật toán này chậm là do ta phải tính lại nhiều lần cùng một giá trị. Như trong Hình trên, các giá trị $F_i$ phải tính lại rất nhiều lần. Số lần được thể hiện trong bảng sau:

| Hàm | phải tính lại |
|---|---|
| $F_5$ | 1 lần |
| $F_4$ | 1 lần |
| $F_3$ | 2 lần |
| $F_2$ | 3 lần |
| $F_1$ | 5 lần |
| $F_0$ | 3 lần |

**Bài tập:** Hãy sửa lại đoạn mã của `fib1` để in ra màn hình số lần các giá trị $F_i$ phải tính lại.

# Thuật toán nhanh hơn

Ta có thể thiết kế thuật toán nhanh hơn bằng cách tránh việc gọi lại. Cụ thể, ta sử dụng một mảng `f[MAX]`, với `f[i]` lưu lại các giá trị $F_i$ đã tính toán để không cần tính lại. Thuật toán cài đặt như sau:

```
long fib2 (int n){
    int f[MAX];
    if (n == 0) return 0;
    f[0] = 0;  f[1] = 1;
    for (int i = 1; i <= n; i++)
        f[i] = f[i-1] + f[i-2];    //f[i] đã tính toán từ bước trước
    return f[n];
}
```

Một cải tiến nhỏ của thuật toán trên giúp ta tiết kiệm bộ nhớ: Ta không cần tạo ra một mảng `f[MAX]` để lưu các giá trị trung gian, thay vào đó ta dùng biến `f` thay cho các `f[i-1]` , và biến `g` thay cho các `f[i]`.

```
long fib2 (int n){
    int f, g;
    int tmp;
    if (n == 0) return 0;
    f = 0;  g = 1;
    for (int i = 1; i <= n; i++) {
        tmp = g;
        g = f + g;
        f = tmp;
    }
    return f;
}
```

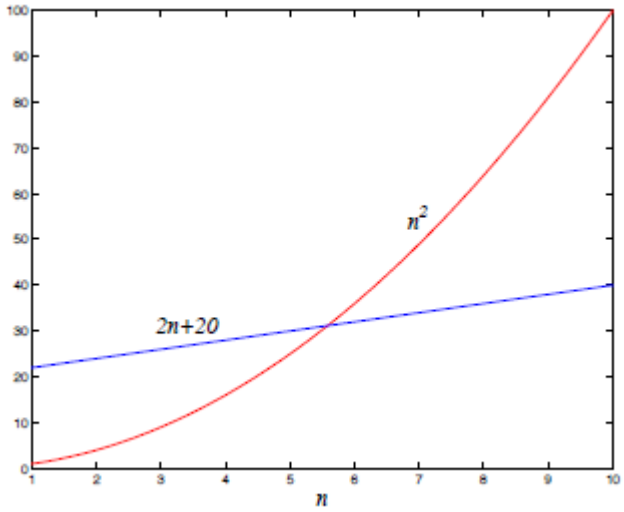
**Câu hỏi:** Thuật toán này chạy trong thời gian bao lâu, tính thời gian như một hàm $T(n)$ theo $n$?

**Trả lời:** Khi $n \geq 1$, thời gian chạy của thuật toán là $T(n) = 3n$.

# Ký hiệu $O$-lớn

**Định nghĩa:** Xét hai hàm $f, g : \mathbb{N}^+ \to \mathbb{N}^+$. Ta nói rằng $f = O(g)$ (có nghĩa rằng $f$ *tăng không nhanh hơn* $g$) nếu có một hằng số $c > 0$ sao cho $f(n) \leq c \cdot g(n)$.

Viết $f = O(g)$ rất giống với viết $f \leq g$. Nó chỉ khác ở hằng số $c$. Ví dụ, $10n = O(n)$. Hằng số này giúp chúng ta không quá tập trung vào các giá trị $n$ nhỏ. Một chương trình chạy trong $f_1(n) = n^2$ bước, trong khi chương trình khác chạy trong $f_2(n) = 2n + 20$ bước (Hình dưới đây). Chương trình nào chạy nhanh hơn? Khi $n \leq 5$, rõ ràng $f_1$ tốt hơn; tuy nhiên khi giá trị $n$ tăng dần lên, rõ ràng $f_2$ tăng nhanh hơn.




Ta có thể thấy điều này bởi ký hiệu $O$-lớn: $f_2 = O(f_1)$, bởi vì

$$\frac{f_2(n)}{f_1(n)} = \frac{2n + 20}{n^2} \leq 22$$

với mọi $n$; mặt khác $f_1 \neq O(f_2)$, vì tỉ lệ $f_1(n)/f_2(n) = n^2(2n + 20)$ có thể lớn tùy ý khi $n$ tăng lên.

**Câu hỏi:**

- Đúng hay sai: $10n = O(n)$ ?
- Xét $f_1(n) = n^2$ và $f_2(n) = 2n + 20$. Vậy thì
$$f_1 = O(f_2) \quad \text{hay} \quad f_2 = O(f_1) \quad ?$$
- Xét $f_2(n) = 2n + 20$ và $f_3(n) = n + 1$. Vậy thì
$$f_2 = O(f_3) \quad \text{hay} \quad f_3 = O(f_2) \quad ?$$

## Một số luật cho $O$ lớn

- Bỏ qua phép nhân với hằng số. Ví dụ, $14n^2$ trở thành $n^2$
- $n^a$ sẽ bị *lấn át* bởi $n^b$ nếu $a > b$. Ví dụ, $n^2$ sẽ *lấn át* $n$
- Hàm mũ sẽ lấn át hàm đa thức. Ví dụ, $3^n$ lấn át $n^5$.
- Tương tự, đa thức sẽ lấn át hàm logarit. Ví dụ, $n$ lấn át $(\log n)^3$.

# Ký hiệu $\Omega$ và $\Theta$

Xem $O(.)$ tương tự như $\leq$ ta có thể định nghĩa ký hiệu tương tự như ký hiệu $\geq$ và $=$ như sau đây:

$$
\begin{aligned}
f = \Omega(g) \quad &\iff \quad g = O(f) \\
f = \Theta(g) \quad &\iff \quad f = O(g) \text{ và } f = \Omega(g).
\end{aligned}
$$

**Bài tập:** Hãy chỉ ra xem liệu $f = O(g)$ hay $f = \Omega(g)$ hay cả hai $f = \Theta(g)$.

| $f(n)$ | $g(n)$ | $O$ **hay** $\Omega$ **hay** $\Theta$ |
|---|---|---|
| $n - 100$ | $n - 200$ | |
| $n \log n$ | $10n \log 10n$ | |
| $\sqrt{n}$ | $(\log n)^3$ | |
| $n2^n$ | $3^n$ | |
| $n!$ | $2^n$ | |
| $2^n$ | $2^{n+1}$ | |
| $(\log n)^{\log n}$ | $2^{(\log_2 n)^2}$ | |

**Bài tập:** Chứng minh rằng nếu $c$ là một số thực dương thì $g(n) = 1 + c + c^2 + \cdots + c^n$ là

- $\Theta(1)$ nếu $c < 1$,
- $\Theta(n)$ nếu $c = 1$, và
- $\Theta(c^n)$ nếu $c > 1$.

# Quay trở lại với số Fibonacci

**Bài tập:** Trong bài tập này ta sẽ chứng minh rằng dãy Fibonacci tăng nhanh như hàm mũ.

1. Dùng quy nạp để chứng minh rằng $F_n \geq 2^{0.5n}$ với $n \geq 6$.
2. Tìm hằng số $c < 1$ sao cho $F_n \leq 2^{cn}$ với mọi $n \geq 0$. Chứng minh câu trả lời của bạn.
3. Tìm giá trị $c$ lớn nhất sao cho $F_n = \Omega(2^{cn})$?

**Bài tập:** Liệu có cách tính số Fibonacci thứ $n$ nhanh hơn hàm `fib2` ? Ý tưởng tính toán sau đây liên quan đến ma trận.

Chúng ta bắt đầu bằng cách viết phương trình $F_1 = F_1$ và $F_2 = F_0 + F_1$ theo ký hiệu ma trận:

$$
\begin{bmatrix} F_1 \\ F_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} F_0 \\ F_1 \end{bmatrix}
$$

Tương tự,

$$
\begin{bmatrix} F_2 \\ F_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} F_1 \\ F_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^2 \cdot \begin{bmatrix} F_0 \\ F_1 \end{bmatrix}
$$

và tổng quát ta được

$$
\begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} F_0 \\ F_1 \end{bmatrix}
$$

Vậy để tính $F_n$, ta chỉ cần lấy ma trận $2 \times 2$ này, ta gọi nó là $X$, và tính $X^n$.

1. Chứng minh rằng để nhân 2 ma trận $2 \times 2$ ta chỉ cần dùng $4$ phép cộng và $8$ phép nhân.
2. Chứng minh rằng chỉ cần $O(\log n)$ phép nhân ma trận là đủ để tính $X^n$.

Vậy số phép toán số học cần bởi thuật toán tính $F_n$ dùng nhân ma trận, ta gọi nó là `fib3`, chỉ là $O(\log n)$. Có phải chứng ta đã phá vỡ giới hạn lũy thừa cho phép tính số fibonacci thứ $n$?