VNU-HUS MAT3500: Toán rời rạc

Thuật toán II

Thuật toán đệ quy, thuật toán tham lam

Hoàng Anh Đức

Bộ môn Tin học, Khoa Toán-Cơ-Tin học
Đại học KHTN, ĐHQG Hà Nội
hoanganhduc@hus.edu.vn

Nội dung

Thuật toán đệ quy
- Giới thiệu
- Tính giai thừa và lũy thừa
- Tìm kiếm tuyến tính
- Tìm kiếm nhị phân
- Sắp xếp nổi bọt
- Sắp xếp chèn
- Sắp xếp trộn

Ước lượng hệ thức truy hồi
- Giới thiệu
- Định lý thợ
- Cây đệ quy

Thuật toán tham lam
- Giới thiệu
- Bài toán Lập lịch

Thuật toán đệ quy
Giới thiệu

- Định nghĩa theo đệ quy không những có thể áp dụng cho các hàm và tập hợp mà còn cho cả các thuật toán
- Một *thủ tục đệ quy (recursive procedure)* là một thủ tục gọi chính nó
- Một *thuật toán đệ quy (recursive algorithm)* là một thuật toán giải một bài toán bằng cách chuyển về việc giải chính bài toán đó nhưng với đầu vào có kích thước nhỏ hơn
  - *Kỹ thuật chia để trị (divide-and-conquer technique):* giải một bài toán ban đầu thông qua việc chia nó thành các bài toán nhỏ hơn cùng loại và giải chúng

Thuật toán đệ quy
Giới thiệu

Một thuật toán đệ quy thường có cấu trúc như sau:

Trường hợp cơ sở Giải trực tiếp bài toán với đầu vào có kích thước đủ nhỏ hoặc đầu vào đặc biệt mà không cần gọi đệ quy

Chia thành các bài toán con Với đầu vào không thuộc trường hợp cơ sở, thuật toán chia bài toán thành một hoặc nhiều *bài toán con* có kích thước nhỏ hơn

Giải các bài toán con Thuật toán gọi đệ quy chính nó để giải các bài toán con đã được định nghĩa

Kết hợp các lời giải Thuật toán kết hợp các lời giải của các bài toán con để tạo ra lời giải cho bài toán ban đầu

Chứng minh tính đúng đắn của thuật toán đệ quy bằng quy nạp mạnh

- **Phát biểu điều cần chứng minh:** Một điểm quan trọng là cần *chỉ rõ "thuật toán đúng" nghĩa là gì*
- **Bước cơ sở:** Các trường hợp khi *thuật toán cho ra kết quả một cách trực tiếp* mà không cần thông qua gọi đệ quy chính nó là các trường hợp cần xét trong bước cơ sở
  - Sử dụng mô tả của thuật toán để chỉ ra thuật toán sẽ trả lại gì trong trường hợp cơ sở
  - Chỉ ra rằng giá trị trả lại của thuật toán là đúng
- **Bước quy nạp:** Giả thiết rằng thuật toán đúng cho *mọi đầu vào kích thước nhỏ hơn*. Chỉ ra rằng thuật toán cũng đúng cho đầu vào hiện tại
  - Phát biểu giả thiết quy nạp: Giả sử thuật toán đúng với mọi đầu vào giữa trường hợp cơ sở và các đầu vào có kích thước nhỏ hơn một đơn vị so với đầu vào hiện tại
  - Mô tả cụ thể thuật toán trả lại gì với đầu vào hiện tại dựa trên các lần gọi đệ quy
  - Sử dụng giả thiết quy nạp để thay mỗi lần gọi đệ quy bằng đáp án chính xác. Chỉ ra rằng những điều này dẫn tới đáp án đúng cho trường hợp hiện tại
  - Nếu bạn xét nhiều trường hợp trong thuật toán thì cần thực hiện hai điều trên với từng trường hợp một

Phân tích thời gian chạy của thuật toán đệ quy:

- *Thiết lập hệ thức truy hồi*
  - Biểu diễn thời gian chạy $T(n)$ dưới dạng các hàm của thời gian chạy cho các bài toán con
- *Giải hoặc ước lượng hệ thức truy hồi*
  - Giải hệ thức truy hồi (đã đề cập trong phần “Quy nạp và Đệ quy”)
    - Đoán nghiệm và chứng minh bằng quy nạp
    - Sử dụng hàm đặc trưng
    - Sử dụng hàm sinh
  - Ước lượng hệ thức truy hồi (sẽ đề cập trong phần sau)
    - Định lý thợ (Master Theorem)
    - Cây đệ quy (Recursion Tree)

Với mọi số nguyên không âm n

$$0! = 1$$
$$n! = n \cdot (n-1)! \quad \forall n \geq 1$$

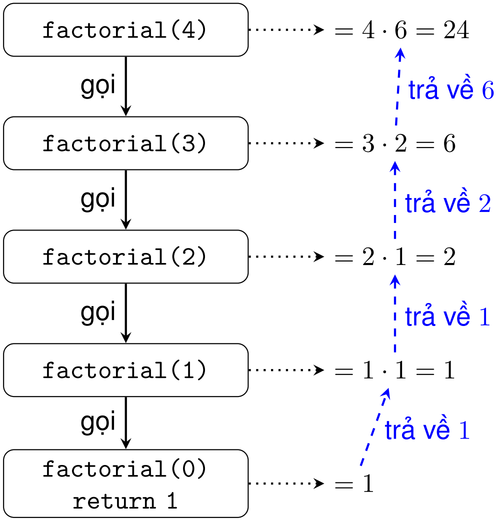
Thuật toán 1: Tính $n!$

Input: n: số nguyên không âm

Output: $n!$

```
procedure factorial(n):
    if n = 0 then
        return 1
    else
        return n · factorial(n − 1)
```

Ví dụ 1 (`factorial(4)`)



Ví dụ 2 (Thuật toán đệ quy tính giai thừa là đúng)

Ta chứng minh tính đúng đắn của Thuật toán 1 bằng quy nạp. Gọi `factorial`(n) là giá trị trả lại bởi Thuật toán 1

- Ta chứng minh `factorial`$(n) = n!$ với mọi $n \geq 0$
- **Bước cơ sở:** Khi $n = 0$, `factorial`$(n) = 1 = n!$
- **Bước quy nạp:** Giả sử `factorial`$(k) = k!$ với số nguyên $k \geq 0$ nào đó. Ta chứng minh `factorial`$(k+1) = (k+1)!$. Thật vậy, Thuật toán 1 trả lại `factorial`$(k+1) = (k+1) \cdot$ `factorial`(k). Theo giả thiết quy nạp, `factorial`$(k) = k!$. Do đó, `factorial`$(k+1) = (k+1) \cdot k! = (k+1)!$

Ví dụ 3 (Thời gian chạy của thuật toán đệ quy tính giai thừa)

$$T(n) = \max\{O(1), T(n-1) + O(1)\} + O(1) = T(n-1) + O(1)$$

nghĩa là tồn tại hằng số C thỏa mãn $T(n) = T(n-1) + C$. Suy ra $T(n) = O(n)$

Với mọi số thực $a \neq 0$ và số nguyên không âm n,

$$a^0 = 1$$

$$a^n = a \cdot a^{n-1} \quad \forall n \geq 1$$

Thuật toán 2: Tính a^n

Input: a: số thực khác 0, n: số nguyên không âm

Output: a^n

```
procedure power(a, n):
    if n = 0 then
        return 1
    else
        return a · power(a, n − 1)
```

Bài tập 1

Với Thuật toán 2, hãy

(a) Chứng minh tính đúng đắn

(b) Xây dựng hệ thức truy hồi để đánh giá thời gian chạy

Thuật toán 3: Tìm kiếm tuyến tính (Linear Search)

Input: $a_1, a_2, \ldots, a_n$: dãy số nguyên, i, j, x: số nguyên, $1 \leq i \leq j \leq n$

Output: Nếu $x \in \{a_i, a_{i+1}, \ldots, a_j\}$ thì trả lại $k \in \{i, i+1, \ldots, j\}$ sao cho $x = a_k$. Ngược lại thì trả lại 0

procedure `LinearSearch`(i, j, x):
    if $a_i = x$ **then** `// Ở đúng vị trí? Trả lại kết quả`
        return i
    else
        if $i = j$ **then** `// Không tìm thấy`
            return 0
        else `// Tìm trong phần còn lại`
            return `LinearSearch`$(i+1, j, x)$

Ví dụ 4 ($\texttt{LinearSearch}(1, 5, 8)$)

5	2	8	14	7
1	2	3	4	5

$\texttt{LinearSearch(1,5,8)}$ ⋯⋯→ $= 3$ (return from call)

gọi ↓ ↑ trả về 3

$\texttt{LinearSearch(2,5,8)}$ ⋯⋯→ $= 3$ (return from call)

gọi ↓ ↑ trả về 3

$\texttt{LinearSearch(3,5,8)}$ ⋯⋯→ $= 3$ (found)

Bài tập 2

Với Thuật toán 3, hãy

(a) Chứng minh tính đúng đắn

(b) Xây dựng hệ thức truy hồi để đánh giá thời gian chạy

Thuật toán 4: Tìm kiếm nhị phân (Binary Search)

Input: $a_1, a_2, \ldots, a_n$: dãy số nguyên thực sự tăng, i, j, x: số nguyên, $1 \leq i \leq j \leq n$

Output: Nếu $x \in \{a_i, a_{i+1}, \ldots, a_j\}$ thì trả lại $k \in \{i, i+1, \ldots, j\}$ sao cho $x = a_k$. Ngược lại thì trả lại 0

procedure `BinarySearch`(i, j, x)**:**
- $m := \lfloor (i+j)/2 \rfloor$ `// Đi đến giữa dãy`
- **if** $x = a_m$ **then** `// Đúng vị trí?`
  - **return** m
- **else**
  - **if** $x < a_m$ *và* $i < m$ **then** `// x ở nửa bên trái?`
    - **return** `BinarySearch`$(i, m-1, x)$
  - **else**
    - **if** $x > a_m$ *và* $j > m$ **then** `// x ở nửa bên phải?`
      - **return** `BinarySearch`$(m+1, j, x)$
    - **else** `// Không tìm thấy`
      - **return** 0

Ví dụ 5 (`BinarySearch(1, 10, 23)`)

2	5	8	12	16	23	38	56	72	91
1	2	3	4	5	6	7	8	9	10

`BinarySearch(1,10,23)` ⋯→ $= 6$ (return from call)

gọi $m = \lfloor (1+10)/2 \rfloor = 5, a_5 = 16 < 23$ trả về 6

`BinarySearch(6,10,23)` ⋯→ $= 6$ (return from call)

gọi $m = \lfloor (6+10)/2 \rfloor = 8, a_8 = 56 > 23$ trả về 6

`BinarySearch(6,7,23)` ⋯→ $= 6$ (found)

$$m = \lfloor (6+7)/2 \rfloor = 6, a_6 = 23 = 23$$

Bài tập 3

Xây dựng hệ thức truy hồi để đánh giá thời gian chạy của Thuật toán 4

Thuật toán đệ quy

Sắp xếp nổi bọt

Thuật toán 5: Sắp xếp nổi bọt (Bubble Sort)

Input: $A = a_1, a_2, \ldots, a_n$: dãy số nguyên, n: số nguyên dương

Output: Dãy số nguyên sắp thứ tự tăng dần

```
procedure BubbleSort(A, n):
    if n = 1 then
        return A        // Trường hợp cơ sở: mảng có 1 phần tử đã được sắp xếp
    for i ← 1 to n − 1 do
        if a_i > a_{i+1} then
            Hoán đổi giá trị của a_i và a_{i+1}
    BubbleSort(A, n − 1)  // Đệ quy với phần tử cuối cùng đã được đẩy đến đúng vị trí
```

Ví dụ 6 (BubbleSort($[5, 1, 4, 2], 4$))

Mảng ban đầu: | 5 | 1 | 4 | 2 |

BubbleSort($[5, 1, 4, 2], 4$) — Lần lượt so sánh: $[5, 1], [1, 4], [4, 2]$

| 1 | 4 | 2 | 5 |

BubbleSort($[1, 4, 2, 5], 3$) — Lần lượt so sánh: $[1, 4], [4, 2]$

| 1 | 2 | 4 | 5 |

BubbleSort($[1, 2, 4, 5], 2$) — Lần lượt so sánh: $[1, 2]$

| 1 | 2 | 4 | 5 |

BubbleSort($[1, 2, 4, 5], 1$) — Trường hợp cơ sở: $n = 1$, mảng đã sắp xếp

Bài tập 4

Xây dựng hệ thức truy hồi để đánh giá thời gian chạy của Thuật toán 5

Thuật toán 6: Sắp xếp chèn (Insertion Sort)

Input: $A = a_1, a_2, \dots, a_n$: dãy số nguyên, n: số nguyên dương
Output: Dãy số nguyên sắp thứ tự tăng dần

procedure `InsertionSort`(A, n):
    if $n \leq 1$ **then**
        return A `// Trường hợp cơ sở: mảng có 0 hoặc 1 phần tử đã được sắp xếp`
    `InsertionSort`($A, n-1$) `// Sắp xếp đệ quy các phần tử từ` a_1 `đến` a_{n-1}
    `// Chèn phần tử` a_n `vào vị trí thích hợp trong mảng đã sắp xếp`
    $key := a_n$
    $i := n - 1$
    while $i \geq 1$ *và* $a_i > key$ **do**
        $a_{i+1} := a_i$
        $i := i - 1$
    $a_{i+1} := key$

Ví dụ 7 ($\texttt{InsertionSort}([5, 2, 4, 1], 4)$)

Mảng ban đầu:

5	2	4	1

$\texttt{InsertionSort}([5, 2, 4, 1], 4)$ — **Sắp xếp đệ quy** $[5, 2, 4]$, sau đó **chèn** 1

gọi ↓

$\texttt{InsertionSort}([5, 2, 4, 1], 3)$ — **Sắp xếp đệ quy** $[5, 2]$, sau đó **chèn** 4

gọi ↓

$\texttt{InsertionSort}([5, 2, 4, 1], 2)$ — **Sắp xếp đệ quy** $[5]$, sau đó **chèn** 2

gọi ↓

$\texttt{InsertionSort}([5, 2, 4, 1], 1)$ — **Trường hợp cơ sở:** mảng $[5]$ đã sắp xếp

2	5	4	1	**Chèn** 2 vào $[5]$
2	4	5	1	**Chèn** 4 vào $[2, 5]$
1	2	4	5	**Chèn** 1 vào $[2, 4, 5]$

Kết quả cuối cùng

Bài tập 5

Xây dựng hệ thức truy hồi để đánh giá thời gian chạy của Thuật toán 6

Thuật toán đệ quy
Sắp xếp trộn

Thuật toán 7: Sắp xếp trộn (Merge Sort)

Input: $L = a_1, a_2, \ldots, a_n$: dãy số nguyên
Output: Dãy số nguyên sắp thứ tự tăng dần

procedure $\texttt{MergeSort}(L)$:
    if $n > 1$ **then**
        $m := \lfloor n/2 \rfloor$
        $L_1 := a_1, \ldots, a_m$; $L_2 := a_{m+1}, \ldots, a_n$
        $L := \texttt{Merge}(\texttt{MergeSort}(L_1), \texttt{MergeSort}(L_2))$

procedure $\texttt{Merge}(A, B)$:
    Input: $A = (a_1, \ldots, a_{|A|}), B = (b_1, \ldots, b_{|B|})$: dãy số đã sắp thứ tự
    Output: Dãy các số trong cả A và B sắp thứ tự tăng dần
    if $A = \emptyset$ **then**
        return B
    if $B = \emptyset$ **then**
        return A
    if $a_1 < b_1$ **then**
        return $(a_1, \texttt{Merge}(a_2, \ldots, a_{|A|}, B))$
    else
        return $(b_1, \texttt{Merge}(A, b_2, \ldots, b_{|B|}))$

Ví dụ 8 ($\texttt{Merge}([2,5],[1,3])$)

Mảng A:

2	5

Mảng B:

1	3

`merge([2,5], [1,3])` ⋯⋯→ $= [1,2,3,5]$

$1 < 2$ nên lấy 1

gọi ↓ ↑ trả về

`merge([2,5], [3])` ⋯⋯→ $= [2,3,5]$

$2 < 3$ nên lấy 2

gọi ↓ ↑ trả về

`merge([5], [3])` ⋯⋯→ $= [3,5]$

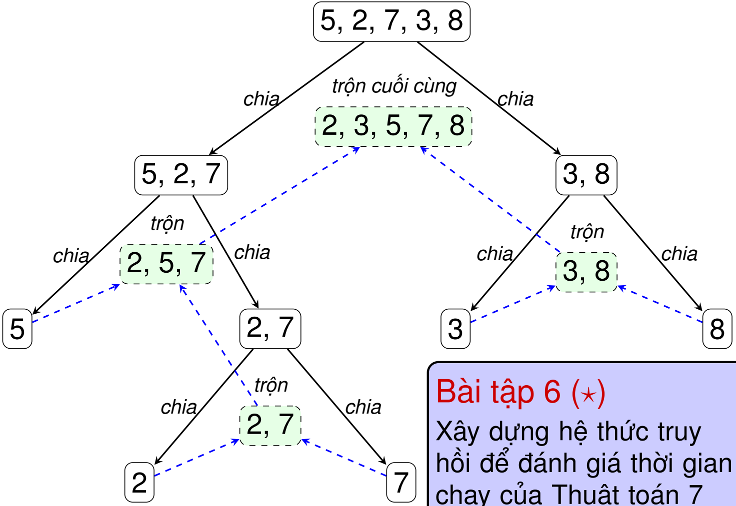
$3 < 5$ nên lấy 3

gọi ↓ ↑ trả về

`merge([5], [])` ⋯⋯→ $= [5]$

$B = \emptyset$ nên trả về A

Ví dụ 9 ($\texttt{MergeSort}([5, 2, 7, 3, 8])$)



Bài tập 6 ($\star$)

Xây dựng hệ thức truy hồi để đánh giá thời gian chạy của Thuật toán 7

- Hệ thức truy hồi thường xuất hiện khi phân tích thuật toán đệ quy
- Ước lượng hệ thức truy hồi giúp ta xác định độ phức tạp thời gian của thuật toán
- Hai phương pháp chính để ước lượng hệ thức truy hồi:
  (1) *Định lý thợ (Master Theorem)*: Công cụ mạnh mẽ giúp ước lượng nhanh chóng các hệ thức có dạng $f(n) = af(n/b) + cn^d$
  (2) *Cây đệ quy (Recursion Tree)*: Biểu diễn trực quan quá trình tính toán

Định lý 1: Định lý thợ (Master Theorem)

Gọi f là một hàm tăng thỏa mãn hệ thức truy hồi

$$f(n) = af(n/b) + cn^d$$

trong đó $n = b^k$ với k là số nguyên dương nào đó, $a \geq 1$, b là số nguyên dương lớn hơn 1, và c, d là các số thực với c dương và d không âm. Ta có

$$f(n) \text{ là } \begin{cases} O(n^d) & \text{nếu } a < b^d \\ O(n^d \log n) & \text{nếu } a = b^d \\ O(n^{\log_b a}) & \text{nếu } a > b^d \end{cases}$$

Ví dụ 10

- Với $T(n) = 2T(n/2) + n$, ta có $T(n) = O(n \log n)$
- Với $T(n) = T(n/2) + n$, ta có $T(n) = O(n)$
- Với $T(n) = 3T(n/2) + n$, ta có $T(n) = O(n^{\log 3})$

Bài tập 7

Sử dụng Định lý thợ (Định lý 1), hãy ước lượng các hệ thức truy hồi sau theo O-lớn, giả sử $T(1) = 1$

(a) $T(n) = 4T(n/3) + n^2$

(b) $T(n) = 4T(n/2) + n^2$

(c) $T(n) = 3T(n/3) + n \log n$

(d) $T(n) = 3T(n/3) + 1$

(e) $T(n) = 2T(n/2) + n^n$

(f) $T(n) = 7T(n/2) + n^2$

(g) $T(n) = 16T(n/4) + n^2$

(h) $T(n) = 2T(n/2) - n^2$

(i) $T(n) = 8T(n/2) + n^3$

(j) $T(n) = 3T(n/3) + \sqrt{n}$

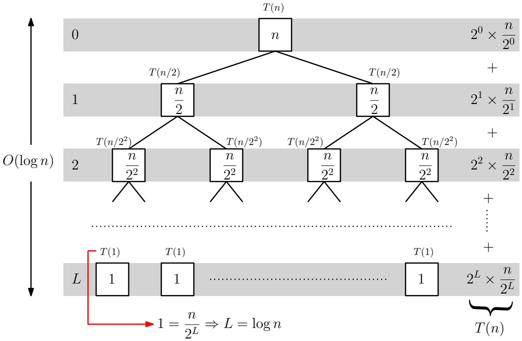
Ước lượng hệ thức truy hồi

Cây đệ quy

- *Cây đệ quy (Recursion Tree)* là biểu diễn trực quan của quá trình chạy thuật toán đệ quy
- Mỗi nút trong cây biểu diễn *chi phí (cost)* của một *lần gọi đệ quy (recursive call)*
- **Các bước xây dựng và phân tích cây đệ quy:**
  (1) Vẽ cây với gốc ứng với lần gọi ban đầu $T(n)$
  (2) Phân tách mỗi nút thành các chi phí: chi phí không đệ quy (non-recursive cost) và các lần gọi đệ quy (recursive calls)
  (3) Tiếp tục phân tách cho đến khi đạt trường hợp cơ sở (base case)
  (4) Tính tổng chi phí theo từng mức (level) của cây
  (5) Tính tổng chi phí các mức để có kết quả cuối cùng
- **Ưu điểm:** Trực quan, giúp hiểu được bản chất của thuật toán và áp dụng được với nhiều dạng hệ thức mà Định lý thợ không áp dụng được
- **Nhược điểm:** Không phải lúc nào cũng dễ vẽ và phân tích được

Ví dụ 11

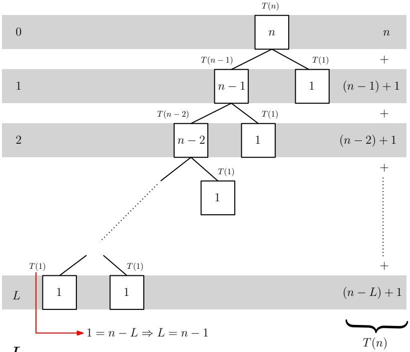
Xét hệ thức truy hồi $T(n) = 2T(n/2) + n$ với điều kiện ban đầu $T(1) = 1$ và $n = 2^k$ với số nguyên $k \geq 1$ nào đó. Ta vẽ cây đệ quy cho hệ thức này như sau



Ta có $T(n) = \sum_{i=0}^{L} 2^i \cdot \frac{n}{2^i} = n(\log n + 1) = O(n \log n)$

Ví dụ 12

Xét hệ thức truy hồi $T(n) = T(n-1) + T(1) + n$ với $n \geq 1$ và điều kiện ban đầu $T(1) = 1$. Ta vẽ cây đệ quy của hệ thức này như sau



Ta có $T(n) \leq \sum_{i=0}^{L} n = O(n^2)$

Bài tập 8

Sử dụng cây đệ quy để ước lượng $T(n)$ cho bởi các hệ thức truy hồi sau

(a) $T(n) = 2T(n/2) + n^2$

(b) $T(n) = T(n/2) + 1$

(c) $T(n) = 2T(n-1) + 1$

(d) ($\star$) $T(n) = 2T(n/2) + n \log n$ (**Đáp án:** $O(n \log^2 n)$)

Bài tập 9

Chứng minh Định lý thợ bằng cách sử dụng cây đệ quy

(a) Vẽ cây đệ quy cho $T(n) = aT(n/b) + cn^d$ trong đó $n = b^k$ với k là số nguyên dương nào đó, $a \geq 1$, b là số nguyên dương lớn hơn 1, và c, d là các số thực với c dương và d không âm

(b) Tính tổng từng hàng và viết công thức của $T(n)$ dưới dạng tổng của các hàng trong cây.

(c) Xét các trường hợp $a < b^d$, $a = b^d$, và $a > b^d$

Bài tập 10

Xây dựng hệ thức truy hồi cho thời gian chạy của các thuật toán đệ quy đã đề cập trong bài giảng. Giải hoặc ước lượng hệ thức bạn tìm được để chứng minh thời gian chạy của các thuật toán như sau:

- Tính lũy thừa (Thuật toán 2): $T(n) = O(n)$
- Tìm kiếm tuyến tính (Thuật toán 3): $T(n) = O(n)$
- Tìm kiếm nhị phân (Thuật toán 4): $T(n) = O(\log n)$
- Sắp xếp nổi bọt (Thuật toán 5): $T(n) = O(n^2)$
- Sắp xếp chèn (Thuật toán 6): $T(n) = O(n^2)$
- Sắp xếp trộn (Thuật toán 7): $T(n) = O(n \log n)$

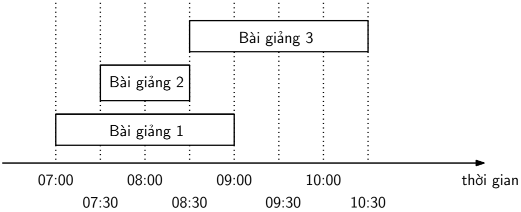
Thuật toán tham lam
Giới thiệu

- *Các bài toán tối ưu (optimization problems)* yêu cầu cực đại hóa hoặc cực tiểu hóa một số tham số xét trên tập tất cả các đầu vào có thể
  - Tìm đường đi giữa hai thành phố với khoảng cách *nhỏ nhất*
  - Tìm cách đặt các trạm phát sóng sao cho diện tích phủ sóng *lớn nhất*
- Một *thuật toán tham lam (greedy algorithm)* thường được sử dụng để giải bài toán tối ưu: luôn chọn biện pháp "tốt nhất" ở mỗi bước địa phương (theo một số tiêu chuẩn cục bộ nào đó) với hi vọng sẽ thu được một lời giải tối ưu trên toàn cục
  - Giải thuật này không nhất thiết xuất ra một lời giải tối ưu cho toàn bộ bài toán, nhưng trong nhiều trường hợp cụ thể nó có thể xuất ra lời giải tối ưu
  - Sau khi mô tả cụ thể "lựa chọn tốt nhất ở từng bước", ta cố gắng chứng minh rằng giải thuật này luôn cho ta một lời giải tối ưu hoặc tìm một phản ví dụ để chỉ ra điều ngược lại

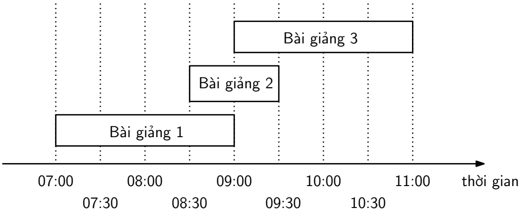
Thuật toán tham lam
Bài toán Lập lịch

- **Bài toán:**
  - **Input:**
    - Một nhóm các bài giảng với thời gian bắt đầu và kết thúc
    - Chỉ có một giảng đường duy nhất
    - Khi một bài giảng bắt đầu, nó tiếp diễn cho đến khi kết thúc
    - Không có hai bài giảng nào được tiến hành ở cùng thời điểm
    - Ngay sau khi một bài giảng kết thúc, một bài giảng khác có thể bắt đầu
  - **Output:** Một danh sách các bài giảng dài nhất có thể
- Ở đây, nếu ta muốn áp dụng giải thuật tham lam, làm thế nào để "lựa chọn tốt nhất" ở mỗi bước của thuật toán? Nói cách khác, ta sẽ chọn bài giảng như thế nào?
  (1) Chọn bài giảng bắt đầu sớm nhất trong số các bài giảng bắt đầu sau các bài giảng ta vừa chọn trước đó?
  (2) Chọn bài giảng ngắn nhất trong số các bài giảng bắt đầu sau các bài giảng ta vừa chọn trước đó?
  (3) Chọn bài giảng kết thúc sớm nhất trong số các bài giảng bắt đầu sau các bài giảng ta vừa chọn trước đó?

(1) Chọn bài giảng bắt đầu sớm nhất? ✗



(2) Chọn bài giảng ngắn nhất? ✗



(3) Chọn bài giảng kết thúc sớm nhất? ✓

Thuật toán 8: Lập lịch tham lam (Greedy Scheduling)

Input: $(s_1, e_1), (s_2, e_2), \ldots, (s_n, e_n)$: thời gian bắt đầu và kết thúc bài giảng $b_1, b_2, \ldots, b_n$

Output: Danh sách bài giảng S có số bài giảng lớn nhất trong đó không có hai bài giảng nào xung đột nhau

Sắp xếp các bài giảng theo thứ tự tăng dần theo thời gian kết thúc và gán lại nhãn bài giảng sao cho $e_1 \leq e_2 \leq \cdots \leq e_n$

$S := \emptyset$

for $j := 1$ **to** n **do**

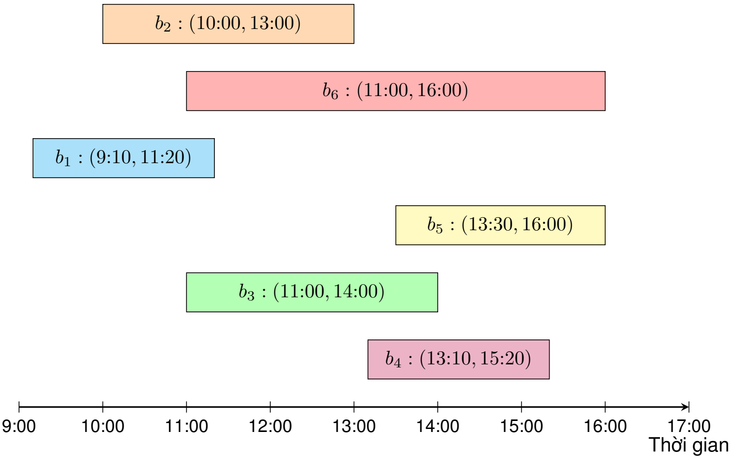
  if *Bài giảng j không xung đột với các phần tử của S* **then**

    $S := S \cup \{j\}$

return S

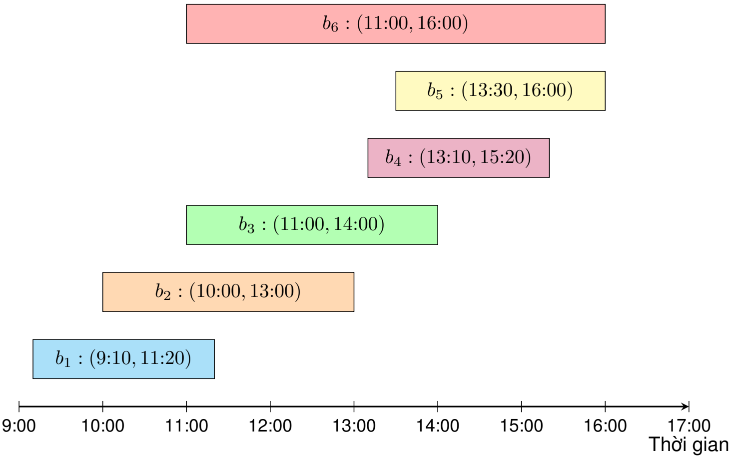
Ví dụ 13

Các bài giảng ban đầu:



Ví dụ 13

Bước 1: Sắp xếp theo thời gian kết thúc

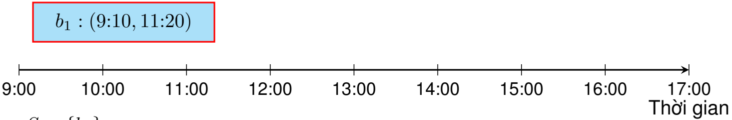


Ví dụ 13

Bước 2: Khởi tạo $S = \emptyset$

Bước 3: Duyệt các bài giảng

Xét $j = 1$: Thêm b_1 vào S



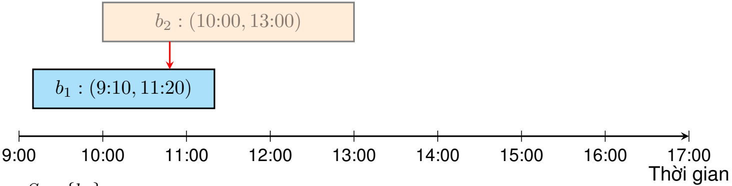
$S = \{b_1\}$

Ví dụ 13

Bước 2: Khởi tạo $S = \emptyset$

Bước 3: Duyệt các bài giảng

Xét $j = 2$: Không thêm b_2 vì xung đột với b_1



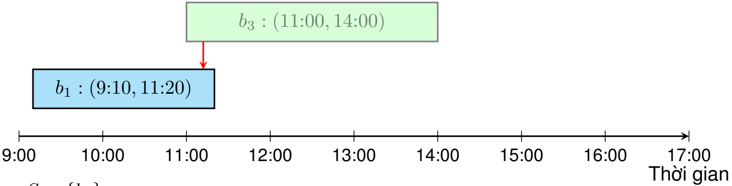
$S = \{b_1\}$

Ví dụ 13

Bước 2: Khởi tạo $S = \emptyset$

Bước 3: Duyệt các bài giảng

Xét $j = 3$: Không thêm b_3 vì xung đột với b_1



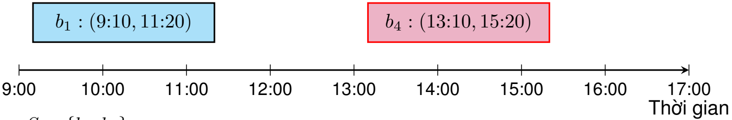
$S = \{b_1\}$

Ví dụ 13

Bước 2: Khởi tạo $S = \emptyset$

Bước 3: Duyệt các bài giảng

Xét $j = 4$: Thêm b_4 vào S



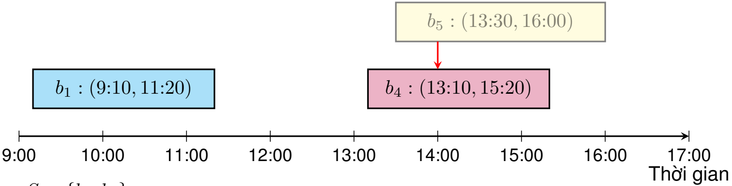
$S = \{b_1, b_4\}$

Ví dụ 13

Bước 2: Khởi tạo $S = \emptyset$

Bước 3: Duyệt các bài giảng

Xét $j = 5$: Không thêm b_5 vì xung đột với b_4



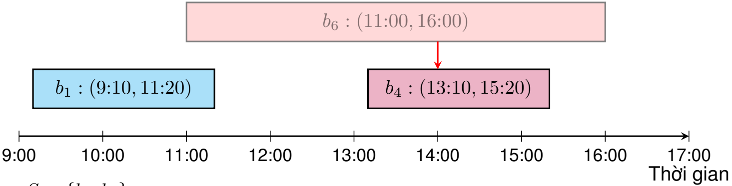
$S = \{b_1, b_4\}$

Ví dụ 13

Bước 2: Khởi tạo $S = \emptyset$

Bước 3: Duyệt các bài giảng

Xét $j = 6$: Không thêm b_6 vì xung đột với b_4



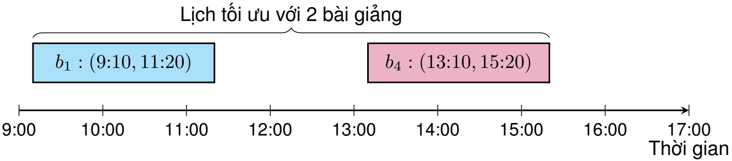
$S = \{b_1, b_4\}$

Ví dụ 13

Bước 2: Khởi tạo $S = \emptyset$

Bước 3: Duyệt các bài giảng

Kết quả: $S = \{b_1, b_4\}$



Định lý 2

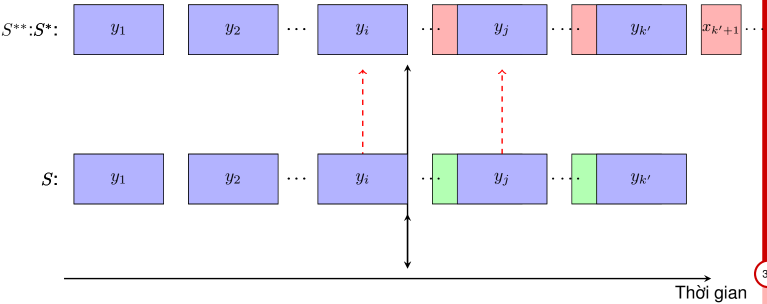
Thuật toán 8 xuất ra một danh sách các bài giảng tối ưu

Chứng minh.

- Giả sử $S^\star = (x_1, x_2, \ldots, x_k)$ là danh sách tối ưu trong đó các bài giảng được sắp xếp theo thứ tự tăng dần theo thời gian kết thúc
- Giả sử $S = (y_1, y_2, \ldots, y_{k'})$ là một danh sách bài giảng xuất ra từ Thuật toán 8 trong đó các bài giảng được sắp xếp theo thứ tự tăng dần theo thời gian kết thúc
- Do $S^\star$ là tối ưu, $k \geq k'$
- Nếu $S = S^\star$ thì ta có điều phải chứng minh. Ngược lại, nếu $S \neq S'$, gọi i là chỉ số đầu tiên trong $\{1, \ldots, k'\}$ thỏa mãn $x_i \neq y_i$, nghĩa là

$$S^\star = \langle x_1 = y_1, x_2 = y_2, \ldots, x_{i-1} = y_{i-1}, x_i, \ldots, x_{k'}, \ldots, x_k \rangle$$

$$S = \langle x_1 = y_1, x_2 = y_2, \ldots, x_{i-1} = y_{i-1}, y_i, \ldots, y_{k'} \rangle$$



Tiếp tục quá trình này cho đến khi $S^* = \langle y_1, y_2, \dots, y_{k'}, x_{k'+1}, \dots, x_k \rangle$
Do thuật toán kết thúc khi chọn $y_{k'}$,
các phần tử $x_{k'+1}, \dots, x_k$ phải xung đột với $y_{k'}$ nên không thể tồn tại
Do đó $S^* = S = \langle y_1, y_2, \dots, y_{k'} \rangle$ là tối ưu

Định lý 3

Thuật toán 8 chạy trong thời gian $O(n \log n)$, với n là số lượng bài giảng

Chứng minh.

Phân tích độ phức tạp:

- Sắp xếp n bài giảng theo thời gian kết thúc: $O(n \log n)$
  - Ví dụ, sử dụng Thuật toán 7 (Sắp xếp trộn)
- Vòng lặp **for** duyệt qua n bài giảng: $O(n)$
- Kiểm tra xung đột giữa bài giảng j với các bài giảng trong S:
  - Phương pháp đơn giản: $O(|S|) = O(n)$ trong trường hợp xấu nhất
  - Phương pháp tối ưu: Chỉ cần kiểm tra thời gian bắt đầu của bài giảng j với thời gian kết thúc của bài giảng cuối cùng trong S: $O(1)$

Part I

Phụ lục

Nội dung

Định lý thợ

Có thể bỏ qua sàn và trần

Một số thuật toán đệ quy khác

- Tìm số Fibonacci thứ n
- Tháp Hà Nội
- Sắp xếp nhanh (Quicksort)

Định lý: (Nhắc lại Định lý 1)

Gọi f là một hàm tăng thỏa mãn hệ thức truy hồi

$$f(n) = af(n/b) + cn^d$$

trong đó $n = b^k$ với k là số nguyên dương nào đó, $a \geq 1$, b là số nguyên dương lớn hơn 1, và c, d là các số thực với c dương và d không âm. Ta có

$$f(n) \text{ là } \begin{cases} O(n^d) & \text{nếu } a < b^d \\ O(n^d \log n) & \text{nếu } a = b^d \\ O(n^{\log_b a}) & \text{nếu } a > b^d \end{cases}$$

Ta có

$$
\begin{aligned}
f(n) &= af(n/b) + cn^d \\
&= a(af(n/b^2) + c(n/b)^d) + cn^d \\
&= a^2 f(n/b^2) + cn^d(1 + \frac{a}{b^d}) \\
&= a^2(af(n/b^3) + c(n/b^2)^d) + cn^d(1 + \frac{a}{b^d}) \\
&= a^3 f(n/b^3) + cn^d(1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2) \\
&= \dots \\
&= a^k f(n/b^k) + cn^d \sum_{i=0}^{k-1} \left(\frac{a}{b^d}\right)^i
\end{aligned}
$$

Xét phương trình $f(n) = a^k f(n/b^k) + cn^d \sum_{i=0}^{k-1} \left(\frac{a}{b^d}\right)^i$

(1) Nếu $a = b^d$, ta có $\sum_{i=0}^{k-1} \left(\frac{a}{b^d}\right)^i = k$

(2) Nếu $a < b^d$, ta có $\sum_{i=0}^{k-1} \left(\frac{a}{b^d}\right)^i \leq \sum_{i=0}^{\infty} \left(\frac{a}{b^d}\right)^i = \frac{1}{1 - \frac{a}{b^d}}$. Do a, b, d là các hằng số, ta có $\sum_{i=0}^{k-1} \left(\frac{a}{b^d}\right)^i = O(1)$

(3) Nếu $a > b^d$, ta có $\sum_{i=0}^{k-1} \left(\frac{a}{b^d}\right)^i = \frac{(a/b^d)^k - 1}{a/b^d - 1} = O((a/b^d)^k)$

Do đó,

$$f(n) = a^k f(n/b^k) + cn^d \cdot \begin{cases} O(k) & \text{nếu } a = b^d \\ O(1) & \text{nếu } a < b^d \\ O((a/b^d)^k) & \text{nếu } a > b^d \end{cases}$$

Từ $n = b^k$, ta có $k = \log_b n$. Do đó,

- Nếu $a = b^d$, ta có $f(n) = O(a^{\log_b n}) + O(n^d \log_b n) = O(n^{\log_b a}) + O(n^d \log_b n) = O(n^d \log n)$. (Chú ý là khi $a \leq b^d$, ta có $n^{\log_b a} \leq n^d$)
- Nếu $a < b^d$, ta có $f(n) = O(n^d) + O(n^d) = O(n^d)$. (Chú ý là khi $a \leq b^d$, ta có $n^{\log_b a} \leq n^d$)
- Nếu $a > b^d$, ta có $f(n) = O(a^{\log_b n}) + O(a^{\log_b n}) = O(a^{\log_b n}) = O(n^{\log_b a})$. (Chú ý là khi $a > b^d$, ta có $(a/b^d)^{\log_b n} = (b^{\log_b (a/b^d)})^{\log_b n} = (b^{\log_b n})^{\log_b (a/b^d)} = n^{\log_b (a/b^d)} = n^{\log_b a - d} = O(n^{\log_b a})$)

Định lý 4: Định lý thợ

Gọi f là một hàm tăng thỏa mãn hệ thức truy hồi

$$f(n) = af(n/b) + g(n)$$

trong đó $a \geq 1$, $b > 1$ là các hằng số nào đó, và $g(n)$ là một hàm thỏa mãn $g(n) \geq 0$ khi n đủ lớn. Ta có

(1) *$f(n)$ là $\Theta(g(n))$ nếu $g(n) = \Omega(n^{\log_a b+\epsilon})$ với $\epsilon > 0$ và $g(n)$ thỏa mãn điều kiện $ag(n/b) \leq cg(n)$ với n đủ lớn và $c < 1$ là hằng số dương nào đó*

(2) *$f(n)$ là $\Theta(n^{\log_b a}\log^{k+1} n)$ nếu $g(n) = \Theta(n^{\log_b a}\log^k n)$ với $k \geq 0$ nào đó. (Trong phần lớn các trường hợp thì $k = 0$.)*

(3) *$f(n)$ là $\Theta(n^{\log_b a})$ nếu $g(n) = O(n^{\log_b a-\epsilon})$ với $\epsilon > 0$*

Có thể bỏ qua sàn và trần

Chú ý

Có thể bỏ qua sàn và trần khi phân tích độ phức tạp của thuật toán

Ví dụ 14 (Thời gian chạy của thuật toán sắp xếp trộn)

- Nếu bạn phân tích giả mã của thuật toán sắp xếp trộn (Thuật toán 7), bạn sẽ nhận ra rằng thời gian chạy của thuật toán tuân theo hệ thức:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) \tag{1}$$

  (Xem Bài tập 6)

- Tuy nhiên, trong nhiều tài liệu, bạn sẽ thấy đề cập rằng thời gian chạy của thuật toán tuân theo hệ thức

$$T(n) = 2T(n/2) + O(n) \tag{2}$$

Có thể bỏ qua sàn và trần

Ví dụ 14 (Thời gian chạy của thuật toán sắp xếp trộn)

Cả hai hệ thức trên đều đúng. Lý do ở đây là ta có thể bỏ qua sàn và trần bằng cách lý luận như sau:

- Đầu tiên, do chúng ta chỉ quan tâm đến chặn trên (vì ta luôn đánh giá theo O-lớn), ta có thể đánh giá $T(n) \leq 2T(\lceil n/2 \rceil) + O(n) \leq 2T(n/2+1) + O(n)$
- Tiếp đó, ta định nghĩa một hàm mới $S(n) = T(n+\alpha)$ bằng cách chọn giá trị của α sao cho $S(n) \leq 2S(n/2) + O(n)$. Ta có

$$
\begin{aligned}
S(n) &= T(n+\alpha) && \text{định nghĩa của } S \\
&\leq 2T(n/2+\alpha/2+1) + O(n+\alpha) && \text{đánh giá cho } T \\
&= 2S(n/2-\alpha/2+1) + O(n+\alpha) && \text{định nghĩa của } S
\end{aligned}
$$

  Chọn $\alpha = 2$, ta có $S(n) \leq 2S(n/2) + O(n)$
- Ước lượng $S(n)$ (ví dụ, bằng Định lý thợ), ta có $S(n) = O(n \log n)$, và do đó $T(n-2) = O((n-2)\log(n-2)) = O(n \log n)$

Có thể bỏ qua sàn và trần

Bài tập 11

Xấp xỉ hệ thức truy hồi $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n - 1$ với điều kiện ban đầu $T(2) = 1$ và $T(1) = 0$. (**Chú ý:** $T(n)$ là thời gian chạy của thuật toán sắp xếp trộn)
(**Gợi ý:** Với $n = 2^k$ ($k \in \mathbb{N}$), chứng minh $T(n) = n \log n - n + 1$. Sử dụng quy nạp, chứng minh với mọi $n \in \mathbb{Z}^+$, $T(n) \approx n \log n - n + 1$.)

Bài tập 12

Cây van Emde Boas (van Emde Boas tree, vEB tree, hoặc van Emde Boas priority queue) là một cấu trúc dữ liệu đệ quy cho phép chúng ta chèn, xóa và tìm kiếm các khóa từ một tập hợp $U = \{1, 2, \dots, n\}$ một cách nhanh chóng. (Nó giải quyết cùng một vấn đề như cây tìm kiếm nhị phân, nhưng thời gian chạy sẽ tính theo kích thước của tập vũ trụ U thay vì số lượng khóa được lưu trữ.) Cây van Emde Boas có thời gian chạy cho bởi $T(n) = T(\sqrt{n}) + 1$ và $T(1) = 1$. Hãy giải hệ thức truy hồi này. (**Gợi ý:** Đặt $S(k) = T(2^k)$. **Đáp án:** $T(n) = O(\log \log n)$)

Bài tập 13

Ước lượng các hệ thức truy hồi sau:

(a) $T(n) = T(\lceil n/3 \rceil) + T(\lfloor 2n/3 \rfloor) + O(n)$

(b) ($\star$) $T(n) = \sqrt{n}T(\lceil \sqrt{n} \rceil) + O(n)$ (**Gợi ý:** Đặt $S(k) = T(2^{2^k})$. **Đáp án:** $T(n) = O(n \log \log n)$)

Có thể bỏ qua sàn và trần

Bài tập 14 ($\star$)

Cho một mảng đã sắp xếp $A[1 \dots n]$, và ta cần tìm chỉ số i thỏa mãn $A[i-1] < x \leq A[i]$. (Bài toán này có thể giải được bằng thuật toán tìm kiếm nhị phân.) Dưới đây là mô tả thuật toán RootSearch để giải quyết bài toán này:

- Nếu n nhỏ (giả sử, nhỏ hơn 100), tìm chỉ số bằng thuật toán tìm kiếm tuyến tính. Ngược lại:
- Định nghĩa $\text{mileposts} := A[\sqrt{n}], A[2\sqrt{n}], A[3\sqrt{n}], \dots, A[n]$ là danh sách các phần tử cách đều nhau $\sqrt{n}$ phần tử trong A.
- Đệ quy, tìm $\text{post} := \textsc{RootSearch}(\text{mileposts}, x)$.
- Trả về $\textsc{RootSearch}(A[(\text{post}-1)\sqrt{n}, \dots, \text{post}\sqrt{n}], x)$.

Có thể bỏ qua sàn và trần

Bài tập 14 (tiếp tục)

```
ROOTSEARCH(A[1...n], x):
  if n < 100 then ⟨⟨Trường hợp cơ sở: mảng nhỏ⟩⟩
      ⟨⟨Tìm kiếm tuyến tính⟩⟩
      for i := 1 to n do
          if A[i − 1] < x ≤ A[i] then
              return i
  ⟨⟨Định nghĩa các cột mốc⟩⟩
  k := ⌊√n⌋
  mileposts := new array of size ⌈n/k⌉
  for i := 1 to ⌈n/k⌉ do
      mileposts[i] := A[min(i · k, n)]
  ⟨⟨Tìm cột mốc thích hợp⟩⟩
  post := ROOTSEARCH(mileposts, x)
  ⟨⟨Tìm kiếm trong đoạn con đã xác định⟩⟩
  return ROOTSEARCH(A[(post − 1) · k + 1 ... min(post · k, n)], x)
```

Hãy xác định hệ thức truy hồi biểu diễn thời gian chạy của thuật toán ROOTSEARCH và ước lượng hệ thức bạn tìm được

Một số thuật toán đệ quy khác

Tìm số Fibonacci thứ n

Bài toán: Tính số Fibonacci thứ n, được định nghĩa như sau:

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \quad \text{với } n \geq 2$$

Thuật toán 9: Thuật toán tính số Fibonacci thứ n

Input: Số nguyên không âm n

Output: Số Fibonacci thứ n

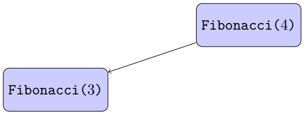
procedure `Fibonacci`(n):
    if $n = 0$ **then**
        return 0
    else
        if $n = 1$ **then**
            return 1
        else
            return `Fibonacci`($n-1$) + `Fibonacci`($n-2$)

Ví dụ 15 (`Fibonacci(4)`)

`Fibonacci(4)`

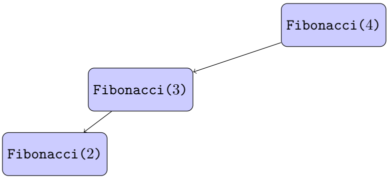
Tính F_4 bằng đệ quy

Ví dụ 15 (Fibonacci(4))



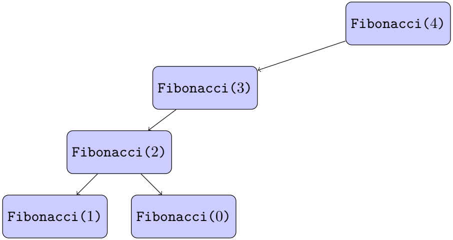
Đầu tiên, tính Fibonacci(3)

Ví dụ 15 (Fibonacci(4))



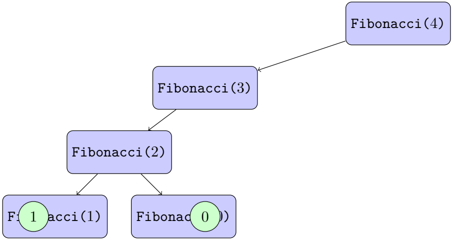
Để tính Fibonacci(3), ta cần tính Fibonacci(2)

Ví dụ 15 (`Fibonacci(4)`)



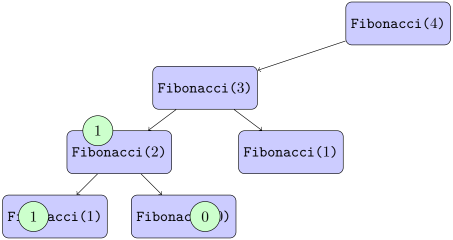
Để tính `Fibonacci(2)`, ta cần `Fibonacci(1)` và `Fibonacci(0)`

Ví dụ 15 (`Fibonacci(4)`)



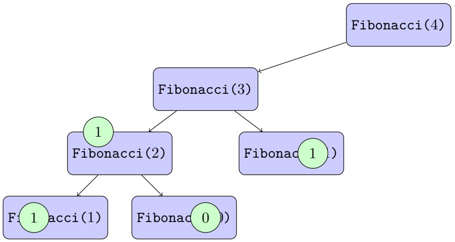
Trường hợp cơ sở: `Fibonacci(1)` $= 1$ và `Fibonacci(0)` $= 0$

Ví dụ 15 ($\texttt{Fibonacci}(4)$)



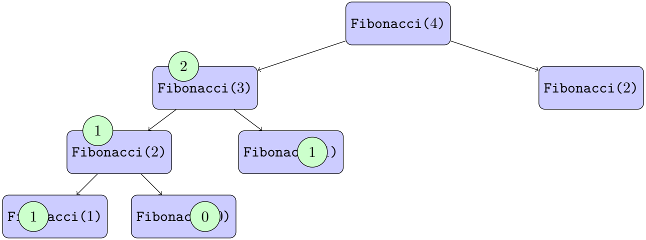
$$\texttt{Fibonacci}(2) = \texttt{Fibonacci}(1) + \texttt{Fibonacci}(0) = 1 + 0 = 1$$

Ví dụ 15 (`Fibonacci(4)`)



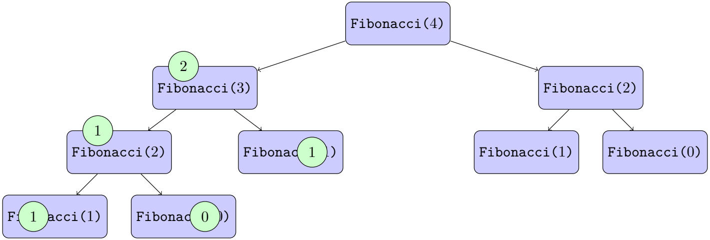
Với `Fibonacci(3)`, ta cũng cần tính `Fibonacci(1)` $= 1$

Ví dụ 15 (`Fibonacci(4)`)



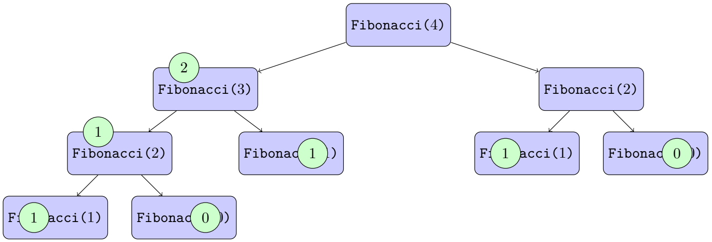
$$\texttt{Fibonacci(3)} = \texttt{Fibonacci(2)} + \texttt{Fibonacci(1)} = 1 + 1 = 2$$

Ví dụ 15 (`Fibonacci(4)`)



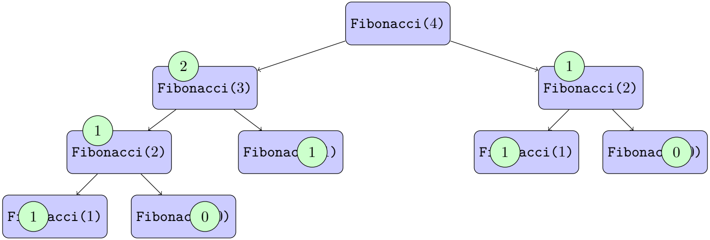
Bây giờ cho phần thứ hai của `Fibonacci(4)`, tính `Fibonacci(2)`

Ví dụ 15 (Fibonacci(4))



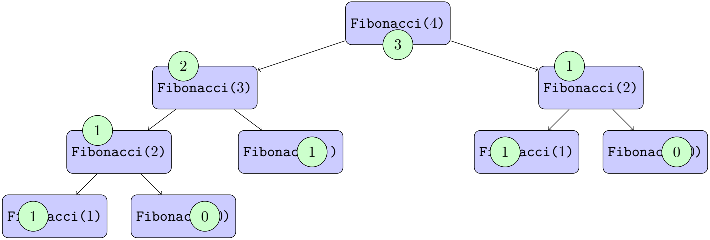
$$\texttt{Fibonacci}(2) = \texttt{Fibonacci}(1) + \texttt{Fibonacci}(0) = 1 + 0 = 1$$

Ví dụ 15 (`Fibonacci(4)`)



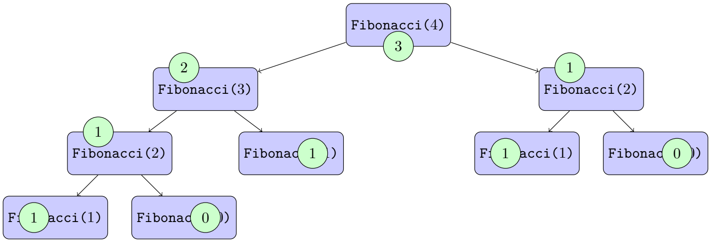
Cuối cùng, $\texttt{Fibonacci}(4) = \texttt{Fibonacci}(3) + \texttt{Fibonacci}(2) = 2 + 1 = 3$

Ví dụ 15 (Fibonacci(4))



Cuối cùng, $\texttt{Fibonacci}(4) = \texttt{Fibonacci}(3) + \texttt{Fibonacci}(2) = 2 + 1 = 3$

Ví dụ 15 (`Fibonacci(4)`)



Bài tập 15

(a) Chứng minh tính đúng đắn của thuật toán tìm số Fibonacci đệ quy

(b) Chứng minh rằng thời gian chạy của thuật toán là $O(2^n)$

(c) ($\star$) Cải tiến thuật toán để giảm độ phức tạp thời gian

Bài toán: Di chuyển n đĩa từ cọc A sang cọc C với sự trợ giúp của cọc B, trong đó không có hai đĩa nào có cùng kích thước, tuân theo các quy tắc:

- Chỉ được di chuyển một đĩa mỗi lượt
- Không được đặt đĩa lớn hơn lên trên đĩa nhỏ hơn

Thuật toán 10: Thuật toán giải bài toán Tháp Hà Nội

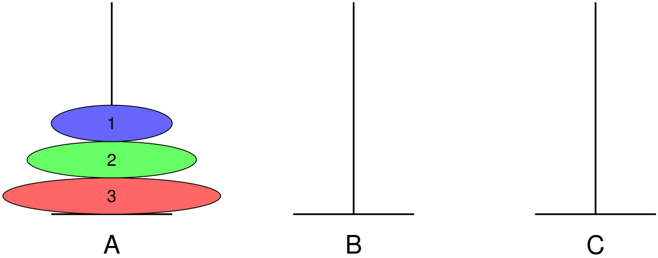
Input: Số nguyên n, cọc nguồn A, cọc đích C, cọc trung gian B

Output: Dãy các bước di chuyển

```
procedure Hanoi(n, A, C, B):
    if n = 1 then
        Di chuyển đĩa trên cùng từ A sang C
        return
    Hanoi(n − 1, A, B, C)
    Di chuyển đĩa trên cùng từ A sang C
    Hanoi(n − 1, B, C, A)
```

Ví dụ 16 (Minh họa giải thuật Tháp Hà Nội với 3 đĩa)

Trạng thái ban đầu: tất cả đĩa ở cọc A



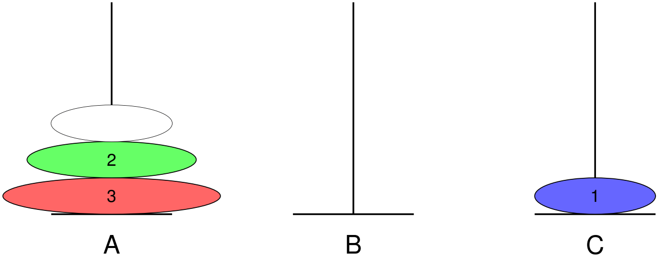
Bài tập 16

(a) Chứng minh tính đúng đắn của thuật toán giải bài toán Tháp Hà Nội

(b) Chứng minh rằng số bước di chuyển tối thiểu để giải quyết bài toán với n đĩa là $2^n - 1$

Ví dụ 16 (Minh họa giải thuật Tháp Hà Nội với 3 đĩa)

Bước 1: Di chuyển đĩa 1 từ A sang C
`Hanoi`$(1, A, C, B)$

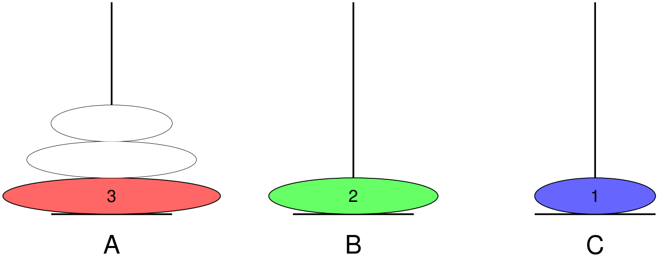


Bài tập 16

(a) Chứng minh tính đúng đắn của thuật toán giải bài toán Tháp Hà Nội
(b) Chứng minh rằng số bước di chuyển tối thiểu để giải quyết bài toán với n đĩa là $2^n - 1$

Ví dụ 16 (Minh họa giải thuật Tháp Hà Nội với 3 đĩa)

Bước 2: Di chuyển đĩa 2 từ A sang B
Đang thực hiện `Hanoi`$(2, A, B, C)$



Bài tập 16

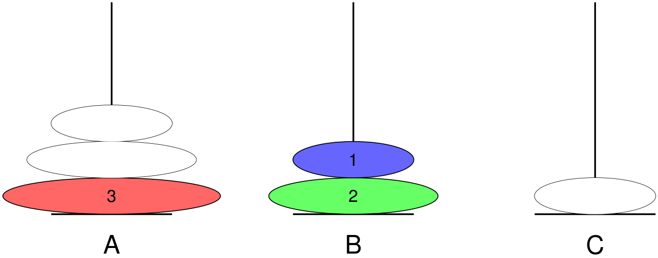
(a) Chứng minh tính đúng đắn của thuật toán giải bài toán Tháp Hà Nội

(b) Chứng minh rằng số bước di chuyển tối thiểu để giải quyết bài toán với n đĩa là $2^n - 1$

Ví dụ 16 (Minh họa giải thuật Tháp Hà Nội với 3 đĩa)

Bước 3: Di chuyển đĩa 1 từ C sang B

`Hanoi`$(1, C, B, A)$



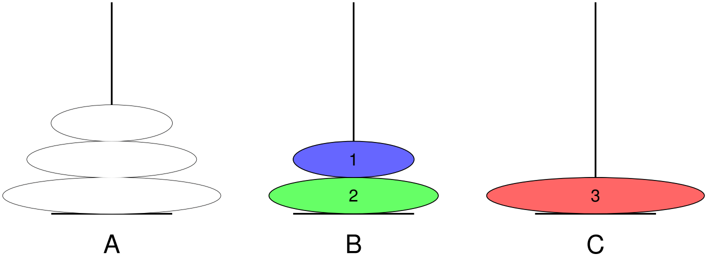
Bài tập 16

(a) Chứng minh tính đúng đắn của thuật toán giải bài toán Tháp Hà Nội

(b) Chứng minh rằng số bước di chuyển tối thiểu để giải quyết bài toán với n đĩa là $2^n - 1$

Ví dụ 16 (Minh họa giải thuật Tháp Hà Nội với 3 đĩa)

Bước 4: Di chuyển đĩa 3 từ A sang C



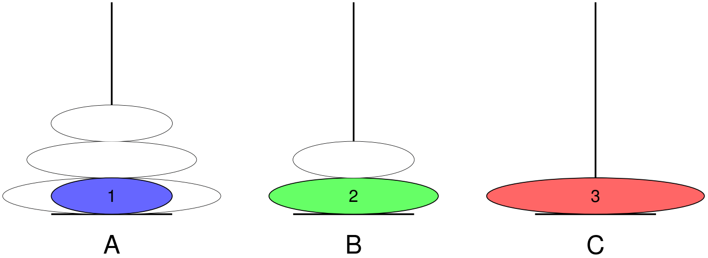
Bài tập 16

(a) Chứng minh tính đúng đắn của thuật toán giải bài toán Tháp Hà Nội

(b) Chứng minh rằng số bước di chuyển tối thiểu để giải quyết bài toán với n đĩa là $2^n - 1$

Ví dụ 16 (Minh họa giải thuật Tháp Hà Nội với 3 đĩa)

Bước 5: Di chuyển đĩa 1 từ B sang A
`Hanoi`$(1, B, A, C)$



Bài tập 16

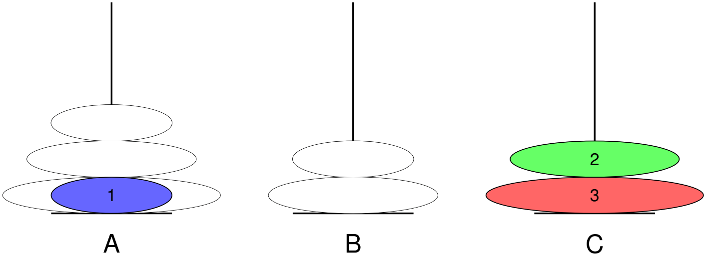
(a) Chứng minh tính đúng đắn của thuật toán giải bài toán Tháp Hà Nội

(b) Chứng minh rằng số bước di chuyển tối thiểu để giải quyết bài toán với n đĩa là $2^n - 1$

Ví dụ 16 (Minh họa giải thuật Tháp Hà Nội với 3 đĩa)

Bước 6: Di chuyển đĩa 2 từ B sang C

Đang thực hiện `Hanoi`$(2, B, C, A)$



Bài tập 16

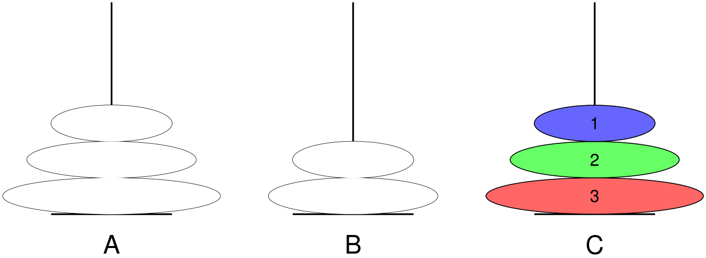
(a) Chứng minh tính đúng đắn của thuật toán giải bài toán Tháp Hà Nội

(b) Chứng minh rằng số bước di chuyển tối thiểu để giải quyết bài toán với n đĩa là $2^n - 1$

Ví dụ 16 (Minh họa giải thuật Tháp Hà Nội với 3 đĩa)

Bước 7: Di chuyển đĩa 1 từ A sang C

`Hanoi`(1, *A*, *C*, *B*)



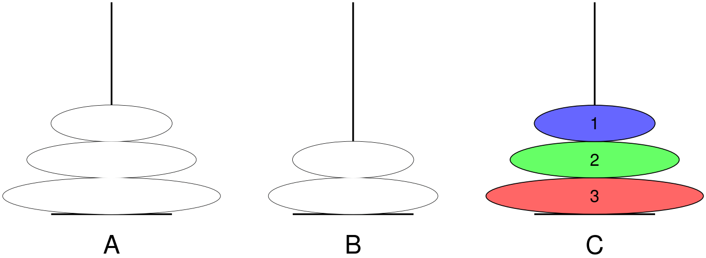
Bài tập 16

(a) Chứng minh tính đúng đắn của thuật toán giải bài toán Tháp Hà Nội

(b) Chứng minh rằng số bước di chuyển tối thiểu để giải quyết bài toán với n đĩa là $2^n - 1$

Ví dụ 16 (Minh họa giải thuật Tháp Hà Nội với 3 đĩa)

Hoàn thành! Tất cả các đĩa đã được chuyển từ cọc A sang cọc C
Total: $2^3 - 1 = 7$ bước



Bài tập 16

(a) Chứng minh tính đúng đắn của thuật toán giải bài toán Tháp Hà Nội

(b) Chứng minh rằng số bước di chuyển tối thiểu để giải quyết bài toán với n đĩa là $2^n - 1$

Một số thuật toán đệ quy khác

Sắp xếp nhanh (Quicksort)

Bài toán: Sắp xếp một mảng các phần tử theo thứ tự tăng dần.

Ý tưởng:

- Chọn một phần tử trong mảng làm *chốt (pivot)*
- Phân hoạch mảng thành hai phần: các phần tử nhỏ hơn pivot và các phần tử lớn hơn pivot
- Đặt pivot vào đúng vị trí của nó
- Đệ quy sắp xếp hai phần trước và sau pivot

Một số thuật toán đệ quy khác

Sắp xếp nhanh (Quicksort)

Thuật toán 11: Thuật toán sắp xếp nhanh (Quicksort)

Input: Mảng A và các chỉ số p và r
Output: Mảng A được sắp xếp trong đoạn từ p đến r

procedure QuickSort(A, p, r):
  // Sắp xếp mảng A từ vị trí p đến r
  if $p < r$ **then**
    // Điều kiện dừng: nếu mảng có ít nhất 2 phần tử
    $q \leftarrow$ Partition(A, p, r)   // q là vị trí cuối cùng của pivot
    QuickSort($A, p, q-1$)   // Đệ quy sắp xếp mảng con bên trái pivot
    QuickSort($A, q+1, r$)   // Đệ quy sắp xếp mảng con bên phải pivot

procedure Partition(A, p, r):
  // Phân hoạch mảng và trả về vị trí của pivot
  $x \leftarrow A[r]$ // Chọn pivot là phần tử cuối cùng của mảng
  $i \leftarrow p-1$ // i theo dõi vị trí cuối cùng của vùng nhỏ hơn pivot
  for $j \leftarrow p$ **to** $r-1$ **do**
    // Duyệt qua tất cả các phần tử trừ pivot
    if $A[j] \leq x$ **then**
      /* Nếu phần tử hiện tại không lớn hơn pivot */
      $i \leftarrow i+1$ // Mở rộng vùng các phần tử nhỏ hơn pivot
      Đổi chỗ $A[i]$ và $A[j]$ // Đưa phần tử nhỏ hơn pivot về đầu mảng
  Đổi chỗ $A[i+1]$ và $A[r]$ // Đặt pivot vào đúng vị trí của nó
  return $i+1$ // Trả về vị trí của pivot trong mảng đã phân hoạch

Ví dụ 17 (Minh họa thuật toán Quicksort)

4	1	3	5	2

Mảng ban đầu, pivot = 2

1	2	3	5	4

Sau phân hoạch: [1] pivot=2 [3,5,4]

1		3	5	4

Phần bên trái chỉ có 1 phần tử, đã sắp xếp

Phân hoạch phần bên phải, pivot = 4

3	4	5

Sau phân hoạch: [3] pivot=4 [5]

3	4	5

Phần bên phải đã sắp xếp

1	2	3	4	5

Mảng đã được sắp xếp hoàn chỉnh

Bài tập 17

(a) Chứng minh tính đúng đắn của thuật toán sắp xếp nhanh

(b) Phân tích độ phức tạp thời gian của thuật toán sắp xếp nhanh:
- Trường hợp tốt nhất: $O(n \log n)$
- Trường hợp xấu nhất: $O(n^2)$