VNU-HUS MAT3500: Toán rời rạc

Thuật toán I
Mô tả, chứng minh, đánh giá thuật toán; Tìm kiếm và sắp xếp

Hoàng Anh Đức

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

Nội dung

Định nghĩa và một số khái niệm

Mô tả thuật toán

- Mô tả bằng ngôn ngữ thông thường
- Mô tả bằng giả mã
- Mô tả bằng sơ đồ khối

Chứng minh thuật toán

- Bất biến vòng lặp

Độ phức tạp tính toán

- Độ tăng của các hàm
- Định nghĩa và khái niệm
- Độ phức tạp tính toán theo thời gian

Tìm kiếm và Sắp xếp

- Một số thuật toán tìm kiếm
  - Tìm kiếm tuyến tính
  - Tìm kiếm nhị phân
- Một số thuật toán sắp xếp
  - Sắp xếp nổi bọt
  - Sắp xếp chèn

Định nghĩa và một số khái niệm

- Một *thuật toán (algorithm)* là một tập hữu hạn các hướng dẫn cụ thể để thực hiện một nhiệm vụ nào đó
  - cộng hai số tự nhiên biểu diễn dưới dạng số thập phân
  - đăng ký môn học trực tuyến
  - đi từ nhà đến trường
- Một *chương trình máy tính (computer program)* là
  - một mô tả của thuật toán nào đó
  - sử dụng một ngôn ngữ đủ chuẩn xác để máy tính có thể hiểu
  - cùng với các phép toán mà máy tính đã biết cách thực hiện

  Ta nói rằng thuật toán được *cài đặt (implement)* cụ thể bằng chương trình máy tính
- Khi mở một phần mềm trong máy tính, ta nói rằng *chương trình hoặc thuật toán của nó được chạy hoặc được thực hiện bởi máy tính*
- Khi có mô tả của một thuật toán, bạn cũng *có thể thực hiện từng bước của thuật toán với giấy và bút*

Định nghĩa và một số khái niệm

Một số tính chất của một thuật toán

Đầu vào (Input) Một thuật toán có các giá trị đầu vào từ một tập đã được xác định trước

Đầu ra (Output) Từ mỗi một tập các giá trị đầu vào, một thuật toán sinh ra các giá trị đầu ra. Các giá trị này chính là lời giải cho bài toán

Tính xác định (Definiteness) Các bước của một thuật toán cần phải được xác định một cách chính xác

Tính đúng đắn (Correctness) Với mỗi tập giá trị đầu vào, một thuật toán cần cho ra kết quả đầu ra đúng

Tính hữu hạn (Finiteness) Với mỗi tập giá trị đầu vào, một thuật toán cần cho ra các giá trị đầu ra mong muốn sau một số hữu hạn (có thể là rất lớn) các bước

Tính hiệu quả (Effectiveness) Mỗi bước của thuật toán cần được thực hiện một cách chính xác và trong thời gian hữu hạn

Tính tổng quát (Generality) Thuật toán phải áp dụng được cho mọi bài toán mong muốn, chứ không phải chỉ với một tập các giá trị đầu vào đặc biệt

Định nghĩa và một số khái niệm

Một *mô tả đầy đủ của một thuật toán* bao gồm ba phần

(1) *Thuật toán (algorithm)*

- Mô tả một cách rõ ràng và chính xác nhất có thể
  - Một *thuật toán* có thể được *mô tả bằng một ngôn ngữ máy tính* (C, Python, Java, v.v...). Tuy nhiên, những mô tả này cần tuân theo các chỉ dẫn cụ thể trong ngôn ngữ máy tính tương ứng. Điều này dẫn đến việc các mô tả theo phương pháp này thường phức tạp và khó hiểu
  - Thay vì dùng một ngôn ngữ máy tính cụ thể để mô tả thuật toán, ta sử dụng *ngôn ngữ thông thường (natural language)*, *giả mã (pseudocode)*, hoặc *sơ đồ khối (flowchart)*
- Thường kèm theo mô tả ngắn gọn về ý tưởng của thuật toán

(2) Một chứng minh về *tính đúng đắn (correctness)* của thuật toán

- Với mọi tập đầu vào hợp lệ, thuật toán cần cho kết quả đầu ra đúng

(3) Một phân tích về *hiệu năng (performance)* của thuật toán

- Thời gian thực thi, không gian bộ nhớ, v.v...

Mô tả thuật toán

Mô tả bằng ngôn ngữ thông thường

Sử dụng *ngôn ngữ thông thường (natural language)* để mô tả từng bước thực hiện thuật toán

- Bước 1 Thực hiện việc X
- Bước 2 Tính Y
- Bước 3 Lặp lại Z
- Bước 4 ...

Ví dụ 1 (Mô tả bằng ngôn ngữ thông thường)

- **Bài toán:**
  - **Input:** Dãy số nguyên $a_1, a_2, \dots, a_n$
  - **Output:** Giá trị của phần tử lớn nhất trong dãy
- **Tìm giá trị của phần tử lớn nhất:**

Bước 1 Gán biến v (lưu giá trị lớn nhất hiện tại) bằng a_1

Bước 2 Lần lượt xét các phần tử $a_2, a_3, \dots, a_n$:

a) Nếu phần tử đang xét lớn hơn v, cập nhật v bằng giá trị của phần tử đó

b) Ngược lại, giữ nguyên giá trị của v

Bước 3 Sau khi xét hết tất cả các phần tử, trả về giá trị của v (chính là phần tử lớn nhất trong dãy)

Ví dụ 2 (Thực hiện thuật toán)

- **Input:** Dãy $a_1 = 7, a_2 = 12, a_3 = 5, a_4 = 16, a_5 = 9$
- **Output:** Giá trị của phần tử lớn nhất trong dãy

a_1	a_2	a_3	a_4	a_5
$\boxed{7}$	$\boxed{12}$	$\boxed{5}$	$\boxed{16}$	$\boxed{9}$
	$i = 2$	$i = 3$	$i = 4$	$i = 5$
$v = 7$	$v = 12$	$v = 12$	$v = 16$	$v = 16$

Mô tả thuật toán

Mô tả bằng giả mã

- *Giả mã (pseudocode)* là một dạng *hỗn hợp giữa ngôn ngữ thông thường và ngôn ngữ lập trình*
- Một *biến (variable)* được sử dụng để biểu diễn vị trí trong bộ nhớ máy tính để lưu trữ một giá trị. Khi ta nói đến một biến X nào đó, trên thực tế, chúng ta muốn sử dụng giá trị lưu tại một vị trí nào đó trong bộ nhớ ứng với X

Phép gán (assignment)

variable := expression

- Tính toán expression
- Lưu kết quả vào (vị trí trong bộ nhớ ứng với) biến variable
- Chú ý rằng $=$ và $:=$ khác nhau. Do đó, trong nhiều tài liệu, ký hiệu $\leftarrow$ được sử dụng thay thế cho $:=$.

Cấu trúc điều kiện (conditional statement)

if condition **then**
    S_1
else
    S_2

- Tính condition; trả lại `True` hoặc `False`
- Nếu `True`, thực hiện S_1; nếu `False`, thực hiện S_2
- Sau đó tiếp tục với các lệnh tiếp theo sau cấu trúc điều kiện

*Vòng lặp **for** (**for** loop)*

for variable := initial_value **to** final_value **do**
    S

- Khởi tạo variable với giá trị initial_value
- Nếu variable $\leq$ final_value, thực hiện đoạn mã S
- Sau mỗi lần lặp, tăng variable thêm 1
- Kiểm tra lại điều kiện; nếu vẫn thỏa mãn, tiếp tục lặp
- Khi variable $>$ final_value, thoát vòng lặp, thực hiện các lệnh tiếp theo ngay sau vòng **for**

*Vòng lặp **while** (**while** loop)*

while condition **do**
    S

- Kiểm tra giá trị của condition
- Nếu condition là `True`, thực hiện đoạn mã S
- Sau khi thực hiện S, kiểm tra lại condition
- Nếu vẫn `True`, thực hiện S lần nữa
- Khi condition trở thành `False`, thoát khỏi vòng lặp và thực hiện các lệnh tiếp theo ngay sau vòng **while**

*Vòng lặp **do-while** (**do-while** loop)*

do
  S
while condition

- Thực hiện đoạn mã S trước
- Sau đó, kiểm tra giá trị của condition
- Nếu condition là `True`, thực hiện S lần nữa
- Tiếp tục lặp lại quá trình cho đến khi condition trở thành `False`
- Khác với vòng lặp **while**, vòng lặp **do-while** luôn thực hiện S ít nhất một lần
- Một số giáo trình đề cập cấu trúc **repeat...until** tương tự, với **until** sử dụng điều kiện ngược lại để dừng lặp

*Lệnh trả về (**return** statement)*

return expression

- Trả về giá trị của expression làm kết quả đầu ra của thuật toán
- Khi gặp lệnh **return**, thuật toán sẽ kết thúc ngay lập tức
- Nếu **return** nằm trong một hàm, nó sẽ trả về giá trị cho nơi gọi hàm và kết thúc việc thực thi hàm

Nhận xét và chú thích (comments)

- Nhận xét là văn bản giải thích mà chương trình sẽ bỏ qua khi thực thi
- Được sử dụng để làm rõ code hoặc giải thích logic
- Các dạng phổ biến:
  - `// Nhận xét một dòng`
  - `/* Nhận xét nhiều dòng */`
  - ⟪ Nhận xét ⟫

Ví dụ 3 (Mô tả thuật toán bằng giả mã)

Thuật toán 1: Tìm giá trị của phần tử lớn nhất

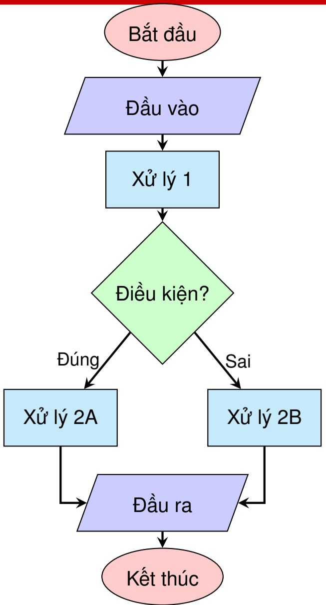
Input: $a_1, a_2, \dots, a_n$: dãy số nguyên
Output: Giá trị của phần tử lớn nhất trong dãy

$v := a_1$ `// phần tử lớn nhất đến hiện tại`
for $i := 2$ **to** n **do** `// lần lượt xét` $a_2, \dots, a_n$
  if $a_i > v$ **then** `//` a_i `> phần tử lớn nhất hiện tại?`
    $v := a_i$ `// bây giờ` v `lớn nhất trong` $a_1, \dots, a_i$
return v

Mô tả thuật toán

Mô tả bằng sơ đồ khối

- *Sơ đồ khối (flowchart)* là một loại biểu đồ biểu diễn một luồng công việc hoặc quy trình
- Hiển thị các bước dưới dạng các hộp hình dạng khác nhau
- Thứ tự thực hiện được kết nối bằng các mũi tên
- Các thành phần chính:
  - Bắt đầu/Kết thúc (hình elip)
  - Xử lý (hình chữ nhật)
  - Điều kiện (hình thoi)
  - Đầu vào/Đầu ra (hình thang)

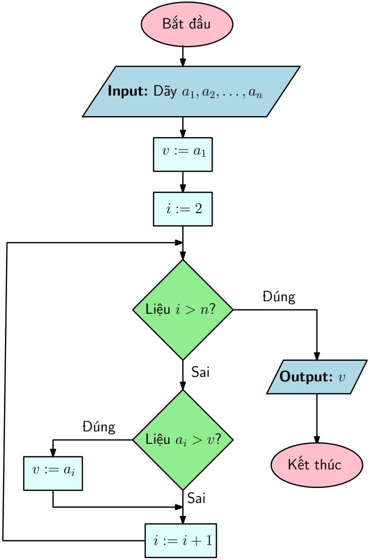


Ví dụ 4 (Mô tả thuật toán bằng sơ đồ khối)

Bài toán:

- **Input:** Dãy số nguyên $a_1, a_2, \ldots, a_n$
- **Output:** Giá trị của phần tử lớn nhất trong dãy

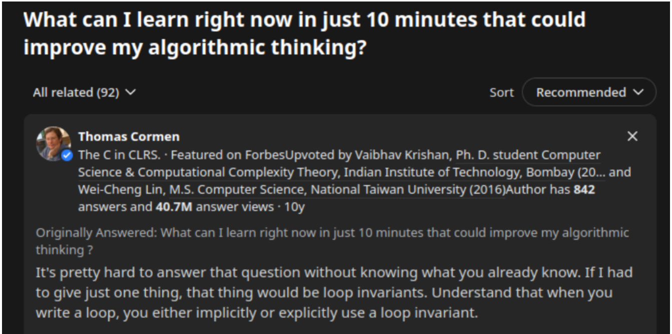
Ý tưởng: Duyệt qua từng phần tử của dãy, lưu giữ giá trị lớn nhất đã tìm thấy cho đến hiện tại.



Chứng minh thuật toán

Bất biến vòng lặp

- Có rất nhiều phương pháp khác nhau để *chứng minh tính đúng đắn của một thuật toán*
- Một trong số đó là sử dụng *bất biến vòng lặp (loop invariant)*—một phương pháp được xây dựng dựa trên phương pháp quy nạp toán học
  - *Vòng lặp (loop):* **for**, **while**, v.v...
  - Một *bất biến vòng lặp* là *một phát biểu luôn đúng trước và sau mỗi lần lặp (iteration) của một vòng lặp (loop)*

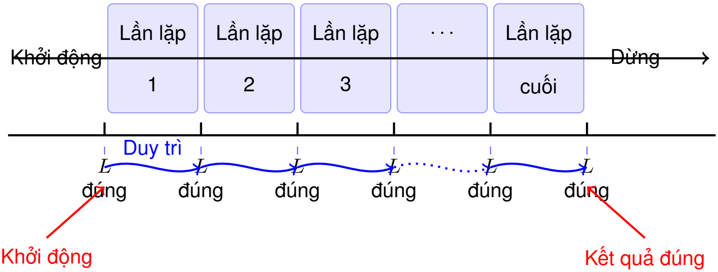


Một bất biến vòng lặp L thỏa mãn ba tính chất:

Khởi động (Initialization) L đúng trước lần lặp đầu tiên của vòng lặp

Duy trì (Maintainance) Nếu L đúng trước một lần lặp của vòng lặp thì nó cũng đúng trước lần lặp tiếp theo

Dừng (Termination) Khi vòng lặp dừng, bất biến vòng lặp cho ta một tính chất hữu ích để chứng minh thuật toán đúng



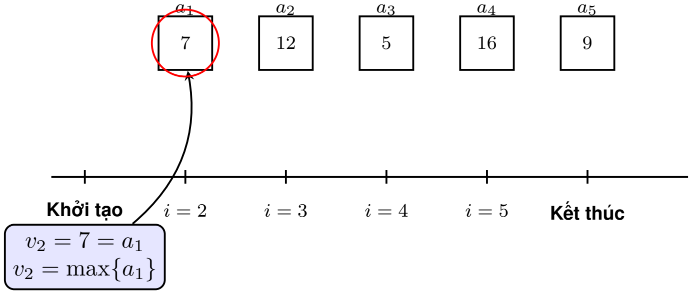
Hình: Bất biến vòng lặp

Ví dụ 5

Một bất biến vòng lặp trong Thuật toán 1 (vòng **for** ở Dòng 2–4) tìm giá trị lớn nhất trong dãy số nguyên $a_1, \dots, a_n$

$L =$ "*Ở trước lần lặp với biến* i, $v = \max\{a_1, a_2, \dots, a_{i-1}\}$"

Gọi v_i là *giá trị của* v *trước lần lặp với biến* i



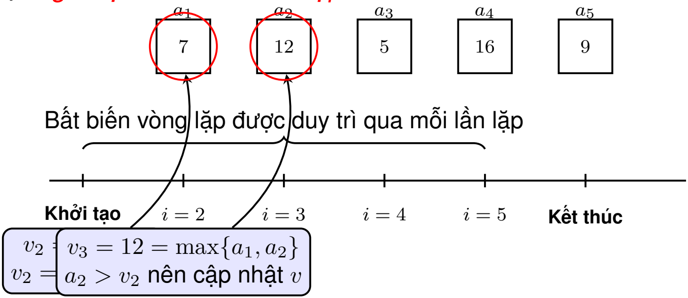
Khởi động: $v_2 = a_1 = \max\{a_1\}$

Ví dụ 5

Một bất biến vòng lặp trong Thuật toán 1 (vòng **for** ở Dòng 2–4) tìm giá trị lớn nhất trong dãy số nguyên $a_1, \dots, a_n$

$L =$ "*Ở trước lần lặp với biến* i, $v = \max\{a_1, a_2, \dots, a_{i-1}\}$"

Gọi v_i là *giá trị của* v *trước lần lặp với biến* i



Khởi động: $v_2 = a_1 = \max\{a_1\}$

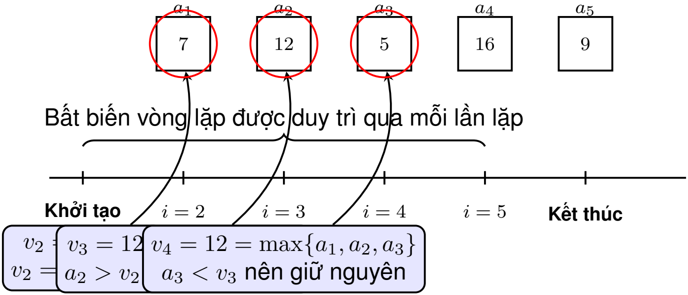
Duy trì:
Trước $i = k$: $v_k = \max\{a_1, \dots, a_{k-1}\}$
Trước $i = k + 1$: $v_{k+1} = \max\{a_1, \dots, a_k\}$

Ví dụ 5

Một bất biến vòng lặp trong Thuật toán 1 (vòng **for** ở Dòng 2–4) tìm giá trị lớn nhất trong dãy số nguyên $a_1, \dots, a_n$

$$L = \text{“}\textit{Ở trước lần lặp với biến } i,\ v = \max\{a_1, a_2, \dots, a_{i-1}\}\text{”}$$

Gọi v_i là *giá trị của v trước lần lặp với biến i*



Khởi động: $v_2 = a_1 = \max\{a_1\}$

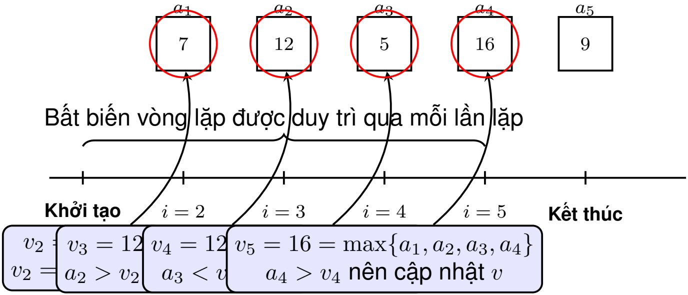
Duy trì:
Trước $i = k$: $v_k = \max\{a_1, \dots, a_{k-1}\}$
Trước $i = k+1$: $v_{k+1} = \max\{a_1, \dots, a_k\}$

Ví dụ 5

Một bất biến vòng lặp trong Thuật toán 1 (vòng **for** ở Dòng 2–4) tìm giá trị lớn nhất trong dãy số nguyên $a_1, \dots, a_n$

$$L = \text{“}\textit{Ở trước lần lặp với biến } i,\ v = \max\{a_1, a_2, \dots, a_{i-1}\}\text{”}$$

Gọi v_i là *giá trị của v trước lần lặp với biến i*



Khởi động: $v_2 = a_1 = \max\{a_1\}$

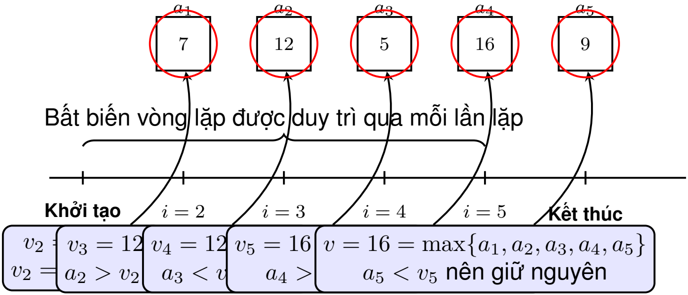
Duy trì:
Trước $i = k$: $v_k = \max\{a_1, \dots, a_{k-1}\}$
Trước $i = k+1$: $v_{k+1} = \max\{a_1, \dots, a_k\}$

Ví dụ 5

Một bất biến vòng lặp trong Thuật toán 1 (vòng **for** ở Dòng 2–4) tìm giá trị lớn nhất trong dãy số nguyên $a_1, \dots, a_n$

$L =$ "*Ở trước lần lặp với biến* i, $v = \max\{a_1, a_2, \dots, a_{i-1}\}$"

Gọi v_i là *giá trị của v trước lần lặp với biến i*



Khởi động: $v_2 = a_1 = \max\{a_1\}$

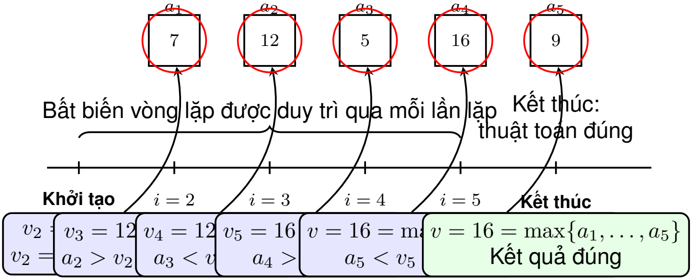
Duy trì:
Trước $i = k$: $v_k = \max\{a_1, \dots, a_{k-1}\}$
Trước $i = k + 1$: $v_{k+1} = \max\{a_1, \dots, a_k\}$

Ví dụ 5

Một bất biến vòng lặp trong Thuật toán 1 (vòng **for** ở Dòng 2–4) tìm giá trị lớn nhất trong dãy số nguyên $a_1, \dots, a_n$

$L =$ "*Ở trước lần lặp với biến* i*,* $v = \max\{a_1, a_2, \dots, a_{i-1}\}$"

Gọi v_i là *giá trị của* v *trước lần lặp với biến* i



Khởi động: $v_2 = a_1 = \max\{a_1\}$

Duy trì: Trước $i = k$: $v_k = \max\{a_1, \dots, a_{k-1}\}$
Trước $i = k+1$: $v_{k+1} = \max\{a_1, \dots, a_k\}$

Dừng: $v = \max\{a_1, \dots, a_n\}$

Chứng minh tính chất của bất biến vòng lặp.

$$L = \text{“}\textit{Ở trước lần lặp với biến } i\text{, } v = \max\{a_1, a_2, \dots, a_{i-1}\}\text{”}$$

Gọi v_i là giá trị của v trước lần lặp với biến i

- **Khởi động ($i = 2$):** Ta cần chỉ ra rằng trước vòng **for**, $v_2 = \max\{a_1\} = a_1$, và điều này hiển nhiên đúng do Thuật toán 1 gán v bằng a_1 ở Dòng 1
- **Duy trì:** Giả sử L đúng ở trước lần lặp với $i = k$ nào đó, nghĩa là $v_k = \max\{a_1, \dots, a_{k-1}\}$. Ta chứng minh L đúng ở trước lần lặp với $i = k + 1$, nghĩa là $v_{k+1} = \max\{a_1, \dots, a_{k-1}, a_k\}$. Ta xét các trường hợp dựa trên điều kiện ở Dòng 3
  - Nếu $a_k > v = v_k$ sai, giá trị của v không thay đổi, và do đó $v_{k+1} = v_k$. Ta có $\max\{a_1, \dots, a_{k-1}, a_k\} = \max\{v_k, a_k\} = v_k$. Suy ra $v_{k+1} = \max\{a_1, \dots, a_{k-1}, a_k\}$
  - Nếu $a_k > v = v_k$ đúng, giá trị của v được gán bằng a_k, và do đó $v_{k+1} = a_k$. Ta cũng có $\max\{a_1, \dots, a_{k-1}, a_k\} = \max\{v_k, a_k\} = a_k$. Suy ra $v_{k+1} = \max\{a_1, \dots, a_{k-1}, a_k\}$
- **Dừng:** Sau khi kết thúc lần lặp $i = n$ (hoặc, trước khi bắt đầu lần lặp $i = n + 1$ mà sẽ không bao giờ được thực hiện), $v = \max\{a_1, \dots, a_n\}$ và do đó là giá trị lớn nhất của các phần tử trong dãy đầu vào

□

Bài tập 1

Một thuật toán tính x^n với $x \in \mathbb{R}^+$ và $n \in \mathbb{N}$ được mô tả như sau

Thuật toán 2: Tính x^n.

Input: x: số thực dương, n: số tự nhiên
Output: Giá trị của x^n

```
answer := 1
m := n
while m > 0 do
    answer := answer · x
    m := m − 1
return answer
```

Hãy chứng minh phát biểu L sau là một bất biến vòng lặp cho vòng **while**

$$L = \text{“Ở trước lần lặp với biến } m,\ \textit{answer} = x^{n-m}\text{”}$$

Độ phức tạp tính toán

Độ tăng của các hàm

- *Phân tích thuật toán (algorithm analysis)* nghiên cứu cách đánh giá *hiệu quả* của thuật toán bằng cách ước lượng lượng *thời gian (time)* và *bộ nhớ (space)* cần thiết
- *Độ phức tạp (complexity)* của thuật toán là thước đo định lượng về tài nguyên (thời gian/bộ nhớ) mà thuật toán tiêu tốn
  - Thường được biểu diễn dưới dạng các hàm của *kích thước đầu vào (input size)*
  - Giúp ta so sánh và lựa chọn thuật toán hiệu quả nhất cho bài toán
- Với các hàm độ phức tạp $f : \mathbb{R} \to \mathbb{R}$ hoặc $f : \mathbb{N} \to \mathbb{R}$, ta cần hiểu rõ tốc độ tăng trưởng của chúng
  - Ta nói f *tăng nhanh hơn* g nếu với mọi giá trị x đủ lớn, $f(x) > g(x)$
  - Giúp ta đánh giá hiệu quả tương đối giữa các thuật toán khi kích thước đầu vào tăng lên
  - Là nền tảng để phân loại các thuật toán theo hiệu quả tiệm cận
- Để đánh giá và so sánh độ phức tạp một cách chính xác, ta sử dụng *ký hiệu O-lớn (big-O notation)* và các ký hiệu tiệm cận khác
  - Cho phép ta mô tả *tốc độ tăng trưởng (growth rate)* của các hàm độ phức tạp
  - Tập trung vào các thành phần chính ảnh hưởng đến tốc độ tăng trưởng, bỏ qua các yếu tố không quan trọng khi kích thước đầu vào lớn

Ký hiệu O-lớn

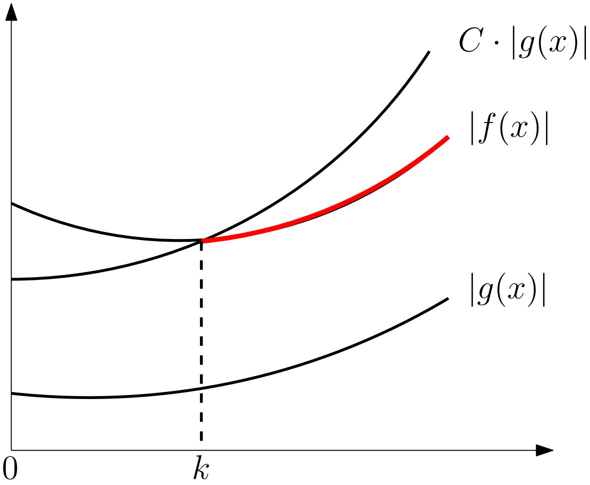
Cho f và g là các hàm $\mathbb{R} \to \mathbb{R}$. Ta nói rằng *$f$ là $O(g)$* (đọc là "f là O-lớn của g" hoặc "f thuộc lớp $O(g)$") nếu *tồn tại các hằng số C và k sao cho $|f(x)| \leq C|g(x)|$ với mọi $x > k$*

Ký hiệu O-lớn

Cho f và g là các hàm $\mathbb{R} \to \mathbb{R}$. Giả sử $g(x) \neq 0$ với $x \in \mathbb{R}$ đủ lớn. Ta nói rằng *f là $O(g)$* (đọc là "f là O-lớn của g" hoặc "f thuộc lớp $O(g)$") nếu $\lim_{x\to\infty} \frac{|f(x)|}{|g(x)|}$ *là hữu hạn*

- f là $O(g)$ nếu từ sau điểm k nào đó, giá trị của hàm f không vượt quá giá trị của một hằng số nhân với giá trị của hàm g. Ta cũng nói "f bị chặn trên bởi g"
- Các hằng số C và k được gọi là các *bằng chứng (witness)* cho mối liên hệ giữa f và g. *Để xác định f là $O(g)$, chỉ cần một cặp bằng chứng là đủ*

Độ phức tạp tính toán
Độ tăng của các hàm



f là $O(g)$

Ví dụ 6

Ta chứng minh hàm $f : \mathbb{R} \to \mathbb{R}$ cho bởi $f(x) = x^2 + 2x + 1$ là $O(g)$ với $g(x) = x^2$ (Ta cũng viết $x^2 + 2x + 1$ là $O(x^2)$)

Cách 1:

- Chú ý rằng khi $x > 1$, ta có $x < x^2$ và $1 < x^2$
- Do đó với mọi $x > 1$, ta có

$$|f(x)| = |x^2 + 2x + 1| \leq |x^2 + 2x^2 + x^2| = 4|x^2|$$

- Ta chọn $C = 4$ và $k = 1$

Cách 2: Do $\lim_{x\to\infty} \frac{|f(x)|}{|g(x)|} = \lim_{x\to\infty} \frac{|x^2 + 2x + 1|}{|x^2|} = 1$, ta có $f = O(g)$

Chú ý về quy tắc chia (division law of limit)

$$\lim_{x \to a} \frac{f(x)}{g(x)} = \frac{\lim_{x \to a} f(x)}{\lim_{x \to a} g(x)}$$

chỉ đúng khi $\lim_{x \to a} f(x)$ và $\lim_{x \to a} g(x)$ đều tồn tại[a] và $\lim_{x \to a} g(x) \neq 0$

[a] Nghĩa là các giới hạn này là các số hữu hạn

Nhắc lại: Quy tắc L'Hospital (L'Hospital's rule)

Nếu $\lim_{x \to \infty} f(x) = \infty$ và $\lim_{x \to \infty} g(x) = \infty$, thì $\lim_{x \to \infty} \frac{f(x)}{g(x)} = \lim_{x \to \infty} \frac{f'(x)}{g'(x)}$, trong đó $f'(x)$ và $g'(x)$ lần lượt là đạo hàm của $f(x)$ và $g(x)$

Ví dụ 7

Ta chứng minh x^2 là $O(2^x)$ bằng cách sử dụng định nghĩa O-lớn theo giới hạn. Ta có

$$\begin{aligned}\lim_{x\to\infty}\frac{|x^2|}{|2^x|} &= \lim_{x\to\infty}\frac{x^2}{2^x} \\ &= \lim_{x\to\infty}\frac{2x}{2^x\cdot\ln 2} && \text{Quy tắc L'Hospital} \\ &= \frac{2}{\ln 2}\lim_{x\to\infty}\frac{x}{2^x} \\ &= \frac{2}{\ln 2}\lim_{x\to\infty}\frac{1}{2^x\cdot\ln 2} && \text{Quy tắc L'Hospital} \\ &= 0.\end{aligned}$$

Do đó, $x^2 = O(2^x)$

Chú ý

Nếu không đề cập gì thêm thì $\log(n) = \log_2(n)$

Bài tập 2

Chứng minh

(a) $7x$ là $O(x^3)$
(b) x^3 không là $O(x^2)$
(c) $1 + 2 + \cdots + n$ là $O(n^2)$
(d) $n! = 1 \times 2 \times \cdots \times n$ là $O(n^n)$
(e) $\log(n!)$ là $O(n \log n)$
(f) n^3 là $O(2^n)$
(g) $\log n$ là $O(n)$
(h) Với các hằng số $b > 1$ và $k > 0$, $\log_b(n^k)$ là $O(\log n)$

Bài tập 3

Hãy giải thích một hàm $f : \mathbb{R} \to \mathbb{R}$ là $O(1)$ nghĩa là gì

Bài tập 4

Chứng minh rằng

(a) x^3 là $O(x^4)$ nhưng x^4 không là $O(x^3)$

(b) $3x^4 + 1$ là $O(x^4/2)$ và $x^4/2$ là $O(3x^4 + 1)$

(c) $x \log x$ là $O(x^2)$ nhưng x^2 không là $O(x \log x)$

(d) 2^n là $O(3^n)$ nhưng 3^n không là $O(2^n)$

Bài tập 5

Chứng minh rằng nếu $f(x)$ là $O(x)$ thì $f(x)$ cũng là $O(x^2)$

Bài tập 6

Chứng minh hoặc tìm phản ví dụ cho phát biểu: Nếu $f_(x)$ là $O(g_1(x))$ và $f_2(x)$ là $O(g_2(x))$ thì $f_1(x) - f_2(x)$ là $O(g_1(x) - g_2(x))$

Bài tập 7

Chứng minh rằng nếu $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$ với $a_0, a_1, \ldots, a_n$ là các số thực (nghĩa là, $f(x)$ là một đa thức bậc n) thì f là $O(x^n)$

- Ký hiệu $f(n) = O(g(n))$ thường được sử dụng để chỉ $f(n)$ là $O(g(n))$
  - Ký hiệu này không hoàn toàn chặt chẽ về mặt toán học, do $f(n)$ là một hàm còn *$O(g(n))$ là một tập hợp các hàm*
  - $f(n) = O(g(n))$ trên thực tế nghĩa là $f(n) \in O(g(n))$, do đó có thể viết $n = O(n^2)$ nhưng *không nên viết $O(n^2) = n$*
- Bạn có thể gặp biểu thức dạng "$f(n) + O(g(n)) = O(h(n))$"
  - Dấu "$=$" ở đây nghĩa là "$\subseteq$". Cụ thể, biểu thức trên cần được hiểu là tập hợp S gồm các hàm $f(n) + g_1(n)$ với $g_1(n) \in O(g(n))$ là tập con của tập $O(h(n))$
- Bạn có thể gặp biểu thức dạng "$f(n) \leq g(n) + O(h(n))$ với mọi $n \geq 0$" hoặc tương tự
  - Nghĩa là tồn tại $e(n)$ sao cho (a) $f(n) \leq g(n) + e(n)$ với mọi $n \geq 0$ và (b) $e(n) \in O(h(n))$
- Một số tác giả định nghĩa O-lớn bằng cách thay điều kiện $|f(x)| \leq C|g(x)|$ bằng $0 \leq f(x) \leq C(g(x))$. (Làm việc với giá trị tuyệt đối và khả năng các hàm $f(x)$ và $g(x)$ có thể nhận giá trị âm thường khó hơn là chỉ làm việc với các hàm nhân giá trị dương.) Định nghĩa theo cách này không hoàn toàn chặt chẽ. Ví dụ như hàm $\log x$ có thể nhận giá trị âm với x nhỏ

Bài tập 8

Giải thích ý nghĩa của các ký hiệu sau:

(a) $n^{O(k)}$

(b) $n^2 + O(n)$

(c) $n^2 + O(n) = O(n^2)$

Bài tập 9

Sử dụng ký hiệu O-lớn để đánh giá liên hệ giữa các cặp hàm sau. Giải thích kết quả của bạn

(a) n^{124} và 1.24^n

(b) $\sqrt[124]{n}$ và $(\log n)^{124}$

(c) $n \log n$ và $n^{\log n}$

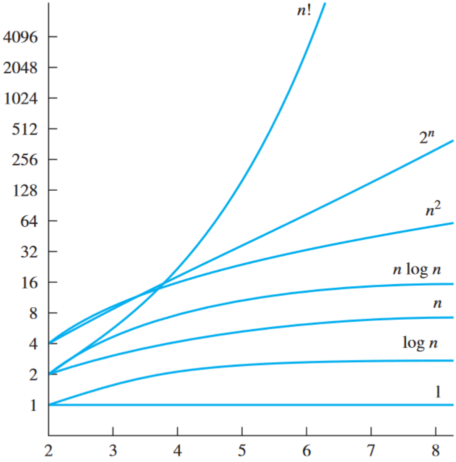
(d) $\sqrt{n}$ và $2^{\sqrt{\log n}}$

(e) $\sqrt{n}$ và $n^{\sin(n)}$

(f) $(n + \log n)^2$ và $n^2 + n \log n$

Độ phức tạp tính toán

Độ tăng của các hàm



Hình: Độ tăng của một số hàm thường dùng khi đánh giá với ký hiệu O-lớn [Rosen 2012]

Một số xấp xỉ hữu ích

(1) Nếu $d > c > 1$, thì n^c là $O(n^d)$, nhưng n^d không là $O(n^c)$

(2) Nếu $b > 1$ và c, d là các số dương, thì $(\log_b n)^c$ là $O(n^d)$ nhưng n^d không là $O((\log_b n)^c)$

(3) Nếu $b > 1$ và d là số dương, thì n^d là $O(b^n)$ nhưng b^n không là $O(n^d)$

(4) Nếu $c > b > 1$, thì b^n là $O(c^n)$ nhưng c^n không là $O(b^n)$

Một số tính chất quan trọng

(a) Nếu $f_1(x) = O(g_1(x))$ và $f_2(x) = O(g_2(x))$ thì $(f_1 + f_2)(x) = O(g(x))$ trong đó $g(x) = \max\{|g_1(x)|, |g_2(x)|\}$ với mọi $x \in \mathbb{R}$

(b) Nếu $f_1(x) = O(g_1(x))$ và $f_2(x) = O(g_2(x))$ thì $(f_1 f_2)(x) = O(g_1(x) g_2(x))$

Bài tập 10

Ước lượng theo O-lớn hàm $f(n) = 3n \log n! + (n^2 + 3) \log n$

Bài tập 11

Sắp xếp các hàm sau trong một danh sách sao cho mỗi hàm là O-lớn của hàm tiếp theo

(a) $\sqrt{n}$, $1000\log n$, $n\log n$, $2n!$, 2^n, 3^n, và $\dfrac{n^2}{1000000}$

(b) $(1.5)^n$, n^{100}, $(\log n)^3$, $\sqrt{n}\log n$, 10^n, $(n!)^2$, và $n^{99}+n^{98}$

Bài tập 12

Giả sử bạn có hai thuật toán khác nhau để giải một bài toán. Để giải một bài toán có kích thước n,

(1) thuật toán thứ nhất sử dụng đúng $n(\log n)$ phép toán và thuật toán thứ hai sử dụng đúng $n^{3/2}$ phép toán;

(2) thuật toán thứ nhất sử dụng đúng $n^2 2^n$ phép toán và thuật toán thứ hai sử dụng đúng $n!$ phép toán.

Khi n tăng, trong mỗi trường hợp trên, thuật toán nào sử dụng ít phép toán hơn?

Bài tập 13

Hãy đưa ra ước lượng O-lớn chính xác nhất có thể cho mỗi hàm sau

(a) $(n^2 + 8)(n + 1)$

(b) $(n \log n + n^2)(n^3 + 2)$

(c) $(n! + 2^n)(n^3 + \log(n^2 + 1))$

(d) $(n^3 + n^2 \log n)(\log n + 1) + (17 \log n + 19)(n^3 + 2)$

(e) $(2n + n^2)(n^3 + 3n)$

(f) $(n^n + n2^n + 5^n)(n! + 5n)$

(g) $n \log(n^2 + 1) + n^2 \log n$

(h) $(n \log n + 1)^2 + (\log n + 1)(n^2 + 1)$

(i) $n^{2^n} + n^{n^2}$

Độ phức tạp tính toán

Độ tăng của các hàm

Bài tập 14

Giả sử $T(1) = O(1)$ và $T(n) = T(n-1) + O(1)$ với mọi $n > 1$. Chứng minh sau sai ở đâu? Hãy đánh giá $T(n)$ theo ký hiệu O-lớn và giải thích đánh giá của bạn.

Giả sử $P(n)$ là phát biểu $T(n) = O(1)$. Ta chứng minh $\forall n \in \mathbb{Z}^+ \; P(n)$ bằng quy nạp.

- **Bước cơ sở:** $P(1)$ đúng vì $T(1) = O(1)$.
- **Bước quy nạp:** Giả sử $P(k)$ đúng với số nguyên $k \geq 1$ nào đó, nghĩa là $T(k) = T(k-1) + O(1)$. Ta chứng minh $P(k+1)$ đúng, nghĩa là chứng minh $T(k+1) = T(k) + O(1)$. Thật vậy, ta có:

$$\begin{aligned}
T(k+1) &= T(k) + O(1) && \text{giả thiết} && (1)\\
&= (T(k-1) + O(1)) + O(1) && \text{giả thiết quy nạp} && (2)\\
&= T(k-1) + O(1) + O(1) && && (3)\\
&= T(k-1) + O(1). && && (4)
\end{aligned}$$

  Do đó, $P(k+1)$ đúng.

Vậy theo nguyên lý quy nạp, ta có $P(n)$ đúng với mọi $n \in \mathbb{Z}^+$, tức là $T(n) = O(1)$ với mọi $n \in \mathbb{Z}^+$.

Ký hiệu Ω-lớn

Cho f và g là các hàm $\mathbb{R} \to \mathbb{R}$. Ta nói rằng *$f$ là $\Omega(g)$* nếu *tồn tại các hằng số $C > 0$ và k sao cho $|f(x)| \geq C|g(x)|$ với mọi $x > k$*

Ký hiệu Ω-lớn

Cho f và g là các hàm $\mathbb{R} \to \mathbb{R}$. Giả sử $g(x) \neq 0$ với $x \in \mathbb{R}$ đủ lớn. Ta nói rằng *f là $\Omega(g)$* nếu $\lim_{x\to\infty} \frac{|f(x)|}{|g(x)|}$ *khác* 0

Bài tập 15

Chứng minh rằng f là $\Omega(g)$ khi và chỉ khi g là $O(f)$

Chú ý

f là $O(g)$ theo nghĩa nào đó là *"độ tăng của $f \leq$ độ tăng của g"*. Tuy nhiên, bạn cần cẩn thận! *Với hai số thực $a, b \in \mathbb{R}$, các bất đẳng thức $a \leq b$ và $b \leq a$ không thể cùng sai. Nhưng tồn tại hàm f sao cho $f = O(g)$ và $f = \Omega(g)$ cùng sai*

Bài tập 16 ($\star$)

Tìm các ví dụ của các hàm $f : \mathbb{R} \to \mathbb{R}$ thỏa mãn các điều kiện (a) – (d) tương ứng

	$f(n)$ là $O(n^3)$	$f(n)$ không là $O(n^3)$
$f(n)$ là $\Omega(n^3)$	(a)	(b)
$f(n)$ không là $\Omega(n^3)$	(c)	(d)

Cụ thể, ở (a), bạn cần tìm ví dụ về một hàm $f(n)$ đồng thời là $O(n^3)$ và $\Omega(n^3)$ và chứng minh ví dụ bạn tìm ra là đúng. Tương tự cho các phần (b), (c), và (d)

Ký hiệu Θ-lớn

Cho f và g là các hàm $\mathbb{R} \to \mathbb{R}$. Ta nói rằng *$f$ là $\Theta(g)$* nếu *f là $O(g)$ và f là $\Omega(g)$*

Ký hiệu Θ-lớn

Cho f và g là các hàm $\mathbb{R} \to \mathbb{R}$. Giả sử $g(x) \neq 0$ với $x \in \mathbb{R}$ đủ lớn. Ta nói rằng *f là $\Theta(g)$* nếu $\lim_{x\to\infty} \frac{|f(x)|}{|g(x)|}$ *là hữu hạn và khác 0*

Bài tập 17

(a) Chứng minh rằng $1 + 2 + \cdots + n$ là $\Theta(n^2)$

(b) Các tập $O(1)$ và $\Theta(1)$ có bằng nhau không? Tại sao?

Bài tập 18

Chứng minh rằng với các hàm f, g từ $\mathbb{R}$ đến $\mathbb{R}$, f là $\Theta(g)$ khi và chỉ khi tồn tại các hằng số dương C_1, C_2, và k sao cho $C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$ với mọi $x > k$

Ký hiệu o-nhỏ

Cho f và g là các hàm $\mathbb{R} \to \mathbb{R}$. Ta nói rằng *$f$ là $o(g)$* nếu *với mọi $C > 0$ tồn tại k sao cho $|f(x)| < C|g(x)|$ với mọi $x > k$*

Ký hiệu o-nhỏ

Cho f và g là các hàm $\mathbb{R} \to \mathbb{R}$. Giả sử $g(x) \neq 0$ với $x \in \mathbb{R}$ đủ lớn. Ta nói rằng *f là $o(g)$* nếu $\lim_{n\to\infty} \frac{|f(x)|}{|g(x)|}$ *bằng 0*

Ký hiệu o-nhỏ

Cho f và g là các hàm $\mathbb{R} \to \mathbb{R}$. Ta nói rằng *$f$ là $o(g)$* nếu *f là $O(g)$ nhưng f không là $\Omega(g)$*

Ký hiệu ω-nhỏ

Cho f và g là các hàm $\mathbb{R} \to \mathbb{R}$. Ta nói rằng *$f$ là $\omega(g)$* nếu *với mọi $C > 0$ tồn tại k sao cho $|f(x)| > C|g(x)|$ với mọi $x > k$*

Ký hiệu ω-nhỏ

Cho f và g là các hàm $\mathbb{R} \to \mathbb{R}$. Giả sử $g(x) \neq 0$ với $x \in \mathbb{R}$ đủ lớn. Ta nói rằng *f là $\omega(g)$* nếu $\lim_{x\to\infty} \frac{|f(x)|}{|g(x)|}$ *bằng ∞*

Ký hiệu ω-nhỏ

Cho f và g là các hàm $\mathbb{R} \to \mathbb{R}$. Giả sử $g(x) \neq 0$ với $x \in \mathbb{R}$ đủ lớn. Ta nói rằng *f là $\omega(g)$* nếu *f là $\Omega(g)$ nhưng f không là $O(g)$*

Tóm lại, với các hàm f và g từ $\mathbb{R}$ đến $\mathbb{R}$

- f là $O(g)$ "f tăng không nhanh hơn g" tương tự "$\leq$"

$$\exists C, k \; \forall x > k \quad |f(x)| \leq C|g(x)|$$

- f là $\Omega(g)$ "f tăng ít nhất nhanh như g" tương tự "$\geq$"

$$\exists C > 0, k \; \forall x > k \quad |f(x)| \geq C|g(x)|$$

- f là $\Theta(g)$ "f tăng nhanh như g" tương tự "$=$"

$$\exists C_1 > 0, C_2 > 0, k \; \forall x > k \quad C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$$

- f là $o(g)$ "f tăng chậm hơn g" tương tự "$<$"

$$\forall C > 0 \, \exists k \, \forall x > k \quad |f(x)| < C|g(x)|$$

- f là $\omega(g)$ "f tăng nhanh hơn g" tương tự "$>$"

$$\forall C > 0 \, \exists k \, \forall x > k \quad |f(x)| > C|g(x)|$$

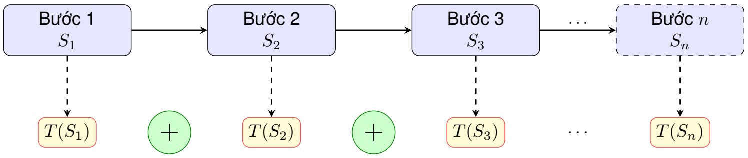
Độ phức tạp tính toán
Định nghĩa và khái niệm

- Một thuật toán tốt phải *luôn đúng về mặt kết quả* và *hiệu quả về mặt tài nguyên*
- *Độ phức tạp (complexity)* đo lường mức độ khó khăn của một tính toán dựa trên tài nguyên cần thiết:
  - *Độ phức tạp theo thời gian (time complexity):* Số lượng các phép toán cơ bản hoặc số bước thực hiện
  - *Độ phức tạp theo không gian (space complexity):* Lượng bộ nhớ (số bit) cần thiết để thực hiện tính toán
- Độ phức tạp của thuật toán thay đổi theo *kích thước đầu vào (input size)*
  - Ví dụ: Tìm kiếm một phần tử trong dãy có 100000 phần tử sẽ tốn nhiều tài nguyên hơn so với dãy có chỉ vài phần tử
- Do đó, độ phức tạp thường được biểu diễn dưới dạng một *hàm số* $f(n)$ với n là kích thước đầu vào
  - Hàm này giúp ta dự đoán tài nguyên cần thiết khi kích thước đầu vào tăng lên
- Khi phân tích thuật toán, chúng ta không quan tâm đến chính xác số lượng phép toán, mà chỉ cần *đánh giá tiệm cận (asymptotic estimate)* về mức độ tài nguyên tiêu thụ khi kích thước đầu vào tăng lên

- Các ký hiệu tiệm cận là những công cụ toán học thiết yếu để đặc trưng hóa và so sánh độ phức tạp các thuật toán
- Trong phân tích *độ phức tạp tính toán theo thời gian (time complexity)*, ta xem xét ba loại trường hợp:
  - *Độ phức tạp trường hợp xấu nhất (worst-case complexity):* Thời gian tối đa mà thuật toán có thể tiêu tốn với bất kỳ đầu vào nào có kích thước cụ thể—**thường được sử dụng phổ biến nhất trong phân tích thuật toán**
  - *Độ phức tạp trường hợp trung bình (average-case complexity):* Thời gian trung bình khi xét tất cả các đầu vào có khả năng xuất hiện với cùng kích thước—**thường khó xác định**
  - *Độ phức tạp trường hợp tốt nhất (best-case complexity):* Thời gian tối thiểu với đầu vào thuận lợi nhất có kích thước cụ thể—**ít có giá trị thực tiễn trong đánh giá thuật toán**
- Trong thực hành, ta thường mô tả thời gian chạy của thuật toán theo *ký hiệu O-lớn* để đơn giản hóa việc phân tích. *Ký hiệu Θ-lớn* cung cấp đánh giá chính xác hơn về độ phức tạp thực sự nhưng đôi khi khó xác định
- Cần lưu ý rằng nhiều tài liệu sử dụng biểu thức "$f(n)$ là $O(g(n))$" khi thực chất họ đang muốn biểu diễn "$f(n)$ là $\Theta(g(n))$"

Độ phức tạp tính toán
Độ phức tạp tính toán theo thời gian

Độ phức tạp của một chuỗi tuần tự các bước là tổng của độ phức tạp của từng bước



Tổng độ phức tạp = $T(S_1) + T(S_2) + T(S_3) + \ldots + T(S_n)$

Thời gian thực hiện các lệnh gán (:=), trả lại (**return**) là $O(1)$, và giả sử thời gian thực hiện các phép toán cơ bản (cộng, trừ, nhân, chia, so sánh, v.v...) cũng là $O(1)$

- Trong thực tế, việc cộng hai số nguyên độ dài n bit biểu diễn dưới dạng số nhị phân có độ phức tạp $O(n)$ chứ không phải $O(1)$

Phép gán: $x := y$ → $O(1)$

Phép trả lại: **return** z → $O(1)$

Phép cộng: $a + b$ → $O(1)$

Phép nhân: $c \times d$ → $O(1)$

Phép so sánh: $x < y$ → $O(1)$

Độ phức tạp tính toán
Độ phức tạp tính toán theo thời gian

Thời gian thực hiện cấu trúc **if...then...else** là thời gian lớn nhất thực hiện các lệnh sau **then** hoặc sau **else** cộng với thời gian kiểm tra điều kiện

if condition **then**
    S_1 // khối lệnh
else
    S_2 // khối lệnh

Kiểm tra điều kiện: T(condition)

Thời gian chạy S_1: $T(S_1)$

Thời gian chạy S_2: $T(S_2)$

Tổng thời gian = $T(\text{condition}) + \max\{T(S_1), T(S_2)\}$
= $\max\{T(\text{condition}) + T(S_1), T(\text{condition}) + T(S_2)\}$

Độ phức tạp tính toán
Độ phức tạp tính toán theo thời gian

Thời gian thực hiện vòng lặp là tổng thời gian thực hiện các lần lặp và thời gian kiểm tra điều kiện lặp. Nếu thời gian thực hiện mỗi lần lặp là giống nhau, thì tổng thời gian thực hiện các lần lặp là tích của số lần lặp và thời gian thực hiện mỗi lần lặp

for $i := 1$ **to** n **do**
    S // khối lệnh

Khởi tạo:
$T(i := 1)$ = $O(1)$

Thời gian chạy S ở lần lặp với i:
$T_i(S)$

Kiểm tra và tăng i:
$T(i \leq n, i++)$ = $O(1)$

Tổng thời gian = $T(i := 1) + \sum_{i=1}^{n}[T_i(S) + T(i \leq n, i++)]$

= $O(1) + \sum_{i=1}^{n} T_i(S) + n \times O(1) = O(1) + \sum_{i=1}^{n} T_i(S) + O(n)$

Độ phức tạp tính toán
Độ phức tạp tính toán theo thời gian

Thời gian thực hiện vòng lặp là tổng thời gian thực hiện các lần lặp và thời gian kiểm tra điều kiện lặp. Nếu thời gian thực hiện mỗi lần lặp là giống nhau, thì tổng thời gian thực hiện các lần lặp là tích của số lần lặp và thời gian thực hiện mỗi lần lặp

while condition **do**
    S // khối lệnh

Kiểm tra điều kiện: T(condition)

Thời gian chạy S ở lần lặp thứ i: $T_i(S)$

Tổng thời gian = $T(\text{condition}) + \sum_{i=1}^{k} [T(\text{condition}) + T_i(S)]$

Nếu $T_i(S) = T(S)$ với mọi i, ta có: $(k+1) \times T(\text{condition}) + k \times T(S)$

với k là số lần lặp thực tế của vòng lặp **while** $(k \geq 0)$

Chú ý: Nếu điều kiện sai ngay từ đầu $(k = 0)$, khối lệnh S không được thực hiện lần nào

Thời gian thực hiện vòng lặp là tổng thời gian thực hiện các lần lặp và thời gian kiểm tra điều kiện lặp. Nếu thời gian thực hiện mỗi lần lặp là giống nhau, thì tổng thời gian thực hiện các lần lặp là tích của số lần lặp và thời gian thực hiện mỗi lần lặp

do
    S // khối lệnh
while condition

Thời gian chạy S ở lần lặp thứ i: $T_i(S)$

Kiểm tra điều kiện: T(condition)

Tổng thời gian = $\sum_{i=1}^{k}[T_i(S) + T(\text{condition})]$

Nếu $T_i(S) = T(S)$ với mọi i, ta có: $k \times T(S) + k \times T(\text{condition})$ = $k \times [T(S) + T(\text{condition})]$

với k là số lần lặp thực tế của vòng lặp **do-while** ($k \geq 1$)

Chú ý: Khác vòng lặp **while**, vòng lặp **do-while** luôn thực hiện S ít nhất một lần

Độ phức tạp tính toán
Độ phức tạp tính toán theo thời gian

Bảng: Một số thuật ngữ thường dùng

Độ phức tạp	Thuật ngữ
$O(1)$	Độ phức tạp hằng số (constant complexity)
$O(\log n)$	Độ phức tạp lôgarit (logarithmic complexity)
$O(n)$	Độ phức tạp tuyến tính (linear complexity)
$O(n \log n)$	Độ phức tạp $n \log n$ (linearithmic complexity)
$O(n^b)$	Độ phức tạp đa thức (polynomial complexity)
$O(b^n)$, với $b > 1$	Độ phức tạp hàm mũ (exponential complexity)
$O(n!)$	Độ phức tạp giai thừa (factorial complexity)

Thuật toán 1: Tìm giá trị của phần tử lớn nhất

Input: $a_1, a_2, \dots, a_n$: dãy số nguyên

Output: Giá trị của phần tử lớn nhất trong dãy

$v := a_1$ — t_1

for $i := 2$ **to** n **do** — t_2

  if $a_i > v$ **then** — t_4^i

    $v := a_i$ — t_5

return v — t_3

$$T(n) = t_1 + t_2 + t_3$$

t_1, t_3 là $O(1)$

$$t_2 = O(1) + \sum_{i=2}^{n} t_4^i + O(n)$$

$$t_4^i = t_5 + (\text{t.g. kiểm tra } a_i > v) = O(1) + O(1) = O(1)$$

$\boxed{T(n) \text{ là } O(n)}$ (xấu nhất)

$\boxed{T(n) \text{ là } O(n)}$ (tốt nhất)

Bài tập 19

(a) Thiết kế thuật toán để tính tổng của tất cả các số hạng trong một dãy số nguyên $a_1, a_2, \ldots, a_n$ cho trước
(b) Chứng minh thuật toán bạn thiết kế là đúng
(c) Đánh giá thời gian chạy của thuật toán bạn thiết kế

Bài tập 20

Một chuỗi ký tự được gọi là *chuỗi đối xứng (palindrome)* khi viết từ trái qua phải và viết từ phải qua trái thì chuỗi không thay đổi. Một ví dụ là chuỗi `madam`.

(a) Thiết kế thuật toán để kiểm tra xem một chuỗi ký tự có phải là chuỗi đối xứng hay không
(b) Đánh giá thời gian chạy của thuật toán bạn thiết kế

Bài tập 21

Cho $f : A \to B$ là một hàm với các tập A, B là các tập con hữu hạn của $\mathbb{Z}$. Hãy thiết kế một thuật toán để kiểm tra xem

(a) liệu f có là đơn ánh không;
(b) liệu f có là toàn ánh không.

Trong mỗi trường hợp, hãy đánh giá thời gian chạy của thuật toán bạn thiết kế

Bài toán tìm kiếm

Cho một dãy n phần tử $a_1, a_2, \dots, a_n$ và một phần tử x. Tìm x trong dãy đã cho hoặc kết luận rằng x không có trong dãy

- Tìm kiếm tuyến tính (Linear Search)
- Tìm kiếm nhị phân (Binary Search)

- **Bài toán:**
  - **Input:** $a_1, \dots, a_n$: dãy số nguyên, x: số nguyên
  - **Output:** Chỉ số i thỏa mãn $x = a_i$ hoặc 0 nếu x không có trong dãy
- **Tìm kiếm tuyến tính:**
  (1) Bắt đầu từ phần tử đầu tiên của dãy (a_1)
  (2) So sánh phần tử hiện tại (a_i) với giá trị cần tìm x
  (3) Nếu $a_i = x$, trả về vị trí i và kết thúc thuật toán
  (4) Nếu $a_i \neq x$, chuyển sang phần tử tiếp theo (a_{i+1})
  (5) Lặp lại bước (2) đến (4) cho đến khi tìm thấy x hoặc đã duyệt hết dãy
  (6) Nếu đã duyệt hết dãy mà không tìm thấy x, trả về giá trị 0

Ví dụ 8 (Tìm kiếm tuyến tính)

- **Input:** Dãy $a_1 = 2, a_2 = 5, a_3 = 6, a_4 = 8, a_5 = 12$ và $x = 8$
- **Output:** Chỉ số i thỏa mãn $x = a_i$ hoặc 0 nếu x không có trong dãy

a_1	a_2	a_3	a_4	a_5
$\boxed{2}$	$\boxed{5}$	$\boxed{6}$	$\boxed{8}$	12
$i = 1$	$i = 2$	$i = 3$	$i = 4$	
$\neq x$	$\neq x$	$\neq x$	$= x$	

Thuật toán
Tìm kiếm tuyến tính

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

Input: $a_1, \ldots, a_n$: dãy số nguyên, x: số nguyên
Output: Chỉ số i thỏa mãn $x = a_i$ hoặc 0 nếu x không có trong dãy

$i := 1$ `// Bắt đầu từ đầu dãy`
while $i \leq n$ *và* $x \neq a_i$ **do** `// Chưa xong và chưa tìm thấy`
    $i := i + 1$ `// Đi tới vị trí tiếp theo trong dãy`
if $i \leq n$ **then**
    location $:= i$ `// Tìm thấy` x `trong dãy`
else
    location $:= 0$ `// Không tìm thấy` x `trong dãy`
return *location*

Một bất biến vòng lặp trong Thuật toán 3 (vòng **while** ở Dòng 2–3) tìm kiếm tuyến tính số nguyên x trong dãy $a_1, \dots, a_n$

$$L = \text{“}\textit{Ở trước lần lặp với biến } i, x \notin \{a_1, \dots, a_{i-1}\}\text{”}$$

- **Khởi động ($i = 1$):** Do $i - 1 = 0$, tập $\{a_1, \dots, a_{i-1}\}$ là tập rỗng, và do đó $x \notin \{a_1, \dots, a_{i-1}\}$, nghĩa là L đúng
- **Duy trì:** Giả sử L đúng ở trước lần lặp với $i = k$ nào đó, nghĩa là $x \notin \{a_1, \dots, a_{k-1}\}$. Ta chứng minh L đúng ở trước lần lặp với $i = k + 1$, nghĩa là $x \notin \{a_1, \dots, a_k\}$. Thật vậy, để thực hiện lần lặp $i = k$, điều kiện ở vòng **while** cần được thỏa mãn, nghĩa là $k \leq n$ và $x \neq a_k$. Kết hợp với giả thiết, ta có điều cần chứng minh
- **Dừng:** Vòng lặp **while** kết thúc khi $i = n + 1$ hoặc $x = a_i$ với $1 \leq i \leq n$. Với trường hợp đầu tiên, bất biến vòng lặp L cho ta $x \notin \{a_1, \dots, a_n\}$ và do đó kết luận không tìm được x. Với trường hợp thứ hai, x hiển nhiên thuộc dãy đã cho

Thuật toán
Tìm kiếm tuyến tính

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

Input: $a_1, \dots, a_n$: dãy số nguyên, x: số nguyên
Output: Chỉ số i thỏa mãn $x = a_i$ hoặc 0 nếu x không có trong dãy

$i := 1$ ——— t_1
while $i \le n$ và $x \neq a_i$ **do** ——— t_2
    $i := i + 1$ ——— t_5^i
if $i \le n$ **then** ——— t_3
    location $:= i$ ——— t_6
else
    location $:= 0$ ——— t_7
return *location* ——— t_4

$T(n) = t_1 + t_2 + t_3 + t_4$

t_1, t_4 là $O(1)$

$\boxed{T(n) \text{ là } O(n)}$ (xấu nhất)

$\boxed{T(n) \text{ là } O(1)}$ (tốt nhất)

$t_3 = \max\{t_6, t_7\}$ + (thời gian kiểm tra $i \le n$) $= O(1)$

$t_2 = \sum_{\{i | i \le n \wedge x \neq a_i\}} \left[t_5^i + (\text{t.g. kiểm tra } i \le n \text{ và } x \neq a_i) \right]$

- **Bài toán:**
  - **Input:** $a_1, \ldots, a_n$: dãy số nguyên *thực sự tăng*, x: số nguyên
  - **Output:** Chỉ số i thỏa mãn $x = a_i$ hoặc 0 nếu x không có trong dãy
- **Tìm kiếm nhị phân:** (Một ví dụ về kỹ thuật *chia để trị (divide and conquer)* trong thiết kế thuật toán)
  (1) Tính $m = \lfloor (1+n)/2 \rfloor$. Phần tử ở giữa của dãy là a_m
  (2) Chia dãy $a_1, \ldots, a_n$ thành hai dãy con (a) $a_1, \ldots, a_m$ và (b) $a_{m+1}, \ldots, a_n$. Nếu $x > a_m$ thì ta chỉ tìm x trong dãy con (b), còn ngược lại thì ta chỉ tìm x trong dãy con (a)
  (3) Làm tương tự cho đến khi không gian tìm kiếm chỉ còn một phần tử a_i. Nếu $x = a_i$ thì trả lại vị trí i của x, còn ngược lại thì trả lại 0

Ví dụ 9 (Tìm kiếm nhị phân)

- **Input:** Dãy $a_1 = 2, a_2 = 5, a_3 = 6, a_4 = 8, a_5 = 12$ và $x = 8$
- **Output:** Chỉ số i thỏa mãn $x = a_i$ hoặc 0 nếu x không có trong dãy

a_1	a_2	a_3	a_4	a_5
2	5	6	8	12
i		m		j
		$< x$		

a_1	a_2	a_3	a_4	a_5
2	5	6	8	12
			$i = m$	j
			$= x$	

a_1	a_2	a_3	a_4	a_5
2	5	6	8	12
			$i = j$	
			$= x$	

Thuật toán
Tìm kiếm nhị phân

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

Input: $a_1, \ldots, a_n$: dãy số nguyên *thực sự tăng*, x: số nguyên
Output: Chỉ số i thỏa mãn $x = a_i$ hoặc 0 nếu x không có trong dãy

$i := 1$ // Chỉ số bắt đầu khoảng tìm kiếm
$j := n$ // Chỉ số kết thúc khoảng tìm kiếm
while $i < j$ **do** // Khi khoảng tìm kiếm có > 1 phần tử
  $m := \lfloor (i+j)/2 \rfloor$ // Chỉ số của phần tử ở giữa
  if $x > a_m$ **then**
    $i := m + 1$
  else
    $j := m$
if $x = a_i$ **then**
  location $:= i$
else
  location $:= 0$
return *location*

Thuật toán
Tìm kiếm nhị phân

Một bất biến vòng lặp trong Thuật toán 4 (vòng **while** ở Dòng 3–8) tìm kiếm nhị phân số nguyên x trong dãy thực sự tăng $a_1, \dots, a_n$

$$L := \textit{Ở trước mỗi lần lặp với các biến } i, j, \textit{ nếu } x \in \{a_1, \dots, a_n\} \textit{ thì } x \in \{a_i, a_{i+1}, \dots, a_j\}$$

- **Khởi động ($i = 1, j = n$):** L hiển nhiên đúng
- **Duy trì:** Giả sử ở trước lần lặp với các biến k, ℓ, nếu $x \in \{a_1, \dots, a_n\}$ thì $x \in \{a_k, a_{k+1}, \dots, a_\ell\}$. Ta chứng minh rằng ở trước lần lặp kế tiếp với các biến (a) $k, \lfloor (k+\ell)/2 \rfloor$ hoặc (b) $\lfloor (k+\ell)/2 \rfloor + 1, \ell$, nếu $x \in \{a_1, \dots, a_n\}$ thì tương ứng (a') $x \in \{a_k, \dots, a_{\lfloor (k+\ell)/2 \rfloor}\}$ hoặc (b') $x \in \{a_{\lfloor (k+\ell)/2 \rfloor + 1}, \dots, a_\ell\}$. Với (a), điều kiện ở Dòng 7 cần được thỏa mãn, nghĩa là $x \leq a_{\lfloor (k+\ell)/2 \rfloor}$. Do đó, nếu $x \in \{a_k, a_{k+1}, \dots, a_\ell\}$ thì (a') đúng. Với (b), điều kiện ở Dòng 5 cần được thỏa mãn, nghĩa là $x > a_{\lfloor (k+\ell)/2 \rfloor}$. Do đó, nếu $x \in \{a_k, a_{k+1}, \dots, a_\ell\}$ thì (b') đúng
- **Dừng:** Vòng lặp **while** dừng khi $i = j$, và từ L, ta có nếu $x \in \{a_1, \dots, a_n\}$ thì $x \in \{a_i\}$. Do đó Thuật toán 4 trả lại vị trí chính xác của x hoặc kết luận không tìm được x (Dòng 9–13)

Độ phức tạp tính toán

Tìm kiếm nhị phân

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

Input: $a_1, \dots, a_n$: dãy số nguyên thực sự tăng, x: số nguyên

Output: Chỉ số i thỏa mãn $x = a_i$ hoặc 0 nếu x không có trong dãy

$i := 1$ — t_1
$j := n$
while $i < j$ **do** — t_2
  $m := \lfloor (i+j)/2 \rfloor$ — $t_5^{i,j}$
  if $x > a_m$ **then** — $t_6^{i,j}$
    $i := m + 1$ — t_7
  else
    $j := m$ — t_8
if $x = a_i$ **then** — t_3
  location $:= i$ — t_9
else
  location $:= 0$ — t_{10}
return *location* — t_4

Độ phức tạp tính toán

Tìm kiếm nhị phân

$$T(n) = t_1 + t_2 + t_3 + t_4$$

$$t_1, t_4, t_5^{i,j}, t_7, \ldots, t_{10} \text{ là } O(1)$$

$$t_2 = \sum_{\text{lần lặp \textbf{while} với cặp } i,j} \Big[(t_5^{i,j} + t_6^{i,j}) + \\ + (\text{thời gian kiểm tra } i < j) \Big]$$

$$t_6^{i,j} = \max\{t_7, t_8\} \\ + (\text{thời gian kiểm tra } x > a_m)$$

$$t_3 = \max\{t_9, t_{10}\} \\ + (\text{thời gian kiểm tra } x = a_i)$$

cặp i, j trong vòng **while** là $O(\log n)$

$\boxed{T(n) \text{ là } O(\log n)}$ (xấu nhất)

$\boxed{T(n) \text{ là } O(\log n)}$ (tốt nhất)

*Tại sao # cặp i, j trong vòng lặp **while** là $O(\log n)$?*

- Lần lặp 0: dãy $a_i, \ldots, a_j$ có độ dài $\leq \lceil n/2^0 \rceil$
- Lần lặp 1: dãy $a_i, \ldots, a_j$ có độ dài $\leq \lceil n/2^1 \rceil$
- $\ldots$
- Lần lặp h (cuối cùng): dãy $a_i, \ldots, a_j$ có độ dài $\leq \lceil n/2^h \rceil$
- # cặp i, j trong vòng lặp **while** là $h + 1$
- Vòng **while** kết thúc khi $i = j$, nghĩa là khi dãy $a_i, \ldots, a_j$ có độ dài 1
- Vì h là lần lặp cuối cùng, h phải là số nguyên dương nhỏ nhất thỏa mãn $\lceil n/2^h \rceil \leq 1$ (Nếu có số $h' < h$ thỏa mãn $\lceil n/2^{h'} \rceil \leq 1$ thì không thể có lần lặp nào sau h', trái với giả thiết h là lần lặp cuối cùng)
- Từ $\lceil n/2^h \rceil \leq 1$, suy ra $n/2^h \leq 1$, nghĩa là $h \geq \log n$. Giá trị nhỏ nhất của h là $\lceil \log n \rceil$. Do đó, # cặp i, j trong vòng lặp **while** là $h + 1 = \lceil \log n \rceil + 1 = O(\log n)$

Bài toán sắp xếp

Cho một dãy n phần tử và một cách so sánh hai phần tử bất kỳ trong dãy. Hãy sắp xếp dãy theo thứ tự tăng dần

- Sắp xếp nổi bọt (Bubble Sort)
- Sắp xếp chèn (Insertion Sort)

- **Bài toán:**
  - **Input:** $a_1, a_2, \ldots, a_n$: dãy số thực ($n \geq 2$)
  - **Output:** Dãy đã cho được sắp xếp theo thứ tự tăng dần
- **Sắp xếp nổi bọt:**
  (1) So sánh các phần tử liên tiếp, bắt đầu với cặp (a_1, a_2)
  (2) Nếu $a_1 > a_2$, hoán đổi giá trị của chúng
  (3) Lặp lại (1) và (2) với các cặp (a_2, a_3), (a_3, a_4), $\ldots$, (a_{n-1}, a_n). Lúc này, a_n là phần tử lớn nhất trong dãy
  (4) Lặp lại (1) – (3) với dãy $a_1, \ldots, a_{n-1}$, và sau đó với dãy $a_1, \ldots, a_{n-2}$, dãy $a_1, \ldots, a_{n-3}$, $\ldots$, cho đến dãy a_1, a_2

Ví dụ 10

- **Input:** Dãy $a_1 = 34$, $a_2 = 13$, $a_3 = 21$, $a_4 = 3$, $a_5 = 89$
- **Output:** Dãy sắp xếp theo thứ tự tăng dần

	a_1	a_2	a_3	a_4	a_5
	34	13	21	3	89
$i = 1$	13	21	3	34	89
$i = 2$	13	3	21	34	
$i = 3$	3	13	21		
$i = 4$	3	13			
	3	13	21	34	89

Thuật toán

Sắp xếp nổi bọt

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

Input: $a_1, a_2, \ldots, a_n$: dãy số thực ($n \geq 2$)
Output: Dãy đã cho được sắp xếp theo thứ tự tăng dần

```
for i := 1 to n − 1 do                          // Lặp lại n − 1 lần
    for j := 1 to n − i do
        if a_j > a_{j+1} then
            Hoán đổi giá trị của a_j và a_{j+1}
    // a_{n−i+1}, ..., a_n đã được sắp xếp
// a_1, ..., a_n đã được sắp xếp
```

Thuật toán 5 có hai vòng lặp **for**: vòng lặp trong ở Dòng 2–4 và vòng lặp ngoài (chứa vòng lặp trong) ở Dòng 1–4

- Một bất biến vòng lặp cho vòng lặp ngoài là

  Ở trước lần lặp i, dãy $a_{n-i+1}, \dots, a_n$ là dãy tăng chứa các phần tử lớn hơn hoặc bằng mọi phần tử trong $a_1, \dots, a_{n-i}$
- Một bất biến vòng lặp cho vòng lặp trong là

  Ở trước lần lặp j, $a_j = \max\{a_1, \dots, a_j\}$

Sơ đồ chứng minh:

- Chứng minh bước **Khởi động** cho bất biến vòng lặp ngoài
- Ở bước **Duy trì** cho vòng lặp ngoài
  - Chứng minh bất biến vòng lặp trong (**Khởi động**, **Duy trì**, **Dừng**)
  - Sử dụng bất biến vòng lặp trong để chứng minh cho vòng lặp ngoài
- Chứng minh bước **Dừng** cho bất biến vòng lặp ngoài

Thuật toán

Sắp xếp nổi bọt

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

Input: $a_1, a_2, \dots, a_n$: dãy số thực $(n \geq 2)$

Output: Dãy đã cho được sắp xếp theo thứ tự tăng dần

for $i := 1$ **to** $n-1$ **do** — t_1

  for $j := 1$ **to** $n-i$ **do** — t_2^i

    if $a_j > a_{j+1}$ **then** — t_3^j

      Hoán đổi giá trị của a_j và a_{j+1} — t_4

$$T(n) = t_1 = \sum_{i=1}^{n-1} \left[t_2^i + (\text{t.g. tăng } i \text{ và kiểm tra } i \leq n-1) \right]$$

$$t_2^i = \sum_{j=1}^{n-i} \left[t_3^j + (\text{t.g. tăng } j \text{ và kiểm tra } j \leq n-i) \right]$$

t_4 là $O(1)$ $(v := a_j,\ a_j := a_{j+1},\ a_{j+1} := v)$

$t_3^j = t_4 + (\text{thời gian kiểm tra } a_j > a_{j+1})$

$\boxed{T(n) \text{ là } O(n^2)}$

(xấu nhất)

$\boxed{T(n) \text{ là } O(n^2)}$

(tốt nhất)

- **Bài toán:**
  - **Input:** $a_1, a_2, \ldots, a_n$: dãy số thực ($n \geq 2$)
  - **Output:** Dãy đã cho được sắp xếp theo thứ tự tăng dần
- **Sắp xếp chèn:**

(1) Ban đầu, ta xem a_1 như một dãy đã được sắp xếp

(2) Để xử lý phần tử a_i ($i = 2, 3, \ldots, n$):
  - Lưu giá trị a_i vào một biến tạm m
  - Duyệt các phần tử $a_{i-1}, a_{i-2}, \ldots, a_1$ từ phải sang trái
  - Đẩy mỗi phần tử lớn hơn m sang phải một vị trí
  - Chèn m vào vị trí thích hợp (ngay sau phần tử đầu tiên nhỏ hơn hoặc bằng m)

(3) Sau mỗi bước, dãy $a_1, a_2, \ldots, a_i$ đã được sắp xếp theo thứ tự tăng dần

(4) Lặp lại quá trình cho đến khi xử lý xong a_n

- **Input:** Dãy $a_1 = 34$, $a_2 = 13$, $a_3 = 21$, $a_4 = 3$, $a_5 = 89$
- **Output:** Dãy sắp xếp theo thứ tự tăng dần

	a_1	a_2	a_3	a_4	a_5
	34	13	21	3	89
$i = 2$	13	34	21	3	89
$i = 3$	13	21	34	3	89
$i = 4$	3	13	21	34	89
$i = 5$	3	13	21	34	89

Thuật toán 6: Sắp xếp chèn (Insertion Sort)

Input: $a_1, a_2, \ldots, a_n$: dãy số thực ($n \geq 2$)

Output: Dãy đã cho được sắp xếp theo thứ tự tăng dần

for $i = 2$ **to** n **do**
  $m := a_i$ // m sắp được chèn vào dãy $a_1, \ldots, a_{i-1}$
  $j := i - 1$
  while $j \geq 1$ *và* $m < a_j$ **do** // Nếu $m < a_j$, đẩy a_j sang phải để có chỗ chèn m
    $a_{j+1} := a_j$
    $j := j - 1$
  $a_{j+1} := m$ // Chèn m
  // Dãy $a_1, \ldots, a_i$ đã được sắp thứ tự

Thuật toán 6 có hai vòng lặp: vòng lặp trong **while** ở Dòng 4–6 và vòng lặp ngoài **for** (chứa vòng lặp trong) ở Dòng 1–7

- Một bất biến vòng lặp cho vòng lặp ngoài là

  Ở trước lần lặp i*, dãy* $a_1, \dots, a_{i-1}$ *là dãy tăng*

- Một bất biến vòng lặp cho vòng lặp trong là

  Ở trước lần lặp j*,* $m \leq \min\{a_{j+1}, \dots, a_i\}$

Thuật toán
Sắp xếp chèn

Thuật toán 6: Sắp xếp chèn (Insertion Sort)

Input: $a_1, a_2, \dots, a_n$: dãy số thực $(n \geq 2)$
Output: Dãy đã cho được sắp xếp theo thứ tự tăng dần

```
for i = 2 to n do                                  t_1
    m := a_i                                       t_2^i
    j := i − 1                                     t_3^i
    while j ≥ 1 và m < a_j do                      t_4^i
        a_{j+1} := a_j                             t_6^{i,j}
        j := j − 1
    a_{j+1} := m                                   t_5^i
```

$$T(n) = t_1 = \sum_{i=2}^{n} \left[\sum_{k=2}^{5} t_k^i + (\text{t.g. tăng } i \text{ và kiểm tra } i \leq n) \right]$$

$t_2^i, t_3^i, t_5^i, t_6^{i,j}$ là $O(1)$

$$t_4^i = \sum_{1 \leq j \leq i-1 \text{ và } m < a_j} [t_6^{i,j} + (\text{t.g. kiểm tra } j \geq 1 \text{ và } m < a_j)]$$

$\boxed{T(n) \text{ là } O(n^2)}$ (xấu nhất)

$\boxed{T(n) \text{ là } O(n)}$ (tốt nhất)