# Competitive Programming:
# Kỹ Thuật Cơ Bản

Ngày 29 tháng 5 năm 2019

## Mục lục

Nội dung trong tài liệu được dịch và biên soạn lại từ các nguồn. Tôi sẽ giả sử rằng bạn đã nắm được lập trình cơ bản.

Trong tài liệu này, tôi sẽ sử dụng ngôn ngữ Pascal và mã giả (pseudocode). *Lưu ý: để tiện viết, trong thân chương trình, khi tôi viết khối lệnh (nhiều hơn 2 lệnh) thụt vào thì ta ngầm hiểu nó nằm trong cặp* `begin - end`.

Xin cảm ơn và xin lỗi những người đầu tiên đã review tài này về những sai sót trong quá trình biên soạn. Mọi ý kiến đóng góp xin gửi về mail: `minhnhat@linuxmail.org`.

# 1 Độ phức tạp thời gian

Độ **phức tạp thời gian** của thuật toán (time complexity) không cho biết một thuật toán chạy bao lâu mà nó *ước lượng thuật toán đó chạy nhanh đến mức nào*. Ý tưởng là biểu diễn tốc độ bằng một hàm số, còn tham số của hàm đó là kích thước của đầu vào (input). Bằng cách tính phức tạp thời gian của thuật toán, chúng ta có thể biết được liệu thuật toán có đủ nhanh chưa mà không cần thực thi nó.

## 1.1 Quy tắc tính

Độ phức tạp thời gian của thuật toán (để tiện, từ giờ tôi sẽ gọi là runtimes) kí hiệu là $O(\cdots)$. Thông thường, biến số $n$ được dùng để ký hiệu cho kích thước input. Ví dụ, nếu input là một mảng (array), $n$ sẽ là kích thước của mảng, và nếu input là một chuỗi ký tự (string), thì $n$ chính là độ dài chuỗi.

### Vòng lặp

Một lý do thuật toán chậm là do chứa nhiều vòng lặp (loop) lồng nhau. Càng nhiều vòng lặp thế này, thuật toán chạy càng chậm. Nếu số vòng lặp lồng nhau là $k$, độ phức tạp sẽ là $O(n^k)$.

Ví dụ, độ phức tạp của đoạn code này là $O(n)$:

```
for i := 1 to n do
    // code
```

Và đoạn sau là $O(n^2)$:

```
for i := 1 to n do
    for j := 1 to n do
        // code
```

*Lưu ý: độ phức tạp thời gian không cho ta biết chính xác số lần lặp của code bên trong. Trong những ví dụ dưới đây, code được thực thi* $3n$, $n+5$ *và* $\lfloor n/2 \rfloor$ *lần, nhưng phức tạp thời gian đều là* $O(n)$.

```
for i := 1 to 3*n do
    // code
```

```
for i := 1 to n+5 do
    // code
```

```
for i := 1 to n div 2 do
    // code
```

Tương tự với một ví dụ khác, vẫn là $O(n^2)$:

```
for i := 1 to n do
    for j := i+1 to n do
        // code
```

## Nhiều vòng lặp rời rạc

Nếu chương trình có nhiều vòng lặp rời rạc, thì runtimes của cả chương trình là runtimes lớn nhất trong các vòng lặp, bởi vì đoạn code chạy lâu nhất thường là nút thắt của chương trình.

Ví dụ, chương trình sau có ba vòng lặp với phức tạp thời gian lần lượt là $0(n), O(n^2)$ và $O(n)$. Do đó, độ phức tạp thời gian của toàn bộ chương trình là $O(n)$.

```
for i := 1 to n do
    // code

for i := 1 to n do
    for j := 1 to n do
        // code

for i := i to n do
    // code
```

## Nhiều biến số khác nhau

Nhiều lúc, độ phức tạp thời gian dựa trên nhiều biến số khác nhau. Trong trường hợp này, phức tạp thời gian là tích của các biến. Ví dụ, đoạn code sau có phức tạp thời gian là $O(nm)$:

```
for i := 1 to n do
    for j := 1 to m do
        // code
```

## Đệ quy

Runtimes của một hàm đệ quy dựa trên số lần bản thân nó được gọi và runtimes của mỗi lần gọi. Độ phức tạp của hàm đệ quy bằng tích hai giá trị trên. Ví dụ:

pseudocode

```
function f(n):
    begin
    if (n = 1) then return;
    f(n-1);
    end;
```

Chương trình gọi lại hàm $f(n)$ $n$ lần, và phức tạp thời gian của mỗi lần gọi là $n$ nên tổng phức tạp thời gian là $O(n)$.

Một ví dụ khác:

pseudocode

```
function g(n):
    begin
    if (n = 1) then return;
    g(n-1);
    g(n-1);
    end;
```

| Hàm gọi | Số lần gọi |
|---|---|
| $g(n)$ | 1 |
| $g(n-1)$ | 2 |
| $g(n-2)$ | 4 |
| ... | ... |
| $g(1)$ | $2^{n-1}$ |

Vậy phức tạp thời gian là

$$1 + 2 + 4 + ... + 2^{n-1} = 2^n - 1 = O(2^n)$$

## 1.2 Các loại phức tạp thời gian

**O(1)** Thời gian chạy là một **hằng số** và không phụ thuộc vào kích thước input. Điển hình cho loại này là các thuật toán dùng công thức để tìm lời giải.

**O(log n)** Thuật toán **logarit**[1] thường chia đôi kích thước input ở mỗi bước, bởi vì $log_2 n$ là số lần $n$ phải chia cho 2 để được 1.

**O($\sqrt{n}$)** Thuật toán **căn bậc hai** (square root) chậm hơn $O(logn)$ và nhanh hơn $O(n)$. Tuy nhiê cần lưu ý rằng $\sqrt{n} = n/\sqrt{n}$.

**O(n)** Thuật toán **tuyến tính**[2] (linear) có số lần chạy bằng input. Đây thường là trường hợp tốt nhất có thể, bởi vì ta luôn phải "đụng vào" mỗi phần tử trong input ít nhất một lần.

**O(n.log n)** Thường là các thuật toán sắp xếp, với mỗi phần tử là $O(logn)$.

**O($n^2$)** Thường duyệt mảng hai chiều.

**O($2^n$)** Các thuật toán này thường liệt kê ra các tổ hợp không lặp.

**O(n!)** Các thuật toán này thường liệt kê ra các hoán vị không lặp.

[1]Khi đề cập đến log, nếu không nói gì thêm, ta hiểu đó là log cơ số 2

[2]Tuyến tính nghĩa là có tính chất của đường thẳng. Giả sử đặt y = f(x) ta luôn có đồ thị là một đường thẳng.

## 1.3 Bảng ước lượng phức tạp thời gian tối thiểu

Bằng cách tính độ phức tạp thời gian, chúng ta có thể kiểm tra được tốc độ của thuật toán trước khi cài đặt. Với mỗi độ lớn của input, ta có thể *đoán* được runtimes như thế nào là cần thiết.

| kích thước input | phức tạp thời gian tối thiểu |
|---|---|
| $n \leq 10$ | $O(n!)$ |
| $n \leq 20$ | $O(2^n)$ |
| $n \leq 500$ | $O(n^3)$ |
| $n \leq 5000$ | $O(n^2)$ |
| $n \leq 10^6$ | $O(nlogn)$ hoặc $O(n)$ |
| $n$ rất lớn | $O(1)$ hoặc $O(logn)$ |

*Cần chú ý rằng runtimes cũng chỉ để ước lượng, bởi vì* $5n$ *hay* $n/5$ *cũng đều được coi là* $O(n)$*. Điều này có thể ảnh hưởng rất lớn đến tốc độ thật sự của thuật toán.*

## 1.4 Tổng lớn nhất của dãy con

Có rất nhiều thuật toán để giải một bài toán với các độ phức tạp thời gian khác nhau. Phần này sẽ thảo luận về một bài toán kinh điển có thể giải ở $O(n^3)$. Tuy nhiên, khi thiết kế một thuật toán tốt hơn thì nó có thể giải ở $O(n^2)$, thậm chí $O(n)$.

Cho một mảng gồm $n$ số nguyên, việc của chúng ta là đi tính tổng lớn nhất của dãy con trong mảng. Ví dụ,

| −1 | 2 | 4 | −3 | 5 | 2 | −5 | 2 |
|---|---|---|---|---|---|---|---|

Và tổng lớn nhất của dãy con là 10:

| −1 | 2 | 4 | −3 | 5 | 2 | −5 | 2 |
|---|---|---|---|---|---|---|---|

Chúng ta giả sử một dãy không có phần tử nào cũng là dãy con và có tổng bằng 0.

### Thuật toán 1

Cách làm kiểu "trâu bò" là đi duyệt qua toàn bộ các dãy con có thể có rồi tính tổng ở mỗi dãy, sau đó cập nhật tổng:

```
best := 0;
for sub_start := 1 to n do
    for sub_end := sub_start to n do
        sum := 0;
        for k := sub_start to sub_end do
            sum := sum + array[k];

        best := max(best, sum);

write(best);
```

Hai biến[3] `sub_start` và `sub_end` là chỉ số đầu và cuối mảng, tổng của mảng con lưu ở biến `sum`, biến `best` lưu tổng lớn nhất sau mỗi lần duyệt. Độ phức tạp thời gian của thuật toán là $O(n^3)$.

## Thuật toán 2

Ý tưởng của thuật toán này là tính tổng cùng với lúc phía cuối mảng con di chuyển:

```
best := 0;
for sub_start := 1 to n do
    sum := 0;
    for sub_end := sub_start to n do
        sum := sum + array[sub_end];
        best := max(best, sum);

write(best);
```

Phức tạp thời gian là $O(n^2)$.

## Thuật toán 3

Lần này, ta sẽ giải quyết bài toán trên chỉ bằng 1 vòng lặp, tức $O(n)$. Nếu ta muốn tìm dãy con với tổng lớn nhất dừng ở *k*, thì dãy con ở vị trí *k-1* cũng cần có tổng là lớn nhất (dừng ở *k-1*). do đó, ta hoàn toàn có thể tìm kết quả của bài toán lớn bằng cách tính tổng lớn nhất với mỗi vị trí dừng:

```
best := 0; sum := 0;
for k := 1 to n do
    sum := max(array[k], sum+array[k]);
    best := max(best, sum);

write(best);
```

[3]Tất nhiên bạn có thể dùng biến `a`, `b`, `c` tùy thích, nhưng đây là bài hướng dẫn nên tôi sẽ cố gắng trình bày để bạn tiếp thu dễ nhất.

## So sánh các thuật toán

Ở mỗi bộ test, input được sinh ra ngẫu nhiên, thời gian đọc input không được tính vào.

| array size $n$ | Algorithm 1 | Algorithm 2 | Algorithm 3 |
|---|---|---|---|
| $10^2$ | 0.0 s | 0.0 s | 0.0 s |
| $10^3$ | 0.1 s | 0.0 s | 0.0 s |
| $10^4$ | > 10.0 s | 0.1 s | 0.0 s |
| $10^5$ | > 10.0 s | 5.3 s | 0.0 s |
| $10^6$ | > 10.0 s | > 10.0 s | 0.0 s |
| $10^7$ | > 10.0 s | > 10.0 s | 0.0 s |

Khi input lớn thì khác biệt về tốc độ càng rõ ràng.

# 2 Toán học

## 2.1 Toán tử mod

Phép toán `mod` trả về số dư của phép chia. Ví dụ, 17 mod 5 = 2 bởi vì 17 = 3 . 5 + 2.

Trong một vài trường hợp, kết quả của bài toán là một số rất lớn nên chỉ cần đưa ra output là kết quả đã `mod` cho một số m. Tất nhiên có một cách ai cũng nghĩ được là sử dụng kiểu dữ liệu thật to để lưu kết quả tính toán để `mod` sau. Nhưng may mắn là chúng ta không cần tốn nhiều tài nguyên mà vẫn có thể giải quyết vấn đề này bằng các tính chất sau đây:

$$(a+b)\ mod\ m = (a\ mod\ m + b\ mod\ m)\ mod\ m \quad (1)$$
$$(a-b)\ mod\ m = (a\ mod\ m - b\ mod\ m)\ mod\ m \quad (2)$$
$$(a\cdot b)\ mod\ m = (a\ mod\ m \cdot b\ mod\ m)\ mod\ m \quad (3)$$
$$x^n\ mod\ m = (x\ mod\ m)^n\ mod\ m \quad (4)$$

Do đó, chúng ta có thể `mod` sau mỗi lần tính và kết quả sẽ không quá lớn. Ví dụ, đoạn code sau đây thực hiện yêu cầu tính $n!$ (giai thừa) rồi `mod` cho m:

```
x := 1;
for i := 2 to n do
    x := (x*i) mod m;
write(x mod m);
```

Thông thường, chúng ta muốn số dư phải nằm trong khoảng từ 0...m-1. Tuy nhiên, trong một số ngôn ngữ, *số dư của một số âm* có thể bằng 0 hoặc cũng là một số âm. Lúc này ta chỉ cần cộng thêm $m$ vào kết quả:

```
x := x mod m;
if x < 0 then x := x+m;
```

## 2.2 Dãy số

Nếu tổng có dạng sau đây:

$$\sum_{x=1}^{n} x^k = 1^k + 2^k + 3^k + ... + n^k \quad (5)$$

với $k$ là số nguyên dương, thì công thức trên có dạng đa thức bậc $k+1$ (xin không đề cập đa thức ở đây). Ví dụ[4],

$$\sum_{x=1}^{n} x = 1 + 2 + 3 + ... + n = \frac{n(n+1)}{2} \quad (6)$$

[4]Cũng có một công thức tổng quát khác cho việc tính tổng, là công thức của Faulhaber, nhưng nó quá phức tạp để trình bày ở đây.

và

$$\sum_{x=1}^{n} x^2 = 1^2 + 2^2 + 3^2 + ... + n^2 = \frac{n(n+1)(2n+1)}{6} \tag{7}$$

Một dãy **cấp số cộng** là một dãy số mà các cặp phần tử liền kề đều chênh lệch nhau bằng một hằng số. Ví dụ,

$$3, 7, 11, 15, 19$$

là một cấp số cộng với chênh lệch (differencece, gọi là *công sai*) bằng 4. Chúng ta có công thức tính tổng dãy cấp số cộng gồm $n$ phần tử:

$$S_n = a_1 + a_2 + a_3 + ... + a_n = \frac{n(a_1 + a_n)}{2} \tag{8}$$

Ví dụ,

$$3 + 7 + 11 + 15 + 19 = \frac{5.(3 + 19)}{2} = 55$$

Một dãy **cấp số nhân** là một dãy số mà các cặp phần tử liền kề đều chênh lệch nhau theo một tỉ lệ (ratio, gọi là *công bội*) cố định. Ví dụ,

$$3, 6, 12, 24$$

là một dãy cấp số nhân với công bội bằng 2. Tổng của một dãy cấp số nhân gồm $n$ phần tử có thể được tính bằng công thức:

$$a + ak + ak^2 + ... + b = \frac{bk - a}{k - 1} \tag{9}$$

với $a$ là phần tử đầu, $b$ là phần tử cuối và $k$ là công bội. Ví dụ,

$$3 + 6 + 12 + 24 = \frac{24.2 - 3}{2 - 1} = 45$$

Một trường hợp đặc biệt của tổng cấp số nhân là:

$$1 + 2 + 4 + 8 + ... + 2^{n-1} = 2^n - 1 \tag{10}$$

## 2.3 Tập hợp

***Tập hợp*** *(set) là một nhóm hữu hạn hoặc vô hạn các đối tượng nào đó.* Ví dụ, tập hợp

$$X = \{2, 4, 7\}$$

chứa các phần tử 2, 4 và 7. Khi kí hiệu một tập hợp bằng $\phi$, ta hiểu đó là một tập rỗng, và $|S|$ kí hiệu kích thước (số phần tử) của tập $S$. Ví dụ, tập hợp trên, $|X| = 3$.

Nếu tập $S$ chứa $x$ ta nói $x$ thuộc $S$ kí hiệu $x \in S$ ngược lạ, ta viết $x \notin S$.

Một số phép toán tập hợp:

**Phép giao** $A \cap B$ trả về các phần tử tồn tại trong $A$ và $B$. Ví dụ, nếu A = {1, 2, 5} và B = {2, 4} thì $A \cap B = \{2\}$.

**Phép hợp** $A \cup B$ trả về các phần tử tồn tại trong cả $A$ và $B$. Ví dụ, nếu A = {3, 7} và B = {2, 3, 8} thì $A \cup B = \{2, 3, 7, 8\}$.

**Phần bù** $\bar{A}$ là các phần tử *không* tồn tại trong $A$, thường phụ thuộc vào **tập vũ trụ** (universal set) - chứa tất cả các phần tử. Ví dụ, nếu A = {1, 2, 5, 7} và tập vũ trụ gồm {1, 2,..., 10}, thì $\bar{A} = \{3, 4, 6, 8, 9, 10\}$.

**Hiệu** của hai tập hợp $A \setminus B = A \cap \bar{B}$ là những phần tử tồn tại trong $A$ nhưng không tồn tại trong $B$. Ví dụ, nếu A = {2, 3, 7, 8} và B = {3, 5, 8} thì $A \setminus B = \{2, 7\}$.

Nếu tất cả các phần tử của A tồn tại trong S thì ta nói A là tập con của S, kí hiệu $A \subset S$. Ví dụ,các tập con của {2, 4, 7} là

$$\phi, \{2\}, \{4\}, \{7\}, \{2, 4\}, \{4, 7\}, \{2, 4, 7\}.$$

Những tập hợp hay được sử dụng là $\mathbb{N}$ (số tự nhiên), $\mathbb{Z}$ (số nguyên), $\mathbb{Q}$ (số hữu tỉ) và $\mathbb{R}$ (số thực).

## 2.4 Phép toán logic

Kết quả của phép toán logic luôn hoặc là **true** (1) hoặc là **false** (0). Những phép toán quan trọng bao gồm: $\neg$ (**phủ định logic**), $\wedge$ (**tích logic, OR**), $\vee$ (**tổng logic, AND**), $\Rightarrow$ (**kéo theo**) và $\Leftrightarrow$ (**tương đương**). Dưới đây là bảng chân lí:

| $A$ | $B$ | $\neg A$ | $\neg B$ | $A \wedge B$ | $A \vee B$ | $A \Rightarrow B$ | $A \Leftrightarrow B$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |

Phép phủ định $\neg A$ trả về kết quả đảo ngược của $A$. Tích $A \wedge B$ là true nếu cả A và B đều là true, tổng $A \vee B$ là true nếu A hoặc B là true. Mệnh đề kéo theo $A \Rightarrow B$ trả về false khi và chỉ khi B là false. Mệnh đề tương đương $A \Leftrightarrow B$ là true khi cả A và B đều là true.

## 2.5 Hàm số

Hàm (function) $\lfloor x \rfloor$ làm tròn x xuống thành một số nguyên, và hàm $\lceil x \rceil$ làm tròn x lên. Ví dụ,

$$\lfloor 3/2 \rfloor = 1; \quad \lceil 3/2 \rceil = 2$$

Hàm $min(x_1, x_2, ..., x_n)$ và $max(x_1, x_2, ..., x_n)$ lần lượt cho ta các giá trị nhỏ nhất và lớn nhất của $x_1, x_2, ..., x_n$. Ví dụ,

$$min(1, 2, 3) = 1; \quad max(1, 2, 3) = 3$$

Giai thừa $n!$ có thể được định nghĩa

$$\Pi_{x=1}^{n} x = 1 \cdot 2 \cdot 3 \cdot ... \cdot n \tag{11}$$

hoặc định nghĩa theo đệ quy (recursion):

$$\text{factorial}(n) \begin{cases} 1 & \text{if n = 1} \\ n.(n-1)! & \text{otherwise} \end{cases} \tag{12}$$

Dãy số **fibonacci** cũng được định nghĩa đệ quy như sau:

$$\text{f(n)} = \begin{cases} 0 & \text{if n = 0} \\ 1 & \text{if n = 1} \\ f(n-1) + f(n-2) & \text{otherwise} \end{cases} \tag{13}$$

Những số đầu của dãy là

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...$$

Fibonacci có một công thức tính thường gọi là **công thức Binet**:

$$f(n) = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}} \tag{14}$$

Tuy nhiên, cần lưu ý việc tính toán với căn bậc hai dễ dẫn đến sai số. Có một cách khác $O(logn)$ là dùng ma trận, nhưng sẽ không đề cập ở đây.

## 2.6 Logarithms

Hàm **logarit** của một số x được ký hiệu $log_k(x)$, k gọi là cơ số (base). Theo định nghĩa, **log** là phép toán ngược của lũy thừa, $log_k(x) = a$ chính xác tuyệt đối khi $k^a = x$.

Hiểu một cách đơn giản, $log_k(x)$ chính là *số lần* chúng ta phải chia x cho k để được 1. Ví dụ, $log_2(32) = 5$ bởi vì cần 5 phép chia cho 2:

$$32 \to 16 \to 8 \to 4 \to 2 \to 1$$

Logarit thường dùng trong phân tích thuật toán, cụ thể rất nhiều thuật toán hiệu quả thường chia đôi $n$ mỗi bước. Do đó, chúng ta có thể ước lượng hiệu quả của thuật toán bằng logarit. Dưới đây là một vài tính chất:

$$log_k(ab) = log_k(a) + log_k(b) \tag{15}$$
$$log_k(x^n) = n \cdot log_k(x) \tag{16}$$
$$log_k(\frac{a}{b}) = log_k(a) - log_k(b) \tag{17}$$
$$log_u(x) = \frac{log_k(x)}{log_k(u)} \tag{18}$$

**Logarit tự nhiên** (natural logarithm) *ln(x)* của một số x cũng là hàm logarit với cơ số $e \approx 2.71828$. Một tính chất của logarit là *số chữ số* dùng để biểu diễn x trong hệ cơ số b là $\lfloor log_b(x) + 1 \rfloor$. Ví dụ, 123 trong hệ nhị phân là 1111011 và $\lfloor log_2(123) + 1 \rfloor = 7$ (dùng 7 bit để biểu diễn).

## 2.7 Số nguyên tố

*Một số $n > 1$ được gọi là số nguyên tố (prime) nếu nó chỉ chia hết cho 1 và chính nó.* Ví dụ, các số nguyên tố đầu tiên là:

$$2, 3, 5, 7, 11, 13, 17, 19, 23, ...$$

Chúng ta có thuật toán kiểm tra `n` có phải số nguyên tố không, với runtimes là $O(\sqrt{n})$:

```
prime := true;

if n = 1 then prime := false;
for i := 2 to trunc(sqrt(n)) do
    if (n mod i = 0) then
        prime := false;
        break;

write(prime);
```

Mọi số nguyên n > 1 đều có thể viết dưới dạng tích (prime factorization):

$$n = p_1^{a_1} p_2^{a_2} ... p_k^{a_k} \tag{19}$$

với $p_1, p_2, ...p_k$ là các số nguyên tố khác nhau và $a_1, a_2, ...a_k$ là các số nguyên dương. Ví dụ,

$$84 = 2^2 \cdot 3^1 \cdot 7^1$$

Số lượng các ước của $n$ là

$$\Pi_{i=1}^{k}(a_i + 1) = (a_1 + 1) \cdot (a_2 + 1) \cdot ... \cdot (a_k + 1) \tag{20}$$

Ví dụ, số lượng các ước của 84 là: 3·2·2 = 12 (1, 2, 4, 6, 7, 12, 14, 21, 28, 42 và 84). Tổng của các ước của $n$ được tính bằng

$$\Pi_{i=1}^{k} \frac{p_i^{a_i+1} - 1}{p_i - 1} \tag{21}$$

Ví dụ tổng của các ước của 84 là:

$$\frac{2^3 - 1}{2 - 1} \cdot \frac{3^2 - 1}{3 - 1} \cdot \frac{7^2 - 1}{7 - 1} = 7 \cdot 4 \cdot 8 = 224$$

Một số $n$ được gọi là *số hoàn thiện* (perfect number) nếu có tổng các ước từ 1 đến n-1 bằng chính nó. Ví dụ 28 là số hoàn thiện vì 28 = 1 + 2 + 4 + 7 + 14.

### Thuật toán sàng nguyên tố

Sàng nguyên tố (the sieve of Eratosthenes) là thuật toán xây dựng một mảng để kiểm tra số nguyên tố từ 2 đến $n$. Thuật toán duyệt qua từng số 2...$n$, khi tìm thấy một số nguyên tố, thuật toán ghi lại các bội của $x$ (2x, 3x, 4x,...) không phải số nguyên tố, bởi vì nó chia hết cho $x$. Ví dụ, nếu $n = 20$, chúng ta sẽ có mảng sau:

| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 2 | 0 | 3 | 0 | 2 | 3 | 5 | 0 | 3 | 0 | 7 | 5 | 2 | 0 | 3 | 0 | 5 |

Trong đó chỉ số vị trí (index) của mảng là các số nguyên từ 2 đến n; giá trị trong mảng bằng 0 nếu đó là số nguyên tố, ngược lại đó là một ước nguyên tố của index.

Dưới đây là cài đặt của thuật toán, giả sử mảng `sieve` (sàng) được khởi tạo chỉ có các giá trị là 0:

```
for x := 2 to n do
    if (sieve[x] <> 0) then continue;
    u := 2*x;
    while u <= n do
        sieve[u] := x;
        u := u+x;
```

Vòng lặp `while` của thuật toán lặp $n/x$ lần với mỗi $x$. Do đó phức tạp thời gian là:

$$\sum_{x=2}^{n} = n/2 + n/3 + ... + n/n = O(n\ log\ n)$$

Thực tế, thuật toán chạy nhanh hơn, bởi vì `while` chỉ chạy khi $x$ là số nguyên tố. Điều này cho thấy runtimes của thuật toán chỉ là $O(n\ loglog\ n)$, rất gần với $O(n)$.

## 2.8 Thuật toán Euclid

**Ước chung lớn nhất** (UCLN, greatest common divisor) của hai số $a$ và $b$, gdc(a, b) là số lớn nhất mà cả $a$ và $b$ đều chia hết cho nó; và **bội chung nhỏ nhất** (BCNN, least common multiple) của $a$ và **b**, lcm(a, b) là số nhỏ nhất chia hết cho cả $a$ và $b$. Ví dụ, gcd(24, 36) = 12 và lcm(24, 36) = 72.

UCLN và BCNN liên hệ qua công thức:

$$lcm(a, b) = \frac{ab}{gdc(a, b)} \tag{22}$$

**Thuật toán Euclid** (Euclid's algorithm) cho chúng ta cách tìm UCLN của hai số. Thuật toán dựa trên công thức sau:

$$\text{gdc(a, b)} \begin{cases} a & \text{if b} = 0 \\ gdc(b, a\ mod\ b) & \text{otherwise} \end{cases} \tag{23}$$

Ví dụ,

$$gcd(24, 36) = gcd(36, 24) = gcd(24, 12) = gcd(12, 0) = 12$$

Thuật toán có thể được cài đặt:

```
function gcd(a, b: integer): integer;
    begin
    if (b = 0) then gcd := a;
    gcd := gcd(b, a mod b);
    end;
```

Thuật toán Euclid có runtimes $O(logn)$, khi $n = min(a, b)$. Trường hợp xấu nhất (worst case) là khi a và b là hai số liền nhau của dãy Fibonacci. Ví dụ,

$$gcd(13, 8) = gcd(8, 5) = gcd(5, 3) = gcd(3, 2) = gcd(2, 1) = gcd(1, 0) = 1$$

# 3 Sắp xếp

**Sắp xếp** (sorting) là một trong những thuật toán cơ sở của khoa học máy tính. Rất nhiều chương trình sử dụng sắp xếp như là một công cụ để xử lí dữ liệu dễ dàng hơn khi các phần tử đã nằm theo thứ tự nhất định. Ở chương này sẽ giới thiệu một vài thuật toán dựa trên độ phức tạp thời gian.

Chúng ta sẽ xét bài toán sắp xếp mảng theo thứ tự tăng dần. Ví dụ, mảng

$$[1, 3, 8, 2, 9, 2, 5, 6]$$

sẽ được sắp xếp thành:

$$[1, 2, 2, 3, 5, 6, 8, 9]$$

## 3.1 Thuật toán $O(n^2)$: Bubble sort

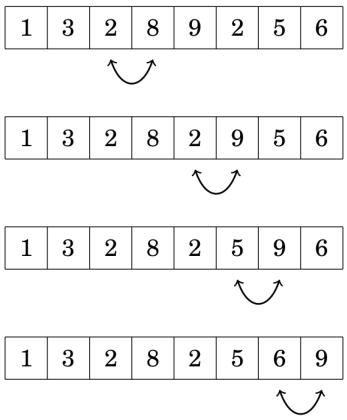
Thuật toán sắp xếp đơn giản thường chạy ở $O(n^2)$. Một thuật toán $O(n^2)$ nổi tiếng là **bubble sort**. Bubble sort gồm $n$ vòng (round). Ở mỗi vòng, thuật toán duyệt qua tất cả các phần tử trong mảng. Bất cứ khi nào thấy hai phần tử kề nhau ở sai vị trí, thuật toán sẽ đổi chỗ chúng. Thuật toán có thể được cài đặt như sau:

```
for round := 1 to n do
    for i := 1 to n-1 do
        if (array[i] > array[i+1]) then
            temp := array[i];
            array[i] := array[i+1];
            array[i+1] := temp;
```

Sau vòng đầu tiên, phần tử lớn nhất sẽ được đưa về cuối mảng, tổng quát, sau $k$ vòng, phần tử lớn thứ $k$ sẽ được đưa về đúng vị trí. Do đó, sau $n$ vòng, toàn bộ mảng đều đã được sắp xếp. Ví dụ, chúng ta có mảng sau:

$$[1, 3, 8, 2, 9, 2, 5, 6]$$

vòng đầu tiên của thuật toán sẽ chạy:



Chúng ta sẽ xem trường hợp xấu nhất, tức các phần tử nằm ngược lại so với thứ tự ta cần sắp xếp:

$$[5, 4, 3, 2, 1]$$

Mảng có các cặp phần tử sai vị trí là (5, 4); (5, 3); (4, 3); (5; 2),... Tức từ index thứ 2 trở về trước thì có 1 cặp sai, tương tự index thứ 3 thì có 2 cặp sai, cứ như vậy đến index thứ n có n-1 cặp sai, do đó ta có công thức tổng quát của số các cặp sai vị trí là:

$$1 + 2 + ... + (n - 1) = \frac{n(n-1)}{2} = O(n^2)$$

## 3.2 Thuật toán $O(n\ log\ n)$: Merge sort

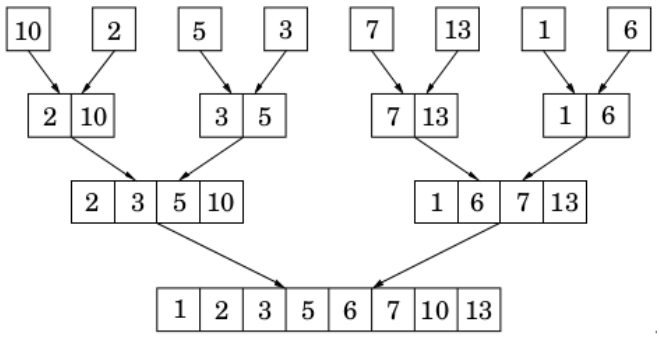
Ở phần này, ta sẽ thảo luận về thuật toán chạy ở $O(n\ log\ n)$ không bị "giới hạn" bởi việc đổi chỗ (swap) các cặp phần tử kề nhau như bubble sort. Một thuật toán nổi tiếng là **Merge sort**, dựa trên đệ quy.

Các bước của Merge sort để sắp xếp một mảng `array[a..b]` như sau:

1. Nếu $a = b$ thì không thực hiện, vì mảng chỉ có 1 phần tử (đã sắp xếp).
2. Tính vị trí của phần tử giữa (middle element): $mid = \lfloor (a + b)/2 \rfloor$.
3. Gọi đệ quy sắp xếp mảng con `array[a..mid]`.
4. Gọi đệ quy sắp xếp mảng con `array[mid+1..b]`.
5. *Trộn* (merge) hai mảng con đã được sắp xếp `array[a..mid]` và `array[mid+1.. b]` thành một mảng con được sắp xếp `array[a..b]`.

Merge sort là một thuật toán hiệu quả,nó chia đôi kích thước của mảng ở mỗi bước. Thuật toán gọi đệ quy biểu diễn một cây (tree) có chiều cao $O(log\ n)$, và xử lý ở mỗi level của cây tốn runtimes $O(n)$. Việc trộn hai mảng con `array[a..mid]` và `array[mid+1.. b]` có thể thực hiện ở runtimes tuyến tính, bởi vì cả hai mảng đều đã được sắp xếp.

Ví dụ, mảng $[10, 2, 5, 3, 7, 13, 1, 6]$ sẽ được chia đôi ra thành hai mảng con $[10, 2, 5, 3]$ và $[7, 13, 1, 6]$, mỗi mảng đó lại tiếp tục chia đôi ra và trộn lại như sau:



Merge sort có thể được cài đặt như sau (bạn đọc tự định nghĩa kiểu dữ liệu):

```
procedure MergeSort(A: array[left..right], left, right);
    var M: array[left..right];
    sub_left , sub_right, i;

    begin
    if right > 1 then
        mid := right div 2;
        MergeSort(A, left, mid);
        MergeSort(A, mid+1, right);

        sub_left := 1; sub_right := mid+1;  // merge into M
        for i := 1 to right do
            if (sub_right > right) then
                M[i] := A[sub_left];
                inc(sub_left);
            else if (sub_left > mid) then
                M[i] := A[sub_right];
                inc(sub_right);
            else if (A[sub_left] < A[sub_right]) then
                M[i] := A[sub_left];
                inc(sub_left);
            else
                M[i] := A[sub_right];
                inc(sub_right);

        for i := 1 to right do
            A[i] := M[i];
    end;
```

Hai biến `sub_left` và `sub_right` được khởi tạo là index bắt đầu của hai mảng con. Hai lệnh `if` đầu trong vòng lặp `for` để kiểm tra nếu một trong hai mảng con đã được trộn hết.

Lưu ý rằng chúng ta không thể sắp xếp nhanh hơn $O(n\ log\ n)$ với thuật toán dựa trên so sánh các phần tử.

## 3.3 Thuật toán $O(n)$: Counting sort

**Counting sort** không so sánh các phần tử trong mảng mà dùng "thông tin" khác để sắp xếp. Thuật toán tạo một mảng gọi là *bookkeeping*, với giá trị là số lần xuất hiện của từng phần tử trong mảng. Ví dụ mảng $[1, 3, 6, 9, 9, 3, 5, 9]$, tạo ra bookkeeping tương ứng:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 2 | 0 | 1 | 1 | 0 | 0 | 3 |

Ví dụ, giá trị ở index thứ 3 bằng 2 vì phần tử 3 xuất hiện 2 lần trong mảng ban đầu.