

# Toán học trong Tin học

## Toán học trong Tin học

Bài viết gốc: [Mathematics for Topcoders](#) - đăng bởi [dimkadimon](#) trên [Topcoder](#) [🔗](#)

### Giới thiệu:

Nhiều người thi Topcoder phàn nàn khi trong đề có quá nhiều Toán – điểm yếu của họ. Cá nhân tôi là một người rất yêu Toán, vì vậy có lẽ tôi sẽ có chút thiên vị trong vấn đề này. Nhưng tôi hoàn toàn tin tưởng rằng, các bài tập nên có ít nhất một chút Toán học và Khoa học máy tính phải đi liền với Toán học. Thật khó có thể tưởng tượng khi mà cả hai lĩnh vực này cũng tồn tại mà không hề có bất kỳ sự tương tác nào lẫn nhau. Ngày nay, Toán học được áp dụng rất nhiều trong Tin học để giải những hệ phương trình hàng nghìn ẩn hay tìm nghiệm xấp xỉ đối với những phương trình mà không có công thức nghiệm tổng quát. Toán học còn được sử dụng rộng rãi trong việc nghiên cứu về Khoa học máy tính, cũng như là áp dụng cho các thuật toán về đồ thị (**graph algorithms**) và lĩnh vực thị giác máy tính (**Computer Vision**).

Bài viết này sẽ phân tích về lý thuyết và ứng dụng của một vài cấu trúc Toán học phổ biến. Các chủ đề được đề cập tới bao gồm:

1. Số nguyên tố.
2. Ước chung lớn nhất (GCD).
3. Hình học cơ bản.
4. Hệ cơ số.
5. Phân số và số phức.

### Số nguyên tố:

Một số tự nhiên là số nguyên tố khi và chỉ khi nó chỉ chia hết cho 1 và chính nó. Ví dụ như 2, 3, 5, 79, 311 và 1931 đều là số nguyên tố, trong khi 21 thì không phải, bởi nó chia hết cho 3 và 7. Để xác định xem một số tự nhiên  $n$  có phải là số nguyên tố hay không, ta chỉ cần đơn giản kiểm tra xem nó có chia hết cho bất kỳ số nào nhỏ hơn nó và lớn 1 hay không. Chúng ta có thể sử dụng phép chia có dư (toán tử  $\%$ ) để kiểm tra khả năng chia hết của nó:

#### Java

```
1 | for (int i = 2; i < n; i++)
2 |     if (n % i == 0) return false;
3 |
4 | return true;
```

## Pascal

```

1  for i := 2 to n - 1 do
2      if (n mod i = 0) then exit(false);
3
4  exit(true);

```

Chúng ta còn có thể làm cho đoạn mã này chạy nhanh hơn nữa bằng việc nhận ra rằng, ta chỉ cần xét khả năng chia hết cho mọi giá trị  $i$  lớn hơn 1 và nhỏ hơn hoặc bằng với phần nguyên của căn bậc 2 của  $n$ , tạm gọi là  $m$ . Tại sao lại thế nhỉ? Đây là bởi vì nếu  $n$  chia hết cho bất kỳ số nào lớn hơn  $m$  thì kết quả nhận được cũng chỉ là một số nhỏ hơn hoặc bằng  $m$ . Một cải tiến nữa là: ta thấy rằng không tồn tại bất kỳ một số nguyên tố chẵn nào lớn hơn 2. Như vậy, sau khi đã kiểm tra rằng  $n$  là số lẻ thì ta có thể an tâm tăng biến  $i$  lên 2 giá trị. Đây sẽ là đoạn mã áp dụng phương thức kiểm tra số nguyên tố mới nhất này:

## Java

```

1  public boolean isPrime (int n)
2  {
3      if (n <= 1) return false;
4      if (n == 2) return true;
5      if (n % 2 == 0) return false;
6      int m = Math.sqrt(n);
7
8      for (int i = 3; i <= m; i += 2)
9          if (n % i == 0)
10             return false;
11
12     return true;
13 }

```

## Pascal

```

1  function IsPrime (n : Integer): Boolean;
2  var
3      i : Integer;
4      m : Integer;
5  begin
6      if (n <= 1) exit(false);
7      if (n = 2) exit(true);
8      if (n mod 2 = 0) exit(false);
9      i := 3;
10     m := trunc(sqrt(n));
11     while (i <= m) do
12         begin
13             if (n mod i = 0) exit(false);
14             inc(i, 2);

```

```

15 |         end;
16 |         exit(true);
17 |     end;

```

Giả sử chúng ta cần phải tìm mọi số nguyên tố trong khoảng  $[1, 100000]$ , thì chẳng lẽ ta phải gọi hàm kiểm tra 100000 lần sao? Việc lặp đi lặp lại các phép kiểm tra tương tự thật sự không hiệu quả chút nào. Trong tình huống này, phương pháp tối ưu nhất chính là sử dụng **Sàng nguyên tố Eratosthenes** (Sieve of Eratosthenes). Sàng nguyên tố Eratosthenes sẽ xác định toàn bộ những số nguyên tố trong khoảng  $[2, n]$ . Trước hết, sàng nguyên tố sẽ giả định mọi số tự nhiên từ 2 đến  $n$  là số nguyên tố. Kế đến, nó sẽ chọn ra số nguyên tố đầu tiên và xóa đi những bội nhỏ hơn bằng  $n$  của số nguyên tố đó. Và nó lại tiếp tục quá trình bằng lựa một số nguyên tố tiếp theo và xóa đi các bội của nó. Quá trình này sẽ tiếp tục diễn ra cho đến khi mà mọi số đều đã xử lý. Ví dụ, ta sẽ tìm các số nguyên tố trong khoảng  $[2, 20]$ . Trước hết, ta hãy ghi ra dãy số đó:

```

1 | 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

```

Bởi vì 2 là số nguyên tố đầu tiên, nên ta sẽ loại bỏ đi tất cả những bội của 2 và nhỏ hơn bằng 20.

```

1 | 2 3 5 7 9 11 13 15 17 19

```

Bởi vì 3 là số đầu tiên chưa bị bỏ, nên ta sẽ lựa chọn 3 là số nguyên tố tiếp theo và loại bỏ đi tất cả những bội của 3 và nhỏ hơn bằng 20.

```

1 | 2 3 5 7 11 13 17 19

```

Tiếp đến, ta sẽ chọn 5, ta thấy không còn bội nào của 5 nhỏ hơn bằng 20 còn sót nên ta lại tiếp tục xét đến 7, ...

Và giờ thì tất cả những số còn sót lại chính là các số nguyên tố mà ta cần tìm. Dưới đây là đoạn mã cho Sàng nguyên tố Eratosthenes:

## Java

```

1 | public boolean[] sieve(int n)
2 | {
3 |     boolean[] prime=new boolean[n+1];
4 |     Arrays.fill(prime,true);
5 |     prime[0] = false;
6 |     prime[1] = false;
7 |     int m = Math.sqrt(n);
8 |
9 |     for (int i = 2; i <= m; i++)
10 |         if (prime[i])
11 |             for (int k = i * i; k <= n; k += i)
12 |                 prime[k] = false;
13 |
14 |

```

```

14 |     return prime;
15 | }

```

## Pascal

```

1 | //mảng kiểm tra số nguyên tố prime[0..n]
2 |
3 | function sieve(n : Integer): Boolean;
4 | var
5 |     i, m : Integer;
6 | begin
7 |     fillchar(prime, sizeof(prime), 1);
8 |     prime[0] := false;
9 |     prime[1] := false;
10 |
11 |     m := trunc(sqrt(n));
12 |     for i := 2 to m do
13 |         if (prime[i]) then
14 |             for j := i to n div i do
15 |                 prime[i * j] := false;
16 | end;

```

Ở đoạn mã trên, ta sẽ tạo một mảng prime, nơi sẽ chứa các giá trị nguyên trong khoảng  $[0, n]$ . `Prime[i]` là true nếu  $i$  là số nguyên tố và ngược lại. Vòng lặp bên ngoài sẽ tìm kiếm số nguyên tố tiếp theo, trong khi vòng lặp bên trong sẽ loại bỏ đi tất cả những bội nhỏ hơn bằng  $n$  của số nguyên tố đó.

## Ước chung lớn nhất

**Ước chung lớn nhất** (Greatest Common Divisor) của hai số tự nhiên  $a$  và  $b$  là số lớn nhất mà cả  $a$  và  $b$  đều chia hết. Để tìm được  $GCD(a, b)$ , một cách đơn giản, ta có thể lấy số nhỏ hơn trong hai số  $a$  và  $b$  rồi giảm dần cho đến khi nào có được một số mà cả  $a$  và  $b$  đều cùng chia hết cho số đó.

## Java

```

1 | for (int i = Math.min(a, b); i >= 1; i--)
2 |     if (a % i == 0 && b % i == 0)
3 |         return i;

```

## Pascal

```

1 | for i := min(a, b) downto 1 do
2 |     if (a mod i = 0) and (b mod i = 0) then exit(i);

```

Mặc dù phương pháp này đủ nhanh để đáp ứng phần lớn các trường hợp, song ta vẫn còn một cách nhanh hơn nữa, đó chính là **thuật toán Euclid**. Thuật toán Euclid sẽ lặp đi lặp lại việc xử lý hai số cho đến khi mà phần dư của chúng bằng 0. Ví dụ, giả sử chúng ta muốn tìm ước chung lớn nhất của hai số 2336 và 1314. Đầu tiên, ta sẽ phân tích số lớn hơn là 2336 thành bội của 1314 cộng thêm cho phần dư.

$$1 \mid 2336 = 1314 \times 1 + 1022$$

Ta sẽ thực hiện tương tự với 1314 và 1022.

$$1 \mid 1314 = 1022 \times 1 + 292$$

Ta tiếp tục quá trình cho đến khi phần dư bằng 0.

$$\begin{array}{l} 1 \mid 1022 = 292 \times 3 + 146 \\ 2 \mid 292 = 146 \times 2 + 0 \end{array}$$

Và phần dư cuối cùng mà khác 0 chính là ước chung lớn nhất mà ta cần tìm. Vậy ước chung lớn nhất của 1314 và 2336 chính là 146. Thuật toán trên sẽ được mô phỏng lại bằng một đoạn mã đệ quy đơn giản:

### Java

```
1 // giả định rằng a và b đều khác 0
2 public int GCD(int a, int b)
3 {
4     if (b == 0) return a;
5     return GCD(b, a % b);
6 }
```

### Pascal

```
1 // giả định a và b đều khác 0
2 function GCD(a, b : Integer): Integer;
3 begin
4     if (b = 0) then exit(a)
5     else exit(GCD(b, a mod b));
6 end;
```

Đặc biệt, thuật toán Euclid còn được áp dụng trong việc tìm bội chung nhỏ nhất (LCM) của hai số tự nhiên. Ví dụ, bội chung nhỏ nhất của 6 và 9 là 18 vì 18 là số nhỏ nhất mà chia hết cho cả a lẫn b. Dưới đây là đoạn mã cho việc tìm bội chung nhỏ nhất:

## Java

```

1 public int LCM(int a, int b)
2 {
3     return b * a / GCD(a, b);
4 }

```

## Pascal

```

1 function LCM(a, b : Integer): Integer;
2 begin
3     exit(a * b / GCD(a, b));
4 end;

```


Một chú ý cuối cùng đó là việc thuật toán Euclid còn có thể dùng để giải phương trình tìm nghiệm nguyên Diophantine. Đó là những phương trình có hệ số và ẩn số nguyên ( $a, b, c, x, y \in \mathbb{N}$ ) và được biểu diễn như sau:



$$1 \quad ax + by = c$$

Xem thêm: [Thuật toán Euclid mở rộng](#) 

## Hình học

Đôi khi, bài toán yêu cầu ta tìm phần giao của các hình chữ nhật với các cạnh song song trục tọa độ. Có rất nhiều cách để biểu diễn một hình chữ nhật. Đối với hệ trục tọa độ Cartesian, thì cách biểu diễn phổ biến nhất chính là lưu giữ giá trị tọa độ của góc trái - dưới và góc phải - trên của hình chữ nhật.

Giả sử chúng ta có hai hình chữ nhật  $R1$  và  $R2$ . Đặt  $(x1, y1)$  là tọa độ góc trái - dưới,  $(x2, y2)$  là tọa độ góc phải - trên của hình chữ nhật  $R1$ . Tương tự với  $R2$ , ta đặt  $(x3, y3)$ ,  $(x4, y4)$  lần lượt là tọa độ góc trái - dưới và góc phải - trên của  $R2$ . Phần giao của  $R1$  và  $R2$  sẽ là hình chữ nhật  $R3$  có tọa độ trái - dưới là  $(\max(x1, x3), \max(y1, y3))$  và tọa độ góc phải - trên là  $(\min(x2, x4), \min(y2, y4))$ . Nếu  $\max(x1, x3) > \min(x2, x4)$  hoặc  $\max(y1, y3) > \min(y2, y4)$  thì sẽ không tồn tại hình chữ nhật  $R3$ , hay nói cách khác là  $R1$  và  $R2$  không giao nhau. Công thức này còn được mở rộng ra hơn không gian 2 chiều giống như là trong bài [CuboidJoin \(SRM 191, Div 2 Hard\)](#) .

Thường thì, khi làm bài chúng ta sẽ phải đối mặt với những đa giác mà đỉnh của nó là số nguyên. Những đa giác đó được gọi là [Đa giác lưới](#)  (lattice polygon). Trong phần hướng dẫn cơ bản về khái niệm hình học ([Geometry Concepts](#) ) , lbackstrom đã trình bày một cách ngắn gọn cách tính diện tích thông qua các cạnh của đa giác. Bây giờ, giả sử ta không biết được chính xác vị trí các cạnh mà thay vào đó ta có được các giá trị sau:

- |   |  |
|---|--|
| 1 | B = số lượng điểm nằm ở trên cạnh đa giác (Boundary) |
| 2 | I = số lượng điểm nằm trong đa giác (Inside)         |

Thật vi diệu, thông qua hai giá trị trên ta có thể tính được diện tích đa giác bằng công thức sau:

1	Diện tích = $B/2 + I - 1$
---	---------------------------

Công thức trên được gọi là **định lý Pick** [\[1\]](#) được chứng minh bởi *Georg Alexander Pick* (1859 – 1943). Để chứng minh được rằng định lý Pick có tính chất bao quát cho toàn bộ mọi đa giác lưới, ta chứng minh theo 4 bước:

- ▶ Bước 1: chứng minh định lý Pick đúng với mọi hình chữ nhật lưới, có cạnh song song với trục tọa độ.
- ▶ Bước 2: chứng minh được định lý Pick vẫn đúng với mọi tam giác vuông có hai cạnh song song với trục tọa độ (bởi chúng là phân nửa diện tích một hình chữ nhật lưới có tọa độ một đỉnh là góc vuông của tam giác và có cạnh là các cạnh góc vuông của tam giác).
- ▶ Bước 3: chứng minh định lý Pick đúng với mọi tam giác, bởi một tam giác bất kỳ đều có thể biểu diễn thành một hình chữ nhật bằng cách ghép cặp với một vài tam giác vuông khác.
- ▶ Bước 4: ta thấy rằng mọi cặp 2 đa giác lưới có chung cạnh sẽ tạo thành một đa giác lưới khác bằng cách xóa đi cạnh chung, và mọi đa giác lưới đều có thể tạo ra bằng cách ghép các hình tam giác.

Tổng hợp lại 4 bước trên và ta sẽ thu được kết quả rằng định lý Pick đúng với mọi đa giác lưới. Định lý Pick khá hữu ích khi ta cần tìm số điểm có tọa độ nguyên bên trong một đa giác lớn.

Một công thức đáng nhớ khác đó chính là công thức **Euler** [\[2\]](#) dành cho các khối đa diện. Khối đa diện được định nghĩa là một khối (một đa giác) mà ta có thể chia thành nhiều đa giác nhỏ hơn. Các đa giác nhỏ hơn được gọi là mặt, cách đỉnh của mặt cũng được gọi là đỉnh của khối đa diện và tương tự, các cạnh của mặt cũng gọi là cạnh. Dưới đây là công thức Euler.

1	$V - E + F = 2$ , với:
2	$V$ = số đỉnh của khối
3	$E$ = số cạnh của khối
4	$F$ = số mặt của khối

Ví dụ, xét một hình vuông mà cả hai đường chéo đều được vẽ. Ta sẽ có  $V = 5$ ,  $E = 8$  và  $F = 5$  (phần nằm ngoài hình vuông cũng được coi như là một mặt), ta sẽ có  $V - E + F = 2$ .

Ta có thể sử dụng quy nạp để chứng minh nó như sau: Ta sẽ bắt đầu với  $V = 2$ , vì mỗi đỉnh đều cần phải có ít nhất một cạnh. Và nếu  $V = 2$  thì chỉ có xảy ra duy nhất một dạng khối đa diện mà thôi. Đó là dạng khối đa diện mà hai cạnh sẽ nối với  $E$  đỉnh. Khối đa diện này cũng sẽ có  $E$  mặt, ( $E - 1$  mặt bên trong hình và 1 mặt ngoài hình). Thế nên  $V - E + F = 2 - E + E = 2$  (điều phải chứng minh). Giờ ta giả sử định lý Euler đúng với  $V$  từ  $[2, n]$ , xét  $V = n + 1$ . Chọn một cạnh  $w$  bất kỳ. Giờ giả định  $w$  nối với tất cả những lưới còn lại có  $G$  đỉnh. Nếu bỏ đi cạnh  $w$  này và những đỉnh mà nó nối, ta sẽ có một lưới với  $n$  cạnh,  $E - G$  đỉnh và  $F - G + 1$  mặt. Từ giả định trên ta có:

$$\begin{array}{l|l} 1 & (n) - (E - G) + (F - G + 1) = 2 \\ 2 & (n+1) - E + F = 2 \end{array}$$

Vậy với  $V = n + 1$ , ta cũng vẫn thu được kết quả là  $V - E + F = 2$ . Vậy là với phương pháp quy nạp toán học, ta đã chứng minh được định lý Euler.

## Hệ cơ số:

Một trong những bài toán mà người thi Topcoder đối mặt nhiều nhất đó chính là việc chuyển đổi số từ hệ cơ số nhị phân sang hệ cơ số thập phân và ngược lại (cùng với nhiều hệ cơ số khác).

Vậy hệ cơ số nghĩa là gì? Ta sẽ bắt đầu với hệ cơ số tiêu chuẩn (hệ thập phân). Xét số 4325 ở hệ cơ số 10. Ta thấy 4325 có thể phân tích thành  $5 + 2 \cdot 10 + 3 \cdot 10 \cdot 10 + 4 \cdot 10 \cdot 10 \cdot 10$ . Chú ý rằng mỗi "giá trị" của chữ số tiếp theo sau của số được nhân thêm tích số 10 khi ta xét từ phải qua trái. Hệ nhị phân cũng hoạt động theo cách tương tự như thế. Nó sử dụng hai chữ số 0 và 1 và "giá trị" của từng chữ số sẽ được nhân thêm 2 khi ta xét từ phải qua trái. Ví dụ, số 1011 ở dạng nhị phân có thể biểu diễn thành  $1 + 1 \cdot 2 + 0 \cdot 2 \cdot 2 + 1 \cdot 2 \cdot 2 \cdot 2 = 1 + 2 + 8 = 11$  và trở lại thành số ở hệ thập phân là 11. Và đây chính là cách để ta chuyển từ số ở hệ nhị phân (hoặc bất kỳ hệ số nào) về hệ cơ số thập phân. Sau đây là đoạn mã để chuyển số  $n$  từ một hệ cơ số  $b$  ( $2 \leq b \leq n$ ) thành hệ cơ số thập phân.

### Java

```

1 public int toDecimal(int n, int b)
2 {
3     int result = 0;
4     int multiplier = 1;
5
6     while(n > 0)
7     {
8         result += n % 10 * multiplier;
9         multiplier *= b;
10        n /= 10;
11    }
12
13    return result;
14 }
```

### Pascal

```

1 function ToDecimal(n, b : Integer): Integer;
2 var
3     mul : Integer = 1;
4
5 begin
6     result := 0;
7
8 
```



```

9      while (n > 0) do
10         begin
11             result := result + (n mod 10) * mul;
12             mul := mul * b;
13             n := n div 10;
14         end;
      end;

```

Người dùng Java hẳn sẽ rất vui khi biết rằng vẫn có cách khác đơn giản hơn để thực hiện việc này.

```

1 | return Integer.parseInt("" + n, b);

```

Để chuyển một số từ hệ thập phân về hệ nhị phân cũng khá đơn giản. Giả sử ta muốn chuyển số 43 từ hệ thập phân về hệ nhị phân. Tại bước đầu của công thức ta sẽ chia đôi 43 và lưu lại giá trị phần dư, tiếp tục xử lý với số được chia đôi đó cho đến khi nó bằng 0. Và danh sách số dư sau cuối cùng cũng chính là cách biểu diễn số nhị phân cần tìm.

```

1 | 43/2 = 21 + dư 1
2 | 21/2 = 10 + dư 1
3 | 10/2 = 5  + dư 0
4 | 5/2  = 2  + dư 1
5 | 2/2  = 1  + dư 0
6 | 1/2  = 0  + dư 1

```

Vậy 43 sẽ được biểu diễn thành 101011 ở dạng nhị phân. Bằng việc thay số 2 thành số b, ta có thể chuyển từ hệ thập phân về bất kỳ hệ cơ số b nào (2):

## Java

```

1 | public int fromDecimal(int n, int b)
2 | {
3 |     int result = 0;
4 |     int multiplier = 1;
5 |
6 |     while(n > 0)
7 |     {
8 |         result += n % b * multiplier;
9 |         multiplier *= 10;
10 |        n /= b;
11 |    }
12 |
13 |    return result;
14 | }

```

## Pascal

```

1  function FromDecimal(n, b : Integer): Integer;
2  var
3      mul : Integer = 1;
4  begin
5      result := 0;
6
7      while (n > 0) do
8          begin
9              result := result + (n mod b) * mul;
10             mul := mul * 10;
11             n := n div b;
12         end;
13 end;

```

Đối với trường hợp  $b$  lớn hơn 10, ta sẽ sử dụng các ký tự không phải chữ số để đại diện cho các số từ 10 trở lên. Ví dụ, ta sẽ để 'A' đại diện cho 10, 'B' cho 11, ... Đoạn mã sau đây sẽ cho phép ta chuyển từ số thập phân sang bất kỳ hệ cơ số nào (lên đến 20).

### Java

```

1  public String fromDecimal2(int n, int b)
2  {
3      String chars = "0123456789ABCDEFGHIJ";
4      String result = "";
5
6      while(n > 0)
7      {
8          result = chars.charAt(n%b) + result;
9          n /= b;
10     }
11
12     return result;
13 }

```

### Pascal

```

1  function FromDecimal2(n, b : Integer): String;
2  var
3      chars : string = '0123456789ABCDEFGHIJ';
4  begin
5      result := '';
6
7      while (n > 0) do
8          begin
9              result := chars[n mod b + 1] + result;
10             n := n div b;
11         end;
12     end;
13 end;

```

```

12 |         end;
    |     end;

```

Trong Java, ta có thể sử dụng những cách tắt sau để chuyển từ hệ thập phân sang như hệ số thông dụng khác, như là hệ nhị phân, hệ bát phân, hệ thập lục phân.

```

1 | Integer.toString(n);
2 | Integer.toOctalString(n);
3 | Integer.toHexString(n);

```

## Phân số và số phức

Phân số thường được gặp rất nhiều trong các bài tập. Thường thì vấn đề khó khăn nhất mà ta cần phải đối mặt đó chính là việc biểu diễn các phân số đó. Mặc dù nó hoàn toàn khả thi trong việc tạo ra một lớp (class) khác để lưu trữ một cách đầy đủ các thông tin về phân số, nhưng trong nhiều trường hợp thì cách thường dùng nhất là sử dụng mảng lưu 2 giá trị bằng cách ghép cặp (pair). Ý tưởng là ta sẽ lưu số đầu tiên là tử số, số thứ hai là mẫu số. Ta sẽ bắt đầu bằng việc nhân hai phân số  $a$  và  $b$ :

### Java

```

1 | public int[] multiplyFractions(int[] a, int[] b)
2 | {
3 |     int[] c = {a[0] * b[0], a[1] * b[1]};
4 |     return c;
5 | }

```

### Pascal

```

1 | type
2 |     ps = record
3 |         x, y : Integer;
4 |     end;
5 |
6 | function multiplyFractions(a, b : ps): ps;
7 | var
8 |     c : ps;
9 | begin
10 |     c.x := a.x * b.x;
11 |     c.y := a.y * b.y;
12 |     exit(c);
13 | end;

```

Cộng phân số thì có một chút phức tạp hơn, bởi chỉ có những phân số có cùng mẫu số mới có thể cộng trực tiếp. Trước hết ta sẽ tìm mẫu số chung của hai phân số và nhân tử số hai phân số theo tỉ số giữa mẫu số cũ với mẫu số chung. Mẫu số chung là số có thể chia hết cho cả mẫu của hai phân số, hay đơn giản hơn đó chính là

bội chung nhỏ nhất của hai mẫu số. Ví dụ để cộng hai phân số  $4/9$  và  $1/6$ . Bội chung nhỏ nhất của 9 và 6 là 18. Để chuyển đổi hai phân số ta sẽ lấy phân số thứ nhất nhân cho  $2/2$  và số thứ hai cho  $3/3$ .

$$1 \quad \left| \quad 4/9 + 1/6 = (4 \cdot 2)/(9 \cdot 2) + (1 \cdot 3)/(6 \cdot 3) = 8/18 + 3/18 \right.$$

Một khi hai phân số đã có mẫu số bằng nhau, thì ta chỉ cần đơn giản cộng hai phân số đó lại và nhận kết quả cuối cùng là phân số  $11/18$ . Trừ cũng rất đơn giản, chỉ khác nhau ở bước cuối cùng.

$$1 \quad \left| \quad 4/9 - 1/6 = 8/18 - 3/18 = 5/18 \right.$$

Đây là đoạn mã cho chương trình cộng hai phân số:

### Java

```

1 public int[] addFractions(int[] a, int[] b)
2 {
3     int denom = LCM(a[1], b[1]);
4     int[] c = {denom / a[1] * a[0] + denom / b[1] * b[0], denom};
5     return c;
6 }
```

### Pascal

```

1 function addFractions(a, b : ps): ps;
2 var
3     denom : Integer;
4 begin
5     denom := LCM(a.y, b.y);
6     result.x := denom div a.y * a.x + denom div b.y * b.x;
7     result.y := denom;
8 end;
```

Cuối cùng, nó rất cần thiết để biết được cách tối giản phân số thành phân số tối giản. Và phân số chỉ tối giản khi và chỉ khi ước chung lớn nhất của tử số và mẫu số là 1. Chúng ta sẽ làm như sau:

### Java

```

1 public void reduceFraction(int[] a)
2 {
3     int b = GCD(a[0], a[1]);
4     a[0] /= b;
5
6 }
```

```

    a[1] /= b;
}

```

## Pascal

```

1  procedure reduceFraction(var a : ps);
2  var
3      b : Integer;
4  begin
5      b := GCD(a.x, a.y);
6      a.x := a.x div b;
7      a.y := a.y div b;
8  end;

```

Bằng phương pháp tương tự, ta cũng có thể biểu diễn được số phức. Một cách tổng quát, số phức được biểu diễn dưới dạng  $a + ib$  với  $a, b$  là số thực và  $i$  là căn bậc hai của  $-1$ . Ví dụ, để cộng hai số phức, ta sẽ cộng đơn giản như sau:

$$\begin{aligned}
 & m + n \\
 &= (a + bi) + (c + di) \\
 &= (a + c) + (b + d)i
 \end{aligned}$$

Việc nhân hai số phức cũng tương tự như việc nhân số thực, trừ việc  $i * i$  bằng  $-1$ .

```

1  m * n
2  = (a + ib) * (c + id)
3  = ac + iad + ibc + (i^2)bd
4  = (ac - bd) + i(ad + bc)

```

Bằng việc lưu trữ phần số thực ở phần tử thứ nhất và phần số phức ở phần tử thứ hai trong một mảng hai giá trị như đoạn mã dưới đây:

## Java

```

1  public int[] multiplyComplex(int[] m, int[] n)
2  {
3      int[] prod = {m[0]*n[0] - m[1]*n[1], m[0]*n[1] + m[1]*n[0]};
4      return prod;
5  }

```

## Pascal

```

1  function multiplyComplex(m, n : ps): ps;
2  begin
3

```

```
4 | result.x := m.x * n.x - m.y * n.y;  
5 | result.y := m.x * n.y + m.y * n.x;  
  | end;
```

## Tổng kết:

Tổng kết lại, tôi chỉ muốn nói rằng bạn không thể đạt rating cao ở Topcoder mà không nắm rõ Toán học và những thuật toán nêu trên. Thường thì một trong những chủ đề Toán học thường gặp nhất trong các bài tập đó là số nguyên tố. Tiếp đó là những bài về hệ cơ số, mà nguyên nhân chủ yếu là vì máy tính hoạt động dựa trên hệ nhị phân, nên ta cần phải biết cách chuyển từ hệ nhị phân về hệ thập phân. Còn công thức tính ước chung lớn nhất (GCD) và bội chung nhỏ nhất (LCM) thì được dùng nhiều trong cả những bài tập "thuần toán" và cả hình học. Cuối cùng, tôi viết chủ đề Phân số và số phức là việc nó không chỉ cần thiết cho việc thi Topcoder, mà bởi nó còn vô cùng quan trọng khi làm việc với những con số.

Được cung cấp bởi [Wiki.js](#)