

# Fun with Bits

## Fun with Bits

Bài viết bởi [bmerry](#) [🔗](#).

Nguồn: [Topcoder](#) [🔗](#)

## Giới thiệu

Hầu hết các kĩ thuật tối ưu dùng trong các kì thi Topcoder đều là những kĩ thuật cao cấp, có nghĩa là, các kĩ thuật này tối ưu trực tiếp thuật toán hơn là tối ưu cách cài đặt. Tuy nhiên, có một kĩ thuật tối ưu cơ bản nhưng sử dụng rất hiệu quả là thao tác bit (bit manipulation), hay sử dụng những bit thuộc biểu diễn của một số nguyên để biểu diễn một tập hợp. Nó không chỉ làm tăng tốc độ chạy, giảm dung lượng bộ nhớ, mà còn làm code chúng ta trở nên đơn giản hơn.

Mình sẽ bắt đầu bằng việc nhắc lại một số kiến thức cơ bản về bit, trước khi nghiên cứu những kĩ thuật sâu hơn.

## Cơ bản

Những thứ quan trọng nhất trong thao tác bit là những toán tử trên bit (bit-wise operator): `&` (and), `|` (or), `~` (not) và `^` (xor). Chắc hẳn các bạn đã quá quen với ba toán tử đầu tiên trong các phép toán logic ( `&&` , `||` , và `!` ). Dưới đây là bảng chân trị (truth tables):

A	B	!A	A && B	A    B	A ^ B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Toán tử bit trên số nguyên cũng thực hiện giống vậy, chỉ khác một thứ là thay vì chuyển các tham số về true hoặc false, thì các toán tử bit được thực hiện trên các bit của các tham số. Do đó, nếu **A** là **1010** và **B** là **1100**, thì

1		<b>A &amp; B = 1000</b>
2		<b>A   B = 1110</b>
3		<b>A ^ B = 0110</b>
4		<b>~A = 11110101</b> (số chữ số 1 phụ thuộc vào kiểu dữ liệu của A)

Hai toán tử mà chúng ta cần phải biết nữa đó là toán tử dịch bit **a << b** và **a >> b**. Toán tử đầu tiên là dịch tất cả các bit của a sang trái b vị trí; Toán tử thứ hai cũng giống vậy nhưng dịch sang phải. Với những giá trị không âm (cũng là những số duy nhất mà ta sẽ xét đến), những bit mới xuất hiện (do dịch trái) sẽ bằng 0. Dịch trái (left-shifting) **b** bit đồng nghĩa với việc nhân với  $2^b$  và dịch phải (right-shifting) đồng nghĩa với chia nguyên cho  $2^b$ . Dịch bit được sử dụng nhiều nhất để truy cập vào một bit bất kì, ví dụ, **1 << x** là một số nhị phân với bit thứ x bằng 1 và các bit khác bằng 0 (bit luôn luôn được đếm từ bên phải (least-significant), đếm bắt đầu từ 0).

Thông thường, chúng ta sẽ sử dụng một số nguyên để biểu diễn một tập hợp với miền giá trị lên đến 32 giá trị (hoặc 64 nếu sử dụng số nguyên 64 bit), với bit 1 cho biết phần tử đó có trong tập hợp và bit 0 thì không có. Sau đó thì các phép toán thì không có gì phức tạp, trong đó **ALL\_BITS** trả về số bit 1, tương ứng với số phần tử có trong tập hợp:

Phép toán hợp (Set union)

**A | B**

Phép toán giao (Set intersection)

**A & B**

Phép toán hiệu (Set subtraction)

**A & ~B**

Phép toán phủ định (Set negation)

**ALL\_BIT ^ A**

Gán bit bằng 1 (Set bit)

**A |= 1 << bit**

Gán bit bằng 0 (Clear bit)

**A &= ~(1 << bit)**

Truy cập giá trị (Test bit)

**(A & 1 << bit) != 0**

# Tách từng bit

Trong phần này, chúng ta sẽ bàn đến việc tìm vị trí của bit 1 cao nhất (bit 1 nằm xa nhất về bên trái) và thấp nhất (bit 1 nằm xa nhất về bên phải) trong một số. Đây là những phép toán cơ bản để tách một tập hợp ra thành những phần tử.

Tìm bit 1 thấp nhất khá là đơn giản, chỉ cần kết hợp đúng đắn giữa toán tử bit và phép toán số học. Giả sử chúng ta muốn tìm bit 1 thấp nhất của số  $x$  ( $x$  khác 0). Nếu chúng ta trừ 1 từ  $x$  thì bit này được xoá, nhưng tất cả các bit một khác vẫn còn. Do đó,  $x \& \sim(x - 1)$  chỉ chứa duy nhất bit 1 thấp nhất của  $x$ . Tuy nhiên, cách này chỉ cho chúng ta biết giá trị của bit đó, không phải là vị trí.

Nếu chúng ta muốn biết vị trí của bit 1 cao nhất và thấp nhất, cách tiếp cận đơn giản nhất đó là duyệt qua các bit (từ trái qua hay từ phải qua) cho tới khi tìm được bit 1 đầu tiên. Lúc đầu, ta có thể cảm thấy cách làm này hơi chậm vì không tận dụng được lợi thế gì về bit. Tuy nhiên, nếu xác suất  $2^N$  tập con của miền giá trị  $N$  phần tử đều bằng nhau, thì trung bình vòng lặp chỉ cần 2 lần chạy, thật ra đây là phương pháp nhanh nhất.

CPU 386 có hỗ trợ duyệt bit (bit scanning): BSF (bit scan forward) và BSR (bit scan reverse). GCC cung cấp những hỗ trợ này qua những hàm xây dựng sẵn (built-in functions) `__builtin_ctz` (đếm số chữ số 0 đứng cuối) và `__builtin_clz` (đếm số chữ số 0 đứng đầu). Đây là những cách tiện lợi nhất để tìm vị trí của bit dành cho lập trình viên C++ ở Topcoder. Lưu ý: giá trị trả về là *undefined* nếu tham số đầu vào bằng 0.

Cuối cùng, còn một phương pháp khác để thay thế trong những testcase mà dùng vòng lặp tốn nhiều thời gian. Sử dụng mỗi byte của số nguyên 4 byte hoặc số nguyên 8 byte để tính trước bảng 256 phần tử lưu trữ vị trí của bit 1 cao nhất (thấp nhất) trong byte đó. Bit 1 cao nhất (thấp nhất) của số nguyên là giá trị lớn nhất (giá trị nhỏ nhất) của bảng này. Phương pháp này được đề cập đến để làm đa dạng thêm các phương pháp, tốc độ cũng chưa được đánh giá rõ ràng qua các kì thi Topcoder.

## Đếm số bit

Chúng ta có thể dễ dàng kiểm tra một số có phải là lũy thừa của 2 bằng cách xoá bit 1 thấp nhất và kiểm tra xem nếu kết quả có bằng 0 chưa. Tuy nhiên, trong một số trường hợp chúng ta cần phải biết có bao nhiêu bit đã được set (bit đã được set là bit 1, bit chưa được set là bit bằng 0), chúng ta cần phải thực hiện nhiều việc phức tạp hơn tí.

GCC có một hàm gọi là `__builtin_popcount` thực hiện đúng những thứ ta cần. Tuy nhiên, không giống như hàm `__builtin_ctz`, nó không được chuyển thành những chỉ thị trên phần cứng (ít nhất là trên x86). Thay vào đó, nó sẽ sử dụng phương pháp lưu bảng giống như trên đã trình bày để tìm kiếm bit. Phương pháp này khá là hiệu quả và cũng cực kì tiện lợi.

Những người dùng các ngôn ngữ khác không thể dùng cách này (mặc dù họ có thể cài đặt lại nó). Nếu một số được dự đoán rằng có rất ít bit 1, một phương pháp thay thế là chỉ cần lặp lại quá trình tìm bit 1 thấp nhất và xoá nó.

## Tất cả các tập con

Một ưu điểm lớn của thao tác bit là việc duyệt qua tất cả các tập con của một tập hợp N phần tử rất đơn giản: mỗi số nguyên đại diện cho một tập con. Hơn thế nữa, nếu A là tập con của B thì số nguyên đại diện cho A sẽ nhỏ hơn số nguyên đại diện cho B, rất tiện lợi cho việc kết hợp với quy hoạch động.

Việc duyệt qua tất cả các tập con của một tập con khác cũng rất dễ dàng (được biểu diễn bằng một dãy bit), nếu bạn không quan tâm đến thứ tự duyệt ngược lại (hoặc có thể lưu các tập hợp vào một danh sách và đi ngược lại). Mẹo được sử dụng cũng giống như việc tìm bit 1 thấp nhất vậy. Nếu chúng ta trừ đi 1 từ tập con, thì phần tử đại diện bởi bit 1 đó sẽ được xoá, và mỗi phần tử 0 phía bên phải nó đều trở thành 1. Tuy nhiên, chúng ta chỉ muốn các phần tử được chọn là những phần tử có trong tập cha. Nên bước lặp chỉ cần thay thế ngắn gọn bằng `i = (i - 1) & superset`.

### Cài đặt:

```
1 // xét tất cả các tập con khác rỗng của S
2 for (int i = S; i > 0; i = (i - 1) & S); {
3 }
```

## Chỉ cần một bit sai cũng 0 điểm

Có một số lỗi mà mọi người thường mắc phải khi sử dụng các thao tác bit. Chú ý cẩn thận với code của bạn.

1. Khi thực hiện phép toán dịch, `a << b`, kiến trúc x86 chỉ sử dụng the 5 bit thấp nhất của b (6 cho số nguyên 64 bit). Điều này có nghĩa là dịch trái (hoặc phải) 32 bit đồng nghĩa là không làm gì cả, ngoài việc xoá tất cả các bit. Việc này cũng được nhắc đến trong Java và C#; C99 đề cập rằng việc dịch ít nhất bằng giá trị của số sẽ trả về kết quả là undefined. Lịch sử: máy 8086 sử dụng full shift register, thường thao tác này còn được dùng để kiểm tra processor mới.
2. Toán tử `&` và `|` có độ ưu tiên thực hiện thấp hơn phép so sánh. Điều đó có nghĩa là `x & 3 == 1` bằng với `x & (3 == 1)`, đây là những lỗi bạn không hề muốn mắc phải.

3. Nếu bạn muốn viết những dòng code C/C++ dùng ở nhiều nơi, hãy đảm bảo rằng bạn sử dụng kiểu unsigned, cụ thể trong trường hợp bạn có ý định sử dụng bit cao nhất (top-most). C99 nói rằng dịch bit trên giá trị âm sẽ trả về undefined. Java chỉ có kiểu dữ liệu có dấu: `>>` sẽ tự động thay đổi dấu của giá trị nếu cần thiết (nhưng đây là thứ bạn thường không muốn xảy ra), nhưng toán tử đặc biệt này của Java là `>>>` sẽ thêm các số 0 vào phía trước sau khi dịch.

## Một số mẹo hay (và cute)

Có một số mẹo hay có thể sử dụng với thao tác bit.

Những mẹo này khá hay để đem đi chém gió với bạn bè, nhưng thông thường hiệu quả cũng không cải thiện lắm ở thực tế.

### Đảo thứ tự các bit trong một số nguyên

```
1 | x = ((x & 0xaaaaaaaa) >> 1) | ((x & 0x55555555) << 1);
2 | x = ((x & 0xcccccccc) >> 2) | ((x & 0x33333333) << 2);
3 | x = ((x & 0xf0f0f0f0) >> 4) | ((x & 0x0f0f0f0f) << 4);
4 | x = ((x & 0xff00ff00) >> 8) | ((x & 0x00ff00ff) << 8);
5 | x = ((x & 0xffff0000) >> 16) | ((x & 0x0000ffff) << 16);
```

Bài tập là bạn hãy dùng kĩ thuật phía trên để đếm số bit với kiểu dữ liệu word.

### Duyệt qua tất cả các tập con có k phần tử

```
1 | int s = (1 << k) - 1;
2 | while (!(s & 1 << N))
3 | {
4 |     //làm gì đó với s
5 |     int lo = s & ~(s - 1); //bit 1 thấp nhất
6 |     int lz = (s + lo) & ~s; //bit 0 thấp nhất trên lo
7 |     s |= lz; //thêm lz vào tập hợp
8 |     s &= ~(lz - 1); //reset bit phía dưới lz
9 |     s |= (lz / lo / 2) - 1; //đặt lại đúng số bit ở cuối
10| }
```

Trong C, dòng cuối có thể viết là `s |= (lz >> ffs(lo)) - 1` để tránh phép chia.

Xác định `x ? y : -y`, trong đó x bằng 0 hoặc 1  
`(-x ^ y) + x`

Câu lệnh trên chạy được trên kiến trúc số bù 2 (tồn tại ở hầu hết các máy tính bạn thấy ngày nay), trong đó số âm được biểu diễn bằng cách đảo tất cả các bit và cộng thêm cho 1. Chú ý rằng trên i686 và trước đó, câu lệnh này chạy khá hiệu quả (không cần rẽ nhánh) nhờ vào lệnh **CMOVE** (di chuyển có điều kiện).

## Bài tập mẫu

TCCC 2006, Round 1B Medium [🔗](#)

Với mỗi thành phố, giữ một bit-set của những thành phố kề nó. Một khi một phần của nhà máy đã được chọn (đệ quy), AND những bit-set đó lại sẽ cho ra một bit-set mới mô tả những vị trí có thể của những phần của nhà máy. Nếu bit-set này có  $k$  bit, thì có  $C_m^k$  cách để chọn các phần của nhà máy.

TCO 2006, Round 1 Easy [🔗](#)

Số lượng nút nhỏ cho thấy rằng bài này có thể giải quyết bằng việc xét tất cả các tập con. Với mỗi tập con ta xét 2 trường hợp: nút nhỏ nhất không có trao đổi gì cả, trong trường hợp ta xét tập con mà không có nó, hoặc nó trao đổi với một số nút, ta sẽ xét các tập hợp không có nó và các nút mà nó trao đổi. Code bài giải rất ngắn gọn:


```
1 static int dp[1 << 18];
2
3 int SeparateConnections::howMany(vector <string> mat)
4 {
5     int N = mat.size();
6     int N2 = 1 << N;
7     dp[0] = 0;
8     for (int i = 1; i < N2; i++)
9     {
10         int bot = i & ~(i - 1);
11         int use = __builtin_ctz(bot);
12         dp[i] = dp[i ^ bot];
13         for (int j = use + 1; j < N; j++)
14             if ((i & (1 << j)) && mat[use][j] == 'Y')
15                 dp[i] = max(dp[i], dp[i ^ bot ^ (1 << j)] + 2);
16     }
17     return dp[N2 - 1];
18 }
```

SRM 308, Division Medium [🔗](#)

Cái bảng chứa 36 hình vuông và những con cờ không thể phân biệt được, nên những vị trí có thể được mã hoá vào số nguyên 64 bit. Bước đầu tiên là liệt kê tất cả các bước đi hợp lệ. Bất kì

bước đi hợp lệ nào cũng đều được mã hoá sử dụng 3 trường: 1 trạng thái trước, 1 trạng thái sau và một mask dùng để định nghĩa phần nào của trạng thái trước là quan trọng. Một bước đi có thể được tạo ra từ trạng thái hiện tại nếu  $(current \& mask) == before$ . Nếu nó được tạo ra, thì trạng thái mới là  $(current \& \sim mask) | after$ .

[SRM 320, Division 1 Hard](#) 

Điều kiện cho ta biết rằng chỉ có nhiều nhất 8 cột (nếu có nhiều hơn, ta có thể đổi giữa dòng và cột), nên chúng ta có thể xét từng cách để một dòng. Một khi chúng ta có thông tin này, ta có thể giải quyết vấn đề còn lại của bài toán (xem [tutorial](#)  để biết thêm chi tiết). Do đó ta cần một danh sách tất cả số nguyên n bit mà không có 2 bit 1 kề nhau, và ta cũng cần biết có bao nhiêu bit 1 trong mỗi dòng như vậy. Đây là code của mình:

```
1 | for (int i = 0; i < (1 << n); i++)
2 | {
3 |     if (i & (i << 1)) continue;
4 |     pg.push_back(i);
5 |     pgb.push_back(__builtin_popcount(i));
6 | }
```

Được cung cấp bởi [Wiki.js](#)