

Phép toán bit

Phép toán bit

Người viết: Nguyễn Minh Nhật - HUS High School for Gifted Students

Reviewer:

- ▶ Hồ Ngọc Vĩnh Phát - VNUHCM - University of Science
- ▶ Nguyễn Đức Kiên - VNU - University of Engineering and Technology
- ▶ Lê Minh Hoàng - VNUHCM - University of Science

Giới thiệu

Các phép toán với bit (Bitwise Operators) là một tập hợp các toán tử và hàm dành riêng cho việc thực hiện các thao tác biến đổi và tính toán trên các bit của một số nguyên (ví dụ như `int` hay `long long` trong C++).

Lưu ý trước khi đọc bài viết

Trước khi đọc bài viết này, bạn cần trang bị kiến thức về các chủ đề sau:

- ▶ Biểu diễn nhị phân của số nguyên.
- ▶ Cách sử dụng AND (&) và OR (|) đối với các toán hạng bool trong C++.

Các khái niệm sau được sử dụng xuyên suốt bài viết:

- ▶ **Bảng chân trị (Truth Table)** của một toán tử bit có thể hiểu nôm na là tất cả các trường hợp đầu vào/đầu ra của phép toán đó.
- ▶ **Biểu diễn dạng nhị phân của một số** được đánh dấu bằng tiền tố `0b`. Chẳng hạn, với số `12` có biểu diễn nhị phân là `1100`, ta viết `12 = 0b1100`. Đây cũng là cách viết được chấp nhận trong code C++.

- ▶ **Bitmask** là một chuỗi các chữ số 0, 1, hay các bit. Trong đó, việc bit thứ i được bật tương ứng với việc một phần tử/đại lượng thứ i nào đó được sử dụng.
Ví dụ đơn giản nhất của bitmask là biểu diễn một tập con của một tập hợp A cho trước. Chẳng hạn, $A = \{5, 1, 2, 3, 0, 4\}$, bitmask `0b110010` biểu diễn cho tập con $\{1, 0, 4\}$ của A .
Chú ý: Trong bài viết này, thứ tự các bit của bitmask được đánh số từ phải sang trái, bắt đầu từ 0. Điều này tương tự như chữ số hàng đơn vị, hàng chục, hàng trăm, hàng nghìn trong số thập phân lần lượt được viết từ trái sang phải, từ thấp đến cao.
Trên thực tế, ta sẽ thường biểu diễn bitmask bằng các kiểu số nguyên (ví dụ như các kiểu `int` hay `long long` trong C++).
- ▶ Trong một bitmask, **bit thứ i bật** có nghĩa là bit thứ i của bitmask này có giá trị bằng 1. Tương tự, **bit thứ i tắt** có nghĩa là bit thứ i của bitmask này có giá trị bằng 0.
- ▶ **Undefined Behaviour (UB)** được dùng để chỉ một đoạn code không có hành vi cố định. Nói cách khác, ta không biết đoạn code đó sẽ làm gì. Một đoạn code UB có thể trả về kết quả sai, làm chương trình gặp lỗi, hay trả về các kết quả khác nhau với hai lần chạy khác nhau, hoặc thậm chí là với compiler khác nhau.
- ▶ **Ký hiệu L** được dùng để chỉ số lượng Bit của kiểu số hiện tại đang sử dụng (32 với `int` và 64 với `long long`).

Các toán tử thao tác bit (Bitwise Operators) cơ bản

Toán tử Bitwise AND (&), OR (|) và XOR (^)

Các toán tử này thuộc loại "Toán tử Bit Logic". Việc sử dụng các toán tử loại này có thể được hiểu nôm na là thực hiện các thao tác tương ứng trên từng bit của các toán hạng (operands). Nói cách khác, nếu ký hiệu a_i là bit thứ i của bitmask a , việc thực hiện phép toán $c := a \oplus b$ trong đó a, b, c là các bitmask và \oplus là một phép toán nào đó sẽ tương đương với việc thực hiện $c_i := a_i \oplus b_i \forall 0 \leq i$.

Định nghĩa của các phép toán này như sau:

1. **AND** trả về True khi và chỉ khi cả hai toán hạng là True.

Ví dụ, ta có:

1		<code>0b11100010</code>
2		<code>& 0b10101111</code>
3		<code>= 0b10100010</code>

2. **OR** trả về True khi và chỉ khi ít nhất một toán hạng là True.

Ví dụ, ta có:

```

1 | 0b11100010
2 | | 0b10101111
3 | = 0b11101111

```

3. **XOR** trả về True khi và chỉ khi hai toán hạng có giá trị khác nhau. Một cách hiểu khác cho **XOR** là phép cộng theo modulo 2.

Ví dụ, ta có:

```

1 | 0b11100010
2 | ^ 0b10101111
3 | = 0b01001101

```

Sau đây là bảng chân trị của các toán tử này

a	b	AND	OR	XOR
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Toán tử Bitwise NOT (~)

Toán tử Bitwise NOT có lẽ là toán tử đơn giản nhất. Toán tử này nhận vào một toán hạng *A* và trả về phần bù của toán hạng này. Nói cách khác, định nghĩa của NOT là trả về False khi và chỉ khi toán hạng là True.

Ví dụ, ta có:

```

1 | ~0b10100100
2 | = 0b01011011

```

Chú ý rằng biểu thức trên chỉ đúng trong trường hợp đầu vào là kiểu số có 8 bit. Cụ thể hơn, cần chú ý:

Khi sử dụng phép NOT, những bit không sử dụng ở bên trái cũng sẽ được bật lên. Chẳng hạn, khi thực hiện phép `0b10` với kiểu số `char` (8 bit), ta nhận được `0b11111101` thay vì `0b01`. Trong đa số trường hợp, ta sẽ cần phải tắt các bit được bật thừa này đi.

Toán tử BITSHIFT LEFT (<<)

Định nghĩa của toán tử Bitshift Left là dịch tất cả các bit trong một số nguyên sang trái một lượng nào đó. Nói cách khác, trong phép toán $a \ll b$, các bit của a được dịch sang trái b lần.

Ví dụ, xét số $5 = 0b101$, nếu thực hiện phép toán $0b101 \ll 2$, ta nhận được $0b10100 = 20$.

Nếu quan sát kỹ, bạn sẽ nhận thấy một tính chất thú vị sau của phép toán Bitshift Left: $a \ll b = a * 2^b$. Ta có tính chất này do phép toán Bitshift Left $a \ll b$ có thể hiểu là thêm b chữ số 0 vào cuối biểu diễn nhị phân của số a . Điều này tương tự như việc thêm một chữ số 0 vào cuối biểu diễn thập phân của một số sẽ nhân số đó thêm 10 lần.

Chú ý

Với C++, không nên để phép toán $a \ll b$ của bạn bị tràn số (bit 1 được left shift đến quá giới hạn của kiểu số đang sử dụng) vì sẽ có trường hợp code của bạn bị UB. Để biết cụ thể về các trường hợp này, bạn có thể tham khảo phần [Phép Left Shift tràn số](#).

Ngoài ra, nếu $b < 0$ hoặc $b \geq L$, kết quả trả về của phép toán là không xác định.

Toán tử BITSHIFT RIGHT (>>)

Nếu như Left Shift là thêm chữ số 0 vào bên phải của một số nguyên ở dạng nhị phân, ta có thể hiểu Right Shift là xóa các chữ số ở bên phải.

Ví dụ, xét số $13 = 0b1101$, ta có $0b1101 \gg 2 = 0b11$.

Tương tự với Bitshift Left, ta cũng có tính chất $a \gg b = \lfloor \frac{a}{2^b} \rfloor$ với a nguyên không âm. Nếu $b < 0$ hoặc $b \geq L$, kết quả trả về của phép toán là không xác định. Dù trên thực tế không sử dụng nhiều, nhưng nếu bạn đọc cảm thấy tò mò về trường hợp $a < 0$ thì có thể tham khảo phần [Phân biệt LRS và ARS](#).

Các hàm thao tác Bit

Compiler GCC (là Compiler đi kèm với Codeforces, Dev-C++, và được sử dụng trên các OJ phổ biến) hiện nay hỗ trợ một số các hàm liên quan tới xử lý bit giúp ta thực hiện một số các phép tính thông dụng với độ phức tạp thời gian $O(1)$.

Nếu bạn tới đây để đọc lại tên hàm, đây là bảng TL;DR:

Tên thao tác	Tên hàm	Giá trị trả về	Trường hợp UB
Population Count	<code>__builtin_popcountll(x)</code>	Số Bit bật	
Parity	<code>__builtin_parityll(x)</code>	Số Bit bật modulo 2	
Count Leading Zeroes	<code>__builtin_clzll(x)</code>	Số Bit 0 ở đầu	<code>x == 0</code>
Log2	<code>std::__lg(x)</code>	$\lfloor \log_2(x) \rfloor$	
Count Trailing Zeroes	<code>__builtin_ctzll(x)</code>	Số Bit 0 ở cuối	<code>x == 0</code>
Find First Set	<code>__builtin_ffsll(x)</code>	Số thứ tự của Bit 1 đầu tiên	

Chú ý: Đối với các hàm có dạng `__builtin`, thêm đuôi `ll` sẽ gọi hàm đó với kiểu đầu vào là `unsigned long long`. Để thuận tiện, người viết sẽ bỏ qua đuôi này trong phần tiếp theo.

Hàm Population Count và Parity

GCC cung cấp hàm `__builtin_popcount(x)` (population count) trả về số lượng bit bật trong bitmask x . Chẳng hạn, `__builtin_popcount(0b100101) = 3`.

Ngoài ra, cũng có hàm `__builtin_parity(x)` trả về `__builtin_popcount(x) % 2`. Hàm này thường được sử dụng trong các bài toán liên quan tới bao hàm loại trừ.

Hàm Count Leading Zeroes và Log2

GCC cung cấp hàm `__builtin_clz(x)` Trả về số lượng bit 0 ở bên trái bit 1 cao nhất của biến đầu vào. Chú ý, hàm này trả về kết quả không xác định đối với `x == 0`.

Chẳng hạn, `__builtin_clz(0b10) == 30`. Kết quả này là do kiểu `int` có 32 bit. Cụ thể, số `0b10` khi lưu dưới dạng `int` sẽ có thể được biểu diễn là:

```
0b 0000 0000 0000 0000 0000 0000 0000 0010
```

Ngoài ra cũng có hàm `std::__lg(x) == 31 - __builtin_clz(x) == 63 - __builtin_clzll(x)`. Hàm này trả về $\lfloor \log_2(x) \rfloor$, thường được sử dụng trong cài đặt của Bảng thừa (Sparse Table).

Hàm Count Trailing Zeroes và Find First Set

GCC cung cấp hàm `__builtin_ctz(x)` trả về số lượng bit 0 ở bên phải bit 1 thấp nhất của biến đầu vào. Chú ý, hàm này cũng trả về kết quả không xác định đối với `x == 0`. Chẳng hạn, `__builtin_ctz(0b100100) = 2`.

GCC cũng cung cấp một hàm khác là `__builtin_ffs(x) == __builtin_ctz(x) + 1`. Trong trường hợp `x == 0`, hàm này trả về 0.

Ứng dụng

Truy cập Bit

Một ứng dụng thường thấy của các phép toán bit là đọc và sửa từng bit trong một bitmask.

Chẳng hạn, để truy cập bit thứ i trong bitmask A , ta có thể sử dụng phép toán `A & (1LL<<i)`. Trước khi đọc giải thích của phép toán này, hãy tự mình chạy thử một số ví dụ.

Xét `A = 0b1010010`. Để truy cập bit thứ 4, ta thực hiện phép toán `0b1010010 & (1<<4)` như sau:

```
1 | 0b1010010
2 | & 0b0010000
3 | = 0b0010000
```

Xét phần thứ hai của phép toán, `1<<i`, ta nhận thấy phần này thực hiện thao tác tạo ra một bitmask chỉ có bit thứ i bật. Bitmask này khi được AND với bitmask ban đầu sẽ loại bỏ thông tin của tất cả mọi bit ngoại trừ bit thứ i .

Ngoài ra cũng có một số cách khác để truy cập bit, ví dụ như `(A >> i) % 2` hay `(A >> i) & 1`.

Chú ý: Một lỗi rất hay gặp phải khi sử dụng bitshift để truy cập và chỉnh sửa bit là tràn số. Chẳng hạn, xét dòng code sau đây:

```
1 | bool get_bit(unsigned long long mask, int pos){
2 |     return mask & (1<<pos)
3 | }
```

```
'
}
```

Trong trường hợp $pos \geq 32$, biểu thức $1 << pos$ sẽ bị tràn số do cả 1 và pos đều có kiểu `int`. Để tránh bị tràn số, ta đổi đoạn code trên thành như sau:

```
1 | bool get_bit(unsigned long long mask, int pos){
2 |     return mask & (1ULL << pos);
3 | }
```

Hệ số `ULL` đánh dấu cho compiler biết rằng `1ULL` cần được coi là một số `unsigned long long`. Như vậy, phép $1ULL << pos$ sẽ không còn bị tràn số. Một số các hệ số thường dùng bao gồm: `ULL` cho `unsigned long long`, `LL` cho `long long`, `L` cho `long`, ...

Chỉnh sửa Bit

Sử dụng phương pháp tương tự như phần trên, ta có một số phép sửa bit như sau:

1. Gán một bit bằng 0 với $A \& \sim(1 << i)$.
2. Gán một bit bằng 1 với $A \mid (1 << i)$.
3. Lật (flip) một bit (từ 0 sang 1 hoặc từ 1 sang 0) với $A \wedge (1 << i)$.

Tắt các bit cao nhất của một bitmask

Phép toán $((1 << i) - 1)$ tạo ra bitmask mà trong đó chỉ các bit từ 0 tới $i - 1$ được bật lên.

Như vậy, để tắt tất cả các bit từ vị trí i trở đi, ta có thể sử dụng $A \& ((1 << i) - 1)$. Đây là cách ta loại bỏ các bit thừa sau khi thực hiện phép bitwise NOT.

Biểu diễn tập hợp

Ứng dụng cơ bản nhất của bitmask là biểu diễn một tập con của một tập A cho trước nào đó. Từ ứng dụng này, ta có một dạng bài tên là quy hoạch động trạng thái (dp bitmask).

Từ các phần [Truy cập Bit](#), [Chỉnh sửa Bit](#) và [Tắt các Bit cao nhất](#), ta có một số phép toán cơ bản trên tập hợp như sau:

Hàm	Ký hiệu toán	Code
Giao	$A \cap B$	$A \& B$

Hàm	Ký hiệu toán	Code
Hợp	$A \cup B$	<code>A B</code>
Hiệu	$A \setminus B$	<code>(A ^ B) & A</code>
Hiệu đối xứng	$A \Delta B$	<code>A ^ B</code>
Phần bù	A^C hay A'	<code>~A & (1<<n)-1</code>
Kiểm tra tập con	$A \subseteq B$	<code>(A & B) == A</code>
Tập hợp chỉ có phần tử i	$\{i\}$	<code>1 << i</code>

Lặp qua mọi tập con của tập cho trước

Để lặp qua mọi tập con A của một tập S cho trước, ta viết vòng `for` như sau:

```
1 void loop_subset(const vector<int> &s){
2     for (int mask=0; mask<(1<<s.size()); mask++){
3         vector<int> a;
4         for (int i=0; i<s.size(); i++){
5             if (mask & (1<<i))
6                 a.push_back(s[i]);
7         }
8         // Thực hiện thao tác gì đó với tập con A
9     }
10 }
```

Cài đặt cấu trúc dữ liệu Fenwick Tree

Cách cài đặt [Fenwick Tree](#) ☑️ tối ưu cũng là một trong những ứng dụng thú vị của các toán tử Bit.


Giải các bài toán bao hàm loại trừ

Đề bài

Cho một tập S gồm các số nguyên tố phân biệt. Gọi a là tích các số trong tập S . Trong các số thuộc khoảng $[0, n]$, đếm số số nguyên tố cùng nhau với a .

Thuật toán

Ta lặp qua mọi tập con T của S . Gọi b là tích các số trong tập T , và x là số nhỏ nhất trong khoảng $[0, n]$ chia hết cho b . Nếu T có chẵn phần tử, ta cộng x vào đáp án. Ngược lại, ta trừ x vào đáp án.

Phần chứng minh cho bài toán này bạn đọc có thể tham khảo ở bài viết về [bao hàm loại trừ](#) .

Cài đặt



```
1  unsigned long long solve(const vector<unsigned long long> &a, unsigned long long n) {
2      unsigned long long result = 0;
3      for (int i = 0; i < 1<a.size(); i++){
4          unsigned long long b = 1;
5          for (int j=0; j<a.size(); j++){
6              if (i & 1<j) b *= a[j];
7          }
8          unsigned long long x = result / b + 1;
9          if (__builtin_parity(i)) result -= x;
10         else result += x;
11     }
12     return result;
13 }
```

Chú ý: Đoạn code này chỉ mang tính chất minh họa, do trên thực tế kết quả có thể tràn `unsigned long long`. Tuy nhiên, kết quả của các thao tác tính toán tràn số trên các kiểu `unsigned` được xác định, nên nếu tích các số trong tập A không tràn số, code này sẽ trả về kết quả theo mod 2^{64} .

Tổng hợp một số điều cần chú ý trong C++

Các thao tác bit trong C++ là một bộ công cụ rất mạnh và có hiệu suất cực đỉnh. Tất nhiên, "with great power comes great responsibility". Khi sử dụng bộ công cụ này, rất nhiều những bug thú vị đang chờ đợi bạn.

Thứ tự tính toán (Operator Precedence)

Thứ tự tính toán có thể được hiểu là thứ tự mà C++ sẽ tính toán các biểu thức của bạn, ví dụ như nhân chia trước, cộng trừ sau. Trang [cppreference.com](#)  có dành hẳn một [bài viết](#)  để nói về thứ tự này.

Một số điểm cần chú ý trong thứ tự tính toán của C++ bao gồm:

- Các phép bitshift `<<`, `>>` đứng sau phép `+`, `-`.

► Phép `&`, `^`, `|` đứng sau phép `==` và các phép `+`, `-`, `*`, `/`, `<<`, `>>`.

Trong mọi trường hợp, kể cả khi đã nhớ kỹ thứ tự tính toán của các toán tử, bạn vẫn nên sử dụng dấu `()` khi làm việc với các toán tử bit để giúp code dễ đọc hơn và hạn chế bug.

Toán tử Bitshift

Trong phép Bitshift Left, nếu toán tử đầu tiên của bạn là một số âm, hoặc kết quả tính toán của bạn tràn số, thì code của bạn sẽ bị UB.

Đối với tất cả mọi phép bitshift, nếu toán tử thứ hai của bạn là một số âm, hoặc có giá trị lớn hơn hoặc bằng số lượng bit có trong kiểu số của kết quả, thì code của bạn bị UB. Trong gần như hầu hết trường hợp, chỉ có 5 hoặc 6 bit cuối cùng (lần lượt với hai trường hợp `int` và `long long`) trong toán tử thứ hai được sử dụng, nên chúng ta có thể đoán trước được code sẽ làm gì. Tuy nhiên, không nên dựa vào tính chất này để viết code.

Tràn số khi truy cập bit

Một lỗi thường gặp của những bạn mới làm quen với các toán tử bit là tràn số khi thực hiện phép `1 << pos` để truy cập bit. Bạn đọc có thể xem phần [Truy cập bit](#) để rõ hơn.

Mở rộng

Lặp qua mọi tập con của một bitmask

Để lặp qua mọi tập con của S , ta viết vòng lặp `for` như sau:

```
1 void loop_mask_subset(int S){
2     for (int mask=S; true; mask = (mask-1) & S){
3         // Thực hiện thao tác nào đó với tập con mask của S
4         if (mask == 0) break;
5     }
6 }
```

Độ phức tạp của vòng lặp trên là $2^{\|S\|}$ với $\|S\|$ là số lượng bit bật của S , chính là số tập con của S .

Như vậy, ta có cách để lặp mọi tập S từ 0 tới 2^n , sau đó lặp mọi tập con T của S một cách hiệu quả.

```
1 void loop_subset_of_all_masks(int n){
2     for (int S = 0; S < 1<<n; S++){
3         // Thực hiện thao tác nào đó với tập con S
4     }
```

```

4 |         for (int T=S; true; T = (T-1) & S){
5 |             // Thực hiện thao tác nào đó với tập con T của S
6 |             }
7 |         }
8 |     }

```

Cách cài đặt trên có độ phức tạp thời gian tối ưu do tất cả các lần lặp đều tạo ra một bộ (S, T) thỏa mãn, và đôi một phân biệt. Ta sẽ chứng minh tổng độ phức tạp thời gian của hai vòng lặp này là $O(3^n)$, thay vì $O(4^n)$.

Dễ dàng nhận thấy, số bước lặp của hai vòng lặp trên có thể viết là:

$$\begin{aligned}
 \sum_{S \subseteq 2^n} \sum_{T \subseteq S} 1 &= \sum_{S \subseteq 2^n} 2^{|S|} \\
 &= \sum_{k=0}^n \sum_{S \subseteq 2^n, |S|=k} 2^k \\
 &= \sum_{k=0}^n \binom{n}{k} 2^k \\
 &= 3^n
 \end{aligned}$$

Nếu bạn thấy chứng minh trên khó hiểu, hãy xem chứng minh của Ứng dụng tiếp theo.

Lặp qua mọi bộ x tập con phân biệt

Bài toán

Cho một tập S độ dài n và một số x . Hãy in ra tất cả các cách chia các phần tử trong S vào x tập hợp không giao nhau, sao cho mỗi phần tử nằm trong đúng một tập hợp.

Nhận xét

Rõ ràng, có $O(x^n)$ tập hợp thỏa mãn. Như vậy, độ phức tạp tốt nhất của bài toán này là $O(x^n)$.

Trường hợp $x = 2$

Rõ ràng, trong trường hợp này, ta chỉ cần lặp qua mọi tập con S của A . Với mỗi lần lặp này, ta nhận được cặp tập hợp $(S, A \setminus S)$.

Trường hợp $x = 3$

Ta sẽ cố gắng mở rộng từ trường hợp $x = 2$ để có được thuật toán cho trường hợp này.

Bước đầu tiên, ta sẽ lặp như trường hợp $x = 2$ để có được cặp tập hợp (A, B) .

Bước thứ hai, ta sẽ lặp mọi tập con C của tập A để nhận được hai tập $(C, A \setminus C)$. Như vậy, bộ tập hợp thỏa mãn đề bài mà ta nhận được sẽ là $(B, A \setminus C, C)$. Chú ý, ở bước này, ta sử dụng kỹ thuật ở [Ứng dụng trước](#).

Cài đặt cho trường hợp này như sau:

```
1 void loop_triplets(int n){
2     int S = (1<<n) - 1;
3     for (int A = S; true; A = (A - 1) & S){
4         int B = S ^ A;
5         for (int C = A; true; C = (C - 1) & A){
6             // In ra B, A^C, C
7             if (C == 0) break;
8         }
9         if (A == 0) break;
10    }
11 }
```

Để ý rằng vòng lặp đầu tiên tương đương với việc lặp A trong khoảng $[0, 2^n)$. Nếu thực hiện thay đổi này, ta sẽ nhận được cài đặt tương đương với hàm `void loop_subset_of_all_masks(int n)` ở trên. Đây cũng là một cách hiểu cho độ phức tạp $O(3^n)$ của hàm này.

Trường hợp tổng quát $x \in \mathbb{Z}^+$

```
1 void generate_partitions(vector<int> &sets, int mask, int x){
2     if (x == 1){
3         sets.push_back(mask);
4         // Hàm thực hiện thao tác gì đó đối với sets
5         solve_for_sets(sets);
6         sets.pop_back();
7         return;
8     }
9     for (int s = mask; true; s = (s - 1) & mask){
10        sets.push_back(s);
11        generate_partitions(sets, mask ^ s, x-1);
12        sets.pop_back();
13        if (s == 0) break;
14    }
15 }
16 int main(){
17     int n = 10, parts = 5;
18     vector<int> sets;
19     generate_partitions(sets, (1<<n)-1, parts);
20 }
```

Tăng tốc cho code

Nếu sử dụng kiểu dữ liệu `unsigned long long`, ta có thể thực hiện 64 phép AND, OR, XOR, hoặc NOT trong một thao tác. Trên thực tế, khi dịch, một số các compiler có thể giúp bạn thực hiện 256 hay thậm chí 512 phép toán như vậy cùng một lúc. Như vậy, một số bài toán với giới hạn như $n \leq 5 * 10^4$ hay thậm chí $n < 10^5$ có thể chạy qua được với độ phức tạp $O(n^2)$. Tuy nhiên, do giới hạn của bài viết, chủ đề này sẽ không được bàn đến ở đây.

Phân biệt Logical Right Shift và Arithmetic Right Shift

Riêng đối với Right Shift, hầu hết các cấu trúc máy tính cung cấp hai loại phép toán khác nhau là Logical Right Shift và Arithmetic Right Shift.

Khác biệt duy nhất giữa hai loại phép toán này là Logical Right Shift điền các bit bên trái mới được thêm bằng 0, trong khi Arithmetic Right Shift điền các bit này là giá trị của bit trái cùng trong số ban đầu (bit thứ 31 đối với kiểu `int`, và bit thứ 63 đối với kiểu `long long`).

Chẳng hạn, ta sử dụng kiểu số `char` có 8 bit, và thực hiện phép toán `0b 101 01101 >> 5`. Logical Right Shift sẽ trả về kết quả `0b00000 101`, nhưng Arithmetic Right Shift sẽ trả về `0b11111 101`.

Chắc chắn khi đọc đến đây, các bạn sẽ tự hỏi về ý nghĩa của phép Arithmetic Right Shift. Trong trường hợp toán hạng `a` là số không âm, hai phép toán hoạt động tương đương. Tuy nhiên, trong trường hợp `a` âm, phép Logical Right Shift không có ý nghĩa về mặt toán học, mà đơn giản chỉ là đẩy các bit sang phải. Trong khi đó, phép Arithmetic Right Shift sẽ vẫn đảm bảo tính chất $a \gg b = \lfloor \frac{a}{2^b} \rfloor$. Chú ý rằng kết quả của phép toán sẽ được làm tròn xuống, chẳng hạn như $-7 \gg 2 = \frac{-7}{2^2} = -1.75$ được làm tròn xuống -2 .

Lý do phép toán trên hoạt động là vì các số nguyên âm được biểu diễn dưới dạng two's complement. Do giới hạn của bài viết, người viết sẽ không đi sâu hơn vào loại biểu diễn này.

Trong C++, phép Logical Right Shift sẽ được sử dụng nếu toán tử đầu tiên là một số thuộc loại `unsigned`, còn nếu không thì phép Arithmetic Right Shift sẽ được sử dụng.

Sử dụng Pragma

Đối với gần như tất cả ($> 99\%$) những máy tính mà bạn sẽ gặp trong đời, các hàm phía trên có thể được thực hiện với chỉ 1 instruction, hay nói cách khác là trong $O(1)$. Tuy nhiên, để hỗ trợ những máy tính rất cũ hoặc rất low-end, compiler GCC mặc định cài đặt các hàm trên bằng toán tử bit, chạy trong $O(\log_2 \log_2 n)$, với n là số lượng bit trong kiểu số của bạn.

Để mở khóa các instruction mới, các bạn cần phải sử dụng định hướng biên dịch `#pragma GCC target`.

Hầu hết mỗi hàm đều có một "flag" riêng biệt mà khi bật lên sẽ mở khóa instruction cho hàm đó. Tuy nhiên, nếu bạn mở khóa thừa thì code bạn vẫn chạy. Vì vậy, để code chạy nhanh hơn, bạn chỉ cần paste dòng sau vào đầu code (trước dòng `#include`):

Điều gì xảy ra nếu phép Left Shift tràn số?

Với C++, nếu phép toán `a << b` của bạn bị tràn số (bit 1 được left shift đến quá giới hạn của kiểu số đang sử dụng), sẽ có 2 trường hợp xảy ra:

1. Nếu kiểu số của kết quả là một số `unsigned`, các bit bị tràn sẽ được coi như là 0, và biến mất. Nói cách khác, gọi số bit của kiểu số kết quả là c (c là 32 với `unsigned int`, và 64 với `unsigned long long`), kết quả trả về sẽ được tính theo modulo 2^c .
2. Nếu kiểu số của kết quả là một số `signed`, chương trình của bạn sẽ bị UB. Tuy nhiên, trong hầu hết trường hợp, code của bạn sẽ không bị lỗi, mà chỉ trả về một kết quả không xác định nào đó. Điều tương tự xảy ra nếu toán hạng `a` của bạn là một số âm.

Thư viện `<bit>` của C++20

Chú ý: Tại thời điểm viết bài, hầu hết các kỳ thi chính thức chưa cho phép sử dụng chuẩn C++20. Nếu không chắc chắn, các bạn vui lòng chỉ sử dụng những biến thể không thuộc C++20 của các hàm này.

Một trong những điểm mới của phiên bản C++20 là thư viện `<bit>`. Thư viện này chứa các hàm tương tự như [Các hàm thao tác Bit](#). Điểm lợi của các hàm trong thư viện `<bit>` là có khả năng chạy trên tất cả các Compiler C++ khác nhau, nhưng điểm trừ là chỉ chạy được với chuẩn C++20 trở lên.