[toc]

位运算就是基于整数的二进制表示进行的运算。由于计算机内部就是以二进制来存储数据，位运算是相当快的。

常用的运算符共 6 种，分别为与（`&`）、或（`|`）、异或（`^`）、取反（`~`）、左移（`<<`）和右移（`>>`）。

与、或、异或

与（`&`）或（`|`）和异或（`^`）这三者都是两数间的运算，因此在这里一起讲解。

它们都是将两个整数作为二进制数，对二进制表示中的每一位逐一运算。

运算符	解释
`&`	只有两个对应位都为 1 时才为 1
`|`	只要两个对应位中有一个 1 时就为 1
`^`	只有两个对应位不同时才为 1

异或运算的逆运算是它本身，也就是说两次异或同一个数最后结果不变，即 a ^ b ^ b = a 。

取反

取反是对一个数 num 进行的计算，即单目运算。

`~` 把 num 的补码中的 0 和 1 全部取反（0 变为 1，1 变为 0）。有符号整数的符号位在 `~` 运算中同样会取反。

补码：在二进制表示下，正数和 0 的补码为其本身，负数的补码是将其对应正数按位取反后加一。

举例（有符号整数）：

$$ \begin{aligned} 5&=(00000101)_2\ \text{~}5&=(11111010)_2=-6\ -5\text{ 的补码}&=(11111011)_2\ \text{~}(-5)&=(00000100)_2=4 \end{aligned} $$

左移和右移

`num << i` 表示将 num 的二进制表示向左移动 i 位所得的值。

`num >> i` 表示将 num 的二进制表示向右移动 i 位所得的值。

举例：

$$ \begin{aligned} 11&=(00001011)_2\ 11<<3&=(01011000)_2=88\ 11>>2&=(00000010)_2=2 \end{aligned} $$

移位运算中如果出现如下情况，则其行为未定义：

1. 右操作数（即移位数）为负值；
2. 右操作数大于等于左操作数的位数；

复合赋值位运算符

和 `+=`，`-=` 等运算符类似，位运算也有复合赋值运算符： `&=`，`|=`，`^=`，`<<=`，`>>=` 。（取反是单目运算，所以没有。）

关于优先级

位运算的优先级低于算术运算符（除了取反），而按位与、按位或及异或低于比较运算符（详见 运算页面 ），所以使用时需多加注意，在必要时添加括号。

位运算的应用

位运算一般有三种作用：

1. 高效地进行某些运算，代替其它低效的方式。
2. 表示集合。（常用于 状压 DP 。）
3. 题目本来就要求进行位运算。

需要注意的是，用位运算代替其它运算方式（即第一种应用）在很多时候并不能带来太大的优化，反而会使代码变得复杂，使用时需要斟酌。（但像“乘 2 的非负整数次幂”和“除以 2 的非负整数次幂”就最好使用位运算，因为此时使用位运算可以优化复杂度。）

乘 2 的非负整数次幂

```
int mulPowerOfTwo(int n, int m) {  // 计算 n*(2^m)
  return n << m;
}
```

除以 2 的非负整数次幂

```
int divPowerOfTwo(int n, int m) {  // 计算 n/(2^m)
  return n >> m;
}
```

!!! warning 我们平常写的除法是向 0 取整，而这里的右移是向下取整（注意这里的区别），即当数大于等于 0 时两种方法等价，当数小于 0 时会有区别，如： `-1 / 2` 的值为 0 ，而 `-1 >> 1` 的值为 -1 。

判断一个数是不是 2 的非负整数次幂

```
bool isPowerOfTwo(int n) { return n > 0 && (n & (n - 1)) == 0; }
```

对 2 的非负整数次幂取模

```
int modPowerOfTwo(int x, int mod) { return x & (mod - 1); }
```

取绝对值

在某些机器上，效率比 `n > 0 ? n : -n` 高。

```
int Abs(int n) {
  return (n ^ (n >> 31)) - (n >> 31);
  /* n>>31 取得 n 的符号，若 n 为正数，n>>31 等于 0，若 n 为负数，n>>31 等于 -1
     若 n 为正数 n^0=n，数不变，若 n 为负数有 n^(-1)
     需要计算 n 和 -1 的补码，然后进行异或运算，
     结果 n 变号并且为 n 的绝对值减 1，再减去 -1 就是绝对值 */
}
```

取两个数的最大/最小值

在某些机器上，效率比 `a > b ? a : b` 高。

```
// 如果 a>=b,(a-b)>>31 为 0，否则为 -1
int max(int a, int b) { return b & ((a - b) >> 31) | a & (~(a - b) >> 31); }
int min(int a, int b) { return a & ((a - b) >> 31) | b & (~(a - b) >> 31); }
```

判断符号是否相同

```
bool isSameSign(int x, int y) {  // 有 0 的情况例外
  return (x ^ y) >= 0;
}
```

交换两个数

note: "该方法具有局限性" 这种方式只能用来交换两个整数，使用范围有限。

对于一般情况下的交换操作，推荐直接调用 `algorithm` 库中的 `std::swap` 函数。

```
void swap(int &a, int &b) { a ^= b ^= a ^= b; }
```

获取一个数二进制的某一位

```
// 获取 a 的第 b 位，最低位编号为 0
int getBit(int a, int b) { return (a >> b) & 1; }
```

将一个数二进制的某一位设置为 0

```
// 将 a 的第 b 位设置为 0 ，最低位编号为 0
int unsetBit(int a, int b) { return a & ~(1 << b); }
```

将一个数二进制的某一位设置为 1

```
// 将 a 的第 b 位设置为 1 ，最低位编号为 0
int setBit(int a, int b) { return a | (1 << b); }
```

将一个数二进制的某一位取反

```
// 将 a 的第 b 位取反 ，最低位编号为 0
int flapBit(int a, int b) { return a ^ (1 << b); }
```

表示集合

一个数的二进制表示可以看作是一个集合（0 表示不在集合中，1 表示在集合中）。比如集合 `{1, 3, 4, 8}` ，可以表示成 $(100011010)_2$ 。而对应的位运算也就可以看作是对集合进行的操作。

操作	集合表示	位运算语句
交集	$a \cap b$	`a & b`
并集	$a \cup b$	`a|b`
补集	$\bar{a}$	`~a` （全集为二进制都是 1）
差集	$a \setminus b$	`a & (~b)`
对称差	$a\triangle b$	`a ^ b`

遍历某个集合的子集

```
// 遍历 u 的非空子集
for (int s = u; s; s = (s - 1) & u) {
  // s 是 u 的一个非空子集
}
```

用这种方法可以在 $O(2^{popcount(u)})$ （ $popcount(u)$ 表示 u 二进制中 1 的个数）的时间复杂度内遍历 u 的子集，进而可以在 $O(3^n)$ 的时间复杂度内遍历大小为 n 的集合的每个子集的子集。（复杂度为 $O(3^n)$ 是因为每个元素都有 不在大子集中/只在大子集中/同时在大小子集中 三种状态。）

bitset

如果需要操作的集合非常大，可以使用 bitset 。

题目推荐

Luogu P1225 黑白棋游戏

[^note1]: 适用于 C++14 以前的标准。在 C++14 和 C++17 标准中，若原值为带符号类型，且移位后的结果能被原类型的无符号版本容纳，则将该结果 转换 为相应的带符号值，否则行为未定义。在 C++20 标准中，规定了无论是带符号数还是无符号数，左移均直接舍弃移出结果类型的位。

[^note2]: 适用于 C++20 以前的标准。

[^note3]: 这种右移方式称为算术右移。在 C++20 以前的标准中，并没有规定带符号数右移运算的实现方式，大多数平台均采用算术右移。在 C++20 标准中，规定了带符号数右移运算是算术右移。