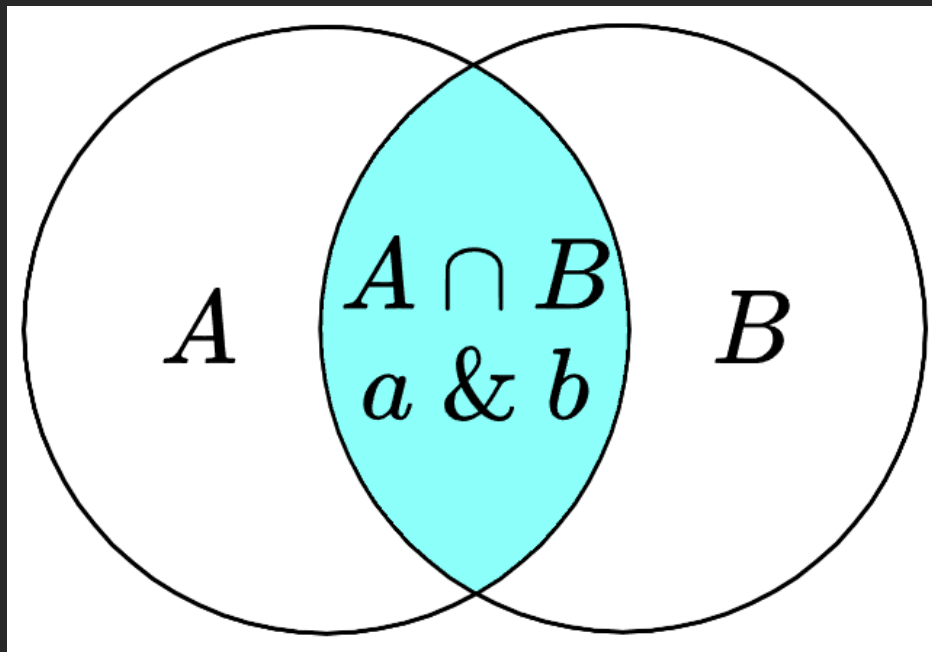


讨论 / 技术交流 / 分享 | 从集合论到位运算，常...

分享 | 从集合论到位运算，常见位运算技巧分类总结！

已关注

灵茶山艾府 发起于 2023-06-16 • 最近编辑于 2024-11-04 • 来自浙江



图：集合交、按位与之间存在某种联系。

前言

本文将扫清位运算的迷雾，在集合论与位运算之间建立一座桥梁。

在高中，我们学了集合论（set theory）的相关知识。例如，包含若干整数的集合 $S = \{0, 2, 3\}$ 。在编程中，通常用哈希表（hash table）表示集合。例如 Java 中的 `HashSet`，C++ 中的 `std::unordered_set`。

在集合论中，有交集 \cap 、并集 \cup 、包含于 \subseteq 等等概念。如果编程实现「求两个哈希表的交集」，需要一个一个地遍历哈希表中的元素。那么，有没有效率更高的做法呢？

该二进制登场了。

集合可以用二进制表示，二进制从低到高第 i 位为 1 表示 i 在集合中，为 0 表示 i 不在集合中。例如集合 $\{0, 2, 3\}$ 可以用二进制数 $1101_{(2)}$ 表示；反过来，二进制数 $1101_{(2)}$ 就对应着集合 $\{0, 2, 3\}$ 。

正式地说，包含非负整数的集合 S 可以用如下方式「压缩」成一个数字：

$$f(S) = \sum_{i \in S} 2^i$$

例如集合 $\{0, 2, 3\}$ 可以压缩成 $2^0 + 2^2 + 2^3 = 13$ ，也就是二进制数 $1101_{(2)}$ 。

利用位运算「并行计算」的特点，我们可以高效地做一些和集合有关的运算。按照常见的应用场景，可以分为以下四类：

1. 集合与集合
2. 集合与元素
3. 遍历集合
4. 枚举集合

一、集合与集合

其中 $\&$ 表示按位与， $|$ 表示按位或， \oplus 表示按位异或， \sim 表示按位取反。

收藏次数

3.7k

参与人数

120

浏览次数

131.6k

相关标签

位运算

数学

动态规划

状态压缩

Go

Java

Python3

周赛

数据结构与算法

双周赛

LeetBook 推荐



LeetCode Cookbook

位运算

动态规划

25,404 人已读



高效算法：竞赛、应...

动态规划

Python3

7,562 人已读



零起步学算法

动态规划

Java

74,573 人已读

🔥 相关讨论

2024 Rewind | 开启你的年...

一鼓作气 | 2024 扣得精彩...

[力扣刷题攻略] Re: 从零...

「力扣编辑器」使用说明

求助 | 本人双非研二 本科...

力扣 (LeetCode) App
随时随地参与讨论

App 扫一扫 即刻参与

两个集合的「对称差」是只属于其中一个集合，而不属于另一个集合的元素组成的集合，也就是不在交集的元素组成的集合。

术语	集合	位运算	集合示例	位运算示例
交集	$A \cap B$	$a \& b$	$\{0, 2, 3\}$ $\cap \{0, 1, 2\}$ $= \{0, 2\}$	1101 & 0111 = 0101
并集	$A \cup B$	$a \mid b$	$\{0, 2, 3\}$ $\cup \{0, 1, 2\}$ $= \{0, 1, 2, 3\}$	1101 0111 = 1111
对称差	$A \Delta B$	$a \oplus b$	$\{0, 2, 3\}$ $\Delta \{0, 1, 2\}$ $= \{1, 3\}$	1101 \oplus 0111 = 1010
差	$A \setminus B$	$a \& \sim b$	$\{0, 2, 3\}$ $\setminus \{1, 2\}$ $= \{0, 3\}$	1101 & 1001 = 1001
差 (子集)	$A \setminus B, B \subseteq A$	$a \oplus b$	$\{0, 2, 3\}$ $\setminus \{0, 2\}$ $= \{3\}$	1101 \oplus 0101 = 1000
包含于	$A \subseteq B$	$a \& b = a$ $a \mid b = b$	$\{0, 2\} \subseteq$ $\{0, 2, 3\}$	$0101 \& 1101 =$ 0101 $0101 \mid 1101 =$ 1101

- 注 1: 按位取反的例子中，仅列出最低 4 个比特位取反后的结果，即 0110 取反后是 1001。
- 注 2: 包含于 (判断子集) 的两种位运算写法是等价的，在编程时只需判断其中任意一种。此外，还可以用 `(a & ~b) == 0` 判断，如果成立，也表示 A 是 B 的子集。
- 注 3: 编程时，请注意运算符的优先级。例如 `==` 在某些语言中优先级比位运算更高。

二、集合与元素

通常会用到移位运算。

其中 `<<` 表示左移，`>>` 表示右移。

注：左移 i 位相当于乘以 2^i ，右移 i 位相当于除以 2^i 。

术语	集合	位运算	集合示例	位运算示例
空集	\emptyset	0		
单元素集合	$\{i\}$	$1 \ll i$	$\{2\}$	$1 \ll 2$
全集	$U = \{0, 1, 2, \dots, n - 1\}$	$(1 \ll n) - 1$	$\{0, 1, 2, 3\}$	$(1 \ll 4) - 1$
补集	$\complement_U S = U \setminus S$	$((1 \ll n) - 1) \oplus s$	$U = \{0, 1, 2, 3\}$ $\complement_U \{1, 2\} = \{0, 3\}$	1111 \oplus 0110 = 1001
属于	$i \in S$	$(s \gg i) \& 1 = 1$	$2 \in \{0, 2, 3\}$	$(1101 \gg 2) \& 1 =$ 1

术语	集合	位运算	集合示例	位运算示例
不属于	$i \notin S$	$(s \gg i) \& 1 = 0$	$1 \notin \{0, 2, 3\}$	$(1101 \gg 1) \& 1 = 0$
添加元素	$S \cup \{i\}$	$s \mid (1 \ll i)$	$\{0, 3\} \cup \{2\}$	$1001 \mid (1 \ll 2)$
删除元素	$S \setminus \{i\}$	$s \& \sim (1 \ll i)$	$\{0, 2, 3\} \setminus \{2\}$	$1101 \& \sim (1 \ll 2)$
删除元素（一定在集合中）	$S \setminus \{i\}, i \in S$	$s \oplus (1 \ll i)$	$\{0, 2, 3\} \setminus \{2\}$	$1101 \oplus (1 \ll 2)$
删除最小元素		$s \& (s - 1)$		见下

```
s = 101100
s-1 = 101011 // 最低位的 1 变成 0，同时 1 右边的 0 都取反，变成 1
s&(s-1) = 101000
```

特别地，如果 s 是 2 的幂，那么 $s \& (s - 1) = 0$ 。

此外，编程语言提供了一些和二进制有关的库函数，例如：

- 计算二进制中的 1 的个数，也就是**集合大小**；
- 计算二进制长度，**减一**后得到**集合最大元素**；
- 计算二进制尾零个数，也就是**集合最小元素**。

调用这些函数的时间复杂度都是 $\mathcal{O}(1)$ 。

术语	Python	Java	C++	Go
集合大小	<code>s.bit_count()</code>	<code>Integer.bitCount(s)</code>	<code>__builtin_popcount(s)</code>	<code>bits.OnesCount(s)</code>
二进制长度	<code>s.bit_length()</code>	<code>32 - Integer.numberOfLeadingZeros(s)</code>	<code>__lg(s)+1</code>	<code>bits.Len(s)</code>
集合最大元素	<code>s.bit_length()-1</code>	<code>31 - Integer.numberOfLeadingZeros(s)</code>	<code>__lg(s)</code>	<code>bits.Len(s)-1</code>
集合最小元素	<code>(s&~s).bit_length()-1</code>	<code>Integer.numberOfTrailingZeros(s)</code>	<code>__builtin_ctz(s)</code>	<code>bits.TrailingZeros(s)</code>

请特别注意 $s = 0$ 的情况。对于 C++ 来说，`__lg(0)` 和 `__builtin_ctz(0)` 是未定义行为。其他语言请查阅 API 文档。

此外，对于 C++ 的 `long long`，需使用相应的 `__builtin_popcountll` 等函数，即函数名后缀添加 `ll`（两个小写字母 L）。`__lg` 支持 `long long`。

特别地，只包含最小元素的子集，即二进制最低 1 及其后面的 0，也叫 lowbit，可以用 `s & -s` 算出。举例说明：

```
s = 101100
~s = 010011
(~s)+1 = 010100 // 根据补码的定义，这就是 -s => s 的最低 1 左侧取反，右侧不变
s & -s = 000100 // lowbit
```

三、遍历集合

设元素范围从 0 到 $n - 1$ ，枚举范围中的元素 i ，判断 i 是否在集合 s 中。

Python3 | Java | C++ | Go

```
for (int i = 0; i < n; i++) {
    if ((s >> i) & 1) { // i 在 s 中
        // 处理 i 的逻辑
    }
}
```

也可以直接遍历集合 s 中的元素：不断地计算集合最小元素、去掉最小元素，直到集合为空。

Python3 | Java | C++ | Go

```
for (int t = s; t; t &= t - 1) {
    int i = __builtin_ctz(t);
    // 处理 i 的逻辑
}
```

四、枚举集合

§4.1 枚举所有集合

设元素范围从 0 到 $n - 1$ ，从空集 \varnothing 枚举到全集 U ：

Python3 | Java | C++ | Go

```
for (int s = 0; s < (1 << n); s++) {
    // 处理 s 的逻辑
}
```

§4.2 枚举非空子集

设集合为 s ，**从大到小枚举 s 的所有非空子集 sub** ：

Python3 | Java | C++ | Go

```
for (int sub = s; sub; sub = (sub - 1) & s) {
    // 处理 sub 的逻辑
}
```

为什么要写成 `sub = (sub - 1) & s` 呢？

暴力做法是从 s 出发，不断减一，直到 0 。但这样做，中途会遇到很多并不是 s 的子集的情况。例如 $s = 10101$ 时，减一得到 10100 ，这是 s 的子集。但再减一就得到 10011 了，这并不是 s 的子集，下一个子集应该是 10001 。

把所有的合法子集按顺序列出来，会发现我们做的相当于「压缩版」的二进制减法，例如

$$10101 \rightarrow 10100 \rightarrow 10001 \rightarrow 10000 \rightarrow 00101 \rightarrow \dots$$

如果忽略掉 10101 中的两个 0 ，数字的变化和二进制减法是一样的，即

$$111 \rightarrow 110 \rightarrow 101 \rightarrow 100 \rightarrow 011 \rightarrow \dots$$

如何快速跳到下一个子集呢？比如，怎么从 10100 跳到 10001 ？

- 普通的二进制减法，是 $10100 - 1 = 10011$ ，也就是把最低位的 1 变成 0 ，同时把最低位的 1 右边的 0 都变成 1 。
- 压缩版的二进制减法也是类似的，对于 $10100 \rightarrow 10001$ ，也会把最低位的 1 变成 0 ，对于最低位的 1 右边的 0 ，并不是都变成 1 ，只有在 $s = 10101$ 中的 1 才会变成 1 。怎么做？减一后 $\& 10101$ 就行，也就是 $(10100 - 1) \& 10101 = 10001$ 。

§4.3 枚举子集（包含空集）

如果要从大到小枚举 s 的所有子集 sub （从 s 枚举到空集 \varnothing ），可以这样写：

Python3 | Java | C++ | Go

```
int sub = s;
do {
    // 处理 sub 的逻辑
    sub = (sub - 1) & s;
} while (sub != s);
```

原理是当 $sub = 0$ 时（空集），再减一就得到 -1 ，对应的二进制为 $111\cdots 1$ ，再 $\&s$ 就得到了 s 。所以当循环到 $sub = s$ 时，说明最后一次循环的 $sub = 0$ （空集）， s 的所有子集都枚举到了，退出循环。

注：还可以枚举全集 U 的所有大小恰好为 k 的子集，这一技巧叫做 **Gosper's Hack**，具体请看 [视频讲解](#)。

§4.4 枚举超集

如果 T 是 S 的子集，那么称 S 是 T 的**超集**（superset）。

枚举超集的原理和上文枚举子集是类似的，这里通过**或运算**保证枚举的集合 S 一定包含集合 T 中的所有元素。

枚举 S ，满足 S 是 T 的超集，也是全集 $U = \{0, 1, 2, \dots, n - 1\}$ 的子集。

Python3 | Java | C++ | Go

```
for (int s = t; s < (1 << n); s = (s + 1) | t) {
    // 处理 s 的逻辑
}
```

关联题单

- [位运算题单](#)
- [数据结构题单](#) 中的「**前缀异或和**」
- [动态规划题单](#) 中的「**状压 DP**」

分类题单

如何科学刷题？

1. 滑动窗口与双指针（定长/不定长/单序列/双序列/三指针）
2. 二分算法（二分答案/最小化最大值/最大化最小值/第K小）
3. 单调栈（基础/矩形面积/贡献法/最小字典序）
4. 网格图（DFS/BFS/综合应用）
5. 位运算（基础/性质/拆位/试填/恒等式/思维）
6. 图论算法（DFS/BFS/拓扑排序/最短路/最小生成树/二分图/基环树/欧拉路径）
7. 动态规划（入门/背包/状态机/划分/区间/状压/数位/数据结构优化/树形/博弈/概率期望）
8. 常用数据结构（前缀和/差分/栈/队列/堆/字典树/并查集/树状数组/线段树）
9. 数学算法（数论/组合/概率期望/博弈/计算几何/随机算法）
10. 贪心与思维（基本贪心策略/反悔/区间/字典序/数学/思维/脑筋急转弯/构造）
11. 链表、二叉树与一般树（前后指针/快慢指针/DFS/BFS/直径/LCA）
12. 字符串（KMP/Z函数/Manacher/字符串哈希/AC自动机/后缀数组/子序列自动机）

[我的题解精选](#)（已分类）

欢迎关注 B站@灵茶山艾府

👍 1760 | 🌟 收藏 | 🔄 分享 | ...

↩ 回复讨论

🔔 接收动态

共 120 个回复

最热 🔥

 天赐细莲 

来自江苏 · 2023-06-16

分享一张图

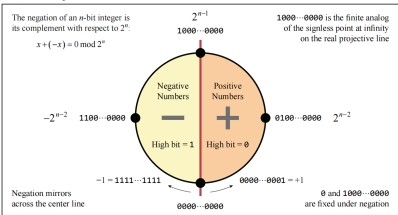
Binary Fundamentals

Powers of Two

Power	Decimal	Hexadecimal
2 ¹	2	0x00000002
2 ²	4	0x00000004
2 ³	8	0x00000008
2 ⁴	16	0x00000010
2 ⁵	32	0x00000020
2 ⁶	64	0x00000040
2 ⁷	128	0x00000080
2 ⁸	256	0x00000100
2 ⁹	512	0x00000200
2 ¹⁰	1024	0x00000400
2 ¹¹	2048	0x00000800
2 ¹²	4096	0x00001000

Power	Decimal	Hexadecimal
2 ¹³	8192	0x00020000
2 ¹⁴	16384	0x00040000
2 ¹⁵	32768	0x00080000
2 ¹⁶	65536	0x00100000
2 ¹⁷	131072	0x00200000
2 ¹⁸	262144	0x00400000
2 ¹⁹	524288	0x00800000
2 ²⁰	1,048,576	0x01000000
2 ²¹	2,097,152	0x02000000
2 ²²	4,194,304	0x04000000
2 ²³	8,388,608	0x08000000
2 ²⁴	16,777,216	0x10000000

Two's Complement



Logical Complement

NOT

Bitwise NOT

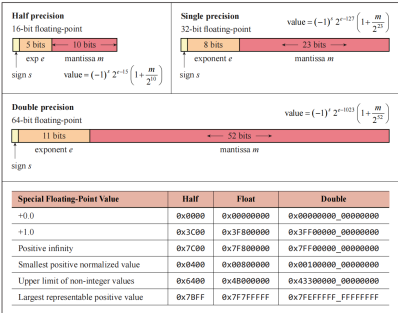
$\sim x$

x	$\sim x$
0	1
1	0

Logical Identities

Unary	Binary
$\sim x = \sim x - 1$	$\sim(x \& y) = \sim x \mid \sim y$
$\sim x = \sim x + 1$	$\sim(x \mid y) = \sim x \& \sim y$
$\sim\sim x = x + 1$	$\sim(x \wedge y) = \sim x \vee \sim y$
$\sim\sim x = x - 1$	$\sim(x \vee y) = \sim x \wedge \sim y$

Floating-Point



Binary Logical Operations

AND Bitwise AND $x \& y$	<table><tr><td>x</td><td>y</td><td>$x \& y$</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	$x \& y$	0	0	0	0	1	0	1	0	0	1	1	1	OR Bitwise OR $x \mid y$	<table><tr><td>x</td><td>y</td><td>$x \mid y$</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	$x \mid y$	0	0	0	0	1	1	1	0	1	1	1	1
x	y	$x \& y$																															
0	0	0																															
0	1	0																															
1	0	0																															
1	1	1																															
x	y	$x \mid y$																															
0	0	0																															
0	1	1																															
1	0	1																															
1	1	1																															
NAND Not AND $\sim(x \& y)$	<table><tr><td>x</td><td>y</td><td>$\sim(x \& y)$</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	$\sim(x \& y)$	0	0	1	0	1	1	1	0	1	1	1	0	NOR Not OR $\sim(x \mid y)$	<table><tr><td>x</td><td>y</td><td>$\sim(x \mid y)$</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	$\sim(x \mid y)$	0	0	1	0	1	0	1	0	0	1	1	0
x	y	$\sim(x \& y)$																															
0	0	1																															
0	1	1																															
1	0	1																															
1	1	0																															
x	y	$\sim(x \mid y)$																															
0	0	1																															
0	1	0																															
1	0	0																															
1	1	0																															
ANDC AND with complement $x \& \sim y$	<table><tr><td>x</td><td>y</td><td>$x \& \sim y$</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	$x \& \sim y$	0	0	0	0	1	0	1	0	1	1	1	0	ORC OR with complement $x \mid \sim y$	<table><tr><td>x</td><td>y</td><td>$x \mid \sim y$</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	$x \mid \sim y$	0	0	1	0	1	0	1	0	1	1	1	1
x	y	$x \& \sim y$																															
0	0	0																															
0	1	0																															
1	0	1																															
1	1	0																															
x	y	$x \mid \sim y$																															
0	0	1																															
0	1	0																															
1	0	1																															
1	1	1																															
XOR Exclusive OR $x \wedge y$	<table><tr><td>x</td><td>y</td><td>$x \wedge y$</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	$x \wedge y$	0	0	0	0	1	1	1	0	1	1	1	0	XNOR Exclusive NOR $\sim(x \wedge y)$	<table><tr><td>x</td><td>y</td><td>$\sim(x \wedge y)$</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	$\sim(x \wedge y)$	0	0	1	0	1	0	1	0	0	1	1	1
x	y	$x \wedge y$																															
0	0	0																															
0	1	1																															
1	0	1																															
1	1	0																															
x	y	$\sim(x \wedge y)$																															
0	0	1																															
0	1	0																															
1	0	0																															
1	1	1																															

Bit Manipulation

Formula	Operation / Effect	Illustration														
$x \& (x - 1)$	Clear lowest 1 bit. If result is zero, then x is zero or 2^k . 000...000 is unchanged.	<table><tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>	1	0	1	1	0	0	0	1	0	1	1	0	0	0
1	0	1	1	0	0	0										
1	0	1	1	0	0	0										
$x \mid (x + 1)$	Set lowest 0 bit. 111...111 is unchanged.	<table><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr></table>	0	1	1	0	0	1	1	0	1	1	0	0	1	1
0	1	1	0	0	1	1										
0	1	1	0	0	1	1										
$x \mid (x - 1)$	Set all bits to right of lowest 1 bit. 000...000 becomes 111...111.	<table><tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	0	1	1	0	0	0	1	0	1	1	1	1	1
1	0	1	1	0	0	0										
1	0	1	1	1	1	1										
$x \& (x + 1)$	Clear all bits to right of lowest 0 bit. If result is zero, then x is zero or $2^k - 1$. 111...111 becomes 000...000.	<table><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	1	1	0	0	1	1	0	1	1	0	0	0	0
0	1	1	0	0	1	1										
0	1	1	0	0	0	0										
$x \& \sim x$	Extract lowest 1 bit. 000...000 is unchanged.	<table><tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	0	1	1	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0										
0	0	0	0	0	0	0										
$\sim x \& (x + 1)$	Extract lowest 0 bit (as a 1 bit). 111...111 becomes 000...000.	<table><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr></table>	0	1	1	0	0	1	1	0	0	0	0	0	1	1
0	1	1	0	0	1	1										
0	0	0	0	0	1	1										

Mask Creation

Formula	Operation / Effect	Illustration														
$\sim x \mid (x - 1)$	Create mask for all bits other than lowest 1 bit. 000...000 becomes 111...111.	<table><tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table>	1	0	1	1	0	0	0	1	1	1	1	0	1	1
1	0	1	1	0	0	0										
1	1	1	1	0	1	1										
$x \mid \sim(x + 1)$	Create mask for all bits other than lowest 0 bit. 111...111 is unchanged.	<table><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table>	0	1	1	0	0	1	1	1	1	1	1	0	1	1
0	1	1	0	0	1	1										
1	1	1	1	0	1	1										
$x \mid \sim x$	Create mask for bits left of lowest 1 bit, inclusive. 000...000 is unchanged.	<table><tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	0	1	1	0	0	0	1	1	1	1	1	1	1
1	0	1	1	0	0	0										
1	1	1	1	1	1	1										
$x \wedge \sim x$	Create mask for bits left of lowest 1 bit, exclusive. 000...000 is unchanged.	<table><tr><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr></table>	1	0	1	1	1	0	0	1	1	1	1	1	0	0
1	0	1	1	1	0	0										
1	1	1	1	1	0	0										
$\sim x \mid (x + 1)$	Create mask for bits left of lowest 0 bit, inclusive. 111...111 becomes 000...000.	<table><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>	0	1	1	0	0	1	1	1	1	1	1	0	0	0
0	1	1	0	0	1	1										
1	1	1	1	0	0	0										
$\sim x \wedge (x + 1)$	Create mask for bits left of lowest 0 bit, exclusive. 111...111 becomes 000...000.	<table><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr></table>	0	1	1	0	0	1	1	1	1	1	1	1	0	0
0	1	1	0	0	1	1										
1	1	1	1	1	0	0										
$x \wedge (x - 1)$	Create mask for bits right of lowest 1 bit, inclusive. 000...000 becomes 111...111.	<table><tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr></table>	1	0	1	1	0	0	0	0	0	0	0	1	1	1
1	0	1	1	0	0	0										
0	0	0	0	1	1	1										
$\sim x \& (x - 1)$	Create mask for bits right of lowest 1 bit, exclusive. 000...000 becomes 111...111.	<table><tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr></table>	1	0	1	1	0	0	0	0	0	0	0	0	1	1
1	0	1	1	0	0	0										
0	0	0	0	0	1	1										
$x \wedge (x + 1)$	Create mask for bits right of lowest 0 bit, inclusive. 111...111 is unchanged.	<table><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr></table>	0	1	1	0	0	1	1	0	0	0	0	1	1	1
0	1	1	0	0	1	1										
0	0	0	0	1	1	1										
$x \& (\sim x - 1)$	Create mask for bits right of lowest 0 bit, exclusive. 111...111 is unchanged.	<table><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr></table>	0	1	1	0	0	1	1	0	0	0	0	0	1	1
0	1	1	0	0	1	1										
0	0	0	0	0	1	1										

By Eric Lengyel
Copyright © 2013

👍 286 | 💬 回复 7 | ⭐ 收藏 | ➦ 分享 ...

➦ 添加回复

👤 灵茶山艾府 🏠 楼主

来自浙江 · 2023-06-16

👤 分享一张图

谢谢分享。位运算的技巧多到可以写一本书，本文列出的是做题时会经常用到的位运算技巧。

👍 134 | 💬 回复 1 | ⭐ 收藏 | ➦ 分享 ...

➦ 添加回复

👤 珂朵莉

来自上海 · 2023-06-16

灵神太强了

👍 32 | 💬 回复 1 | ⭐ 收藏 | ➦ 分享 ...

➦ 添加回复

👤 roylx

未归属地 · 2023-06-16

「压缩版」的二进制减法太帅了！只是 但对于这个 1 右边的 0，只保留在 10101 中的 1 这句话太绕了。

我啰嗦一下，意思是，把10100 的最低位1变0，它的后面有两位 00，都是0。这时候按照普通二进制，会把这两个 00 都变成 11，如果按照压缩版，就只把原来集合里有的 1 变成 1（因为求的是子集），其余的还是 0，原有的集合是 10101，最后两位是 01，所以只保留 01。综合起来就是 10100 先变 10000，然后保留 01，变成 10001。


这样做的效果是从 10100 直接跳到 10001，把中间的 10011 和10010 忽略掉了（普通减法顺序是 10100 - 10011 - 10010 - 10001），因为10011 和 10010不是有效的子集。

👍 20 | 💬 回复 2 | ⭐ 收藏 | ➦ 分享 | ➦ 添加回复

 Automation垃圾专业 来自重庆 · 2024-01-26

应用数学中混进了一个理论数学

👍 8 | 💬 回复 | ⭐ 收藏 | ➦ 分享 | ➦ 添加回复

 灵剑2012 来自北京 · 2023-06-29

枚举固定大小的子集能用bit_count相关指令的话好像还挺容易的，是不是就是每次加lowbit然后在最低位补全1的数量就行

👍 8 | 💬 回复 | ⭐ 收藏 | ➦ 分享 | ➦ 添加回复

 悟 🌸 🌿 来自广东 · 2023-06-16

我的神!!!!


👍 8 | 💬 回复 | ⭐ 收藏 | ➦ 分享 | ➦ 添加回复

 KZHU 🏠 来自上海 · 2024-06-10

 分享一张图

感谢大佬分享 https://terathon.com/binary_fund.pdf 附上链接大家直接下载下来当壁纸吧 😊

👍 6 | 💬 回复 | ⭐ 收藏 | ➦ 分享 | ➦ 添加回复

 灵茶山艾府 🏠 楼主 来自浙江 · 2023-11-04

 请教灵神，怎么理解位运算的「并行计算」这个特点。

二进制每一位是互相独立的。比如 AND 运算，在某个比特位上，两个数必须都是 1，结果才能是 1，把这个运算规则应用到每个比特位就能得到结果。例如 int 有 32 个比特，如果一个个地去算就要算 32 次，但是位运算只需要一次就能全部算完。

👍 5 | 💬 回复 | ⭐ 收藏 | ➦ 分享 | ➦ 添加回复

 yuukilee 来自新加坡 · 2023-06-16

 真正的英雄！

遍历子集泰裤辣

5

|

回复

收藏

分享

...

添加回复

<

1

2

3

4

5

...

12

>

力扣 LeetCode

竞赛

LeetBook

讨论社区

求职

Plus 会员

周边商城

企业服务

在线面试

企业测评

招聘

培训

解决方案

商务

社区合作

活动

赞助竞赛



产品推广

校园合作

关于我们


价值观

工作机会

  知 今

© 2024 领扣网络（上海）有限公司

商务咨询 | 侵权投诉 | 问题反馈 | 加入我们 | 使用条款 | 隐私政策

沪ICP备18019787号-20  沪公网安备31010702007420号 沪ICP证B2-20180578 LeetCode力扣·证照中心
上海市互联网违法和不良信息举报中心 中国互联网违法和不良信息举报中心

https://leetcode.cn/circle/discuss/CaOJ45/

8/8