位运算

计算机中编码表示方式

原码

原码就是一个数字的二进制表示。

反码

对于单个数值（二进制的 0 和 1）而言，对其进行取反操作就是将 0 变为 1，1 变为 0。 对一个数字每一位都进行一次取反，就可以得到它的反码。

补码

英文名 2's complement。是一种用二进制表示有号数的方法，也是一种将数字的正负号变 号的方式，常在计算机科学中使用。补码以有符号比特的二进制数定义。正数和 0 的补码 就是该数字本身。负数的补码则是将其对应正数按位取反（反码）再加 1。

补码系统的最大优点是可以在加法或减法处理中，不需因为数字的正负而使用不同的计算方 式。只要一种加法电路就可以处理各种有号数加法，而且减法可以用一个数加上另一个数的 补码来表示，因此只要有加法电路及补码电路即可完成各种有号数加法及减法，在电路设计 上相当方便。**简单来说，就是可以统一加减法**。

另外，补码系统的 0 就只有一个表示方式，这和反码系统不同（在反码系统中，0 有二种 表示方式），因此在判断数字是否为 0 时，只要比较一次即可。

常见的位运算

符号	描述	运算规则
&	与	两个位都为 1 时，结果才为 1
|	或	两个位都为 0 时，结果才为 0
^	异或	两个位相同为 0，相异为 1
~	取反	0 变 1，1 变 0
<<	左移	各二进位全部左移若干位，高位丢弃，低位补 0
>>	右移	各二进位全部右移若干位，对无符号数，高位补 0，有符号数，各编译器处理方法不一样，有的补符号位（算术右移），有的补 0（逻辑右移）

在算法题中通常考察，并且大家比较容易忽略的的就是异或和位移运算（左移和右移）。

两个数字异或，我们需要对其每一位的数字异或得到结果。如果两位的数字相同则结果为 0，不同则为 1。 因此异或有以下性质：

- 任何数和本身异或则为 `0`
- 任何数和 0 异或是 本身
- 异或运算满足交换律，即: a ^ b ^ c = a ^ c ^ b

一个简单的异或的应用是通过异或运算，在不借助第三个变量的情况下可以实现两数对调：

```
a = a ^ b
b = a ^ b
a = a ^ b
```

常见套路

- 如果你想将某几个二进制位变成 1，其他二进制位不变，可以用或运算。 比如 a | 0xff，就是将 a 的低八位变为 1，其他位不变。
- 如果你想将某几个二进制位不变，其他二进制变成 0，可以用与运算。比如 a| 0xff，就 是将 a 的低八位保持不变，其他位置为 0。
- 当你需要用的异或的自反性的话，就考虑使用异或。自反性指的是异或的“反运算”还是异 或。
- 非运算用的比较少，做题过程我基本没有用过。
- 左移运算等价于乘以 2，但是要比乘法快得多。因此建议大家使用位移，而不是乘以 2 （ 或者 2 的幂）

> 右移也是同理

- 当题目的数据范围有一项是 30 以内，可以考虑是否可以使用状态压缩。这样就可以使用 位运算来优化性能。

位运算常见题型

直接考察位运算性质

- **231. 2 的幂**
- **268. 缺失数字**

求某一个二进制位的值

比如我们要求 a 的第 n 位（从低到高）是多少。那么可以通过 (a >> n) & 1 的方式来获 取。 实际上 (a >> n) & 1 就是一个左边全部是 0，最低位和 a 的第 n 位保持一致的数 。

当然也可以用别的方式。比如 (a & (1 << n)) >> n。

> 当 n == 31 时，程序可能有问题。那么会有什么问题呢?

状态压缩

将状态进行压缩，可使用位来模拟。实际上使用状态压缩和不使用压缩的「思路一模一样， 只是 API 不一样」罢了。这部分主要考察大家灵活使用位运算来解决或者降低时空复杂度 。

举个例子来进行说明。题目描述：

```
给定一组不含重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。

说明：解集不能包含重复的子集。

   示例:

   输入: nums = [1,2,3]
   输出:
   [
   [3],
   [1],
   [2],
   [1,2,3],
   [1,3],
   [2,3],
   [1,2],
   []
   ]
```

这道题我们可以直接使用暴力回溯来解决。

JS Code：

```
function backtrack(list, tempList, nums, start) {
list.push([...tempList]);
for (let i = start; i < nums.length; i++) {
tempList.push(nums[i]);
backtrack(list, tempList, nums, i + 1);
tempList.pop();
}
}
/**

- @param {number[]} nums
```

```
- @return {number[][]}
  \*/
  var subsets = function (nums) {
  const list = [];
  backtrack(list, [], nums, 0);
  return list;
  };
```

复杂度分析

- 时间复杂度：由排列组合原理可知，一共有 2^N 种组合，因此时间复杂度为 $O(2^N)$，其中 N 为数字的个数。
- 空间复杂度：由于调用栈深度最多为 N，且临时数组长度不会超过 N，因此空间复杂 度为 $O(N)$，其中 N 为数字的个数。

实际上，我们还可以对其进行优化， 活地使用位运算的性质，来解决这道题。

例如我们现在有 3 个元素，那我们分别给这 3 个元素编号为 A B C

实际上这三个元素能取出的所有子集就是这 3 个元素的使用与不使用这两种状态的笛卡尔 积。我们使用 0 与 1 分别表示这 3 个元素的使用与不使用的状态。那么这 3 个元素能构 成的的所有情况其实就是：

```
000, 001, 010 ... 111
```

那么我们就依次遍历这些数，将为 1 的元素取出，即为子集:

代码(JS):

```
var subsets = function (nums) {
  let res = [],
    sum = 1 << nums.length,
    temp;
  for (let now = 0; now < sum; now++) {
    temp = [];
    for (let i = 0; now >> i > 0; i++) {
      if (((now >> i) & 1) == 1) {
        temp.push(nums[i]);
      }
    }
    res.push(temp);
  }
  return res;
};
```

时间和空间复杂度均为 $O(n)$，其中 n 为 nums 中数字总和。

更多参考：

- 状压 DP 是什么？这篇题解带你入门

二进制的思维角度

有时间从二进制角度思考问题或许可以实现降维打击的效果。不过这种思维方式不是很容易 掌握，需要大家不断练习来掌握。

题目推荐：

- 从老鼠试毒问题来看二分法

更多

- 力扣专题 - 位运算
- 最高（最低）有效位最高（最 低）有效位这里不讲了，给大家个资料学习一下。强烈推荐看下官方题解 👍

总结

位运算的题目，首先要知道的就是各种位运算有哪些，对应的功能以及性质。很多题目的考 点基本都是围绕性质展开。另外一种题目的考点是状态压缩，大大减少时间和空间复杂度。 使用位运算的状态压缩一点都不神秘，只是 api 不一样罢了。如果你不会，只能说明你堆 位运算 api 不熟悉，多用几次其实就好了。

上一页 下一页