78. Subsets子集

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）。

【分析】把数组中所有的数分配一个状态，true表示这个数在子集中出现，false表示在子集中不出现，那么对于一个长度为n的数组，每个数字都有出现与不出现两种情况，所以共有2n中情况，那么我们把每种情况都转换出来就是子集了，用题目中[1 2 3]数组为例，数组共有8个子集，每个子集的序号的二进制表示，把是1的位对应原数组中的数字取出来就是一个子集，八种情况都取出来就是所有的子集了。

【CODE】class Solution {

public:

vector<vector<int> > subsets(vector<int> &S) {

vector<vector<int> > res;

sort(S.begin(), S.end());

int max = 1 << S.size();

for (int k = 0; k < max; ++k) {

vector<int> out = convertIntToSet(S, k);

res.push\_back(out);

}

return res;

}

vector<int> convertIntToSet(vector<int> &S, int k) {

vector<int> sub;

int idx = 0;

for (int i = k; i > 0; i >>= 1) {

if ((i & 1) == 1) {

sub.push\_back(S[idx]);

}

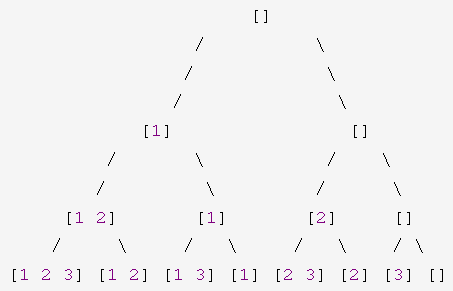
++idx;

}

return sub;

}

};

【分析】递归的解法，相当于一种深度优先搜索，由于原集合每一个数字只有两种状态，要么存在，要么不存在，那么在构造子集时就有选择和不选择两种情况，所以可以构造一棵二叉树，左子树表示选择该层处理的节点，右子树表示不选择，最终的叶节点就是所有子集合，树的结构如图：

【CODE】class Solution {

public:

vector<vector<int> > subsets(vector<int> &S) {

vector<vector<int> > res;

vector<int> out;

sort(S.begin(), S.end());

getSubsets(S, 0, out, res);

return res;

}

void getSubsets(vector<int> &S, int pos, vector<int> &out, vector<vector<int> > &res) {

res.push\_back(out);

for (int i = pos; i < S.size(); ++i) {

out.push\_back(S[i]);

getSubsets(S, i + 1, out, res);

out.pop\_back();

}

}

};

137. Single Number II只出现一次的数字 II

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现了三次。找出那个只出现了一次的元素。

说明：你的算法应该具有线性时间复杂度。 你可以不使用额外空间来实现吗？

【分析】利用计算机按位储存数字的特性来做，这道题就是除了一个单独的数字之外，数组中其他的数字都出现了三次，那么还是要利用位操作 Bit Operation 来解此题。我们可以建立一个**32位的数字**，来统计每一位上1出现的个数，我们知道如果某一位上为1的话，那么如果该整数出现了三次，对3去余为0，我们把每个数的对应位都加起来对3取余，最终剩下来的那个数就是单独的数字。

*但是一定要这么来记录数字出现的次数吗？*我们把数组中数字的每一位累加起来对3取余，剩下的结果就是那个单独数组该位上的数字，由于我们累加的过程都要对3取余，那么每一位上累加的过程就是0->1->2->0，换成二进制的表示为00->01->10->00，那么我们可以写出对应关系：

00 (+) 1 = 01

01 (+) 1 = 10

10 (+) 1 = 00 ( mod 3)

那么我们用ab来表示开始的状态，对于加1操作后，得到的新状态的ab的算法如下：

b = b xor r & ~a;

a = a xor r & ~b;

我们这里的ab就是上面的三种状态00，01，10的十位和各位，刚开始的时候，a和b都是0，当此时遇到数字1的时候，b更新为1，a更新为0，就是01的状态；再次遇到1的时候，b更新为0，a更新为1，就是10的状态；再次遇到1的时候，b更新为0，a更新为0，就是00的状态，相当于重置了；最后的结果保存在b中。

【CODE】class Solution {

public:

int singleNumber(vector<int>& nums) {

int a = 0, b = 0;

for (int i = 0; i < nums.size(); ++i) {

b = (b ^ nums[i]) & ~a;

a = (a ^ nums[i]) & ~b;

}

return b;

}

};

260. Single Number III只出现一次的数字 III

给定一个整数数组 nums，其中恰好有两个元素只出现一次，其余所有元素均出现两次。 找出只出现一次的那两个元素。

注意： 你的算法应该具有线性时间复杂度。你能否仅使用常数空间复杂度来实现？

【分析】这道题其实是很巧妙的利用了Single Number 单独的数字的解法，因为那道解法是可以准确的找出只出现了一次的数字，但前提是其他数字必须出现两次才行。而这题有两个数字都只出现了一次，那么我们如果能想办法把原数组分为两个小数组，不相同的两个数字分别在两个小数组中，这样分别调用Single Number 单独的数字的解法就可以得到答案。那么如何实现呢，首先我们先把原数组全部异或起来，那么我们会得到一个数字，这个数字是两个不相同的数字异或的结果，我们取出其中任意一位为‘1’的位，为了方便起见，我们用 a &= -a 来取出最右端为‘1’的位，然后和原数组中的数字挨个相与，那么我们要求的两个不同的数字就被分到了两个小组中，分别将两个小组中的数字都异或起来，就可以得到最终结果。

【CODE】class Solution {

public:

vector<int> singleNumber(vector<int>& nums) {

int diff = accumulate(nums.begin(), nums.end(), 0, bit\_xor<int>());

diff &= -diff;

vector<int> res(2, 0);

for (auto &a : nums) {

if (a & diff) res[0] ^= a;

else res[1] ^= a;

}

return res;

}

};

187. Repeated DNA Sequences重复的DNA序列

所有 DNA 由一系列缩写为 A，C，G 和 T 的核苷酸组成，例如：“ACGAATTCCG”。在研究 DNA 时，识别 DNA 中的重复序列有时会对研究非常有帮助。

编写一个函数来查找 DNA 分子中所有出现超多一次的10个字母长的序列（子串）。

【分析】用两位来表示一个字符，00表示A，01表示C，10表示G，11表示T，那么总共需要20位就可以表示10个字符流。为了提取出后20位，我们还需要用个mask，取值为0x3ffff，用此mask可取出后18位，再向左平移2位即可。算法的思想是，当取出第十个字符时，将其存在哈希表里，和该字符串出现频率映射，之后每向左移2位替换一个字符，查找新字符串在哈希表里出现次数，如果之前刚好出现过一次，则将当前字符串存入返回值的数组并将其出现次数加一，如果从未出现过，则将其映射到1。为了能更清楚的阐述整个过程，我们用题目中给的例子来分析整个过程：

首先我们取出前九个字符AAAAACCCC，根据上面的分析，我们用2位来表示一个字符，所以这九个字符可以用二进制表示为000000000001010101，然后我们继续遍历字符串，下一个进来的是C，则当前字符为AAAAACCCCC，二进制表示为00000000000101010101，然后我们将其存入哈希表中，用二进制的好处是可以用一个int变量来表示任意十个字符序列，比起直接存入字符串大大的节省了内存空间，然后再读入下一个字符C，则此时字符串为AAAACCCCCA，我们还是存入其二进制的表示形式，以此类推，当某个序列之前已经出现过了，我们将其存入结果res中即可.

【CODE】class Solution {

public:

vector<string> findRepeatedDnaSequences(string s) {

unordered\_set<string> res;

unordered\_set<int> st;

unordered\_map<int, int> m{{'A', 0}, {'C', 1}, {'G', 2}, {'T', 3}};

int cur = 0, i = 0;

while (i < 9) cur = cur << 2 | m[s[i++]];

while (i < s.size()) {

cur = ((cur & 0x3ffff) << 2) | (m[s[i++]]);

if (st.count(cur)) res.insert(s.substr(i - 10, 10));

else st.insert(cur);

}

return vector<string>(res.begin(), res.end());

}

};

【分析】如果我们不需要考虑节省内存空间，那我们可以直接将10个字符组成字符串存入set中，那么也就不需要mask了。

【CODE】class Solution {

public:

vector<string> findRepeatedDnaSequences(string s) {

set<string> res, st;

for (int i = 0; i + 9 < s.size(); ++i) {

string t = s.substr(i, 10);

if (st.count(t)) res.insert(t);

else st.insert(t);

}

return vector<string>{res.begin(), res.end()};

}

};

338. Counting BitsBit位计数

给定一个非负整数 num。 对于范围 0 ≤ i ≤ num 中的每个数字 i ，计算其二进制数中的1的数目并将它们作为数组返回。

示例：比如给定 num = 5 ，应该返回 [0,1,1,2,1,2].

进阶：给出时间复杂度为O(n \* sizeof(integer)) 的解答非常容易。 但是你可以在线性时间O(n)内用一次遍历做到吗？

要求算法的空间复杂度为O(n)。

你能进一步完善解法吗？ 在c ++或任何其他语言中不使用任何内置函数（如c++里的 \_\_builtin\_popcount）来执行此操作。

【分析】列写二进制数，找出规律。可以找到一个规律是，从1开始，遇到偶数时，其1的个数和该偶数除以2得到的数字的1的个数相同，遇到奇数时，其1的个数等于该奇数除以2得到的数字的1的个数再加1。

【CODE】

class Solution {

public:

vector<int> countBits(int num) {

vector<int> res{0};

for (int i = 1; i <= num; ++i) {

if (i % 2 == 0) res.push\_back(res[i / 2]);

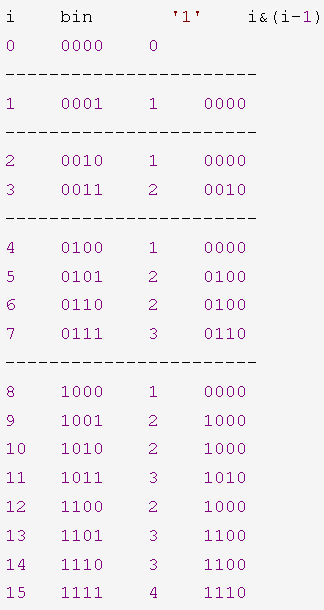
else res.push\_back(res[i / 2] + 1);

}

return res;

}

};

【分析】一种巧妙的规律，利用了i&(i - 1)， 这个本来是用来判断一个数是否是2的指数的快捷方法，比如8，二进制位1000, 那么8&(8-1)为0，只要为0就是2的指数, 那么我们观察一下0到15的数字和其对应的i&(i - 1)值，可以发现每个i值都是i&(i-1)对应的值加1。

【CODE】

class Solution {

public:

vector<int> countBits(int num) {

vector<int> res(num + 1, 0);

for (int i = 1; i <= num; ++i) {

res[i] = res[i & (i - 1)] + 1;

}

return res;

}

};

342. Power of Four 4的幂

给定一个整数 (32位有符整数型)，请写出一个函数来检验它是否是4的幂。

问题进阶：你能***不使用循环/递归***来解决这个问题吗？

【分析】判断一个数是否为4的次方数，那么最直接的方法就是不停的除以4，看最终结果是否为1。这就会利用到循环。

使用换底公式来做：

return num > 0 && int(log10(num) / log10(4)) - log10(num) / log10(4) == 0;

知道num & (num - 1)可以用来判断一个数是否为2的次方数，更进一步说，就是二进制表示下，只有最高位是1，那么由于是2的次方数，不一定是4的次方数，比如8，所以我们还要其他的限定条件，我们仔细观察可以发现，4的次方数的最高位的1都是计数位，那么我们只需与上一个数(0x55555555) <==> 1010101010101010101010101010101，如果得到的数还是其本身，则可以肯定其为4的次方数。

【CODE】class Solution {

public:

bool isPowerOfFour(int num) {

return num > 0 && !(num & (num - 1)) && (num & 0x55555555) == num;

}

};

【分析】或者我们在确定其是2的次方数了之后，发现只要是4的次方数，减1之后可以被3整除，所以可以写出代码。

【CODE】class Solution {

public:

bool isPowerOfFour(int num) {

return num > 0 && !(num & (num - 1)) && (num - 1) % 3 == 0;

}

};

371. Sum of Two Integers两整数之和

不使用运算符 + 和-，计算两整数a 、b之和。

【分析】用异或算不带进位的和，用与并左移1位来算进位，再把两者加起来即可。

【CODE】return b == 0 ? a : getSum(a ^ b, (a & b) << 1);

389. Find the Difference找不同

给定两个字符串 s 和 t，它们只包含小写字母。

字符串 t 由字符串 s 随机重排，然后在随机位置添加一个字母。

请找出在 t 中被添加的字母。

【分析】给我们两个字符串s和t，t是在s的任意一个地方加上了一个字符，让我们找出新加上的那个字符。首先第一反应的方法就是用哈希表来建立字符和个数之间的映射，如果在遍历t的时候某个映射值小于0了，那么返回该字符即可。

也可以使用位操作Bit Manipulation来做，利用异或的性质，相同位返回0，这样相同的字符都抵消了，剩下的就是后加的那个字符。

也可以直接用加和减，相同的字符一减一加也抵消了，剩下的就是后加的那个字符。

【CODE】

class Solution {

public:

char findTheDifference(string s, string t) {

char res = 0;

for (char c : s) res ^= c; // for (char c : s) res -= c;

for (char c : t) res ^= c; // for (char c : t) res += c;

return res;

}

};

【分析】利用了STL的accumulate函数，实际上是上面解法二的改写，一行就写完。

【知识点】STL的accumulate函数（#include <numeric>）

用来计算特定范围内（包括连续的部分和初始值）所有元素的和，除此之外，还可以用指定的二进制操作来计算特定范围内的元素结果。其头文件在numeric中。

accumulate原函数声明定义如下：

template<class InputIterator, class Type>

Type accumulate(

InputIterator \_First,

InputIterator \_Last,

Type \_Val

);

template<class InputIterator, class Type, class Fn2>

Type **accumulate**(

**InputIterator \_First**,

**InputIterator \_Last**,

**Type \_Val**,

**BinaryOperation \_Binary\_op**

);

参数说明：

|  |  |
| --- | --- |
| **\_First** | 指定范围内第一个迭代的值或者结合操作选项使用。 |
| **InputIterator \_Last** | 指定范围内最后一个迭代值或者结合操作项使用。 |
| **\_Val** | 要计算的初始值。 |
| **\_Binary\_op** | 运用于指定范围内所有元素和前面计算得到结果的参数。 |

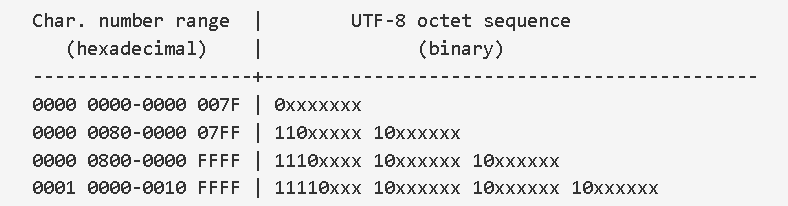
【CODE】 return accumulate(**begin(s)**, **end(s += t)**, **0**, **bit\_xor<int>()**);

393. UTF-8 Validation UTF-8 编码验证

UTF-8 中的一个字符可能的长度为 1 到 4 字节，遵循以下的规则：

对于 1 字节的字符，字节的第一位设为0，后面7位为这个符号的unicode码。

对于 n 字节的字符 (n > 1)，第一个字节的前 n 位都设为1，第 n+1 位设为0，后面字节的前两位一律设为10。剩下的没有提及的二进制位，全部为这个符号的unicode码。

这是 UTF-8 编码的工作方式：

给定一个表示数据的整数数组，返回它是否为有效的 utf-8 编码。

注意:输入是整数数组。只有每个整数的最低 8 个有效位用来存储数据。这意味着每个整数只表示 1 字节的数据。

【分析】UTF-8编码，这种互联网所采用的通用的编码格式的产生是为了解决ASCII只能表示英文字符的局限性，和统一Unicode的实现方式。摘自维基UTF-8编码：

对于UTF-8编码中的任意字节B，如果B的第一位为0，则B独立的表示一个字符(ASCII码)； (非ASCII字符)

如果B的第一位为1，第二位为0，则B为一个多字节字符中的一个字节；

如果B的前两位为1，第三位为0，则B为两个字节表示的字符中的第一个字节；

如果B的前三位为1，第四位为0，则B为三个字节表示的字符中的第一个字节；

如果B的前四位为1，第五位为0，则B为四个字节表示的字符中的第一个字节；

因此，对UTF-8编码中的任意字节，根据第一位，可判断是否为ASCII字符；根据前二位，可判断该字节是否为一个字符编码的第一个字节；根据前四位（如果前两位均为1），可确定该字节为字符编码的第一个字节，并且可判断对应的字符由几个字节表示；根据前五位（如果前四位为1），可判断编码是否有错误或数据传输过程中是否有错误。

那么根据上面的描述，我们可以先来判断第一位，如果是0的话，则说明是ASCII码，我们直接跳过，判断方法是只要比二进制数10000000小的数第一位肯定是0，然后我们来处理第一位是1的情况，由于第一位的1只是个标识符，后面连续跟的1的个数才是表示后面的字节的个数，我们可以统一从第一位开始连续1的个数，然后减去1就是后面的字节的个数，我想的办法是如果该数字大于等于128，则表示第一位是1，然后减去128，如果得到的数大于等于64，则表示第二位是1，依次类推就可以得到连续的个数，我们要注意10000000这个数是不合法的，遇到了直接返回false。我们得到了cnt的个数，只要验证后面的字节是否是以10开头的数即可，验证方法也很简单，只要这个数在10000000 ~ 10111111范围之间，则一定是10开头的。

【CODE】

class Solution {

public:

bool validUtf8(vector<int>& data) {

for (int i = 0; i < data.size(); ++i) {

if (data[i] < 0b10000000) {

continue;

} else {

int cnt = 0, val = data[i];

for (int j = 7; j >= 1; --j) {

if (val >= pow(2, j)) ++cnt;

else break;

val -= pow(2, j);

}

if (cnt == 1) return false;

for (int j = i + 1; j < i + cnt; ++j) {

if (data[j] > 0b10111111 || data[j] < 0b10000000) return false;

}

i += cnt - 1;

}

}

return true;

}

};

【分析】一种非常简洁的方法，也是要记连续1的个数，如果是标识字节，先将其向右平移五位，如果得到110，则说明后面跟了一个字节，否则向右平移四位，如果得到1110，则说明后面跟了两个字节，否则向右平移三位，如果得到11110，则说明后面跟了三个字节，否则向右平移七位，如果为1的话，说明是10000000这种情况，不能当标识字节，直接返回false。在非标识字节中，向右平移六位，如果得到的不是10，则说明不是以10开头的，直接返回false，否则cnt自减1，成功完成遍历返回true。

【CODE】

class Solution {

public:

bool validUtf8(vector<int>& data) {

int cnt = 0;

for (int d : data) {

if (cnt == 0) {

if ((d >> 5) == 0b110) cnt = 1;

else if ((d >> 4) == 0b1110) cnt = 2;

else if ((d >> 3) == 0b11110) cnt = 3;

else if (d >> 7) return false;

} else {

if ((d >> 6) != 0b10) return false;

--cnt;

}

}

return cnt == 0;

}

};

201. Bitwise AND of Numbers Range数字范围按位与

给定范围 [m, n]，其中 0 <= m <= n <= 2147483647，返回此范围内所有数字的按位与（包含 m, n 两端点）。

【分析】先从题目中给的例子来分析，[5, 7]里共有三个数字，分别写出它们的二进制为：101　　110　　111

相与后的结果为100，仔细观察我们可以得出，最后的数是该数字范围内所有的数的左边共同的部分，如果上面那个例子不太明显，我们再来看一个范围[26, 30]，它们的二进制如下：

11010　　11011　　11100　　11101　　11110

发现了规律后，我们只要写代码找到左边公共的部分即可，我们可以从建立一个32位都是1的mask，然后每次向左移一位，比较m和n是否相同，不同再继续左移一位，直至相同，然后把m和mask相与就是最终结果。

【CODE】class Solution {

public:

int rangeBitwiseAnd(int m, int n) {

int d = INT\_MAX;

while ((m & d) != (n & d)) {

d <<= 1;

}

return m & d;

}

};

【分析】另一种解法，不需要用mask，直接平移m和n，每次向右移一位，直到m和n相等，记录下所有平移的次数i，然后再把m左移i位即为最终结果。

【CODE】class Solution {

public:

int rangeBitwiseAnd(int m, int n) {

int i = 0;

while (m != n) {

m >>= 1;

n >>= 1;

++i;

}

return (m << i);

}

};

【分析】递归来做的，如果n大于m，那么就对m和n分别除以2，并且调用递归函数，对结果再乘以2，一定要乘回来，不然就不对了，就举一个最简单的例子，m = 10, n = 11，注意这里是二进制表示的，然后各自除以2，都变成了1，调用递归返回1，这时候要乘以2，才能变回10。

【CODE】class Solution {

public:

int rangeBitwiseAnd(int m, int n) {

return (n > m) ? (rangeBitwiseAnd(m / 2, n / 2) << 1) : m;

}

};

【分析】如果m小于n就进行循环，n与上n-1，举个简单的例子，110与上(110-1)，得到100，这不就相当于去掉最低位的1么，n就这样每次去掉最低位的1，如果小于等于m了，返回此时的n即可。

【CODE】

class Solution {

public:

int rangeBitwiseAnd(int m, int n) {

while (m < n) n &= (n - 1);

return n;

}

};

268. Missing Number缺失数字

给定一个包含 0, 1, 2, ..., n 中 n 个数的序列，找出 0 .. n 中没有出现在序列中的那个数。说明:你的算法应具有线性时间复杂度。你能否仅使用额外常数空间来实现?

【分析】使用位操作Bit Manipulation来解，用到了异或操作的特性。思路是既然0到n之间少了一个数，我们将这个少了一个数的数组合0到n之间完整的数组异或一下，那么相同的数字都变为0了，剩下的就是少了的那个数字了。

【CODE】class Solution {

public:

int missingNumber(vector<int>& nums) {

int res = 0;

for (int i = 0; i < nums.size(); ++i) {

res ^= (i + 1) ^ nums[i];

}

return res;

}

};

318. Maximum Product of Word Lengths最大单词长度乘积

给定一个字符串数组 words，找到 length(word[i]) \* length(word[j]) 的最大值，并且这两个单词不含有公共字母。你可以认为每个单词只包含小写字母。如果不存在这样的两个单词，返回 0。

|  |
| --- |
| 输入: ["abcw","baz","foo","bar","xtfn","abcdef"]  输出: 16  解释: 这两个单词为 "abcw", "xtfn"。 |
| 输入: ["a","ab","abc","d","cd","bcd","abcd"]  输出: 4  解释: 这两个单词为 "ab", "cd"。 |
| 输入: ["a","aa","aaa","aaaa"]  输出: 0  解释: 不存在这样的两个单词。 |

【分析】给我们了一个单词数组，让我们求两个没有相同字母的单词的长度之积的最大值。我开始想的方法是每两个单词先比较，如果没有相同字母，则计算其长度之积，然后每次更新结果就能找到最大值。但是开始想的两个单词比较的方法是利用哈希表先将一个单词的所有出现的字母存入哈希表，然后检查另一个单词的各个字母是否在哈希表出现过，若都没出现过，则说明两个单词没有相同字母，则计算两个单词长度之积并更新结果。但是这种判断方法无法通过OJ的***大数据集***，大神们的解法，都是用了mask，因为题目中说都是**小写字母，那么只有26位，一个整型数int有32位，我们可以用后26位来对应26个字母，若为1，说明该对应位置的字母出现过，那么每个单词的都可由一个int数字表示**，**两个单词没有共同字母的条件是这两个int数想与为0**，用这个判断方法可以通过OJ。

【CODE】class Solution {

public:

int maxProduct(vector<string>& words) {

int res = 0;

vector<int> mask(words.size(), 0);

for (int i = 0; i < words.size(); ++i) {

for (char c : words[i]) {

mask[i] |= 1 << (c - 'a');

}

for (int j = 0; j < i; ++j) {

if (!(mask[i] & mask[j])) {

res = max(res, int(words[i].size() \* words[j].size()));

}

}

}

return res;

}

};

【分析】还有一种写法，借助哈希表，映射每个mask的值和其单词的长度，每算出一个单词的mask，遍历哈希表里的值，如果和其中的mask值相与为0，则将当前单词的长度和哈希表中存的单词长度相乘并更新结果。

【CODE】

class Solution {

public:

int maxProduct(vector<string>& words) {

int res = 0;

unordered\_map<int, int> m;

for (string word : words) {

int mask = 0;

for (char c : word) {

mask |= 1 << (c - 'a');

}

m[mask] = max(m[mask], int(word.size()));

for (auto a : m) {

if (!(mask & a.first)) {

res = max(res, (int)word.size() \* a.second);

}

}

}

return res;

}

};

397. Integer Replacement整数替换

给定一个正整数 n，你可以做如下操作：

1. 如果 n 是偶数，则用 n / 2替换 n。

2. 如果 n 是奇数，则可以用 n + 1或n - 1替换 n。

n 变为 1 所需的最小替换次数是多少？

【分析】看道题的要求，就会感觉用递归很合适，我们直接按照规则写出递归即可，注意由于有n+1的操作，所以当n为INT\_MAX的时候，就有可能**溢出**，所以我们可以先将n转为长整型，然后再进行运算。

【CODE】

class Solution {

public:

int integerReplacement(int n) {

if (n == 1) return 0;

if (n % 2 == 0) return 1 + integerReplacement(n / 2);

else {

long long t = n;

return 2 + min(integerReplacement((t + 1) / 2), integerReplacement((t - 1) / 2));

}

}

};

【分析】也可以使用迭代的解法，那么这里就有小技巧了，当n为奇数的时候，我们什么时候应该加1，什么时候应该减1呢，通过观察来说，除了3和7以外，所有加1就变成4的倍数的奇数，适合加1运算，比如15:

15 -> 16 -> 8 -> 4 -> 2 -> 1

15 -> 14 -> 7 -> 6 -> 3 -> 2 -> 1

对于7来说，加1和减1的结果相同，我们可以不用管，对于3来说，减1的步骤小，所以我们需要去掉这种情况。那么我们如何知道某个数字加1后是否是4的倍数呢，我们可以用个小技巧，由于我们之前判定其是奇数了，那么最右边一位肯定是1，如果其右边第二位也是1的话，那么进行加1运算，进位后右边肯定会出现两个0，则一定是4的倍数，搞定。如果之前判定是偶数，那么除以2即可。

【CODE】

class Solution {

public:

int integerReplacement(int n) {

long long t = n;

int cnt = 0;

while (t > 1) {

++cnt;

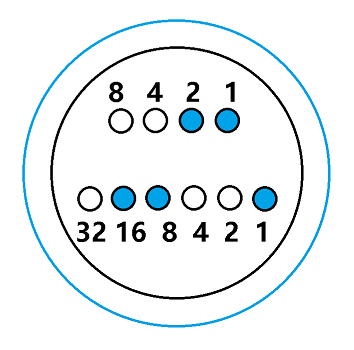
if (t & 1) {

if ((t & 2) && (t != 3)) ++t;

else --t;

} else {

t >>= 1;

 }

}

return cnt;

}

};

401. Binary Watch二进制手表

二进制手表顶部有 4 个 LED 代表小时（0-11），底部的 6 个 LED 代表分钟（0-59）。

每个 LED 代表一个 0 或 1，最低位在右侧。（上图二进制手表读取 “3:25”）

给定一个非负整数 n 代表当前 LED 亮着的数量，返回所有可能的时间。

输入: n = 1

返回: ["1:00", "2:00", "4:00", "8:00", "0:01", "0:02", "0:04", "0:08", "0:16", "0:32"]

注意事项：输出的顺序没有要求。

小时不会以零开头，比如 “01:00” 是不允许的，应为 “1:00”。

分钟必须由两位数组成，可能以零开头，比如 “10:2” 是无效的，应为 “10:02”。

【分析】第一种是遍历，用到的是bitset这个C++ STL模板。

按照数值上分析，最多只有12×60=720种组合，所以只要遍历这720种组合，通过bitset的count函数判断各个组合的1的个数是否等于num，是则添加对应的时间字符串到结果数组中，最后返回结果数组即可。

【CODE】

class Solution {

public:

vector<string> readBinaryWatch(int num) {

vector<string> times;

for (int i = 0; i < 12; i++) {

bitset<4> h((size\_t) i);

for (int j = 0; j < 60; j++) {

bitset<6> m((size\_t) j);

if (h.count() + m.count() == num)

times.push\_back(to\_string(i) + (j < 10? ":0": ":") + to\_string(j));

}

}

return times;

}

};

【分析】除了按数值，也可以从二进制表示上遍历，由于小时有4位，分钟有6位，所以总共10位就可以表示所有的情况，最大的数值是0x2FF (1011 | 111111)，所以遍历从0到0x2FF之间的数值，判断1的个数是否等于num，是则截取出小时和分钟，并拼接成字符串添加到结果数组。

【CODE】class Solution {

public:

vector<string> readBinaryWatch(int num) {

vector<string> times;

if (num < 0 || num > 8)

return times;

for (int i = 0; i < 0x2FF; i++) {

if (bitset<10>(i).count() == num)

add(i, times);

}

return times;

}

void add(int i, vector<string>& t) {

string s;

int m = i & 0x3F;

if (m > 59)

return;

int h = i >> 6 & 0xF;

if (h > 11)

return;

t.push\_back(to\_string(h) + (m < 10? ":0" : ":") + to\_string(m));

}

};

405. Convert a Number to Hexadecimal数字转换为十六进制数

给定一个整数，编写一个算法将这个数转换为十六进制数。对于负整数，我们通常使用 ***补码运算*** 方法。

注意:十六进制中所有字母(a-f)都必须是小写。

十六进制字符串中不能包含多余的前导零。如果要转化的数为0，那么以单个字符'0'来表示；对于其他情况，十六进制字符串中的第一个字符将不会是0字符。

给定的数确保在32位有符号整数范围内。

**不能使用任何由库提供的将数字直接转换或格式化为十六进制的方法。**

【分析】采取位操作的思路，每次取出最右边四位，如果其大于等于10，找到对应的字母加入结果，反之则将对应的数字加入结果，然后num像右平移四位，循环停止的条件是num为0，或者是已经循环了7次。

【CODE】class Solution {

public:

string toHex(int num) {

string res = "";

for (int i = 0; num && i < 8; ++i) {

int t = num & 0xf;

if (t >= 10) res = char('a' + t - 10) + res;

else res = char('0' + t) + res;

num >>= 4;

}

/\* string res = "", str = "0123456789abcdef";

\* int cnt = 0;

\* while (num != 0 && cnt++ < 8) {

\* res = str[(num & 0xf)] + res;

\* num >>= 4;

}\*/

return res.empty() ? "0" : res;

}

};

【分析】把要转换的十六进制的数字字母都放在一个字符串中，按位置直接取亦可。

421. Maximum XOR of Two Numbers in an Array数组中两个数的最大异或值

给定一个非空数组，数组中元素为 a0, a1, a2, … , an-1，其中 0 ≤ ai < 231 。

找到 ai 和aj 最大的异或 (XOR) 运算结果，其中0 ≤ i, j < n 。

你能在O(n)的时间解决这个问题吗？

【分析】这道题是一道典型的位操作Bit Manipulation的题目，开始以为异或值最大的两个数一定包括数组的最大值，但是OJ给了另一个例子{10,23,20,18,28}，这个数组的异或最大值是10和20异或，得到30。那么只能另辟蹊径，正确的做法是按位遍历，题目中给定了数字的返回不会超过231,那么最多只能有32位，我们用一个从左往右的mask，用来提取数字的前缀，然后将其都存入HashSet中，我们用一个变量t，用来验证当前位为1再或上之前结果res，看结果和HashSet中的前缀异或之后在不在HashSet中，这里用到了一个性质，若a^b=c，那么a=b^c，因为t是我们要验证的当前最大值，所以我们遍历HashSet中的数时，和t异或后的结果仍在HashSet中，说明两个前缀可以异或出t的值，所以我们更新res为t，继续遍历，如果上述讲解不容易理解，那么建议自己带个例子一步一步试试，并把每次循环中HashSet中所有的数字都打印出来，基本应该就能理解

3 10 5 25 2 8

11 1010 101 11001 10 1000

我们观察这些数字最大的为25，其二进制最高位在 i=4 时为1，那么我们的循环[31, 5]之间是取不到任何数字的，所以不会对结果res有任何影响。

当 i=4 时，我们此时mask为前28位为‘1’的二进制数，跟除25以外的任何数相‘与’，都会得到0。 然后跟25的二进制数10101相‘与’，得到二进制数10000，存入HashSet中，那么此时HashSet中就有0和16两个数字。此时我们的t为结果res（此时为0）‘或’上二进制数10000，得到二进制数10000。然后我们遍历HashSet，由于HashSet是无序的，所以我们会取出0和16中的其中一个，如果prefix取出的是0，那么t=16‘异或’上0，还等于16，而16是在HashSet中存在的，所以此时结果res更新为16，然后break掉遍历HashSet的循环。实际上prefix先取16的话也一样，那么t=16‘异或’上16，等于0，而0是在HashSet中存在的，所以此时结果res更新为16，然后break掉遍历HashSet的循环。

3 10 5 25 2 8

11 1010 101 11001 10 1000

当 i=3 时，我们此时mask为前29位为‘1’的二进制数，如上所示，跟数字3，5，2中任何一个相‘与’，都会得到0。然后跟10的二进制数1010，或跟8的二进制数1000相‘与’，都会得到二进制数1000，即8。跟25的二进制数11001相‘与’，会得到二进数11000，即24，存入HashSet中，那么此时HashSet中就有0，8，和24三个数字。此时我们的t为结果res（此时为16）‘或’上二进制数1000，得到二进制数11000，即24。此时遍历HashSet中的数，当prefix取出0，那么t=24‘异或’上0，还等于24，而24是在HashSet中存在的，所以此时结果res更新为24，然后break掉遍历HashSet的循环。大家可以尝试其他的数，当prefix取出24，其实也可以更新结果res为24的。但是8就不行啦，因为HashSet中没有16。不过无所谓了，我们只要有一个能更新结果res就可以了。

3 10 5 25 2 8

11 1010 101 11001 10 1000

当 i=2 时，我们此时mask为前30位为‘1’的二进制数，如上所示，跟3的二进制数11相‘与’，会得到二进制数0，即0。然后跟10的二进制数1010相‘与’，会得到二进制数1000，即8。然后跟5的二进制数101相‘与’，会得到二进制数100，即4。然后跟25的二进制数11001相‘与’，会得到二进制数11000，即24。跟数字2和8相‘与’，分别会得到0和8，跟前面重复了。所以最终HashSet中就有0，4，8，和24这四个数字。此时我们的t为结果res（此时为24）‘或’上二进制数100，得到二进制数11100，即28。那么就要验证结果res能否取到28。我们遍历HashSet，当prefix取出0，那么t=28‘异或’上0，还等于28，但是HashSet中没有28，所以不行。当prefix取出4，那么t=28‘异或’上二进制数100，等于24，在HashSet中存在，Bingo！结果res更新为28。其他的数可以不用试了。

3 10 5 25 2 8

11 1010 101 11001 10 1000

当 i=1 时，我们此时mask为前31位为‘1’的二进制数，如上所示，每个数与mask相‘与’后，我们HashSet中会有2，4，8，10，24这五个数。此时我们的t为结果res（此时为28）‘或’上二进制数10，得到二进制数11110，即30。那么就要验证结果res能否取到30。我们遍历HashSet，当prefix取出2，那么t=30‘异或’上2，等于28，但是HashSet中没有28，所以不行。当prefix取出4，那么t=30‘异或’上4，等于26，但是HashSet中没有26，所以不行。当prefix取出8，那么t=30‘异或’上8，等于22，但是HashSet中没有22，所以不行。当prefix取出10，那么t=30‘异或’上10，等于20，但是HashSet中没有20，所以不行。当prefix取出24，那么t=30‘异或’上24，等于6，但是HashSet中没有6，所以不行。遍历完了HashSet所有的数，结果res没有被更新，还是28。

3 10 5 25 2 8

11 1010 101 11001 10 1000

当 i=0 时，我们此时mask为前32位为‘1’的二进制数，如上所示，每个数与mask相‘与’后，我们HashSet中会有2，3，5，8，10，25这六个数。此时我们的t为结果res（此时为28）‘或’上二进制数1，得到二进制数11101，即29。那么就要验证结果res能否取到29。取出HashSet中每一个数字来验证，跟上面的验证方法相同，这里博主偷懒就不写了，最终可以发现，结果res无法被更新，还是28，所以最终的结果就是28。

综上所述，我们来分析一下这道题的核心。我们希望用二进制来拼出结果的数，最终结果28的二进制数为11100，里面有三个‘1’，我们来找一下都是谁贡献了这三个‘1’？在 i=4 时，数字25贡献了最高位的‘1’，在 i=3 时，数字25贡献了次高位的‘1’，在 i=2 时，数字5贡献了第三位的‘1’。而一旦某个数贡献了‘1’，那么之后在需要贡献‘1’的时候，此数就可以再继续贡献‘1’。而一旦有两个数贡献了‘1’后，那么之后的‘1’就基本上只跟这两个数有关了，其他数字有‘1’也贡献不出来。验证方法里使用了前面提到的性质，a ^ b = t，如果t是所求结果话，我们可以先假定一个t，然后验证，如果a ^ t = b成立，说明该t可以通过a和b‘异或’得到。

【CODE】class Solution {

public:

int findMaximumXOR(vector<int>& nums) {

int res = 0, mask = 0;

for (int i = 31; i >= 0; --i) {

mask |= (1 << i);

unordered\_set<int> s;

for (int num : nums) {

s.insert(num & mask);

}

int t = res | (1 << i);

for (int prefix : s) {

if (s.count(t ^ prefix)) {

res = t;

break;

}

}

}

return res;

}

};

461. Hamming Distance汉明距离

两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。

给出两个整数 x 和 y，计算它们之间的汉明距离。注意：0 ≤ x, y < 231.

【分析】两个数字之间的汉明距离就是其二进制数对应位不同的个数，那么最直接了当的做法就是按位分别取出两个数对应位上的数并异或，我们知道异或的性质上相同的为0，不同的为1，我们只要把为1的情况累加起来就是汉明距离。

【CODE】class Solution {

public:

int hammingDistance(int x, int y) {

int res = 0;

for (int i = 0; i < 32; ++i) {

if ((x & (1 << i)) ^ (y & (1 << i))) {

++res;

}

}

return res;

}

};

【分析】对上面的代码进行优化，我们可以一开始直接将两个数字异或起来，然后我们遍历异或结果的每一位，统计为1的个数，也能达到同样的效果。

【CODE】

class Solution {

public:

int hammingDistance(int x, int y) {

int res = 0, exc = x ^ y;

for (int i = 0; i < 32; ++i) {

res += (exc >> i) & 1;

}

/\*while (exc) {

\* ++res;

\* exc &= (exc - 1);

}\*/

return res;

}

};

【分析】上面的遍历每一位的方法并不高效，还可以进一步优化，假如数为num, num & (num - 1)可以快速地移除最右边的bit 1， 一直循环到num为0, 总的循环数就是num中bit 1的个数。

【分析】递归的写法，非常的简洁，递归终止的条件是当两个数异或为0时，表明此时两个数完全相同，我们返回0，否则我们返回异或和对2取余加上对x/2和y/2调用递归的结果。异或和对2取余相当于检查最低位是否相同，而对x/2和y/2调用递归相当于将x和y分别向右移动一位，这样每一位都可以比较到，也能得到正确结果。

【CODE】

class Solution {

public:

int hammingDistance(int x, int y) {

if ((x ^ y) == 0) return 0;

return (x ^ y) % 2 + hammingDistance(x / 2, y / 2);

}

};

477. Total Hamming Distance汉明距离总和

两个整数的 汉明距离 指的是这两个数字的二进制数对应位不同的数量。

计算一个数组中，任意两个数之间汉明距离的总和。

【分析】需要用异或来求每个位上的情况，那么我们需要来找出某种规律来，比如我们看下面这个例子，4，14，2和1：

4: 0 1 0 0

14: 1 1 1 0

2: 0 0 1 0

1: 0 0 0 1

我们先看最后一列，有三个0和一个1，那么它们之间相互的汉明距离就是3，即1和其他三个0分别的距离累加，然后在看第三列，累加汉明距离为4，因为每个1都会跟两个0产生两个汉明距离，同理第二列也是4，第一列是3。我们仔细观察累计汉明距离和0跟1的个数，我们可以发现其实就是0的个数乘以1的个数，发现了这个重要的规律，那么整道题就迎刃而解了，只要统计出每一位的1的个数即可。

【CODE】

class Solution {

public:

int totalHammingDistance(vector<int>& nums) {

int res = 0, n = nums.size();

for (int i = 0; i < 32; ++i) {

int cnt = 0;

for (int num : nums) {

if (num & (1 << i)) ++cnt;

}

res += cnt \* (n - cnt);

}

return res;

}

};

476. Number Complement数字的补数

给定一个正整数，输出它的补数。补数是对该数的二进制表示取反。

注意:给定的整数保证在32位带符号整数的范围内。

你可以假定二进制数不包含前导零位。

输入: 5 输出: 2

解释: 5的二进制表示为101（没有前导零位），其补数为010。所以你需要输出2。

输入: 1 输出: 0

解释: 1的二进制表示为1（没有前导零位），其补数为0。所以你需要输出0。

【分析】位操作里面的取反符号～本身就可以翻转位，但是如果直接对num取反的话就是每一位都翻转了，而最高位1之前的0是不能翻转的，所以我们只要用一个mask来标记最高位1前面的所有0的位置，然后对mask取反后，与上对num取反的结果即可。

【CODE】

class Solution {

public:

int findComplement(int num) {

int mask = INT\_MAX;

while (mask & num) mask <<= 1;

return ~mask & ~num;

}

};

【分析】迭代的写法，一行搞定，思路就是每次都右移一位，并根据最低位的值先进行翻转，如果当前值小于等于1了，就不用再调用递归函数。

【CODE】

class Solution {

public:

int findComplement(int num) {

return (1 - num % 2) + 2 \* (num <= 1 ? 0 : findComplement(num / 2));

}

};

693. Binary Number with Alternating Bits交替位二进制数

给定一个正整数，检查他是否为交替位二进制数：换句话说，就是他的二进制数相邻的两个位数永不相等。

【分析】利用了0和1的交替的特性，进行错位相加，从而组成全1的二进制数，然后再用一个检测全1的二进制数的trick，就是‘与’上加1后的数，因为全1的二进制数加1，就会进一位，并且除了最高位，其余位都是0，跟原数相‘与’就会得0，所以我们可以这样判断。比如n是10101，那么n>>1就是1010，二者相加就是11111，再加1就是100000，二者相‘与’就是0。

【CODE】

class Solution {

public:

bool hasAlternatingBits(int n) {

return ((n + (n >> 1) + 1) & (n + (n >> 1))) == 0;

}

};

【分析】将n右移两位，再和原来的n亦或，得到的新n其实就是除了最高位，其余都是0的数，然后再和自身减1的数相‘与’，如果是0就返回true，反之false。比如n是10101，那么n/4是101，二者相‘亦或’，得到10000，此时再减1，为1111，二者相‘与’得0。

【CODE】

class Solution {

public:

bool hasAlternatingBits(int n) {

return ((n ^= n / 4) & (n - 1)) == 0;

}

};

762. Prime Number of Set Bits in Binary Representation二进制表示中质数个计算置位

给定两个整数 L 和 R ，找到闭区间 [L, R] 范围内，计算置位位数为质数的整数个数。（注意，计算置位代表二进制表示中1的个数。例如 21 的二进制表示 10101 有 3 个计算置位。还有，1 不是质数。）

|  |  |
| --- | --- |
| 输入: L = 6, R = 10  输出: 4  解释:  6 -> 110 (2 个计算置位，2 是质数)  7 -> 111 (3 个计算置位，3 是质数)  9 -> 1001 (2 个计算置位，2 是质数)  10-> 1010 (2 个计算置位，2 是质数) | 输入: L = 10, R = 15  输出: 5  解释:  10 -> 1010 (2 个计算置位, 2 是质数)  11 -> 1011 (3 个计算置位, 3 是质数)  12 -> 1100 (2 个计算置位, 2 是质数)  13 -> 1101 (3 个计算置位, 3 是质数)  14 -> 1110 (3 个计算置位, 3 是质数)  15 -> 1111 (4 个计算置位, 4 不是质数) |

【分析】C++的内置函数\_\_builtin\_popcount来快速的求出非零位的个数cnt，然后又利用到了20以内的数，只要不能被2和3的一定是质数，又可以快速判断了质数。

【CODE】

class Solution {

public:

int countPrimeSetBits(int L, int R) {

int res = 0;

for (int i = L; i <= R; ++i) {

int cnt = \_\_builtin\_popcount(i);

res += cnt < 4 ? cnt > 1 : (cnt % 2 && cnt % 3);

}

return res;

}

};