编程中的数学

1：快速求最大公因数和最小公倍数

9.2 公倍数与公因数

利用辗转相除法，我们可以很方便地求得两个数的最大公因数（greatest common divisor，gcd）；将两个数相乘再除以最大公因数即可得到最小公倍数（least common multiple, lcm）。

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a% b);
}

int lcm(int a, int b) {
    return a * b / gcd(a, b);
}
```

进一步地，我们也可以通过扩展欧几里得算法（extended gcd）在求得 a 和 b 最大公因数的同时，也得到它们的系数 x 和 y，从而使 ax + by = gcd(a, b)。

```
int xGCD(int a, int b, int &x, int &y) {
    if (!b) {
       x = 1, y = 0;
       return a;
    }
    int x1, y1, gcd = xGCD(b, a % b, x1, y1);
    x = y1, y = x1 - (a / b) * y1;
    return gcd;
}
```

2：快速判断是否是质数（质数一定大于1）

204. 计数质数

难度 简单 729

统计所有小于非负整数 n 的质数的数量。

示例 1:

```
输入: n = 10
输出: 4
解释: 小于 10 的质数一共有 4 个, 它们是 2, 3, 5, 7 。
```

题解

埃拉托斯特尼筛法（Sieve of Eratosthenes，简称埃氏筛法）是非常常用的，判断一个整数是否是质数的方法。并且它可以在判断一个整数 n 时，同时判断所小于 n 的整数，因此非常适合这道题。其原理也十分易懂：从 1 到 n 遍历，假设当前遍历到 m，则把所有小于 n 的、且是 m 的倍数的整数标为和数；遍历完成后，没有被标为和数的数字即为质数。

```
class Solution {
public:
    int countPrimes(int n) {
        if(n<=2)return 0;
        int count=n-2;
        vector<bool>judge(n,true);
        for(int i=2;i<n;i++){
            if(judge[i]){
                for(int j=2;j*i<n;j++)judge[j*i]=false;
            }else --count;
        }
        return count;
    }
};
```

3：洗牌算法

384. 打乱数组

难度 中等 152

给你一个整数数组 nums ，设计算法来打乱一个没有重复元素的数组。

实现 `Solution` class:

- `Solution(int[] nums)` 使用整数数组 `nums` 初始化对象
- `int[] reset()` 重设数组到它的初始状态并返回
- `int[] shuffle()` 返回数组随机打乱后的结果

示例:

```
输入
["Solution", "shuffle", "reset", "shuffle"]
[[[1, 2, 3]], [], [], []]
输出
[null, [3, 1, 2], [1, 2, 3], [1, 3, 2]]

解释
Solution solution = new Solution([1, 2, 3]);
solution.shuffle();    // 打乱数组 [1,2,3] 并返回结果。任何 [1,2,3]的排列返回的概率应该相同。例如，返回 [3, 1, 2]
solution.reset();      // 重设数组到它的初始状态 [1, 2, 3] 。返回 [1, 2, 3]
solution.shuffle();    // 随机返回数组 [1, 2, 3] 打乱后的结果。例如，返回 [1, 3, 2]
```

题解

我们采用经典的 Fisher-Yates 洗牌算法，原理是通过随机交换位置来实现随机打乱，有正向和反向两种写法，且实现非常方便。注意这里 “reset” 函数以及类的构造函数的实现细节。

```
class Solution {
    vector<int> origin;
public:
    Solution(vector<int> nums): origin(std::move(nums)) {}

    vector<int> reset() {
        return origin;
    }

    vector<int> shuffle() {
        if (origin.empty()) return {};
        vector<int> shuffled(origin);
        int n = origin.size();
        // 可以使用反向或者正向洗牌，效果相同。
        // 反向洗牌：
        for (int i = n - 1; i >= 0; --i) {
            swap(shuffled[i], shuffled[rand() % (i + 1)]);
        }
        // 正向洗牌：
        // for (int i = 0; i < n; ++i) {
        //     int pos = rand() % (n - i);
        //     swap(shuffled[i], shuffled[i+pos]);
        // }
        return shuffled;
    }
};
```

4：用前缀和的方式来求不同权重的概率算法

题目描述：

528. 按权重随机选择

难度 中等 108

给定一个正整数数组 `w` ，其中 `w[i]` 代表下标 `i` 的权重（下标从 `0` 开始），请写一个函数 `pickIndex` ，它可以随机地获取下标 `i`，选取下标 `i` 的概率与 `w[i]` 成正比。

例如，对于 `w = [1, 3]`，挑选下标 `0` 的概率为 `1 / (1 + 3) = 0.25` （即，25%），而选取下标 `1` 的概率为 `3 / (1 + 3) = 0.75`（即，75%）。

也就是说，选取下标 `i` 的概率为 `w[i] / sum(w)` 。

题目分析:

题解

我们可以先使用 partial_sum 求前缀和（即到每个位置为止之前所有数字的和），这个结果对于正整数数组是单调递增的。每当需要采样时，我们可以先随机产生一个数字，然后使用二分法查找其在前缀和中的位置，以模拟加权采样的过程。这里的二分法可以用 lower_bound 实现。

以样例为例，权重数组 [1,3] 的前缀和为 [1,4]。如果我们随机生成的数字为 1，那么 lower_bound 返回的位置为 0；如果我们随机生成的数字是 2、3、4，那么 lower_bound 返回的位置为 1。

关于前缀和的更多技巧，我们将在接下来的章节中继续深入讲解。

```cpp
class Solution {
vector<int> sums;
public:
    Solution(vector<int> weights): sums(std::move(weights)) {
        partial_sum(sums.begin(), sums.end(), sums.begin());
    }

    int pickIndex() {
        int pos = (rand() % sums.back()) + 1;
        return lower_bound(sums.begin(), sums.end(), pos) - sums.begin();
    }
};
```

5：蓄水池算法

题目描述：

382. 链表随机节点

难度 中等 147

给定一个单链表，随机选择链表的一个节点，并返回相应的节点值。保证每个节点**被选的概率一样**。

进阶:
如果链表十分大且长度未知，如何解决这个问题？你能否使用常数级空间复杂度实现？

蓄水池算法介绍：

- 假设数据流中含有 N 个数据，要保证每条数据被抽取到的概率相等，那么每个数被抽取的概率必然是 $\dfrac{k}{N}$
  - 对于前 k 个数 $n_1, n_2, \ldots, n_k$，我们保留下来，则 $p(n_1) = p(n_2) = \ldots = p(n_k) = 1$（下面连等采用 $p(n_{1-k})$ 的形式
  - 对于第 k+1 个数 n_{k+1}，以 $\dfrac{k}{k+1}$ 的概率保留它（这里只是指本次保留下来），那么前 k 个数中的 $n_r(r \in 1-k)$ 被保留的概率可以这样表示：$p(n_r\text{被保留}) = p(\text{上一轮}n_r\text{被保留}) \times (p(n_{k+1}\text{被丢弃}) + p(n_{k+1}\text{被保留}) \times p(n_r\text{未被替换}))$，即 $p_{1-k} = \dfrac{1}{k+1} + \dfrac{k}{k+1} \times \dfrac{k-1}{k} = \dfrac{k}{k+1}$
  - 对于第 k+2 个数 n_{k+2}，以 $\dfrac{k}{k+2}$ 的概率保留它（这里只是指本次保留下来），那么前 k+1 个被保留下来的数中的 $n_r(r \in 1-k+1)$ 被保留的概率为：$p_{1-k} = \dfrac{k}{k+1} \times \dfrac{2}{k+2} + \dfrac{k}{k+1} \times \dfrac{k-1}{k+2}$
  - ……
  - 对于第 i(i>k) 个数 n_i，以 $\dfrac{k}{i}$ 的概率保留它，前 i-1 个数中的 $n_r(r \in 1-i-1)$ 被保留的概率为：$p_{1-k} = \dfrac{k}{i-1} \times \dfrac{i-k}{i} + \dfrac{k}{i-1} \times \dfrac{k-1}{i} = \dfrac{k}{i}$

对于前 k 个数，全部保留，对于第 i(i>k) 个数，以 $\dfrac{k}{i}$ 的概率保留第 i 个数，并以 $\dfrac{1}{k}$ 的概率与前面已选择的 k 个数中的任意一个替换。

我们提供一个简单的，对于水库算法随机性的证明。对于长度为 n 的链表的第 m 个节点，最后被采样的充要条件是它被选择，且之后的节点都没有被选择。这种情况发生的概率为 $\frac{1}{m} \times \frac{m}{m+1} \times \frac{m+1}{m+2} \times \cdots \times \frac{n-1}{n} = \frac{1}{n}$。因此每个点都有均等的概率被选择。

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
private:
//蓄水池算法
ListNode*Head;
public:
    /** @param head The linked list's head.
        Note that the head is guaranteed to be not null, so it contains
at least one node. */
    Solution(ListNode* head):Head(head) {

    }

    /** Returns a random node's value. */
    int getRandom() {
        ListNode*head=Head;
        int ans=head->val,i=2;//初始时设置答案为头，概率为1
        head=head->next;
        while(head){
            //由于rand()生成的是伪随机数，rand()%i==0的概率为1/i;
            //当有1/i的概率改变值时，前面的取相应值的概率都会改变
            if(rand()%i==0){
                ans=head->val;
            }
            head=head->next;
            ++i;
        }
        return ans;
    }
};

/**
 * Your Solution object will be instantiated and called as such:
 * Solution* obj = new Solution(head);
```

```
 * int param_1 = obj->getRandom();
 */
```

6：前缀和后缀

238. 除自身以外数组的乘积

难度 中等 891

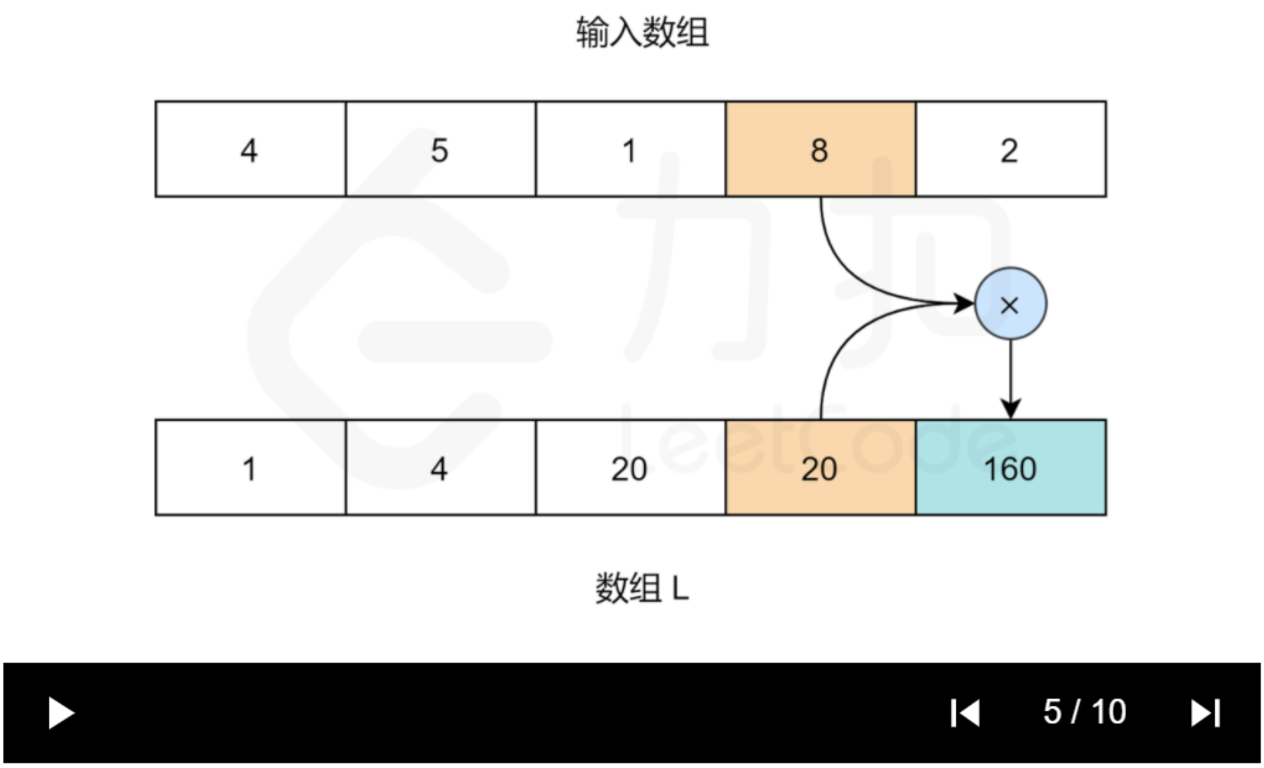
给你一个长度为 n 的整数数组 `nums`，其中 $n > 1$，返回输出数组 `output`，其中 `output[i]` 等于 `nums` 中除 `nums[i]` 之外其余各元素的乘积。

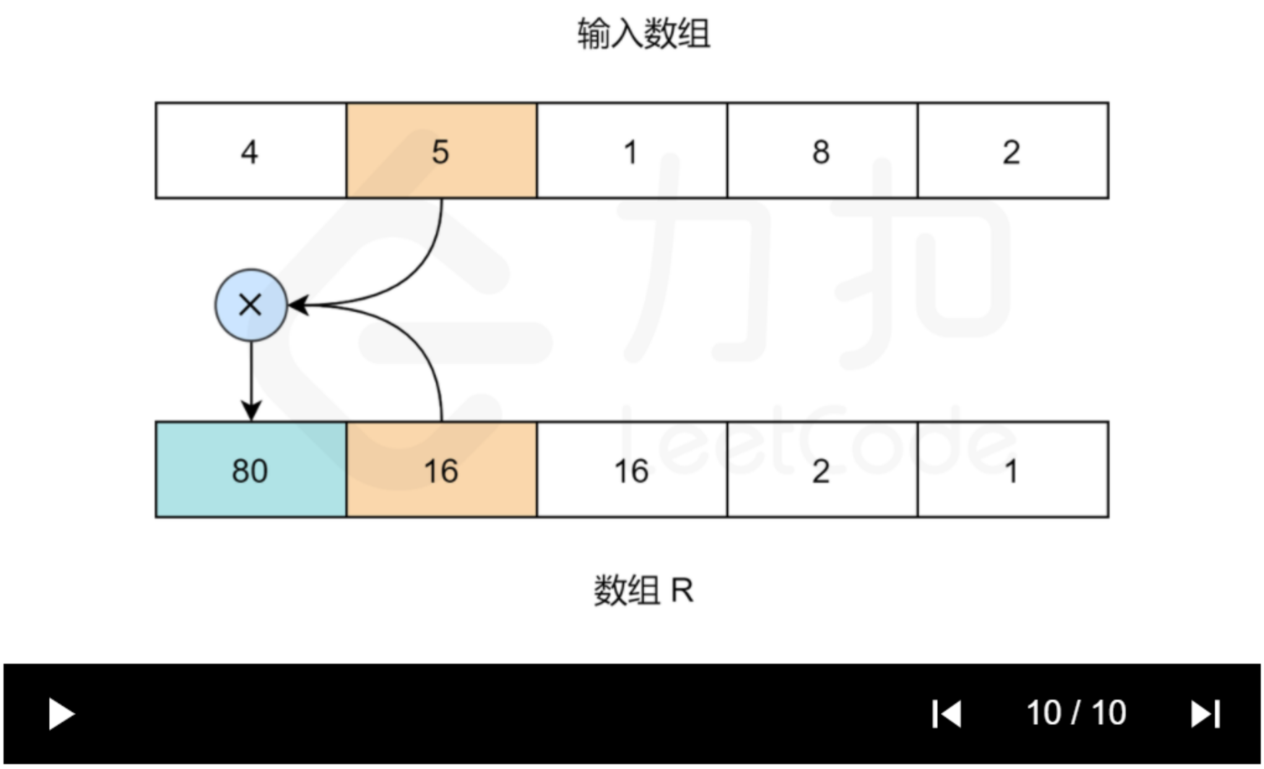
示例:

```
输入: [1,2,3,4]
输出: [24,12,8,6]
```

提示：题目数据保证数组之中任意元素的全部前缀元素和后缀（甚至是整个数组）的乘积都在 32 位整数范围内。

说明: 请**不要使用除法，**且在 $O(n)$ 时间复杂度内完成此题。






```
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        //前后缀矩阵列表
        int n=nums.size();
        vector<int>res(n,1);
        for(int i=1;i<n;i++)res[i]=nums[i-1]*res[i-1];
        int temp=1;
        for(int i=n-2;i>=0;i--){
```

```
            temp=nums[i+1]*temp;
            res[i]*=temp;
        }
        return res;
    }
};
```