LeetCode

索引

快速求出数组元素到某个值的距离——LeetCode 2602. 使数组元素全部相等的最少操作次数

数学相关

二进制小数的表示

LeetCode 面试题 05.02. 二进制数转字符串

设十进制小数 num 的二进制表示为 $0.b_1b_2b_3\cdots b_k$ ，考虑将其转为二进制字符串

每次乘 2 将小数点向右移动一位，变为 $b_1.b_2b_3\cdots b_k$ ，将 b_1 记录下来并减去，至多循环 32 次。如果 num 最终变为 0 ，那么便可以直接返回

不难写出如下代码：

```
class Solution {
public:
    string printBin(double num)
```

```
    {
        string ans = "0.";
        for(int i = 0; i < 32; i ++)
        {
            num *= 2;
            if(num < 1)
            {
                ans += "0";
            }
            else
            {
                ans += "1";
                if(--num == 0) return ans;
            }
        }
        return "ERROR";
    }
};
```

优化：

- 任何一个**有限位** p 进制小数均可以表示成**最小分数** $\frac{a}{p^k}$ ，其中 a 和 p^k 互质

num 最多为十进制**六位**小数，设其表示为 $\frac{a}{10^6}$ ，设其对应二进制表示为 $\frac{b}{2^k}$ ，有：

$$\frac{a}{10^6}=\frac{b}{2^k}\rightarrow\frac{a}{2^65^6}=\frac{b}{2^k}\rightarrow b=\frac{a2^{k-6}}{5^6}$$

由于 b 与 2 互质，因此 $0\le k\le 6$ ，即对于六位十进制小数而言，其对应二进制小数最多只有六位，因此只需要枚举 6 次即可

完整代码：

```
class Solution {
public:
    string printBin(double num)
    {
        string ans = "0.";
        for(int i = 0; i < 6; i ++)
        {
            num *= 2;
            if(num < 1)
            {
                ans += "0";
            }
            else
            {
                ans += "1";
                if(--num == 0) return ans;
            }
        }
```

```
        return "ERROR";
    }
};
```

位运算

下面给出常用位运算技巧：

我们用一个二进制数来表示一个集合，那么有：

1. 将 x 变为集合：`1 << x`
2. 判断 x 是否在集合 A 中：`((A >> x) & 1) == 1`
3. 计算两个集合 A, B 的并集 $A \cup B$：`A | B`
4. 计算 $A \setminus B$，表示从集合 A 中去掉集合 B 中的元素：`A & ~B` 。例如 `110 & ~11 = 100`
5. 得到全集 $U = 0, 1, 2, \cdots, n-1$ ：`(1 << n) - 1`

排列相关

求全局 & 局部倒置对数量

原题链接：LeetCode 775. 全局倒置与局部倒置

整理一下题意，对于满足 $nums[i] > nums[j], i < j$ 的逆序对称为全局倒置；对于满足 $nums[i] > nums[i+1]$ 的逆序对称为局部倒置

如果全局倒置的数量**等于**局部倒置的数量，则返回 `true`

贪心

注意到局部倒置的数量为全局倒置数量的**子集**，也就是说如果我们统计所有满足「属于全局倒置而不属于局部倒置」的逆序对，只要这个结果为 0 ，就表明全局倒置和局部倒置的数量相等

如果当前值比满足「属于全局倒置而不属于局部倒置」后缀的最小值要大，那么说明必然不符合题意

也就是说，我们需要从 $n-3$ 开始枚举，而后缀最小值从 $n-1$ 开始维护，每次用 $i+1$ 来进行更新

完整代码如下：

```
class Solution {
public:
    bool isIdealPermutation(vector<int>& nums)
    {
        int n = nums.size();
        int mn = nums[n - 1];
        for(int i = n - 3; i >= 0; i --)
        {
            if(nums[i] > mn) return false;
            mn = min(mn, nums[i + 1]);
        }
```

```
            return true;
        }
    };
```

时间复杂度：$O(n)$

树状数组

对于 x ，我们需要求出前面所有比其大的数的个数，这恰好可以用树状数组（前缀和）来实现

我们以**元素值为下标**，数组对应值表示该元素出现的**次数**，对于 x 而言，所有比其小的数的个数为 $\sum_{i=1}^{x-1} s_i$

完整代码如下：

```
class Solution {
public:

    static const int N = 1e5 + 10;
    int n;
    int tr[N];
    int lowbit(int x)
    {
        return x & -x;
    }
    void add(int x, int c)
    {
        for(int i = x; i <= n; i += lowbit(i))
            tr[i] += c;
    }
    int query(int x)
    {
        int ans = 0;
        for(int i = x; i > 0; i -= lowbit(i))
            ans += tr[i];
        return ans;
    }
    bool isIdealPermutation(vector<int>& nums)
    {
        n = nums.size();
        memset(tr, 0, sizeof tr);
        add(nums[0] + 1, 1);
        long a = 0, b = 0;
        for(int i = 1; i < n; i ++)
        {
            int v = nums[i] + 1;
            a += query(n) - query(v);
            b += nums[i] < nums[i - 1] ? 1 : 0;
            add(v, 1);
        }
        return a == b;
```

```
    }
};
```

时间复杂度：$O(n\log n)$ ，树状数组的插入和查询均为 $O(\log n)$

基于排列性质

由于整个数组是一个从 $0\sim n-1$ 的排列，因此对于每个数组元素，将整个数组排序后都有与其**唯一对应**的位置（也就是以**元素下标**）

具体地，对于排列 $[3,4,0,2,1]$ ，有：

3 应当放在下标为 3 的位置，4 应当放在下标为 4 的位置，0 应当放在下标为 0 的位置，2 应当放在下标为 2 的位置，1 应当放在下标为 1 的位置

回到本题，对于元素 $nums[i]$ 而言，其应当放在位置 i ，如果我们期望由 $nums[i]$ 构成的逆序对满足「属于全局倒置而不属于局部倒置」，这就要求 $nums[i]$ **距离** i **不能超过** 1，也就是不能超过局部倒置的范围，即：

$$|nums[i]-i|\leq 1$$

所以我们可以对每个数依次判断它与对应位置的距离，完整代码如下：

```
class Solution {
public:
    bool isIdealPermutation(vector<int>& nums)
    {
        for(int i = 0; i < nums.size(); i ++)
            if(abs(nums[i] - i) >= 2) return false;
        return true;
    }
};
```

数组

LeetCode 1630. 等差子数组

这道题可以用最暴力的做法，每次将子数组进行排序，然后考虑子数组是否为等差数列，这里我们直接给出代码，不做过多讨论

```
class Solution {
public:
    vector<bool> checkArithmeticSubarrays(vector<int>& nums, vector<int>& l,
vector<int>& r)
    {
        int n = nums.size(), m = l.size();
        vector<bool>ans(m, true);
        for(int i = 0; i < m; i ++)
        {
```

```
            vector<int>tmp(nums.begin() + l[i], nums.begin() + r[i] + 1);
            sort(tmp.begin(), tmp.end());
            int d = tmp[1] - tmp[0];
            for(int j = 2; j < tmp.size(); j ++)
            {
                if(tmp[j] - tmp[j - 1] != d)
                {
                    ans[i] = false;
                    break;
                }
            }
        }
        return ans;
    }
};
```

这么做的时间复杂度为：$O(mn\log n)$，其实有 $O(nm)$ 的做法

对于一个序列，设最大值为 $maxv$，最小值为 $minv$，元素个数为 l，如果从新排列使其能够成为等差序列，其公差必然为：

$$\frac{maxv-minv}{l-1}$$

如果无法整除，说明该序列无法构成等差数列

更进一步，我们需要确认将数从新排列后是否出现冲突，因此需要确认每个数在等差数列当中的位值，如果出现冲突则说明该序列无法构成等差数列

具体地，对于数 x 而言，其在等差数列当中的位置为：

$$idx=\frac{x-minv}{d}$$

因此这一步我们可以开一个哈希表来判断是否出现冲突，如果都没有出现冲突，则说明该子数组重新排列后可以构成等差数列

完整代码如下：

```
class Solution {
public:
    vector<bool> checkArithmeticSubarrays(vector<int>& nums, vector<int>& l,
vector<int>& r)
    {
        int n = nums.size(), m = l.size();
        vector<bool>ans;
        for(int i = 0; i < m; i ++)
        {
            int left = l[i], right = r[i];
            int maxv = *max_element(nums.begin() + left, nums.begin() + right +
1);
            int minv = *min_element(nums.begin() + left, nums.begin() + right +
```

```
1);
            if((maxv - minv) % (right - left) != 0)//无法整除的情况
            {
                ans.push_back(false);
                continue;
            }
            if(maxv == minv)//数列中每个数都相同
            {
                ans.push_back(true);
                continue;
            }
            int d = ((maxv - minv) / (right - left));
            bool flag = true;
            vector<int>hash(right - left + 1, 0);
            for(int j = left; j <= right; j ++)
            {
                if((nums[j] - minv) % d != 0)//无法整除的情况
                {
                    flag = false;
                    break;
                }
                int t = (nums[j] - minv) / d;
                if(hash[t])
                {
                    flag = false;
                    break;
                }
                else hash[t]++;
            }
            ans.push_back(flag);
        }
        return ans;
    }
};
```

删除部分元素使数组有序

LeetCode 1574. 删除最短的子数组使剩余数组有序

双指针 + 二分

我们需要删除一个子数组使得剩余数组呈**非递减**状态

首先从左往右遍历，得到**最长非递减前缀**下标 $left$ ，此时数组 $[0 \cdots left]$ 均为**非递减数组**

此前从右往左遍历，得到**最长非递减后缀**下标 $right$，此时数组 $[right \cdots n-1]$ 均为**非递减数组**

此时可以我们可以删去 $[0 \cdots right-1]$ 或者 $[left+1 \cdots n-1]$ ，此时可以保证剩余数组一定有序，即 $ans = \min(right, n-1-left)$

此后，我们枚举区间 $[0 \cdots left]$ （设下标为 l），在 $[right \cdots n-1]$ 中找到第一个**大于等于**的数（设下标为 r），此时删去区间 $[l+1, r-1]$ 可保证剩余数组一定有序，即 $ans = \min(ans, r-l-1)$

> 注：lower_bound 可以找到第一个**大于等于**的数，upper_bound 可以找到第一个**大于**的数

完整代码如下：

```
class Solution {
public:
    int findLengthOfShortestSubarray(vector<int>& arr)
    {
        int n = arr.size();
        int left = 0, right = n - 1;
        while(left + 1 <= n - 1 && arr[left] <= arr[left + 1]) left++;
        while(right - 1 >= 0 && arr[right - 1] <= arr[right]) right--;
        if(left >= right) return 0;
        int ans = min(right, n - left - 1);//删去[left + 1, n - 1] 与 [0, right -
1]
        for(int l = 0; l <= left; l ++)
        {
            int r = lower_bound(arr.begin() + right, arr.end(), arr[l]) -
arr.begin();
            ans = min(ans, r - l - 1);//删去[l + 1, r - 1]
        }
        return ans;
    }
};
```

双指针

类似于上面的做法，我们首先找到**最长非递减后缀** $right$，此时 $ans = right$

左指针 $left$ 从左往右遍历，我们需要删除的区间为开区间 $(left, right)$

对左指针而言，首先需要保证非递减，因此必然需要满足 $arr[left - 1] = arr[left]$

再者，对于元素 $arr[left]$ ，我们需要**右移**右指针 $right$ ，使得 $arr[right] \geq arr[left]$

完整代码如下：

```
class Solution {
public:
    int findLengthOfShortestSubarray(vector<int>& arr)
    {
        int n = arr.size();
        int right = n - 1;
        while(right >= 1 && arr[right - 1] <= arr[right]) right--;
        if(right == 0) return 0;
        int ans = right;//删除[0,right-1]
        for(int left = 0; left == 0 || arr[left - 1] <= arr[left]; left ++)
        {
            while(right <= n - 1 && arr[left] > arr[right]) right++;
            ans = min(ans, right - left - 1);
```

```
        }
        return ans;
    }
};
```

快速求出数组元素到某个值的距离

原题链接：LeetCode 2602. 使数组元素全部相等的最少操作次数

前缀和 + 二分

将 $nums[i]$ 中的元素全部变为 q，相当于求 $nums[i]$ **中每个元素到 q 的距离**

由于 $nums[i]$ 的顺序与最终结果无关，因此可以先对其**排序**

此后我们二分，找到第一个**大于等于** q 的数的下标 k，此时左边 $< q$，右边 $\geq q$

然后我们从面积的角度来考虑，左边的距离和为：q 对应面积减去元素面积；右边距离和为：元素面积减去 q 对应面积，即：

$$l = k \times q - s[k] \quad r = (s[n] - s[k]) - (n - k) \times q$$

完整代码如下：

```
class Solution {
public:
    vector<long long> minOperations(vector<int>& nums, vector<int>& queries)
    {
        sort(nums.begin(), nums.end());
        int n = nums.size(), m = queries.size();
        vector<long long>s(n + 1, 0);
        for(int i = 1; i <= n; i ++)
            s[i] = s[i - 1] + nums[i - 1];
        vector<long long>ans(m);
        for(int i = 0; i < m; i ++)
        {
            long long k = lower_bound(nums.begin(), nums.end(), queries[i]) -
nums.begin();
            long long l = k * queries[i] - s[k];
            long long r = (s[n] - s[k]) - (n - k) * queries[i];
            ans[i] = l + r;
        }
        return ans;
    }
};
```

时间复杂度：$O(n \log n)$

考虑距离和的增量

LeetCode 2615. 等值距离和

前缀和

我们先对**所有相同元素的下标进行分组**，之后再单独考虑每个组内部的情况

对于某个组 q，其每个元素均为下标，我们需要求**每个元素到其余元素的距离之和**，那么本题与上一题就一样了

完整代码如下：

```
class Solution {
public:
    vector<long long> distance(vector<int>& nums)
    {
        int n = nums.size();
        unordered_map<int, vector<int>>hash;
        for(int i = 0; i < n; i ++)
            hash[nums[i]].push_back(i);
        vector<long long>s(n + 1, 0), ans(n);
        for(auto& [_, q] : hash)
        {
            int m = q.size();
            for(int i = 1; i <= m; i ++)
                s[i] = s[i - 1] + q[i - 1];
            for(int i = 0; i < m; i ++)
            {
                long long v = q[i];
                long long l = i * v - s[i];
                long long r = s[m] - s[i] - v * (m - i);
                ans[v] = l + r;
            }
        }
        return ans;
    }
};
```

时间复杂度：$O(n)$，由于数组 q 本身有序且每次求的 v 均为数组元素，因此不需要二分

贪心

设数组 q 长度为 n，下标从 0 开始，s_i 表示下标 i 对应元素到其余元素的距离

假设当前已经求出了 s_0，我们考虑 s_1 与 s_0 之间的关系：

从 q_0 到 q_1，有 1 **个**元素增大了 $q_1 - q_0$，$n-1$ 个元素减小了 $q_1 - q_0$，即：
$s_1 = s_0 + (q_1 - q_0) - (n-1)(q_1 - q_0)$

更一般地，s_i 与 s_{i-1} 之间，有 i 个元素增大了 $q_i - q_{i-1}$，有 $n-i$ 个元素减小了 $q_i - q_{i-1}$，即：

$$s_i = s_{i-1} + i \times (q_i - q_{i-1}) - (n-i) \times (q_i - q_{i-1}) = s_{i-1} + (2i - n) \times (q_i - q_{i-1})$$

在代码实现上，我们先求出 s_0 ，然后不断更新 s_i 即可

完整代码如下：

```
class Solution {
public:
    vector<long long> distance(vector<int>& nums)
    {
        int n = nums.size();
        unordered_map<int, vector<int>>hash;
        for(int i = 0; i < n; i ++)
            hash[nums[i]].push_back(i);
        vector<long long>ans(n);
        for(auto& [_, q] : hash)
        {
            int m = q.size();
            long long s = 0;
            for(int i = 0; i < m; i ++) s += (q[i] - q[0]);
            ans[q[0]] = s;
            for(int i = 1; i < m; i ++)
                ans[q[i]] = s += (long long)(2 * i - m) * (q[i] - q[i - 1]);
        }
        return ans;
    }
};
```

数据结构应用题

原题链接：846. 一手顺子

出于便利，我们用 m 代替 $groupSize$

只要 $hand$ 和 m 唯一给定，那么划分的方式也就确定了，因此本题实质上是**模拟**的过程

在这里我们用「哈希表」来对每个数进行频率统计，之后**从小到大**枚举每个数

设当前枚举数为 t ，我们考虑以 t 为**左端点**的序列

如果序列 $t,t+1,t+2,\cdots,t+m-1$ 当中的任意一个数均在哈希表中出现过，则将对应数的出现频率减一

这个过程中，只要出现有某个数出现频率变为负数，说明发生冲突，直接返回 `false`

而对于当前枚举的左端点 t ，如果其在哈希表中对应值为 0 ，表明当前数已经与前面的数相匹配，直接跳过

由于我们每次都需要取当前的较小值，因此考虑用「小根堆」解决

完整代码如下：

```
class Solution {
public:
```

```
    bool isNStraightHand(vector<int>& hand, int groupSize)
    {
        if(hand.size() % groupSize) return false;
        unordered_map<int, int>hash;
        priority_queue<int, vector<int>, greater<int>>heap;
        for(int x : hand) hash[x]++, heap.push(x);
        while(heap.size())
        {
            int t = heap.top();
            heap.pop();
            if(hash[t] == 0) continue;
            for(int i = 0; i < groupSize; i ++)
            {
                hash[t + i]--;
                if(hash[t + i] < 0) return false;
            }
        }
        return true;
    }
};
```

字符串

子串思考的通用思路

LeetCode 1638. 统计只差一个字符的子串数目

如果需要枚举一个字符串的子串，需要知道两个量：结束位置与起始位置

如果需要枚举**一对**字符串的子串，在长度相同的情况下，需要知道三个量：`s` 中的结束位置 i 、`t` 中的结束位置 j 、子串长度（起始位置 k）

首先考虑 $i = j$ 的情况

此时该子串的结束位置是固定的，我们**向前枚举**，找到**上一个不同字符的下标** $k1$，与**上上个不同字符的下标** $k0$

此时统计的个数为区间 $(k0, k1]$ 当中的字符数，即 $k1 - k0$

二者的初值可以直接为 -1（二者的意义均为：不同字符的位置）

对于 $i \neq j$ 的情况，我们枚举二者的**差值** $d = i - j$

由于 $1 \leq i \leq n, 1 \leq j \leq m$ ，因此 $1 - m \leq d \leq n - 1$

我们**始终以 `s` 为基准**，考察 d 的范围：

- 若 $d > 0$ ，此时有 $i > j$ ，相当于将 `t` 右移 d 个单位，此时 `s` 的起始遍历位置为 d ，`t` 的起始遍历位置为 0，将 $k0, k1$ 初始化为 $d - 1$
- 若 $d < 0$ ，此时有 $i < j$ ，相当于将 `t` 左移 d 个单位，此时 `s` 的起始遍历位置为 0 ，`t` 的起始遍历位置为 d ，将 $k0, k1$ 初始化为 -1

完整代码如下：

```
class Solution {
public:
    int countSubstrings(string s, string t)
    {
        int n = s.length(), m = t.length(), ans = 0;
        for(int d = 1 - m; d <= n - 1; d ++)//d = i - j => j = i - d
        {
            for(int i = max(0, d), k1 = i - 1, k0 = k1; i < n && i - d < m; i ++)
            {
                if(s[i] != t[i - d])
                    k0 = k1, k1 = i;
                ans += k1 - k0;
            }
        }
        return ans;
    }
};
```

时间复杂度：$O((n+m)\min(n,m)) = O(nm)$

回文字符串

原题链接：1400. 构造 K 个回文字符串

对于回文字符串有一个结论：对于长度为 n 的回文字符串，一定可以分解为**个数或长度**为 d, $d \in [1,n]$ 的回文字符串

我们考虑回文字符串的组成，必然有两种可能：

- 长度为奇数，其中心为**一个字符**
- 长度为偶数，其中心为**两个相等字符**

我们随意取一个回文字符串 `abcdfdcba` ，我们去掉中心字符，得到 `abcddcba` 和 `f` ，进一步，我们对 `abcddcba` 进行拆分，有：`abccba`, `dd`, `f`

这个过程可以持续下去，我们可以得到一个结论：

对于一个长度为 n 的回文字符串，对其拆分必然可以得到个数为 $1,2,3,\cdots,n$ 的回文字符串同时也可以得到长度为 $1,2,3,\cdots,n$ 的回文字符串

回到本题，对于字符串 `s` ，如果我们求出其能表示出的**最少回文字符串 l 和最多回文字符串 r** ，那么只要 k 处于 $[l,r]$ 内，必然是符合条件

设 `s` 中有 p 个字符出现了奇数次，有 q 个字符出现了偶数次，那么 `s` 所能构成的回文字符串的个数最小值是多少?

对于出现偶数次的字符，我们可以任意组合，但对于出现奇数次的字符，我们需要保证其**必须处于一个回文字符串的中心**，因此这最少能表示的字符个数为 p

特别地，如果如果 $p == 0$ ，表明全部由偶数个字符组成，此时个数为 1 ，因此总体为 $\max(p, 1)$

完整代码如下：

```
class Solution {
public:
    bool canConstruct(string s, int k)
    {
        unordered_map<char, int>hash;
        for(char c : s) hash[c]++;
        int cnt = 0, tot = s.length();
        for(auto p : hash)
            if(p.second & 1) cnt++;
        return k >= max(cnt, 1) && k <= tot;
    }
};
```

二叉树

原题链接：1110. 删点成林

我们考虑如果要将一个节点加入答案数组需要满足什么条件：

- 当前点不能有父节点
- 当前点不在 `to_delete` 中
- 当前点的左右子树均处理完毕

我们根据这三个条件着手设计递归函数

基于前两点，我们需要用 u 来表示当前递归到的节点，用 has_father 来表示当前节点是否有父节点

在返回值部分，我们需要**根据当前节点的左孩子与右孩子是否被删除来判断是否需要对当前节点进行更改**，因此需要用 `bool` 作为返回类型

意思是说，如果当前节点的左孩子需要被删除，那么我们需要将当前节点的左孩子置空，这对于右孩子也是一样

基于此设计，在返回时我们需要**返回当前节点是否会被删除**

在代码实现上，考虑去重可以直接用哈希表

```
class Solution {
public:
    vector<TreeNode*> delNodes(TreeNode* root, vector<int>& to_delete)
    {
        unordered_set<int>hash(to_delete.begin(), to_delete.end());
        vector<TreeNode*>ans;
        function<bool(TreeNode*, bool)> dfs = [&](TreeNode* u, bool has_father) ->
bool
```

```
        {
            //当前节点为nullptr，直接返回true
            if(!u) return true;
            bool is_delete = hash.count(u->val);//为true表示当前节点需要删除
            if(dfs(u->left, !is_delete))
                u->left = nullptr;
            if(dfs(u->right, !is_delete))
                u->right = nullptr;
            //当前节点不能被删除且没有父节点
            if(!has_father && !is_delete)
                ans.push_back(u);
            return is_delete;
        };
        dfs(root, false);
        return ans;
    }
};
```