数学

进制

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 $0\leq k\leq 6$ ，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";
    }
};
```

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;
    }
};
```

字符串

字符串的通用思路

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：$O((n+m)\min(n,m))=O(nm)$