基础算法

双指针及其使用条件

双指针使用的条件为两个指针都必须具有**单调性**，所谓单调性，就是当右指针向右走的时候，左指针**不允许**出现向左走的情况，具体需要依据题目来确定，但必须要满足这一点才能够用双指针

能够使用双指针

LeetCode 1574. 删除最短的子数组使剩余数组有序

不能使用双指针

原题连接：LeetCode 1590. 使数组和能被 P 整除

求区间和，首先用前缀和处理，将区间和转化为两个数的差，有：

```
int n = nums.size();
vector<int>prefix(n + 1, 0);
for(int i = 1; i <= n; i ++)
    prefix[i] = (prefix[i - 1] + nums[i - 1]) % p;
```

设需要删除的区间为 $[l,r]$ ，此时区间和为 $prefix[r]-prefix[l-1]$ ，若满足

$$prefix[n]-(prefix[r]-prefix[l-1])\equiv 0(\bmod\ p)$$

那么该区间便是局部解

这里不能用双指针的点在于，$prefix[i] \bmod\ p$ 并不是单调的，也就是说当右指针递增时，我们**无法保证左指针一定不会出现递减的情况**

关于「单调性」的理解：

在这里我将其理解为「二分性」

回顾二分查找算法：在一个单调递增的数组中，找到某个数 x 的最小下标 对于一个单调递增的区间而言，它是具有「二分性」的，整个数组可以划分为两个集合——「满足性质 p」以及「不满足性质 p」，在这个案例中，「性质 p」为是否「大于等于 x」

这两个集合没有交集，而二分找的就是两个集合的两个分界点

回到这里，当右指针递增的时候，如果此时左右指针构成的区间能够保证二分性，那么便说明可以使用双指针

对于区间 $[l,r]$ ，我们无法找到某个点 idx 将区间划分为两个**不相交**的集合使得左集合内所有元素均满足 $prefix[n]-(prefix[r]-prefix[l])\equiv 0(\bmod\ p)$

这道题正确的做法是用哈希表

由于每次枚举的是右端点，因此为确定值，将上式移项，有：

$$ prefix[l-1]\equiv prefix[r]-prefixn $$

因此，只需要用哈希表记录**最近一次**$prefix[r]-prefixn$出现的下标，然后取最小值即可

这道题有两个细节：

- 哈希表需要预先将 0 存入，因为前缀和是从 0 开始的
- 求前缀和时，需要对 p 取模，因为会爆 `int`

完整代码如下：

```
class Solution {
public:
    int minSubarray(vector<int>& nums, int p)
    {
        int n = nums.size();
        vector<int>prefix(n + 1, 0);
        for(int i = 1; i <= n; i ++)
            prefix[i] = (prefix[i - 1] + nums[i - 1]) % p;
        int cnt = 0x3f3f3f3f;
        unordered_map<int, int>Hash;
        for(int i = 0; i <= n; i ++)//需要将0提前放入哈希表中
        {
            Hash[prefix[i]] = i;//用于记录prefix[i]在哈希表中最后一次出现得到下标
            int left = (prefix[i] - prefix[n] + p) % p;//保证不出现负数
            if(Hash.find(left) != Hash.end())
                cnt = min(cnt, i - Hash[left]);
        }
        return cnt == 0x3f3f3f3f || cnt == n ? -1 : cnt;
    }
};
```

如果用 `long long` ，不对前缀和取模，需要在哈希表插入时取模，因为 $prefix[r]-prefixn的值不会超过p$

```
class Solution {
public:
    int minSubarray(vector<int>& nums, int p)
    {
        int n = nums.size();
        vector<long long>prefix(n + 1, 0);
        for(int i = 1; i <= n; i ++)
            prefix[i] = (prefix[i - 1] + nums[i - 1]);
        int cnt = 0x3f3f3f3f;
        unordered_map<int, int>Hash;
        for(int i = 0; i <= n; i ++)//需要将0提前放入哈希表中
        {
            Hash[prefix[i] % p] = i;//用于记录prefix[i]在哈希表中最后一次出现得到下标
            int left = ((prefix[i] - prefix[n]) % p + p) % p;//前面一定要先取一次模，因为前面一定会超出p
            if(Hash.find(left) != Hash.end())
                cnt = min(cnt, i - Hash[left]);
        }
        return cnt == 0x3f3f3f3f || cnt == n ? -1 : cnt;
    }
};
```

原题链接：LeetCode 面试题 17.05. 字母与数字

是字母还是数字并不重要，我们将字母设成 -1 ，数字设成 1 此时原数组便可以转化成一个**只含** $0,1$ **的整数数组**

我们需要求最长的子数组保证里面的数字与字母相同，等价于，在整数数组中求一段子数组使得 0 与 1 的个数相同，即这段区间和为 0

考虑用前缀和优化，这时问题转变成求两个数 $prefix[r],prefinx[l-1]$ 使得 $prefix[r]=prefix[l-1]$ 并且期望区间最大

子区间并不具有「二分性」，因此不能用双指针，考虑用哈希表

对于当前枚举到的数 $prefix[i]$ 而言

- 如果不在哈希表中，那么将其加入哈希表内
- 如果在哈希表中，则计算一遍区间大小，取最大的区间赋值给 l,r

完整代码：

```
class Solution {
public:
    vector<string> findLongestSubarray(vector<string>& array)
    {
        int n = array.size();
        vector<int>num(n + 1, 0), prefix(n + 1, 0);
        for(int i = 0; i < n; i ++)
        {
            if(isalpha(array[i][0])) num[i] = -1;
            else num[i] = 1;
            prefix[i + 1] = prefix[i] + num[i];
        }
        unordered_map<int, int>Hash;
        int r = 0, l = 0;
        for(int i = 0; i <= n; i++)
```

```
        {
            auto it = Hash.find(prefix[i]);
            if(it == Hash.end())//第一次遇到，则对哈希表赋值
                Hash[prefix[i]] = i;
            else if(i - it->second > r - l)//第二次遇到，对区间赋值
                r = i, l = it->second;
        }
        return {array.begin() + l, array.begin() + r};
    }
};
```

哈希表记录已出现元素的几种模板

设当前元素为 x

- 利用哈希表查找之前出现的不同于 x 的元素（设为 y）

先将元素放入哈希表中，随后再考虑 y 是否在哈希表中出现过，如果在哈希表中则更新答案

典型例题：

LeetCode 1590. 使数组和能被 P 整除

```
unordered_map<int, int>Hash;
for(int i = 0; i <= n; i ++)
{
    Hash[prefix[i] % p] = i;
    int left = ((prefix[i] - prefix[n]) % p + p) % p;
    if(Hash.find(left) != Hash.end())
        cnt = min(cnt, i - Hash[left]);
}
```

- 利用哈希表查找 x

先考虑 x 是否在哈希表中，如果不在将其放入，在的话则更新答案

典型例题：

LeetCode 面试题 17.05. 字母与数字

```
unordered_map<int, int>Hash;
int r = 0, l = 0;
for(int i = 0; i <= n; i++)
{
    auto it = Hash.find(prefix[i]);
    if(it == Hash.end())
        Hash[prefix[i]] = i;
    else if(i - it->second > r - l)
        r = i, l = it->second;
}
```

LeetCode 6317. 统计美丽子数组数目

这题思路跟 LeetCode 面试题 17.05. 字母与数字 一样的，在此不过多说明

都是先考虑当前值是否在哈希表中，然后再将这个值插入哈希表中

完整代码：

```
class Solution {
public:
    long long beautifulSubarrays(vector<int>& nums)
    {
        long long ans = 0;
        long long x = 0;
        unordered_map<int, int>hash{{0,1}};
        for(int i : nums)
        {
            x ^= i;//求前缀异或和
            ans += hash[x]++;//如果不在哈希表中，则为0，随后再将其插入哈希表中
        }
        return ans;
    }
};
```

何时用 DFS 与 BFS

如果从题目中我们分析出，**从一个状态可以单向转移到多个其他的状态**，并且在数据量不大的情况下，我们可以考虑**暴力枚举**

暴力枚举分为 BFS 和 DFS ，具体是使用需要依据题目来

需要注意的是，BFS 是可以求最短路径的（在路径权重均相同的情况下）

典型例题：

[LeetCode 797. 所有可能的路径]

这是一道非常简单的题目，每一个节点都可以走到其他的节点，即满足「从一个状态可以单向转移到其他状态」，因此考虑用暴力枚举

注意到这道题是非常经典的回溯问题，因此考虑 DFS

完整代码：

```
class Solution {
public:

    vector<vector<int>>ans;
    vector<int>path;

    void dfs(vector<vector<int>>& graph, int pos, int n)
    {
        if(pos == n)
        {
            ans.push_back(path);
            return;
        }
        for(int x : graph[pos])
        {
            path.push_back(x);
            dfs(graph, x, n);
            path.pop_back();
        }
    }

    vector<vector<int>> allPathsSourceTarget(vector<vector<int>>& graph)
    {
        path.push_back(0);
        dfs(graph, 0, graph.size() - 1);
        return ans;
    }
};
```

[LeetCode 1625. 执行操作后字典序最小的字符串]

我们首先去掉执行「任意」多次这个条件，因为这个条件会干扰我们分析问题

考虑对字符串执行一次操作，由于操作只有两个，因此**一个字符串可以生成两个不同的字符串**，即满足「从一个状态可以单向转移到其他多个状态」，可以考虑暴力枚举

思考到这里，我们发现，如果我们对初始字符串执行**任意**多次操作，本质上是生成了一个**有向图**，即对于任意一个**可达**的状态，我们都可以通过多次两种操作的组合生成

我们需要求的是在整个图中，字典序最小的字符串，直接遍历整个图即可，这里我们选择 BFS

每次从队头取出节点并将其弹出，然后求一次最小值，我们遍历该节点的所有出边（也就是两种可以生成的字符串）

为了避免遍历到重复节点，我们需要用一个哈希表来进行去重，如果当前节点**没有**在哈希表中出现过，我们将当前节点加入队列中

完整代码：

```
class Solution {
public:

    /*
        每个字符串都可以都可以通过这两个操作生成另外两个字符串
        将每个字符串抽象成一个点，每个点都会有两条出边，此时我们得到了一个有向图
        用BFS遍历所有点，找到值最小的即可
    */

    string findLexSmallestString(string s, int a, int b)
    {
        int n = s.length();
        queue<string>q;
        unordered_set<string>st;
```

```
        q.push(s);
        st.insert(s);
        string ans = s;
        while(q.size())
        {
            string t = q.front();
            q.pop();
            ans = min(ans, t);
            string s1 = t.substr(n - b) + t.substr(0, n - b);
            string s2 = t;
            for(int i = 1; i < n; i += 2)
                s2[i] = (s2[i] - '0' + a) % 10 + '0';
            for(string x : {s1, s2})
            {
                if(!st.count(x))
                {
                    st.insert(x);
                    q.push(x);
                }
            }
        }
        return ans;
    }
};
```

其他例题：

AcWing 187. 导弹防御系统

二分边界条件的特殊判断

我们首先给出二分的模板：

二分的使用首先需要保证整个区间具有「二分性」，即区间可以**恰好划分成**两个**没有交集**的区间，使得**前者满足性质 p 而后者不满足性质 p**

而二分就是为了去找那**两个端点**的

具体地，对于区间 $[1\cdots n]$ ，如果 $[1\cdots l]$ 满足性质 p ，$[l+1\cdots n]$ 不满足性质 p ，那么二分可以寻找分界点 l 与 $l+1$ ，分别对应两个不同的模板

寻找左端点的模板：

```
int l = 1, r = n;
while(l < r)
{
    int mid = l + r + 1;//l开头的话需要加1
    if(check(mid)) l = mid;
    else r = mid - 1;
}
if(check(l))//一定要去判断最后的结果是否满足条件
    ...
```

寻找右端点的模板：

```
int l = 1, r = n;
while(l < r)
{
    int mid = l + r >> 1;//r开头不需要加1
    if(check(mid)) r = mid;
    else l = mid + 1;
}
if(check(l))
    ...
```

需要注意的是，二分的最终出口一定是 $l=r$，因此 l 与 r 在此模板当中是等价的

有一些题目需要我们判断一下边界的情况，比如**如果所有元素都满足性质 p 或者都不满足性质 p** （对应的出口最终为 $l=1$ 或者 $l=n$），这时我们需要额外考虑

缺失的数（重点）

原题链接：LeetCode 1539. 第 k 个缺失的正整数

设 i 从 1 开始，对于元素 a_i 而言，$1\sim i$ 中**没有出现的元素个数**为 a_i-i，具体情况如下表：

i的取值	1	2	3	4	5	6

i的取值	1	2	3	4	5	6
`a[i]`	2	5	7	9	11	15
`a[i]-i`	1	3	4	5	6	9
缺失数	`1`	`1,3,4`	`1,3,4,6`	`1,3,4,6,8`	`1,3,4,6,8,10`	`1,3,4,6,8,10,12,13,14`

设 $p_i = a_i - i$ ，我们需要找到某个 i ，使得 $p_i < k \le p_i + 1$

也就是，我需要找出**严格小于** k 的数当中最大的那个

我们可以将性质 p 定义为：严格小于 k，随后二分即可，最终答案为：$k - p_i + a_i$

对于边界的判断：

如果数组 p_i 所有元素均满足性质 p ，也就是一定存在某个 p_i 使得 $p_i < k$ 成立，此种情况合法

如果数组 p_i 所有元素均不满足性质 p ，说明不存在某个 p_i 使得 $p_i < k$ 成立，此时我们需要直接返回 k

完整代码如下：

```
class Solution {
public:
    int findKthPositive(vector<int>& arr, int k)
    {
        int l = 1, r = arr.size();
        while(l < r)
        {
            int mid = l + r + 1 >> 1;
            if(arr[mid - 1] - mid < k) l = mid;
            else r = mid - 1;
        }
        if(arr[l - 1] - l < k) return l + k;
        else return k;
    }
};
```

前缀和的两种写法

前缀和一定是相比于原数组**右移一位的**

如果数组 $a[i]$ 的下标为 $0 \sim n-1$

那么 $a[i]$ 的前缀和数组 $s[i]$ 的下标为 $1 \sim n$ （下标为 0 的地方对应的值为 0）

$s[i]$ 的前缀和数组 $ss[i]$ 的下标为 $2 \sim n+1$ （下标为 $0,1$ 的地方均为 0）

往后以此类推

以元素序号为基准

对于数组 $a_1, a_2, a_3, \cdots, a_n$ ，我们定义数组 $s[i]$ 表示数组 $a[i]$ 的前缀和，有：

\begin{align*}
s[i]=\left(\begin{array}{l}
a[0], &i=0\
\sum_{j=1}^{i}a[j], &i\ge 1 \end{array} \right.
\end{align*}

因此对于区间 $[l,r]$ （l,r 均从 1 开始），其和为 $s[r]-s[l-1]$，即：

$$\sum_{i=l}^{r} a[i] = s[r] - s[l-1]$$

以元素下标为基准

对于数组 $a_0, a_1, a_2, \cdots, a_{n-1}$ ，我们定义数组 $s[i]$ 为其前缀和，有：

\begin{align*}
s[i]=\left(\begin{array}{l}
a[0],&i=0\
\sum_{j=0}^{i-1}a[j],&i\ge 1
\end{array}\right.
\end{align*}

因此对于区间 $[l,r]$ （l,r 均从 0 开始），其和为 $s[r+1]-s[l]$，即：

$$\sum_{i=l}^{r+1} a[i] = s[r+1] - s[l]$$

归纳：

- 两种前缀和的构造方面，完全是等价的，不同的点在于区间的意义上
- 如果区间表示的是**序号**（从 1 开始），那么选第一个
- 如果区间表示的是**下标**（从 0 开始），那么选第二个

单调栈 + 前缀和的前缀和

[LeetCode 2281. 巫师的总力量和]

看到「子数组」和「最弱」，其实应该马上想到有可能会用到「单调栈」了

对于某个元素 $w[i]$ 而言，我们需要找到左边第一个**严格小于**的元素 $w[L]$ 和右边**严格小于**的元素 $w[R]$ ，此时子数组 $[L+1, R-1]$ 就是我们待枚举区间

当然这么定义会有一个问题是，有可能会枚举到重复的子数组，例如：

$[1,3,1,2]$ ，对于 $w[0]$ 而言，$L=-1, R=n$；对于 $w[2]$ 而言，$L=-1, R=n$，因此区间 $[-1,n]$ 被我们枚举了两次

此时我们需要修改定义，我们找到左边第一个**严格小于**的元素 $w[L]$ 和右边第一个**大于等于**的元素 $w[R]$ ，此时便可保证不重不漏

对于**待枚举**区间 $[L,R]$（下标从 0 开始）而言，设 $L \le l \le i \le r \le R$ ，其中 l,r 均为变量，此时有：

$$\sum_{i=l}^{r} a[i] = s[r+1] - s[l]$$

由于 l,r 均为变量，有：

$$\sum_{r=i}^{R}\sum_{l=L}^{i} s[r+1]-s[l] = \sum_{r=i+1}^{R+1}\sum_{l=L}^{i}(s[r]-s[l]) \qquad = \left((i-L+1)\sum_{r=i+1}^{R+1} s[r]\right) - \left((R-i+1)\sum_{l=L}^{i} s[l]\right) \qquad = (i-L+1)(ss[R+2]-ss[i+1]) - (R-i+1)$$

如果待枚举区间 $[L,R]$（下标从 1 开始），则：

$$\sum_{r=i}^{R}\sum_{l=L}^{i} s[r]-s[l-1] = \sum_{r=i}^{R}\sum_{l=L-1}^{i-1} s[r]-s[l] \qquad = \left((i-L+1)\sum_{r=i}^{R} s[r]\right) - \left((R-i+1)\sum_{l=L-1}^{i-1} s[l]\right) \qquad = (i-L+1)(ss[R]-ss[i-1]) - (R-i+1)(ss[i$$

子数组下标从 0 开始：

```
class Solution {
public:
    const int mod = 1e9 + 7;
    int totalStrength(vector<int>& strength)
    {
        int n = strength.size();
        vector<int> left(n, -1);//左侧严格小于i的元素下标
        vector<int> right(n, n);//右侧小于等于i的元素下标
        vector<int> st;
        for(int i = 0; i < n; i ++)
        {
            while(!st.empty() && strength[st.back()] >= strength[i])
            {
                right[st.back()] = i;//对st.back而言刚好满足，当前元素小于等于它
                st.pop_back();
            }
            left[i] = st.empty() ? -1 : st.back();
            st.push_back(i);
        }
        vector<long long> ss(n + 2, 0);
        long long s = 0;
        for(int i = 1; i <= n; i ++)
        {
            s += strength[i - 1];
            ss[i + 1] = (ss[i] + s) % mod;
        }
        int cnt = 0;
        for(int i = 0; i < n; i ++)
        {
            int L = left[i] + 1, R = right[i] - 1;//这里的L和R均为下标，并不是元素序号
            long long tot = ((i - L + 1) * (ss[R + 2] - ss[i + 1]) - (R - i + 1) * (ss[i + 1] - ss[L])) % mod;
            cnt = (cnt + tot * strength[i]) % mod;
        }
        return (cnt + mod) % mod;
    }
};
```

子数组下标从 1 开始：

```
class Solution {
public:

    const int mod = 1e9 + 7;

    int totalStrength(vector<int>& strength)
    {
        int n = strength.size();
        vector<int> left(n, 1);//左侧严格小于i的元素下标
        vector<int> right(n, n + 2);//右侧小于等于i的元素下标
        vector<int> st;
        for(int i = 1; i <= n; i ++)
        {
            while(!st.empty() && strength[st.back() - 1] >= strength[i - 1])
            {
                right[st.back() - 1] = i + 1;//对st.back而言刚好满足，当前元素小于等于它
                st.pop_back();
            }
            left[i - 1] = st.empty() ? 1 : st.back() + 1;
            st.push_back(i);
        }
        vector<long long> ss(n + 2, 0);
        long long s = 0;
        for(int i = 1; i <= n; i ++)
        {
            s += strength[i - 1];
            ss[i + 1] = (ss[i] + s) % mod;
        }
        int cnt = 0;
        for(int i = 2; i <= n + 1; i ++)
        {
            int L = left[i - 2] + 1, R = right[i - 2] - 1;//这里的L和R均为下标，并不是元素序号
            long long tot = ((i - L + 1) * (ss[R] - ss[i - 1]) - (R - i + 1) * (ss[i - 1] - ss[L - 2])) % mod;
            cnt = (cnt + tot * strength[i - 2]) % mod;
        }
        return (cnt + mod) % mod;
    }
};
```