

基础算法

- 基础算法
 - 双指针及其使用条件
 - 不能用双指针
 - 哈希表记录已出现元素的几种模板
 - 何时用 DFS 与 BFS

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

不能用双指针

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

$$prefix[n] - (prefix[r] - prefix[l - 1]) \equiv 0 \pmod{p}$$

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 $prefix[n] - (prefix[r] - prefix[l]) \equiv 0 \pmod{p}$

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：

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

因此，只需要用哈希表记录最近一次 $prefix[r] - prefix[n]$ 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 $prefix[r] - prefix[n]$ 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 $prefix[r], prefix[l-1]$ 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](#)