【第四十三课】傅里叶变换与信息隐写术（三）

2187. 完成旅途的最少时间

给你一个数组 `time` ，其中 `time[i]` 表示第 `i` 辆公交车完成 **一趟旅途** 所需要花费的时间。

每辆公交车可以 **连续** 完成多趟旅途，也就是说，一辆公交车当前旅途完成后，可以 **立马开始** 下一趟旅途。每辆公交车 **独立** 运行，也就是说可以同时有多辆公交车在运行且互不影响。

给你一个整数 `totalTrips` ，表示所有公交车 **总共** 需要完成的旅途数目。请你返回完成 **至少** `totalTrips` 趟旅途需要花费的 **最少** 时间。

示例 1：

```
输入：time = [1,2,3], totalTrips = 5
输出：3
解释：
- 时刻 t = 1 ，每辆公交车完成的旅途数分别为 [1,0,0] 。
  已完成的总旅途数为 1 + 0 + 0 = 1 。
- 时刻 t = 2 ，每辆公交车完成的旅途数分别为 [2,1,0] 。
  已完成的总旅途数为 2 + 1 + 0 = 3 。
- 时刻 t = 3 ，每辆公交车完成的旅途数分别为 [3,1,1] 。
  已完成的总旅途数为 3 + 1 + 1 = 5 。
所以总共完成至少 5 趟旅途的最少时间为 3 。
```

```cpp
class Solution {
public:
    long long check(long long t, vector<int>& time) {
        long long sum = 0;
        for (auto x : time) sum += t / x;
        return sum;
    }
    long long binary_search(vector<int>& time, long long l, long long r, long long totalTrips) {
        long long mid;
        while (l < r) {
            mid = (l + r) >> 1;
            if (check(mid, time) < totalTrips) l = mid + 1;
            else r = mid;
        }
        return l;
    }
    long long minimumTime(vector<int>& time, int totalTrips) {
        long long l = 0, r = 1LL * totalTrips * time[0];
        for (auto x : time) r = min(r, 1LL * totalTrips * x);
        return binary_search(time, l, r, totalTrips);
    }
};
```

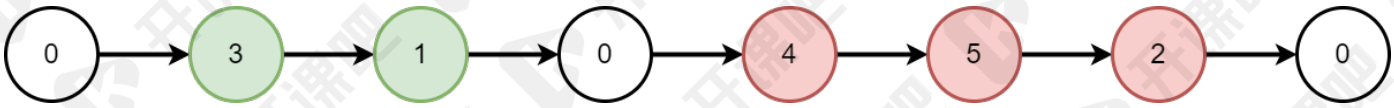
2181. 合并零之间的节点

给你一个链表的头节点 `head` ，该链表包含由 `0` 分隔开的一连串整数。链表的 **开端** 和 **末尾** 的节点都满足 `Node.val == 0` 。

对于每两个相邻的 `0` ，请你将它们之间的所有节点合并成一个节点，其值是所有已合并节点的值之和。然后将所有 `0` 移除，修改后的链表不应该含有任何 `0` 。

返回修改后链表的头节点 `head` 。

示例 1：




```
输入：head = [0,3,1,0,4,5,2,0]
输出：[4,11]
解释：
上图表示输入的链表。修改后的链表包含：
- 标记为绿色的节点之和：3 + 1 = 4
- 标记为红色的节点之和：4 + 5 + 2 = 11
```

```cpp
/**
 * 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 {
public:
    ListNode* mergeNodes(ListNode* head) {
        if (head->next == nullptr) return nullptr;
        ListNode *p = head;
        while (p->next->val != 0) {
            p->val += p->next->val;
            p->next = p->next->next;
        }
        p->next = mergeNodes(p->next);
        return head;
    }
};
```

2182. 构造限制重复的字符串

给你一个字符串 `s` 和一个整数 `repeatLimit` ，用 `s` 中的字符构造一个新字符串 `repeatLimitedString` ，使任何字母 **连续** 出现的次数都不超过 `repeatLimit` 次。你不必使用 `s` 中的全部字符。

返回 **字典序最大的** ** `repeatLimitedString` 。

如果在字符串 `a` 和 `b` 不同的第一个位置，字符串 `a` 中的字母在字母表中出现时间比字符串 `b` 对应的字母晚，则认为字符串 `a` 比字符串 `b` **字典序更大** 。如果字符串中前 `min(a.length, b.length)` 个字符都相同，那么较长的字符串字典序更大。

示例 1：

```
输入：s = "cczazcc", repeatLimit = 3
输出："zzcccac"
解释：使用 s 中的所有字符来构造 repeatLimitedString "zzcccac"。
字母 'a' 连续出现至多 1 次。
字母 'c' 连续出现至多 3 次。
字母 'z' 连续出现至多 2 次。
因此，没有字母连续出现超过 repeatLimit 次，字符串是一个有效的 repeatLimitedString 。
该字符串是字典序最大的 repeatLimitedString ，所以返回 "zzcccac" 。
注意，尽管 "zzcccca" 字典序更大，但字母 'c' 连续出现超过 3 次，所以它不是一个有效的 repeatLimitedString 。
```

```cpp
class Solution {
public:
    string repeatLimitedString(string s, int repeatLimit) {
        sort(s.begin(), s.end(), greater<char>());
        string temp, ans = "";
        int i = 0, j;
        while (i < s.size()) {
            temp = "";
            temp += s[i], i += 1;
            while (s[i] == temp[0]) temp += s[i], i += 1;
            j = 0;
            while (temp.size() - j > repeatLimit) {
                ans += temp.substr(j, repeatLimit);
                if (s[i] == 0) return ans;
                ans += s[i], i += 1;
                j += repeatLimit;
            }
            if (j < temp.size()) ans += temp.substr(j, repeatLimit);
        }
        return ans;
    }
};
```

2178. 拆分成最多数目的正偶数之和

给你一个整数 `finalSum` 。请你将它拆分成若干个 **互不相同** 的正偶数之和，且拆分出来的正偶数数目 **最多** 。

- 比方说，给你 `finalSum = 12` ，那么这些拆分是 **符合要求** 的（互不相同的正偶数且和为 `finalSum`）：`(2 + 10)` ，`(2 + 4 + 6)` 和 `(4 + 8)` 。它们中，`(2 + 4 + 6)` 包含最多数目的整数。注意 `finalSum` 不能拆分成 `(2 + 2 + 4 + 4)` ，因为拆分出来的整数必须互不相同。

请你返回一个整数数组，表示将整数拆分成 **最多** 数目的正偶数数组。如果没有办法将 `finalSum` 进行拆分，请你返回一个 **空** 数组。你可以按 **任意** 顺序返回这些整数。

示例 1：

```
输入：finalSum = 12
输出：[2,4,6]
解释：以下是一些符合要求的拆分：(2 + 10)，(2 + 4 + 6)和(4 + 8) 。
(2 + 4 + 6) 为最多数目的整数，数目为 3 ，所以我们返回 [2,4,6] 。
[2,6,4] ，[6,2,4] 等等也都是可行的解。
```

```cpp
class Solution {
public:
    vector<long long> maximumEvenSplit(long long finalSum) {
        vector<long long> ret;
        if (finalSum % 2 == 1) return ret;
        for (int i = 2; i <= finalSum; i += 2) {
            ret.push_back(i);
            finalSum -= i;
        }
        ret[ret.size() - 1] += finalSum;
        return ret;
    }
};
```

2170. 使数组变成交替数组的最少操作数

给你一个下标从 **0** 开始的数组 `nums` ，该数组由 `n` 个正整数组成。

如果满足下述条件，则数组 `nums` 是一个 **交替数组** ：

- `nums[i - 2] == nums[i]` ，其中 `2 <= i <= n - 1` 。
- `nums[i - 1] != nums[i]` ，其中 `1 <= i <= n - 1` 。

在一步 **操作** 中，你可以选择下标 `i` 并将 `nums[i]` **更改** 为 **任一** 正整数。

返回使数组变成交替数组的 **最少操作数** 。

示例 1：

```
输入：nums = [3,1,3,2,4,3]
输出：3
解释：
使数组变成交替数组的方法之一是将该数组转换为 [3,1,3,1,3,1] 。
在这种情况下，操作数为 3 。
可以证明，操作数少于 3 的情况下，无法使数组变成交替数组。
```

```cpp
class Solution {
public:
    typedef pair<int, int> PII;
    void getMaxNum(vector<int>& nums, int p, PII &x1, PII &x2) {
        x1.first = x1.second = x2.first = x2.second = 0;
        unordered_map<int, int> cnt;
        for (int i = p, n = nums.size(); i < n; i += 2) cnt[nums[i]] += 1;
        for (auto z : cnt) {
            if (z.second > x1.second) x2 = x1, x1 = z;
            else if (z.second > x2.second) x2 = z;
        }
        return ;
    }
    int minimumOperations(vector<int>& nums) {
        if (nums.size() == 1) return 0;
        PII x1, x2, y1, y2;
        getMaxNum(nums, 0, x1, x2);
        getMaxNum(nums, 1, y1, y2);
        int n = nums.size(), n0 = (n + 1) / 2, n1 = n - n0;
        if (x1.first != y1.first) return (n0 - x1.second) + (n1 - y1.second);
        return min(
            (n0 - x1.second) + (n1 - y2.second),
            (n0 - x2.second) + (n1 - y1.second)
        );
    }
};
```

2179. 统计数组中好三元组数目

给你两个下标从 **0** 开始且长度为 `n` 的整数数组 `nums1` 和 `nums2` ，两者都是 `[0, 1, ..., n - 1]` 的 **排列** 。

好三元组 指的是 `3` 个 **互不相同** 的值，且它们在数组 `nums1` 和 `nums2` 中出现顺序保持一致。换句话说，如果我们将 `pos1v` 记为值 `v` 在 `nums1` 中出现的位置，`pos2v` 为值 `v` 在 `nums2` 中的位置，那么一个好三元组定义为 `0 <= x, y, z <= n - 1` ，且 `pos1x < pos1y < pos1z` 和 `pos2x < pos2y < pos2z` 都成立的 `(x, y, z)` 。

请你返回好三元组的 **总数目** 。

示例 1：

输入：nums1 = [2,0,1,3], nums2 = [0,1,2,3]
输出：1
解释：
总共有 4 个三元组 (x,y,z) 满足 pos1x < pos1y < pos1z，分别是 (2,0,1) ，(2,0,3) ，(2,1,3) 和 (0,1,3) 。
这些三元组中，只有 (0,1,3) 满足 pos2x < pos2y < pos2z 。所以只有 1 个好三元组。

```cpp
class FenwickTree {
public :
    FenwickTree(int n) : n(n), c(n + 1) {};
    #define lowbit(x) (x & (-x))
    void add(int i, int x) {
        while (i <= n) c[i] += x, i += lowbit(i);
        return ;
    }
    int query(int i) {
        int sum = 0;
        while (i) sum += c[i], i -= lowbit(i);
        return sum;
    }

private:
    int n;
    vector<int> c;
};
class Solution {
public:
    long long goodTriplets(vector<int>& nums1, vector<int>& nums2) {
        long long n = nums1.size(), ans = 0;
        FenwickTree ind(n);
        unordered_map<int, int> ind_map;
        for (int i = 0; i < n; i++) ind_map[nums2[i]] = i;
        for (int i = 0; i < n; i++) {
            int x = nums1[i], j = ind_map[x];
            int cnti = ind.query(j + 1);
            int green = i - cnti;
            int cntj = n - j - 1 - green;
            ans += 1LL * cnti * cntj;
            ind.add(j + 1, 1);
        }
        return ans;
    }
};
```

2165. 重排数字的最小值

给你一个整数 `num` 。**重排** `num` 中的各位数字，使其值 **最小化** 且不含 **任何** 前导零。

返回不含前导零且值最小的重排数字。

注意，重排各位数字后，`num` 的符号不会改变。

示例 1：

```
输入：num = 310
输出：103
解释：310 中各位数字的可行排列有：013、031、103、130、301、310 。
不含任何前导零且值最小的重排数字是 103 。
```

```cpp
class Solution {
public:
    long long smallestNumber(long long num) {
        if (num == 0) return 0;
        int flag = 1, cnt[10] = {0};
        if (num < 0) flag = -1, num = -num;
        while (num) {
            cnt[num % 10] += 1;
            num /= 10;
        }
        long long ans = 0;
        if (flag == 1) {
            for (int i = 1; i < 10; i++) {
                if (cnt[i] == 0) continue;
                ans = i;
                cnt[i] -= 1;
                break;
            }
        }
        for (int i = (flag == 1 ? 0 : 9), I = 9 - i; i * flag <= I; i += flag) {
            for (int j = 0; j < cnt[i]; j++) {
                ans = ans * 10 + i;
            }
        }
        return ans * flag;
    }
};
```

1744. 你能在你最喜欢的那天吃到你最喜欢的糖果吗?

给你一个下标从 **0** 开始的正整数数组 `candiesCount` ，其中 `candiesCount[i]` 表示你拥有的第 `i` 类糖果的数目。同时给你一个二维数组 `queries` ，其中 `queries[i] = [favoriteTypei, favoriteDayi, dailyCapi]` 。

你按照如下规则进行一场游戏：

- 你从第 `0` 天开始吃糖果。
- 你在吃完 **所有** 第 `i - 1` 类糖果之前，**不能** 吃任何一颗第 `i` 类糖果。
- 在吃完所有糖果之前，你必须每天 **至少** 吃 **一颗** 糖果。

请你构建一个布尔型数组 `answer` ，用以给出 `queries` 中每一项的对应答案。此数组满足：

- `answer.length == queries.length` 。`answer[i]` 是 `queries[i]` 的答案。
- `answer[i]` 为 `true` 的条件是：在每天吃 **不超过** `dailyCapi` 颗糖果的前提下，你可以在第 `favoriteDayi` 天吃到第 `favoriteTypei` 类糖果；否则 `answer[i]` 为 `false` 。

注意，只要满足上面 3 条规则中的第二条规则，你就可以在同一天吃不同类型的糖果。

请你返回得到的数组 ** `answer` 。

```cpp
class Solution {
public:
    vector<bool> canEat(vector<int>& candiesCount, vector<vector<int>>& queries) {
        int n = candiesCount.size(), m = queries.size();
        vector<long long> sum(n); sum[0] = candiesCount[0];
        for (int i = 1; i < n; i++) sum[i] = sum[i - 1] + candiesCount[i];
        vector<bool> ans(m);
        for (int i = 0; i < m; i++) {
            int t = queries[i][0], d = queries[i][1], cap = queries[i][2];
            long long totali = sum[t], totali1 = (t - 1 >= 0 ? sum[t - 1] : 0);
            if (totali < d + 1 || totali1 / cap - 1 >= d) ans[i] = false;
            else ans[i] = true;
        }
        return ans;
    }
};
```

1745. 回文串分割 IV

给你一个字符串 `s` ，如果可以将它分割成三个 **非空** 回文子字符串，那么返回 `true` ，否则返回 `false` 。

当一个字符串正着读和反着读是一模一样的，就称其为 **回文字符串** 。

示例 1：

```
输入：s = "abcbdd"
输出：true
解释："abcbdd" = "a" + "bcb" + "dd"，三个子字符串都是回文的。
```

```cpp
class Solution {
public:
    unordered_set<int> pos0, posn;
    void extract(string &s, int i, int j) {
        while (i >= 0 && s[i] == s[j]) {
            if (i == 0) pos0.insert(j);
            if (s[j + 1] == 0) posn.insert(i);
            --i, ++j;
        }
        return ;
    }
    bool check(string &s, int i, int j) {
        while (i >= 0 && s[i] == s[j]) {
            if (pos0.find(i - 1) != pos0.end()
            && posn.find(j + 1) != posn.end()) return true;
            --i, ++j;
        }
        return false;
    }
    bool checkPartitioning(string s) {
        pos0.clear(), posn.clear();
        for (int i = 0, n = s.size(); i < n; i++) {
            extract(s, i, i);
            extract(s, i, i + 1);
        }
        for (int i = 0, n = s.size(); i < n; i++) {
            if (check(s, i, i)) return true;
            if (check(s, i, i + 1)) return true;
        }
        return false;
    }
};
```