2025 CEIT 寒假集训热身赛

概述

题号	题目名称	难度	通过率	tag
A	伟大的一步	200	27/33	无
B	面试的第一关，居然是电梯?	800	18/79	模拟
C	接！化？发！	1300	5/82	构造，暴力枚举
D	“宝贝”代码优化	1200	4/20	字符串，贪心
E	调频	1400	4/17	贪心，枚举，位运算
F	排列游戏	1200	0/4	交互题，构造，模拟
G	猫方程	1400	0/3	字符串，大模拟
H	空间跃迁(EasyVersion)	1400	1/4	枚举
I	空间跃迁	1700	0/3	动态规划
J	调制器	1500	1/8	暴力枚举，模拟
K	时间环	1400	1/3	DFS / 并查集
L	时间线修复	1500	1/6	动态规划，组合数学
M	Orange实在太多了>_<	1400	2/6	前缀和，动态规划

题解报告

A. 伟大的一步

你不会没签上到罢?

```
print("I'm ready! Let's go!")
```

B. 面试的第一关，居然是电梯?

对于两点之间的移动，不难发现，我们可以单独考虑 x 和 y 轴的移动，对于单个数轴的移动，我们假设要从 a 移动到 b，我们可以直接移动 $|a-b|$ 步而直接到达，也可以利用地图的特性，先移动到地图一个边界，再穿越到另一个边界，再移动到 b，显然，这样的移动步数是 $n-|a-b|$，我们只需要比较这两种方案，取最小值即可。

假设地图大小为 $n \times m$，那么从点 $A(x,y)$ 移动到 $B(x',y')$ 的最小步数为：

$$\min(|x-x'|,n-|x-x'|)+\min(|y-y'|,m-|y-y'|)$$

那么三个点也是同理，先从A到B，再从B到C，最后从C到A，按照上述方法计算即可。

```
typedef pair<int, int> pii;

void solve()
{
    int n, m;
    cin >> n >> m;

    pii a, b, c;
    cin >> a.first >> a.second;
    cin >> b.first >> b.second;
    cin >> c.first >> c.second;

    long long ans = 0;
    // 从 a 到 b 的 x 轴
    ans += min(abs(a.first - b.first), n - abs(a.first - b.first));
    // 从 a 到 b 的 y 轴
    ans += min(abs(a.second - b.second), m - abs(a.second - b.second));
    // 从 b 到 c 的 x 轴
    ans += min(abs(b.first - c.first), n - abs(b.first - c.first));
    // 从 b 到 c 的 y 轴
    ans += min(abs(b.second - c.second), m - abs(b.second - c.second));

    cout << ans << "\n";
}
```

C. 接！化？发！

对于 b_i，我们应该满足 $a_i + b_i$ 是 i 的倍数。同时需要满足 b_i 大于 0，那么我们可以枚举 i 的所有倍数，直到找到第一个之前没有出现过的，且大于 a_i 的数，那么 b_i 就是这个数。每次找到 b_i，可以使用类似set之类的容器记录它已经出现过。

```
void solve()
{
    int n;
    cin >> n;
    vector<int> a(n + 1);
    for(int i = 1; i <= n; i ++) cin >> a[i];
    set<int> st;
    st.insert(0);
    for(int i = 1; i <= n; i ++)
    {
        int x = ((a[i] + i - 1) / i * i - a[i]);
        while(st.count(x)) x += i;
        cout << x << ' ';
        st.insert(x);
    }
    cout << '\n';
}
```

D. “宝贝”代码优化

为了让01串字典序最小，我们需要让 `0` 尽可能考前， `1` 尽可能往后，那么每次交换一定是**将最后面的0与最前面的1交换**，可以使用vector倒序记录所有0的位置，然后顺序扫描字符串，每次遇到 `1` 就与vector内的0交换，直到交换次数用完为止。需要注意的是，如果顺序扫描的指针越过了当前vector内0的位置，应该跳过，因为这样可能导致前面的0和后面的1交换。

```
void solve()
{
    int n, k;
    cin >> n >> k;
    string s;
    cin >> s;
    vector<int> pos;
    for(int i = 0; i < s.size(); i ++)
    {
        if(s[i] == '0') pos.push_back(i);
    }
    reverse(all(pos));
    int p = 0;
    for(int i : pos)
    {
        while(s[p] == '0') p++;
        if(i < p) continue; // 如果 p 越过了 i，那么跳过
        if(k)
        {
            swap(s[p++], s[i]);
            k--;
        }
    }
    cout << s << '\n';
}
```

E. 调频

不难发现，位与运算的结果具有单向性，也就是说，假设 a and $b = c$，那么 a and $c = c$，b and $c = c$，也就是说，c 一定就是最终的结果，那么可以得到结论，每个数最多只需要进行一次位与运算。

那么得出我们的做题策略，进行两次扫描：第一次扫描，我们可以先用一个标记数组记录每个数是否出现，然后依次遍历每个数，如果这个数没有出现过，就将他标记，如果出现过了，那么答案就直接为0，因为已经有两个相同的数了。对于第二次扫描，我们依次标记每个数与上 X，如果之前这个数在第一次扫描的标记数组中出现过，那么答案就是 1，如果在第二次扫描的标记数组中出现过，那么答案就是 2，否则继续扫描，如果没有触发上述三种情况，答案则为 `-1` 。

```
void solve()
{
    int n, x;
    cin >> n >> x;
    vector<int> a(n + 1), cnt(1e5 + 5, 0), cnt1(1e5 + 5, 0);
    for(int i = 1; i <= n; i++) cin >> a[i];
    for(int i = 1; i <= n; i ++)
    {
        if(cnt[a[i]])
        {
            cout << 0 << '\n';
            return;
        }
        cnt[a[i]]++;
    }
    bool flag = 0;
    int ans = 2;
    for(int i = 1; i <= n; i ++)
    {
        if(cnt1[a[i] & x])
        {
            flag = 1;
            ans = min(ans, 1 + (cnt[a[i] & x] == 0));
        }
        cnt1[a[i] & x] ++;
    }
    if(flag) cout << ans << '\n';
    else cout << -1 << '\n';
    return;
}
```

F. 排列游戏

应该注意到排列具有的性质： **每个数都不相同**

那么对于三个位置，我们一定可以确认其中的一个值，例如：我们假设存在三个数 x,y,z，记 Q_{xy} 为询问 x 和 y 的结果，Q_{xz} 为询问 x 和 z 的结果，Q_{yz} 为询问 y 和 z 的结果，那么我们可以得到以下结论：

- 如果 $Q_{xy}=Q_{xz}$ 那么 $x=Q_{xy}=Q_{xz}$
- 如果 $Q_{xy}=Q_{yz}$ 那么 $y=Q_{xy}=Q_{yz}$
- 如果 $Q_{xz}=Q_{yz}$ 那么 $z=Q_{xz}=Q_{yz}$

因为排列中不存在重复的数，所以三个数两两去最小值一定会有一个最小值重复2次，我们可以通过上述结论得到三个位置中至少一个的值。

可以用数组，栈，队列之类的数据结构维护一下顺序，我们每次拿出三个位置，能得到一个位置的值，那么直到最后剩下两个位置的时候，无法得到答案，而刚好符合题目要求的海明距离为2的限制。

> tips:交互题不要关闭同步流，输出后记得刷新缓冲区

```cpp
#include "bits/stdc++.h"
using namespace std;
int main()
{
    int n;
    cin >> n;
    auto ask = [&](int x, int y)
    {
        cout << "? " << x << " " << y << endl;
        int res;
        cin >> res;
        return res;
    };
    vector<int> stk(n + 1);
    int top = 0;
    for(int i = n; i >= 1; i --) stk[top++] = i;
    vector<int> ans(n + 1);
    while (top > 2)
    {
        int x = stk[--top];
        int y = stk[--top];
        int z = stk[--top];
        int xy = ask(x, y);
        int yz = ask(y, z);
        int xz = ask(x, z);
        if(xy == xz)
        {
            ans[x] = xy;
            stk[top++] = y;
            stk[top++] = z;
        }
        else if(xy == yz)
        {
            ans[y] = xy;
            stk[top++] = x;
            stk[top++] = z;
        }
        else
        {
            ans[z] = xz;
            stk[top++] = x;
            stk[top++] = y;
        }
    }
    cout << "! ";
    for(int i = 1; i <= n; i ++) cout << ans[i] << ' ';
    cout << endl;
}
```

G. 猫方程

实际上本题做法非常多，你可以直接枚举每个位置进行插入，然后插入一次之后计算一次答案，这样最多有 n 个位置插入，每次计算答案复杂度为 $O(n)$，而 n 最多只有2000，因此可以接受 $O(n^2)$ 的复杂度。

下面是给出的一种比较优秀的做法，代码较为复杂，仅作为参考。

```cpp
#include <bits/stdc++.h>
using namespace std;
#define IOS                         \
        ios_base::sync_with_stdio(0); \
        cin.tie(0);                   \
        cout.tie(0);
// #define int long long
#define endl '\n'
#define all(_x) _x.begin(), _x.end()
#define range(_x, _st, _ed) (_x.begin() + _st), (_x.begin() + _ed)
#define rep(_x, _y, _z) for (int _x = _y; _x <= _z; _x++)
#define matrix(_x, _y, _z) vector<vector<int>>(_x, vector<int>(_y, _z))
#define debug(_x) cout << #_x << '=' << _x << endl
using i64 = long long;
using i128 = __int128;
using pii = pair<int, int>;
using mat = vector<vector<int>>;
using u64 = unsigned long long;
constexpr int N = 2e5 + 10;
// dont use umap!!!

int calc(vector<vector<int>> poly)
{
        int res = 0;
        for (auto i : poly)
        {
                int t = 1;
                for (int j : i)
                {
                        t *= j;
                }
                res += t;
        }
        return res;
}

int calc(vector<int> poly)
{
        int t = 1;
        for(int i : poly)
        {
                t *= i;
        }
        return t;
}

vector<vector<int>> get(string s)
{
        vector<vector<int>> res;
        vector<int> a;
        int sum = 0;
        int flag = 0;
        for (int i = 0; i < s.size(); i++)
        {
                if (isdigit(s[i]))
                {
                        flag = 1;
                        sum = sum * 10 + (s[i] - '0');
                }
                else
                {
                        if (s[i] == '+')
                        {
                                if (flag)
                                        a.push_back(sum);
                                sum = 0;
                                flag = 0;
                                if (a.size())
```

```
                                res.push_back(a);
                        a.clear();
                }
                else
                {
                        a.push_back(sum);
                        sum = 0;
                        flag = 0;
                }
        }
    }
    if (flag)
        a.push_back(sum);
    if (a.size() > 0)
        res.push_back(a);
    return res;
}

void solve()
{
    string s;
    cin >> s;
   // cout << s.size() << '\n';
    vector<vector<int>> Lpoly, Rpoly;
    int eqpos = s.find('=');
    string Ls = s.substr(0, eqpos), Rs = s.substr(eqpos + 1);
    Lpoly = get(Ls), Rpoly = get(Rs);
    int Lres = calc(Lpoly), Rres = calc(Rpoly);
    if (Lres == Rres)
    {
        cout << "Yes\n";
        return;
    }
    if(Lres < Rres)
    {
        for(int i = 0; i < Lpoly.size(); i ++)
        {
            int tres = calc(Lpoly[i]);
            // 枚举项
            for(int j = 0; j < Lpoly[i].size(); j ++)
            {
                auto bak = Lpoly[i][j];
                string ts = to_string(Lpoly[i][j]);
                auto baks = ts;
                for(int k = 0; k < ts.size(); k ++) // 枚举插入位置
                {
                    for(int l = (k != 0 ? '0' : '1'); l <= '9'; l ++) // 枚举插入数字
                    {
                        ts.insert(k, 1, l);
                        Lpoly[i][j] = stoi(ts);
                        int ttres = calc(Lpoly[i]);
                        if(ttres - tres == Rres - Lres)
                        {
                            cout << "Yes" << '\n';
                            return;
                        }
                        Lpoly[i][j] = bak;
                        ts = baks;
                    }
                }
                // 在末尾插入
                for(int l = '0'; l <= '9'; l ++)
                {
                    ts.push_back(l);
                    Lpoly[i][j] = stoi(ts);
                    int ttres = calc(Lpoly[i]);
                    if(ttres - tres == Rres - Lres)
                    {
                        cout << "Yes" << '\n';
```

```
                                        return;
                                }
                                Lpoly[i][j] = bak;
                                ts = baks;
                        }
                }
        }
        cout << "No\n";
    }
    else
    {
        for(int i = 0; i < Rpoly.size(); i ++)
        {
            int tres = calc(Rpoly[i]);
            for(int j = 0; j < Rpoly[i].size(); j ++)
            {
                auto bak = Rpoly[i][j];
                string ts = to_string(Rpoly[i][j]);
                auto baks = ts;
                for(int k = 0; k < ts.size(); k ++) // 枚举插入位置
                {
                    for(int l = (k != 0 ? '0' : '1'); l <= '9'; l ++) // 枚举插入数字
                    {
                        ts.insert(k, 1, l);
                        Rpoly[i][j] = stoi(ts);
                        int ttres = calc(Rpoly[i]);
                        if(ttres - tres == Lres - Rres)
                        {
                            cout << "Yes" << '\n';
                            return;
                        }
                        Rpoly[i][j] = bak;
                        ts = baks;
                    }
                }
                // 在末尾插入
                for(int l = '0'; l <= '9'; l ++)
                {
                    ts.push_back(l);
                    Rpoly[i][j] = stoi(ts);
                    int ttres = calc(Rpoly[i]);
                    if(ttres - tres == Lres - Rres)
                    {
                        cout << "Yes" << '\n';
                        return;
                    }
                    Rpoly[i][j] = bak;
                    ts = baks;
                }
            }
        }
        cout << "No\n";
    }
}

signed main()
{
    IOS;
    int _ = 1;
    cin >> _;

    while (_--)
        solve();

    return 0;
}
```

H. 空间跃迁EasyVersion

注意到 $k = 5$，而且题目要求了必须在 1 和 n 这两点设置加速站，实际上我们可以移动的加速站就只有3个了，但是如果暴力枚举三个加速站的位置，复杂度是 $O(n^3)$ 的，显然会超时。

我们可以考虑如下策略，枚举中间的加速站，然后以此为中心，向两边枚举加速站，这样复杂度就变成了 $O(n^2)$，可以通过。

```
void solve()
{
        int n, k;
        cin >> n >> k;
        vector<double> pos(n + 1), a(n + 1);
        auto get = [&](double x, double sum, double dist)
        {
                return (long double)x * sum / dist;
        };
        for (int i = 1; i <= n; i++)
                cin >> pos[i];
        for (int i = 1; i <= n; i++)
                cin >> a[i];
        double ans = 0;
        for(int i = 3; i <= n - 2; i ++)
        {
                double t1 = 0;
                for(int j = 2; j < i; j ++)
                {
                        double w = (a[j] + a[1]) * (a[i] + a[j]) / ((pos[j] - pos[1]) * (pos[i] - pos[j]));
                        t1 = max(t1, w);
                }
                for(int j = i + 1; j < n; j ++)
                {
                        double w = t1 * (a[j] + a[i]) * (a[n] + a[j]) / ((pos[j] - pos[i]) * (pos[n] - pos[j]));
                        ans = max(ans, w);
                }
        }
        cout << fixed << setprecision(10) << ans << '\n';
}
```

I. 空间跃迁

本题解除了上一题 $k = 5$ 的限制，我们可以考虑动态规划，假设 $dp[i][j]$ 表示前 i 个点，已经选了 j 个加速站且最后一个加速站在 pos_i 的最大速度，转移方程为：

$$dp[i][j] = \max_{k=1}^{i-1} dp[k][j-1] \times \frac{a[j] + a[i]}{pos[j] - pos[i]}$$

同时，注意一下，最后一个加速站由于必须被放到点 n，因此我们还需要枚举一下第 $n-1$ 个加速站的位置，计算出最后的答案。

复杂度是 $O(n^2k)$ 的，可以通过。

```
#define double long double
void solve()
{
        int n, m;
        cin >> n >> m;
        vector<vector<double>> dp(n + 1, vector<double>(m, -1e9));
        vector<int> pos(n + 1), a(n + 1);
        auto get = [&](double x, double sum, double dist)
        {
                return x * sum / dist;
        };
        for(int i = 1; i <= n; i ++) cin >> pos[i];
        for(int i = 1; i <= n; i ++) cin >> a[i];
        dp[1][1] = 1;
        for(int i = 2; i <= n; i ++)
        {
                for(int j = i - 1; j >= 1; j --)
                {
                        for(int k = 2; k < m; k ++)
                        {
                                dp[i][k] = max(dp[i][k], get(dp[j][k - 1], a[i] + a[j], pos[i] - pos[j]));
                        }
                }
        }
        double ans = 0;
        for(int i = m - 1; i < n; i ++)
        {
                ans = max(ans, get(dp[i][m - 1], a[i] + a[n], pos[n] - pos[i]));
        }
        cout << fixed << setprecision(10) << ans << '\n';
        return;
}
```

J. 调制器

一种错误的做法是，无脑将 X 扩大 2 倍。比如存在一个干扰值为 9，此时 $X=4$，那么最优的方法是将 9 变成 $\lfloor \frac{9}{2} \rfloor = 4$。

那么可不可以特判一下最大值呢？显然不可以，因为可能有多个最大值，以及次大值和 X 也存在上述交叉关系，因此我们无法特判。

综上所述：本题不存在复杂度为 $O(\log n)$ 的做法。

那么本题应该如何做呢？可以考虑暴力枚举，我们每次枚举对 X 的操作次数，每次再分别计算出将其他所有的干扰值降低到小于等于 X 需要的操作次数，这样就是本次的答案，然后我们取出最小的答案即可。

需要注意的是，X 可能为 0，此时需要特判一下，答案为所有数除的log上取整之和。

复杂度为 $O(n \log^2 n)$，可以通过。

```
void solve()
{
    int n;
    i64 X;
    cin >> n >> X;
    // cin >> n;
    vector<int> a(n + 1);
    int mx = -1;
    for(int i = 1; i <= n; i ++)
    {
        cin >> a[i];
        mx = max(a[i], mx);
    }
    auto check = [&](int u)
    {
        int res = 0;
        while(u > X) u /= 2, res++;
        return res;
    };
    int ans = 1e9;
    for(int cnt = 0;; X *= 2, cnt ++)
    {
        int t = 0;
        for(int i = 1; i <= n; i ++)
        {
            t += check(a[i]);
        }
        ans = min(ans, cnt + t);
        if(X > mx or X == 0) break;
    }
    cout << ans << '\n';
}
```

K. 时间环

我们发现，i 点可以到达 a_i 点，将图建出来之后，会发现，图一定形如若干个环，而我们每次的交换操作实际上可以将两个环合并成一个环，而题目最后的要求就是将所有的环合并成一个环。因此答案为环的个数 -1。环可以用dfs也可以用并查集维护。

```
void solve()
{
    int n;
    cin >> n;
    vector<int> a(n + 1), vis(n + 1);
    for(int i = 1; i <= n; i ++) cin >> a[i];
    auto dfs = [&](auto self,int u) -> void
    {
        vis[u] = 1;
        if(vis[a[u]]) return;
        self(self, a[u]);
    };
    int ans = 0;
    for(int i = 1; i <= n; i ++)
    {
        if(!vis[i])
        {
            ans ++;
            dfs(dfs, i);
        }
    }
    cout << ans - 1 << '\n';
}
```

L. 时间线修复

题意实际上是要求出原序列中有多少个子序列是排列。

不难发现排列的数量存在如下递推关系：

我们设 $dp[i]$ 表示子序列为 i 的排列的数量，n_i 表示序列中 i 的数量，则：

$$dp[i] = dp[i-1] \times \binom{n_i}{1}$$

而我们的答案就是所有排列数量之和：

$$ans = \sum_{i=1}^{n} dp[i]$$

```
void solve()
{
    int n;
    cin >> n;
    map<int, int> mp;
    vector<int> a(n + 1), dp(n + 1);
    for(int i = 1; i <= n; i ++)
    {
        cin >> a[i];
        mp[a[i]]++;
    }
    i64 ans = 0;
    for(int i = 1; i <= n; i ++)
    {
        if(i == 1)
            dp[i] = mp[1];
        else
            dp[i] = (i64)dp[i-1] * mp[i] % mod;
        ans += dp[i];
        ans %= mod;
    }
    cout << ans << '\n';
}
```

M. Orange实在太多了

我们假设 $f[i][1]$ 表示的是字母 `o` 的前缀和，$f[i][2]$ 表示的是子序列 `or` 的前缀和，$f[i][3]$ 表示的是子序列 `ora` 的前缀和，$f[i][4]$ 表示的是子序列 `oran` 的前缀和，$f[i][5]$ 表示的是子序列 `orang` 的前缀和，$f[i][6]$ 表示的是子序列 `orange` 的前缀和。

那么我们可以得到如下的递推关系：

$$f[i][1] = f[i-1][1] + [s_i =' o']$$

$$f[i][2] = f[i-1][2] + f[i-1][1] \times [s_i =' r']$$

$$f[i][3] = f[i-1][3] + f[i-1][2] \times [s_i =' a']$$

$$f[i][4] = f[i-1][4] + f[i-1][3] \times [s_i =' n']$$

$$f[i][5] = f[i-1][5] + f[i-1][4] \times [s_i =' g']$$

$$f[i][6] = f[i-1][6] + f[i-1][5] \times [s_i =' e']$$

最终答案就是 $f[n][6]$。

```
constexpr int mod = 1e9+7;
void solve()
{
    string s;
    cin >> s;
    vector<vector<i64>> pre(s.size(), vector<i64>(6));
    string charset = "orange";
    for(int i = 0; i < charset.size(); i ++)
    {
        for(int j = 0; j < s.size(); j ++)
        {
            if(j)
                        {
                pre[j][i] += pre[j - 1][i];
                                pre[j][i] %= mod;
                        }
            if(s[j] == charset[i])
            {
                if(i == 0)
                {
                    pre[j][i] += 1;
                                        pre[j][i] %= mod;
                }
                else
                {
                    pre[j][i] += pre[j][i - 1];
                                        pre[j][i] %= mod;
                }
            }
        }
    }
    cout << pre[s.size() - 1][charset.size() - 1] << '\n';
}
```