T1 对联

模拟，尽量将前面的数字对应成小的数字。即假设当前数字没有对应的数字，那么就从小到大寻找第一个未出现过的数字，对应到当前数字上。（可以用一个 2e6 大小的 bool 数组来表示每一个数字是否出现过，大于 2e6 的数字无需记录是否出现过，因为不可能成为对应的数字）

T2 燃料分配

推一下公式。

考虑贪心，我们应该让尽可能多地机器进行工作，如果把燃料全部投入到少数几个机器上，那么很容易到达上限。而燃料越分散，则越不容易达到上限。

```
#include <cstdio>
#include <algorithm>
#define ll long long
using namespace std;

ll n, m, k, q, t, f1, n1, f2, n2, ans;
int main() {
    scanf("%lld%lld%lld%lld%lld", &n, &m, &k, &q, &t);

    if (m < k) {
        printf("0");
        return 0;
    }

    if (n * k > m) n = m / k;//如果有不能启动的，那么只启动可以启动的
    m -= n * k;//减去启动消耗
    if (m % n) {//如果不能平分剩下的
        f2 = k + m / n + 1;//f2分多1个
        n2 = m % n;//人数
    }
    f1 = k + m / n;
    n1 = n - n2;

    ans += min(f1 * t, q) * n1;
    ans += min(f2 * t, q) * n2;

    printf("%lld", ans);
    return 0;
}
```

T3 学习绝对值

区间最大值和最小值的绝对值相等，有以下两种情况：

- 第一种情况：区间中只有一个数字
- 第二种情况：最大值和最小值互为相反数

第二种情况具体来说，就是最大值1，最小值-1，和最大值2，最小值-2。

考虑如何求解第二种情况。

考虑固定左端点时，如何找合法的右端点：

1. 对于最大值为1最小值-1的情况：右端点需要保证：区间中有1和-1，区间中没有2和-2。这可以通过维护左端点向后的1/-1/2/-2的初始位置找到。右端点的起始位置为max(下一个1的位置，下一个-1的位置)，结束位置为min(下一个2的位置，下一个-2的位置)
2. 对于最大值为2最小值-2的情况：右端点需要保证：区间中有2和-2。维护方法类似上面。

至此本题做完，复杂度为 O(n)。

```
#include <bits/stdc++.h>
using namespace std;
#define ll long long
const int maxn = 5e5 + 10;
```

```cpp
const int inf = 0x3f3f3f3f;
ll n, ans, num[maxn];
int nxt[maxn][5];
int startpos, endpos;
// -2 --> 0 // -1 --> 1
// 1 --> 3  // 2 --> 4
int main() {
    ios::sync_with_stdio(false);
    cin.tie(0); cout.tie(0);
    cin >> n;
    memset(nxt, 0x3f, sizeof nxt);
    for(int i = 1;i <= n; ++i) {
        cin >> num[i];
    }
    ll ret = 1, lst = num[1];
    for(int i = 2;i <= n; ++i) {
        if(num[i] == lst) ++ret;
        else {
            ans += ret * (ret + 1) / 2;
            ret = 1; lst = num[i];
        }
    }
    ans += ret * (ret + 1) / 2;
    for(int i = n;i >= 1; --i) {
        for(int j = 0;j <= 4; ++j) nxt[i][j] = nxt[i + 1][j];
        nxt[i][num[i] + 2] = i;
        int maxpos1 = nxt[i][1 + 2], maxpos2 = nxt[i][2 + 2];
        int minpos1 = nxt[i][-1 + 2], minpos2 = nxt[i][-2 + 2];
        startpos = max(maxpos2, minpos2); endpos = n + 1;
        if(startpos != inf && startpos < endpos) ans += endpos - startpos;
        startpos = max(maxpos1, minpos1); endpos = min(min(maxpos2, minpos2), (int
        if(startpos != inf && startpos < endpos) ans += endpos - startpos;
    }
    cout << ans << '\n';
    return 0;
}
```

T4 方差

考虑换根 dp

首先考虑方差的计算式，假设选择点 为根，各点距离为 ，则此时有

说明我们需要维护各距离的平方和，以及各距离的和。假设 1 为整棵树的根，考虑怎么向上计算出结果。

令 sum1[u] 表示以 为根的子树上各点到 的距离和，sum2[u] 表示以 为根的子树上各点到 的距离平方和。假设已有儿子 的答案，考虑如何转移到父亲 上：显然，儿子上所有店到父亲的距离会在原来的基础上加一。因此发现需要维护子树上的点数量，记为 sz[u]

此时可以写出转移式：

这样我们就可以算出以 1 为根时整棵树的方差，这也是第一次 dfs 做的事。

接下来考虑如果换成其它点作为根，答案会有怎么样的变化。

某一个节点为根时，方差的答案有两个来源：一个是该结点所在的子树的贡献（这里的子树指的是假设 1 为根时的子树），还有一个来源是当前点上方的祖先结点对的贡献。（祖先也是指 1 为根时这个点的祖先）

前者我们已经在第一次 dfs 中求得。我们来考虑第二个问题如何计算。

首先，将子树的影响从父节点中去掉：

v 这个点的子树对父亲结点的贡献为：

i. 对 sz 的贡献：sz[v]

ii. 对 sum1 的贡献：sum1[v]+sz[v]

iii. 对 sum2 的贡献：sum2[v] + 2 *sum1[v]+sz[v]

不妨将它们记为 szu,ret1,ret2

从根向下传时，s1 = ret + szu s2=ret2 + 2 *ret1 + szu

然后从根出发进行 dfs，向下的过程中记录维护 s1 s2 的变化，来获得之后的点作为根时整棵树的方差。

```
#include <bits/stdc++.h>
using namespace std;
#define ll unsigned long long
const int maxn = 1e5 + 10;
ll sum2[maxn], sum1[maxn], sz[maxn], n, res;
vector<int> G[maxn];

void dfs1(int u, int f) {
    for(auto v : G[u]) {
        if(v == f) continue;
        dfs1(v, u);
        sz[u] += sz[v];
        sum1[u] += sum1[v];
        sum2[u] += sum2[v];
    }
    sum2[u] += sz[u] + 2 * sum1[u];
    sum1[u] += sz[u];
    sz[u] += 1;
    return;
}

void dfs2(int u, int f, ll s1, ll s2) {
    res = min(res, n * (s2 + sum2[u]) - (sum1[u] + s1) * (sum1[u] + s1));
    for(auto v : G[u]) {
        if(v == f) continue;
        ll ret1 = sum1[u] - (sum1[v] + sz[v]) + s1;
        ll ret2 = sum2[u] - (sum2[v] + 2 * sum1[v] + sz[v]) + s2;
        ll szu = n - sz[v];
        dfs2(v, u, ret1 + szu, ret2 + 2 * ret1 + szu);
```