



GESP

信息学竞赛

GESP C++ 六级认证 (四)

综合代码训练



相关题目

- B3874 [GESP202309 六级] 小杨的握手问题
- P11247 [GESP202409 六级] 算法学习
- P11375 [GESP202412 六级] 树上游走
- P11376 [GESP202412 六级] 运送物资
- P11962 [GESP202503 六级] 树上漫步
- P13016 [GESP202506 六级] 最大因数



GESP

信息学竞赛

GESP C++ 六级认证 (四)

综合代码训练



B3874 [GESP202309 六级] 小杨的握手问题

I 题意

有 N 名同学，学号从 0 到 $N - 1$ ，按照给定顺序依次进入教室。每位同学进入教室时，需要和已经在教室内且学号小于自己的同学握手。求总共会进行多少次握手。

数据范围： $2 \leq N \leq 3 \times 10^5$ 。



分析

- 问题转化：**对于进入顺序序列中的每个位置 i ，我们需要统计在它之前进入教室且学号小于当前学号的同学数量，然后将所有统计结果求和。
- 关键观察：**这实际上是一个**逆序对计数问题**的变形。在归并排序过程中，当合并两个有序子序列时，可以高效地统计逆序对数量。



1. 算法思路：

- 使用归并排序算法对进入顺序序列进行排序
- 在合并两个有序子序列时，如果左半部分的当前元素大于右半部分的当前元素，说明存在逆序对
- 统计所有逆序对的数量，即为总握手次数



2. 正确性证明：

- 对于任意两个同学 (i, j) , 如果 i 在 j 之前进入教室且学号 $a_i > a_j$, 则 j 进入时不会与 i 握手 (因为学号 $a_j < a_i$)
- 因此, 握手次数 = 所有可能的对数 - 逆序对数量
 - 这里可以优化成, 从大到小排序, 统计逆序对数量即可。
- 所有可能的对数为 $\frac{N(N-1)}{2}$, 逆序对数量可以在归并排序中统计

时间复杂度: $O(N \log N)$, 在 $N \leq 3 \times 10^5$ 时可行。



参考代码

```
#define int long long
const int N = 3e5 + 10;
int n, a[N], b[N], ans;
void msort(int l, int r) {
    if (l >= r)
        return;
    int mid = (l + r) / 2;
    msort(l, mid), msort(mid + 1, r);
    int i = l, j = mid, st = mid + 1, ed = r, len = l;
    while (i <= j && st <= ed) {
        if (a[i] >= a[st])
            b[len++] = a[i++];
        else {
            b[len++] = a[st++];
            // 合并序列过程，存在逆序对，贡献即为 [i, mid]
            ans += j - i + 1;
        }
    }
    while (i <= j)
        b[len++] = a[i++];
    while (st <= ed)
        b[len++] = a[st++];
    for (int i = l; i <= r; i++)
        a[i] = b[i];
}
```



P11247 [GESP202409 六级] 算法学习

题意

小杨需要学习 m 种算法，有 n 道题目，每道题目属于一个知识点 a_i 并提升掌握程度 b_i 。目标是对每种算法的掌握程度至少达到 k ，且不能连续学习两道相同知识点的题目。求最少需要学习的题目数量，如果无法满足条件则输出 -1 。

数据范围： $1 \leq m, n, b_i, k \leq 10^5$, $1 \leq a_i \leq m$ 。



分析

1. **问题转化**: 将题目按知识点分组, 对于每个知识点, 需要选择若干题目使其总掌握程度至少为 k 。同时, 在学习顺序中, 不能连续学习相同知识点的题目。
2. **关键步骤**:

- **分组与贪心**: 对于每个知识点, 将其题目按掌握程度降序排序, 贪心选择题目直到总掌握程度 $\geq k$, 记录每个知识点所需的最少题目数 $\text{cnt}[i]$ 。
- **总题目数**: 计算所有知识点所需题目数的总和 $\text{ans} = \sum \text{cnt}[i]$ 。
- **检查可行性**: 如果任何知识点无法达到 k , 输出 -1 。



3. 学习顺序的约束：不能连续学习相同知识点的题目。这要求在学习顺序中，对于每个知识点，其题目不能被连续安排。

设 max_cnt 为所有知识点中所需题目数的最大值。

为了满足间隔条件，其他知识点的题目数必须至少为 $\text{max_cnt} - 1$ 。

这是因为，如果某个知识点需要 max_cnt 个题目，那么在安排顺序时，每两个该知识点的题目之间至少需要一个其他知识点的题目作为间隔。

因此，总的其他知识点题目数应不少于 $\text{max_cnt} - 1$ 。



4. 优化条件与组合数学原理：

- 如果 $\text{ans} - \text{max_cnt} \geq \text{max_cnt} - 1$, 则总的其他知识点题目数足够提供间隔, 可以直接使用 ans 个题目。
- 否则, 考虑其他知识点中未使用的题目 (即已经达到 k 但还有剩余题目的部分)。设 need 为其他知识点未使用的题目总数。如果 $\text{ans} - \text{max_cnt} + \text{need} \geq \text{max_cnt} - 1$, 则可以通过额外选择一些题目来填充间隔, 使总题目数达到 $2 \times \text{max_cnt} - 1$ 。



- **组合数学基础：**这里的优化基于鸽巢原理和序列安排问题。

对于一个需要 x 个题目的知识点，在学习序列中，这 x 个题目最多可以出现 x 次，但每两个之间必须有一个其他题目的间隔。

因此，整个序列中其他题目的数量至少为 $x - 1$ 。

如果其他题目的数量不足，我们可以从其他知识点中抽取未使用的题目来增加间隔，但这会增加总题目数。

总题目数的最小值取决于最大知识点的题目数和其他知识点的题目可用性，最小可能值为 $2x - 1$ （当其他题目刚好满足间隔时）。



5. 示例说明：

考虑以下示例：

知识点1：题目 $[9, 1]$ → 需要2题 ($9+1=10$)

知识点2：题目 $[10]$ → 需要1题

知识点3：题目 $[10, 1]$ → 需要1题

总题目数 $ans = 2 + 1 + 1 = 4$, 最大题目数 $max_cnt = 2$ 。

检查间隔条件： $ans - max_cnt = 2 \geq max_cnt - 1 = 1$, 满足条件, 因此可以直接输出4。



| B | C |
A、 A

可以这么理解，把 B、C 看作题目 2、3，那么最多的题目 1 设为 A。

ABAC
ABCA
BACA

以上都是合法的放入方案，除了最多题目数量以外的题目数量 ans，如果能够插入最多题目数量的中间空隙，说明方案可以正常摆放。



一个知识点1需要很多题目，而其他知识点题目很少的情况。例如：

知识点1：题目 [1, 1, 1, 1, 1, 1, 1, 1, 1] → 需要10题 (10个1)

知识点2：题目 [10] → 需要1题

知识点3：题目 [10] → 需要1题

总题目数 $ans = 10 + 1 + 1 = 12$, 最大题目数 $max_cnt = 10$ 。

检查间隔条件： $ans - max_cnt = 2 < 10 - 1 = 9$, 不满足。

然后计算剩余题目：知识点2和3都没有剩余题目（因为每个知识点只有1题，已经全部使用）。

所以 $ans - max_cnt + need = 2 + 0 = 2 < 9$, 无法满足，输出-1。



6. 时间复杂度：主要耗时在于对每个知识点的题目排序。每个知识点最多有 n 个题目，但总题目数为 n ，因此排序的总时间复杂度为 $O(n \log n)$ ，在数据范围内可行。



参考代码

```
const int N = 100010;
int n, m, k;
vector<int> p[N]; // p[i]存储知识点i的所有题目掌握程度
int cnt[N]; // cnt[i]表示知识点i所需的最少题目数

int main() {
    cin >> m >> n >> k;
    // 读入每道题的知识点
    vector<int> a(n);
    for (int i = 0; i < n; i++)
        cin >> a[i];

    // 读入每道题的掌握程度并分组
    for (int i = 0, b; i < n; i++)
        cin >> b; p[a[i]].push_back(b);

    int ans = 0, max_cnt = 0, max_idx = 0; // 总题目数 / 最大知识点题目数 / 对应知识点编号
```



```
// 处理每个知识点
for (int i = 1; i <= m; i++) {
    // 如果该知识点没有题目，无法达到k
    if (p[i].empty()) cout << -1 << endl, exit(0);

    // 按掌握程度降序排序
    sort(p[i].begin(), p[i].end(), greater<int>());
    int sum = 0, cnt[i] = 0;
    // 贪心选择题目直到达到k
    for (int val : p[i]) {
        sum += val, cnt[i]++;
        if (sum >= k)
            break;
    }
    // 无法达到k
    if (sum < k)
        cout << -1 << endl, exit(0);

    ans += cnt[i];
    // 更新最大题目数知识点
    if (cnt[i] > max_cnt)
        max_cnt = cnt[i], max_idx = i;
}
```



```
// 检查是否满足间隔条件
if (ans - max_cnt >= max_cnt - 1)
    cout << ans << endl;
else {
    // 计算其他知识点的剩余题目数
    int need = 0;
    for (int i = 1; i <= m; i++)
        if (i != max_idx)
            need += p[i].size() - cnt[i];
    // 检查是否可以通过额外选题满足间隔
    if (ans - max_cnt + need >= max_cnt - 1)
        // 总题目数需要达到2*max_cnt-1
        cout << 2 * max_cnt - 1 << endl;
    else
        cout << -1 << endl;
}

return 0;
}
```



P11375 [GESP202412 六级] 树上游走

I 题意

给定一棵无限节点的二叉树，根节点编号为 1，节点 i 的左儿子为 $2i$ ，右儿子为 $2i + 1$ 。从节点 s 开始，按照给定的移动序列（U: 向上移动，L: 向左儿子移动，R: 向右儿子移动）移动 n 次，求最终所在的节点编号。

数据范围： $1 \leq n \leq 10^6$ ， $1 \leq s \leq 10^{12}$ ，保证最终节点编号不超过 10^{12} 。



分析

1. **问题转化：**在无限二叉树中按照给定序列移动，需要高效处理移动操作，避免直接模拟导致数值溢出或效率低下。
2. **关键观察：**
 - 二叉树节点编号有规律：节点 i 的父节点为 $\lfloor i/2 \rfloor$ ，左儿子为 $2i$ ，右儿子为 $2i + 1$ 。
 - 当节点编号接近 10^{12} 时，继续向下移动可能导致数值溢出，但题目保证最终节点编号不超过 10^{12} ，因此可以安全处理。



3. 算法思路：

- 使用变量 s 记录当前节点编号， tag 记录"虚拟"的向下移动次数。
- 遍历移动序列：
 - 遇到 'U' (向上移动)：如果有 tag 则抵消一次，否则如果当前不是根节点则移动到父节点。
 - 遇到 'L' (向左移动)：如果 $2s > 10^{12}$ 则增加 tag ，否则移动到左儿子。
 - 遇到 'R' (向右移动)：如果 $2s + 1 > 10^{12}$ 则增加 tag ，否则移动到右儿子。
- 最终输出 s 。



4. 正确性证明：

- tag 的作用是记录那些会导致节点编号超过 10^{12} 的向下移动。由于题目保证最终节点编号不超过 10^{12} ，这些“虚拟”的向下移动最终会被向上的移动抵消。
- 当遇到向上移动时，优先抵消 tag ，因为实际的向上移动会回到一个编号更小的节点，而 tag 代表的虚拟向下移动也需要相应的向上移动来抵消。
- 这种方法确保了在移动过程中不会产生超过 10^{12} 的节点编号，同时正确记录了移动的净效果。

5. 时间复杂度： $O(n)$ ，只需遍历一次移动序列。



参考代码

```
const long long INF = 1e12; // 最大节点编号限制
int main() {
    int n;
    long long s;
    cin >> n >> s;
    int tag = 0; // 记录虚拟向下移动次数

    for (int i = 0; i < n; i++) {
        char ch; cin >> ch;
        // 向上移动
        if (ch == 'U') {
            if (tag > 0) tag--; // 抵消一次虚拟向下移动
            else if (s != 1) s /= 2; // 移动到父节点
        }
        // 向左儿子移动
        else if (ch == 'L') {
            if (s * 2 > INF) tag++; // 虚拟向下移动
            else s = s * 2; // 实际移动到左儿子
        }
        // 向右儿子移动
        } else if (ch == 'R') {
            if (s * 2 + 1 > INF) tag++; // 虚拟向下移动
            else s = s * 2 + 1; // 实际移动到右儿子
        }
    }
    cout << s << endl;
    return 0;
}
```



P11962 [GESP202503 六级] 树上漫步

I 题意

给定一棵 n 个节点的树，从每个节点出发，经过偶数步（步数可以为 0）能到达的节点数量。每一步可以移动到相邻节点，且允许重复访问节点。

数据范围： $1 \leq n \leq 2 \times 10^5$ 。



分析

时间复杂度： $O(n)$ ，遍历树一次。

1. **问题转化：**树是一个二分图，可以按节点深度的奇偶性进行染色。从任意节点出发，经过偶数步只能到达与起点深度奇偶性相同的节点，因为每一步都会改变深度的奇偶性。
2. **关键观察：**

- 设节点深度的奇偶性为颜色：深度为偶数的节点颜色为 0，深度为奇数的节点颜色为 1。
- 从颜色为 c 的节点出发，经过偶数步只能到达颜色为 c 的节点。
- 因此，对于每个节点，能到达的节点数量等于与其颜色相同的节点总数。



3. 算法步骤：

- 从根节点（节点 1）开始进行 DFS 或 BFS，计算每个节点的深度（从根节点开始的步数）。
- 统计深度为偶数的节点数量 cnt_0 和深度为奇数的节点数量 cnt_1 。
- 对于每个节点 i ，如果其深度为偶数，则答案为 cnt_0 ；否则为 cnt_1 。

4. 正确性证明：

- 由于树是无环连通图，且边连接不同深度的节点，因此深度奇偶性定义了合法的二分图染色。
- 从任意节点出发，每一步移动都会改变深度奇偶性，因此偶数步后深度奇偶性与起点相同。
- 由于允许重复访问节点，且树是连通的，从起点可以到达所有同色节点（通过来回移动调整步数）。



```
const int N = 200010;
vector<int> graph[N]; // 邻接表存储树
int color[N]; // color[i]表示节点i的颜色: 0表示偶数深度, 1表示奇数深度
int cnt[2]; // cnt[0]记录偶数深度节点数, cnt[1]记录奇数深度节点数

void dfs(int node, int parent, int depth) {
    // 根据当前深度设置颜色并计数
    color[node] = depth % 2, cnt[color[node]]++;
    // 遍历相邻节点
    for (int neighbor : graph[node])
        if (neighbor != parent) // 避免回到父节点
            dfs(neighbor, node, depth + 1);
}

int main() {
    int n;
    cin >> n;
    // 读入边, 构建树
    for (int u, v, i = 0; i < n - 1; i++)
        cin >> u >> v, graph[u].push_back(v), graph[v].push_back(u);

    // 从节点1开始DFS, 初始深度为0 (偶数)
    dfs(1, 0, 0);

    // 输出每个节点的答案
    for (int i = 1; i <= n; i++)
        cout << cnt[color[i]] << " ";
}

return 0;
}
```



P11376 [GESP202412 六级] 运送物资

题意

有 n 个运输站点位于 A 市（坐标 0）和 B 市（坐标 x ）之间，第 i 个站点的坐标为 p_i ($0 < p_i < x$)，最多容纳 c_i 辆货车。有 m 辆货车，第 j 辆货车每天需要向 A 市运送 a_j 次物资，向 B 市运送 b_j 次物资。货车从分配站点出发，前往 A 市或 B 市后返回，每次去 A 市行驶路程为 $2p_i$ ，去 B 市行驶路程为 $2(x - p_i)$ 。求在最优分配站点的情况下，所有货车每天的最短总行驶路程。

数据范围： $1 \leq n, m \leq 10^5$, $2 \leq x \leq 10^8$, $1 \leq c_i \leq 10^5$, $0 \leq a_j, b_j \leq 10^5$, 保证 $\sum c_i \geq m$ 。



分析

1. **问题转化**: 对于货车 j 分配到站点 i , 其每日行驶路程为 $2a_j p_i + 2b_j(x - p_i) = 2p_i(a_j - b_j) + 2b_jx$ 。其中 $2b_jx$ 为常数, 因此最小化总路程等价于最小化 $2p_i(a_j - b_j)$ 的总和。

2. 关键观察:

- 当 $a_j \geq b_j$ 时, $a_j - b_j \geq 0$, 因此 p_i 越小, $2p_i(a_j - b_j)$ 越小。
- 当 $a_j < b_j$ 时, $a_j - b_j < 0$, 因此 p_i 越大, $2p_i(a_j - b_j)$ 越小 (负值绝对值越大)。
- 因此, 应将 $a_j \geq b_j$ 的货车分配到 p_i 较小的站点, 将 $a_j < b_j$ 的货车分配到 p_i 较大的站点。



3. 贪心策略：

- 将货车分为两组： f 组 ($a_j \geq b_j$) 和 g 组 ($a_j < b_j$)。
- 对站点按 p_i 从小到大排序。
- 对 f 组按 $(a_j - b_j)$ 降序排序，优先将 $(a_j - b_j)$ 大的货车分配到 p_i 小的站点，以最大化节省。
- 对 g 组按 $(b_j - a_j)$ 降序排序，优先将 $(b_j - a_j)$ 大的货车分配到 p_i 大的站点，以最大化节省。
- 在站点容量限制下，按顺序分配站点。



4. 正确性证明：

- 对于 f 组，设两辆货车 j 和 k 满足 $(a_j - b_j) > (a_k - b_k)$ ，若分配站点 $p_1 < p_2$ 。
则分配方案 $j \rightarrow p_1$ 、 $k \rightarrow p_2$ 的总代价为 $2p_1(a_j - b_j) + 2p_2(a_k - b_k)$ 。
若交换则代价为 $2p_2(a_j - b_j) + 2p_1(a_k - b_k)$ 。
由于 $(a_j - b_j) > (a_k - b_k)$ 且 $p_1 < p_2$ ，
有 $p_1(a_j - b_j) + p_2(a_k - b_k) < p_2(a_j - b_j) + p_1(a_k - b_k)$ ，因此原方案更优。
- 同理可证 g 组分配策略的最优性。
- 站点容量限制通过顺序分配保证，且数据保证 $\sum c_i \geq m$ ，故所有货车均能被奇思妙学 GESP 六级 分配。



```
struct Station {
    int p, c; // 站点位置和容量
} d[N];

struct Truck {
    int a, b; // 货车向A、B运送次数
} f[N], g[N];

int fcnt, gcnt; // f组和g组的货车数量
int ans;         // 总行驶路程

// 按站点位置升序排序
bool cmpStation(Station a, Station b) {return a.p < b.p;}

// 按(a-b)降序排序, 用于f组
bool cmpF(Truck x, Truck y) {return x.a - x.b > y.a - y.b;}

// 按(b-a)降序排序, 用于g组
bool cmpG(Truck x, Truck y) {return x.b - x.a > y.b - y.a;}
```



```
signed main() {
    int n, m, x;
    cin >> n >> m >> x;

    // 读入站点数据
    for (int i = 1; i <= n; i++)
        cin >> d[i].p >> d[i].c;

    // 读入货车数据并分组
    for (int i = 1; i <= m; i++) {
        int a, b;
        cin >> a >> b;
        if (a >= b)
            f[++fcnt] = {a, b};
        else
            g[++gcnt] = {a, b};
    }
}
```



```
// 排序
sort(d + 1, d + n + 1, cmpStation), sort(f + 1, f + fcnt + 1, cmpF),
sort(g + 1, g + gcnt + 1, cmpG);

// 分配f组货车到小位置站点
int idx = 1; // 指向当前最小站点
for (int i = 1; i <= fcnt; i++) {
    while (d[idx].c == 0) idx++; // 跳过已满站点
    ans += 2 * f[i].a * d[idx].p + 2 * f[i].b * (x - d[idx].p);
    d[idx].c--;
}
// 分配g组货车到大位置站点
idx = n; // 指向当前最大站点
for (int i = 1; i <= gcnt; i++) {
    while (d[idx].c == 0) idx--; // 跳过已满站点
    ans += 2 * g[i].a * d[idx].p + 2 * g[i].b * (x - d[idx].p);
    d[idx].c--;
}
cout << ans << endl;
return 0;
}
```



P13016 [GESP202506 六级] 最大因数

I 题意

给定一棵有 10^9 个结点的有根树，结点编号从 1 到 10^9 ，根结点为 1。对于编号为 k ($2 \leq k \leq 10^9$) 的结点，其父结点编号为 k 的因数中除 k 以外最大的因数。现在有 q 组询问，每组询问给出两个结点编号 x_i, y_i ，要求求出这两个结点在树上的距离（即连接两结点的简单路径的边数）。

数据范围： $1 \leq q \leq 1000$, $1 \leq x_i, y_i \leq 10^9$ 。



分析

根据题意，每个结点（除根结点）的父结点是它除自身外最大的因数。对于偶数，最大因数是 $k/2$ ；对于奇数，我们需要找到它的最小奇因数（因为最小奇因数对应的因数对中的另一个因数就是最大因数），如果这个最小奇因数是 k 本身（即 k 是质数），那么最大因数就是 1。

我们可以通过模拟从 x 和 y 到根结点 1 的路径来求解。两条路径的交集部分（从根结点开始到最近公共祖先的部分）是公共的，我们只需要找到最近公共祖先（LCA），然后距离就是 x 到 LCA 的距离加上 y 到 LCA 的距离。



具体步骤：

1. 定义函数 `find(x)` 用于求 x 的父结点。

- 如果 x 是 1，则返回 0（表示没有父结点）。
- 如果 x 是偶数，则父结点为 $x/2$ 。
- 如果 x 是奇数，则从 3 开始尝试奇数因子直到 \sqrt{x} ，如果找到因子 i ，则返回 x/i （因为 i 是最小奇因子，则 x/i 就是最大因子）。如果没有找到，说明 x 是质数，则返回 1。

2. 定义函数 `find_path(x)`，返回从 x 到根结点 1 的路径上的所有结点（包括 x 和 1）。



3. 对于每组询问 (x, y) :

- 分别求出 x 和 y 的路径 a 和 b 。
- 然后从根结点（即路径的最后一个元素）开始向前比较，找到最后一个相同的结点（即最近公共祖先）。
- 距离的计算：设 i 和 j 分别从 a 和 b 的末尾（根结点）开始向前移动，直到遇到不同的结点。那么 x 到 LCA 的距离为 $i+1$ ， y 到 LCA 的距离为 $j+1$ ，但由于我们是从末尾开始比较的，实际计算时距离为 $i + j + 2$ 。

时间复杂度：每次询问，我们需要分别求出 x 和 y 到根结点的路径。路径的长度最多为 $O(\log x)$ 和 $O(\log y)$ ，因为每次跳转到父结点，结点编号至少减半。然后比较路径也是 $O(\log n)$ 。所以总时间复杂度为 $O(q \log n)$ ，其中 n 是结点编号的最大值 (10^9)。



参考代码

```
// 求结点x的父结点
int find(int x) {
    if (x == 1) return 0; // 根结点没有父结点
    if (x % 2 == 0) return x / 2; // 偶数，父结点为x/2
    // 奇数，寻找最小奇因数
    for (int i = 3; i * i <= x; i += 2) {
        if (x % i == 0) {
            return x / i; // 找到最小奇因数i，则父结点为x/i
        }
    }
    return 1; // 是质数，则父结点为1
}

// 求从x到根结点的路径
vector<int> find_path(int x) {
    vector<int> s;
    while (x != 0) { // 当x不为0（根结点的父结点为0）时，继续向上
        s.emplace_back(x);
        x = find(x);
    }
    return s;
}
```



```
// 计算x和y之间的距离
int solve(int x, int y) {
    vector<int> a = find_path(x); // x的路径
    vector<int> b = find_path(y); // y的路径
    int i = a.size() - 1, j = b.size() - 1;
    // 从根结点开始（路径的最后一个元素）向前比较，直到遇到第一个不同的结点
    while (i >= 0 && j >= 0 && a[i] == b[j]) {
        i--;
        j--;
    }
    // 距离为 (i+1) + (j+1) = i+j+2
    return i + j + 2;
}

int main() {
    int t;
    cin >> t;
    while (t--) {
        int x, y;
        cin >> x >> y;
        cout << solve(x, y) << endl;
    }
    return 0;
}
```



GESP

信息学竞赛

GESP C++ 六级认证 (四)

综合代码训练