算法一：暴力枚举法

算法思想与步骤

对于每一个可能出现在数组中的元素，扫描整个数组统计它出现的次数，若次数 $> \frac{n}{2}$ 则输出。
具体操作时可以用二重循环：外层枚举位置 i 的元素 a_i 作为候选，内层遍历整个数组统计 a_i 的出现次数。

正确性证明

因为绝对多数元素一定存在，穷举所有候选必然能找到它。根据定义，它的出现次数 $> \frac{n}{2}$，当检验到它时必会满足条件。

时间复杂度：$O(n^2)$，空间复杂度：$O(1)$。

C++ 代码

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    freopen("hztp.in", "r", stdin);
    freopen("hztp.out", "w", stdout);

    int n;
    scanf("%d", &n);
    vector<int> a(n + 1);          // 下标从1开始
    for (int i = 1; i <= n; ++i)
        scanf("%d", &a[i]);

    // 暴力枚举：对每个位置i，统计a[i]在整个数组中出现的次数
    for (int i = 1; i <= n; ++i) {
        int cnt = 0;
        for (int j = 1; j <= n; ++j) {
            if (a[j] == a[i]) ++cnt;
        }
        if (cnt > n / 2) {          // 找到超过半数的元素
            printf("%d\n", a[i]);
            break;                  // 找到即可退出
        }
    }
    return 0;
}
```

算法二：排序取中法

算法思想与步骤

将数组从小到大排序，取下标为 $\lfloor n/2 \rfloor + 1$（即 $\frac{n+1}{2}$ 向上取整）的元素，该元素必定是绝对多数元素。
C++ 中可使用标准库 `sort`，时间复杂度 $O(n \log n)$。

正确性证明

设绝对多数元素为 x，其出现次数 $> \frac{n}{2}$。排序后所有 x 会连续排列。不论这段连续的 x 的起始位置在哪里，其长度超过数组长度的一半，因此必会覆盖中间位置。换言之，排序后位于 $\lceil \frac{n}{2} \rceil$ 位置的元素一定是 x。
故直接输出排序后的中间元素即可。

时间复杂度：$O(n \log n)$，空间复杂度：$O(n)$（亦可原地排序，视 `sort` 实现而定）。

C++ 代码

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    freopen("hztp.in", "r", stdin);
    freopen("hztp.out", "w", stdout);

    int n;
    scanf("%d", &n);
    vector<int> a(n + 1);
    for (int i = 1; i <= n; ++i)
        scanf("%d", &a[i]);

    sort(a.begin() + 1, a.end());   // 对下标1~n排序
    // 中间位置为 (n+1)/2，对应下标从1开始
    int ans = a[(n + 1) / 2];
    printf("%d\n", ans);
    return 0;
}
```

算法三：分治法

算法思想与步骤

采用类似归并的递归策略：

1. 将数组分成左右两半，分别递归找出左半部分的多数元素和右半部分的多数元素。
2. 若左、右得到的多数元素相同，则它就是整个数组的多数元素；

3. 若不同，则在整个数组范围内分别统计这两个元素的出现次数，取次数超过半数的那个作为结果。

正确性证明

如果 x 是整个数组的绝对多数元素，那么它至少在左半部分或右半部分也是绝对多数元素吗？不一定。例如 `[1,2,1,2,1]` 左边 `[1,2]` 没有多数元素。因此分治时左右可能都没有多数元素，此时递归返回特殊值（如 -1）表示不存在，最后在全数组范围统计候选。只要候选存在，统计全数组即可找出真正的绝对多数。
该算法正确性基于：若存在全局多数，则合并时正确统计可得。

时间复杂度：$T(n) = 2T(n/2) + O(n) \implies O(n \log n)$，空间复杂度 $O(\log n)$（递归栈）。

C++ 代码

```cpp
#include <bits/stdc++.h>
using namespace std;

vector<int> a; // 全局数组，便于递归访问

// 统计元素x在下标区间[l, r]内出现的次数
int countInRange(int x, int l, int r) {
    int cnt = 0;
    for (int i = l; i <= r; ++i)
        if (a[i] == x) ++cnt;
    return cnt;
}

// 分治函数，返回区间[l, r]内的绝对多数元素；若不存在则返回-1
int majorityDivide(int l, int r) {
    if (l == r) return a[l];                // 只有一个元素，直接返回
    int mid = (l + r) / 2;
    int leftMaj = majorityDivide(l, mid);
    int rightMaj = majorityDivide(mid + 1, r);

    if (leftMaj == rightMaj) return leftMaj; // 左右多数相同
    // 否则统计两个候选在当前区间的出现次数
    int leftCnt = countInRange(leftMaj, l, r);
    int rightCnt = countInRange(rightMaj, l, r);
    if (leftCnt > (r - l + 1) / 2) return leftMaj;
    if (rightCnt > (r - l + 1) / 2) return rightMaj;
    return -1;                              // 理论上不会执行，因全局多数必存在
}

int main() {
    freopen("hztp.in", "r", stdin);
    freopen("hztp.out", "w", stdout);

    int n;
    scanf("%d", &n);
    a.resize(n + 1);
```

```
    for (int i = 1; i <= n; ++i) scanf("%d", &a[i]);

    int ans = majorityDivide(1, n);
    printf("%d\n", ans);
    return 0;
}
```

算法四：位运算法

算法思想与步骤

由于参赛号是整数，可以逐位确定绝对多数元素的二进制表示。
对每一位 bit（从 0 到 $\lfloor \log_2(\max a_i) \rfloor$），统计数组中该位为 1 的元素个数。如果该计数 $> \frac{n}{2}$，说明绝对多数元素在这一位上必为 1，否则为 0。按位拼出最终结果即可。

正确性证明

设绝对多数元素为 x。对于第 k 位，x 在该位上的值要么是 0 要么是 1。在数组中出现次数 $> \frac{n}{2}$ 的 x 必定使得该位的众数与 x 一致。因为该位为 1 的元素个数要么包含全部的 x（当 x 该位为 1），要么完全不包含 x（当 x 该位为 0）。无论哪种情况，绝对多数元素都会使该位的计数值偏向它那一边，故按多数原则逐位决定即可。

时间复杂度：$O(n \log U)$，$U = 3 \times 10^8 < 2^{29}$，因此约 $29n$ 次操作。空间复杂度：$O(1)$。

C++ 代码

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    freopen("hztp.in", "r", stdin);
    freopen("hztp.out", "w", stdout);

    int n;
    scanf("%d", &n);
    vector<int> a(n + 1);
    int maxVal = 0;
    for (int i = 1; i <= n; ++i) {
        scanf("%d", &a[i]);
        if (a[i] > maxVal) maxVal = a[i]; // 记录最大值以确定位数
    }

    int ans = 0;
    // 逐位确定，最大值的位数决定循环上限
    for (int bit = 0; (1LL << bit) <= maxVal; ++bit) {
        int cnt = 0; // 统计该位为1的个数
        for (int i = 1; i <= n; ++i) {
            if (a[i] & (1 << bit)) ++cnt;
        }
        if (cnt > n / 2)           // 多数原则
            ans |= (1 << bit);     // 将该位置1
    }
    printf("%d\n", ans);
    return 0;
}
```

算法五：哈希表统计法

算法思想与步骤

使用哈希表（C++ `unordered_map`）记录每个元素出现的次数。遍历数组，每遇到一个元素就将其计数加 1，同时检查计数是否已超过 $\frac{n}{2}$，若是则立即输出。

正确性证明

哈希表如实统计了每个元素的频率，绝对多数元素必然在某个时刻计数超过半数，算法自然正确。

时间复杂度：期望 $O(n)$（最坏 $O(n^2)$ 当哈希冲突严重时，但概率极低）。空间复杂度：$O(n)$。

C++ 代码

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    freopen("hztp.in", "r", stdin);
    freopen("hztp.out", "w", stdout);

    int n;
    scanf("%d", &n);
    unordered_map<int, int> freq; // 哈希表统计频率
    int ans = 0;
    for (int i = 1; i <= n; ++i) {
        int x;
        scanf("%d", &x);
        if (++freq[x] > n / 2) {  // 找到超过半数的元素
            ans = x;
            break;                // 题目保证存在，找到即可停止
        }
    }
    printf("%d\n", ans);
    return 0;
}
```

算法六：随机化方法

算法思想与步骤

随机选取数组中的一个下标，检验该元素是否出现次数 $> \frac{n}{2}$。若否，则重新随机选取。因为绝对多数元素占据超过一半的位置，每次随机选中的概率 $> \frac{1}{2}$，因此期望尝试次数不超过 2。

正确性证明

每次独立随机选取，选中多数元素的概率 $p > 0.5$，期望次数为 $\frac{1}{p} < 2$。每次检验需 $O(n)$ 时间，因此期望总时间为 $O(n)$。最坏情况下可能无限循环，但概率为 0。

时间复杂度：期望 $O(n)$，空间复杂度 $O(1)$。

C++ 代码

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    freopen("hztp.in", "r", stdin);
    freopen("hztp.out", "w", stdout);

    int n;
    scanf("%d", &n);
    vector<int> a(n + 1);
    for (int i = 1; i <= n; ++i) scanf("%d", &a[i]);

    srand(time(0)); // 初始化随机种子
    int ans;
    while (true) {
        int idx = rand() % n + 1;      // 随机下标[1, n]
        int cand = a[idx];
        int cnt = 0;
        for (int i = 1; i <= n; ++i)   // 校验
            if (a[i] == cand) ++cnt;
        if (cnt > n / 2) {
            ans = cand;
            break;
        }
    }
    printf("%d\n", ans);
    return 0;
}
```

算法七：摩尔投票法（Boyer–Moore Majority Vote）

算法思想与步骤

摩尔投票是解决该问题的最优算法，时间 $O(n)$，空间 $O(1)$。
维护一个 `candidate` 和一个计数器 `cnt`，初始 `cnt = 0`。遍历数组：

- 若 `cnt == 0`，将当前元素设为 `candidate`，`cnt = 1`；
- 否则，若当前元素等于 `candidate`，则 `cnt++`，否则 `cnt--`。
  遍历结束，`candidate` 即为绝对多数元素。

正确性证明

核心思想是 **“多数元素与其他元素一一抵消后仍有剩余”**。将绝对多数元素看作 $+1$，其他元素看作 -1，则整个数组的代数和 > 0。在摩尔投票过程中，每次遇到不同元素就相当于一次“抵消”。因为多数元素数量超过总数一半，无论如何配对抵消，最后存活的 `candidate` 必定是多数元素。严格的数学归纳证明可基于“多数元素在任意前缀中不可能被完全抵消”的性质。

时间复杂度：$O(n)$，空间复杂度：$O(1)$。

C++ 代码

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    freopen("hztp.in", "r", stdin);
    freopen("hztp.out", "w", stdout);

    int n;
    scanf("%d", &n);
    int candidate = 0, cnt = 0; // 候选人，计数器
    for (int i = 1; i <= n; ++i) {
        int x;
        scanf("%d", &x);
        if (cnt == 0) {         // 当前没有候选人，立刻推举x
            candidate = x;
            cnt = 1;
        } else if (x == candidate) {
            ++cnt;              // 遇到相同，投票加一
        } else {
            --cnt;              // 遇到不同，相互抵消
        }
    }
    // 根据题意，绝对多数元素一定存在，candidate 即为答案
    printf("%d\n", candidate);
    return 0;
}
```

总结

算法	时间复杂度	空间复杂度	备注
暴力枚举	$O(n^2)$	$O(1)$	仅用于理论分析，不可处理大数据
排序取中	$O(n \log n)$	$O(n)$ / $O(1)$	简洁易写
分治法	$O(n \log n)$	$O(\log n)$	体现递归思想

位运算法	$O(n \log U)$	$O(1)$	适用于整数且值域不大时
哈希表统计	期望 $O(n)$	$O(n)$	实现简单，但哈希有常数开销
随机化	期望 $O(n)$	$O(1)$	思想巧妙，但不保证最坏复杂度
摩尔投票	$O(n)$	$O(1)$	**最优算法，推荐使用**