二分

CF780B. The Meeting Place Cannot Be Changed(小数二分)

```
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
//#define int long long
#define li __int128
#define vi vector<int>
#define pii pair<int, int>
#define arr3 array<int, 3>
#define lowbit(x) (x & -x)
const int N = 6e4 + 10, M = 2e5 + 10, inf = 0x3f3f3f3f, Inf = INT_MAX, mod
= 1e9 + 7;
int n, a[N], v[N];
/*
- 首先，对于每个朋友的位置、速度 p,v ，我们发现，这个朋友在 t 时刻可以到达
[p-vt,p+vt] 线段上的任意一点。于是，我们只需要判断这 n 条线段有无公共点。
- 我们考虑我们相遇的那个关键点，可以发现所有线段的右端点均在这个关键点右侧，左端点都在这
个关键点左侧。
- 于是我们得到条件：线段左端点的最大值小于关键点坐标，也就进一步小于右端点的最小值。
- 而左端点最大值小于右端点最小值这一条件也是充分的。因为可以取左端点的最大值作为相遇点。
*/
bool check(double mid){
    double maxn = 0, minn = 1e9;
    for(int i = 1; i <= n; i++){
        maxn = max(maxn, a[i] - mid * v[i]);
        minn = min(minn, a[i] + mid * v[i]);
    }
    return maxn <= minn;
}

void solve(){
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i];
    for(int i = 1; i <= n; i++) cin >> v[i];
    double l = 0, r = 1e9;
    while(r - l > 1e-6){
        double mid = (l + r) / 2;
        if(check(mid)) r = mid;
        else l = mid;
    }
    cout << fixed << setprecision(12) << r << endl;

}
```

```
signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T = 1;
    //cin >> T;
    while(T--) solve();
    return 0;
}
```

CF1461D. Divide and Summarize(前缀和 + 二分)

解析

```
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define int long long // 要开long long
const int N = 1e6 + 10, M = 2e5 + 10, inf = 0x3f3f3f3f, Inf = INT_MAX, mod
= 1e9 + 7;
int n, q, a[N], sum[N];
set<int> st;

void dfs(int l, int r){
    st.insert(sum[r] - sum[l - 1]);
    if(l == r) return;
    int mid = upper_bound(a + l, a + r + 1, (a[l] + a[r]) >> 1) - a - 1;
    if(mid >= r) return;
    dfs(l, mid), dfs(mid + 1, r);
}

void solve(){
    st.clear();
    cin >> n >> q;
    for(int i = 1; i <= n; i++) cin >> a[i];
    sort(a + 1, a + n + 1);
    for(int i = 1; i <= n; i++) sum[i] = sum[i - 1] + a[i];
    dfs(1, n);
    while(q--){
        int tmp;
        cin >> tmp;
        cout << (st.count(tmp) ? "Yes" : "No") << endl;
    }
}

signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T = 1;
    cin >> T;
    while(T--) solve();
```

```
    return 0;
}
```

CF231C. To Add or Not to Add(前缀和 + 二分)

解析

```
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define int long long // 要开long long
const int N = 1e5 + 10, M = 2e5 + 10, inf = 0x3f3f3f3f, mod = 1e9 + 7;
int n, k, a[N], sum[N], cnt[N], ans;
/*
- 首先给 a 数组从小到大排序。
- 显然，如果要把数字改成 `a[i]`，显然只能更改 `a[1] ~ a[i-1]` 的数字。
- 首先暴力枚举这个 `i`，考虑将 `a[j] ~ a[i] (j ≤ i)` 改为 `a[i]`。
- 显然 `j`越小，需要更改的次数越大，所以具有单调性，考虑二分答案
*/
void solve(){
    cin >> n >> k;
    for(int i = 1; i <= n; i++) cin >> a[i];
    sort(a + 1, a + n + 1);
    for(int i = 1; i <= n; i++) sum[i] = sum[i - 1] + a[i];
    for(int i = 1; i <= n; i++){
        int l = 1, r = i;
        while(l < r){
            int mid = (l + r) >> 1;
            if((i - mid + 1) * a[i] - (sum[i] - sum[mid - 1]) <= k) r =
mid;
            else l = mid + 1;
        }
        cnt[i] = i - l + 1;
    }
    for(int i = 1; i <= n; i++)
        if(cnt[i] > cnt[ans]) ans = i;
    cout << cnt[ans] << ' ' << a[ans] << endl;
}

signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T = 1;
    //cin >> T;
    while(T--) solve();
    return 0;
}
```

CF847B. Preparing for Merge Sort

解析

```cpp
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define int long long
#define vi vector<int>
const int N = 2e5 + 10, M = 2e5 + 10, inf = 0x3f3f3f3f, mod = 1e9 + 7;
int n, a[N], last[N] = {inf}, cnt;
vi ans[N];
/*
对于每个 a[i]，找到第一个结尾比 a[i] 小的序列，并将 a[i] 插入到序列的末尾。
如果找不到，则新开一个序列，然后将 a[i] 加入这个序列。
*/
void solve(){
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i];
    for(int i = 1; i <= n; i++){
        if(a[i] <= last[cnt]){
            last[++cnt] = a[i];
            ans[cnt].push_back(a[i]);
        }else{
            int l = 1, r = cnt;
            while(l < r){
                int mid = (l + r) >> 1;
                if(a[i] > last[mid]) r = mid;
                else l = mid + 1;
            }
            last[l] = a[i];
            ans[l].push_back(a[i]);
        }
    }
    for(int i = 1; i <= cnt; i++){
        for(auto &tmp : ans[i]) cout << tmp << ' ';
        cout << endl;
    }
    return;
}

signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T = 1;
    //cin >> T;
    while(T--) solve();
    return 0;
}
```

CF1223C. Save the Nature(gcd+lcm + 二分)

解析

```cpp
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define int long long
#define vi vector<int>
const int N = 2e5 + 10, M = 2e5 + 10, inf = 0x3f3f3f3f, mod = 1e9 + 7;
int n, p[N], x, y, a, b, ab, k;
/*
- p[i] 可以被 a,b 的公倍数整除，有 (x+y)% 的贡献；
- p[i] 可以被 a 整除，有 x% 的贡献；
- p[i] 可以被 b 整除，有 y% 的贡献。
*/
int gcd(int a, int b){
    return b ? gcd(b, a % b) : a;
}
int lcm(int a, int b){
    return a / gcd(a, b) * b;
}

bool check(int mid){
    int sum = 0, xn = mid / a, yn = mid / b, xyn = mid / ab;
    xn -= xyn; // 算容斥
    yn -= xyn;
    for(int i = 1; i <= xyn; i++) sum += (x + y) * p[i];
    for(int i = xyn + 1; i <= xyn + xn; i++) sum += x * p[i];
    for(int i = xyn + xn + 1; i <= xyn + xn + yn; i++) sum += y * p[i];
    return sum >= k;
}

void solve(){
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> p[i], p[i] /= 100;
    sort(p + 1, p + n + 1, greater<int>());
    cin >> x >> a >> y >> b >> k;
    if(x < y) swap(x, y), swap(a, b);
    ab = lcm(a, b);
    int l = 1, r = n + 1;
    while(l < r){
        int mid = (l + r) >> 1;
        if(check(mid)) r = mid;
        else l = mid + 1;
    }
    cout << (l == n + 1 ? -1 : l) << endl;
}

signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
```

```
    int T = 1;
    cin >> T;
    while(T--) solve();
    return 0;
}
```

CF380A. Sereja and Prefixes(记录“版本变化” + 反查)

解析

```
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define int long long
#define vi vector<int>
const int N = 2e5 + 10, M = 1e5, inf = 0x3f3f3f3f, mod = 1e9 + 7;
int n, m, op, x, l, c;
vi pre, notes, lengths = {0};
// pre 存储前 M 个构造出来的数，供查询用
// notes 存储每次操作（负数代表 type 1，正数代表复制的长度 l）

// 注意lengths的位置和notes的位置有一位偏差
void solve(){
    cin >> n;
    while(n--){
        cin >> op;
        if(op == 1){
            cin >> x;
            if(pre.size() < M)
                pre.emplace_back(x);
            notes.emplace_back(-x);
            lengths.push_back(lengths.back() + 1);
        }else{
            cin >> l >> c;
            for(int i = 0; i < l * c && pre.size() < M; i++)
                pre.emplace_back(pre[i % l]);
            notes.emplace_back(l);
            lengths.emplace_back(lengths.back() + l * c);
        }
    }
    cin >> m;
    while(m--){
        cin >> x;
        // 二分查找：找到生成位置 x 的操作编号 p
        // lengths[i] 表示前 i 次操作后的序列长度
        // 所以 lengths[p] < x <= lengths[p+1]
        int p = lower_bound(lengths.begin(), lengths.end(), x) -
lengths.begin() - 1;
        if(notes[p] < 0) cout << -notes[p] << ' ';
```

```
        else{
            int idx = (x - lengths[p] - 1) % notes[p];
            cout << pre[idx] << ' ';
        }
    }
    cout << endl;
}

signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T = 1;
    //cin >> T;
    while(T--) solve();
    return 0;
}
```

CF1168A. Increasing by Modulo

解析

我们假设到 i 下标时，得到的最小的能使得前缀单调不减的数字为 x ，则接下来考虑下标 $i+1$ ：

- 如果该数比 x 小，那么我们最好的做法是尽可能把它刚好凑成 x ，如果次数不够就不可行。
- 如果该数比 x 大，那么我们有两种策略：
  - 不改变该数。
  - 增加该数，直到其又从 0 开始增大到 x 。这样可以使得 $i+1$ 位置达到 x ，比原数小。如果这么做，需要操作次数足够，设原数字为 y ，则操作次数至少为 $m-y+x$ 次让其变为 x ，否则只能维持原样。

  根据上述讨论，我们可以贪心得到 $i+1$ 位置能得到的最小的数。如果上述操作可以不断进行，则对应 k 可行，否则 k 不可行。

  那么二分上界如何选取呢？事实上，二分上界为 $m-1$ ，因为 $m-1$ 次操作足以使任何数变成其他任何数。

  于是，通过二分，本题的复杂度为 $\mathcal{O}(n\log n)$.

```
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define int long long
#define vi vector<int>
const int N = 3e5 + 10, M = 2e5 + 10, inf = 0x3f3f3f3f, mod = 1e9 + 7;
int n, m, a[N], b[N];

bool check(int mid){
    for(int i = 1; i <= n; i++) b[i] = a[i];
    for(int i = 1; i <= n; i++){
        if(b[i - 1] > b[i]){
```

```
            if(b[i - 1] - b[i] > mid) return false;
            b[i] = b[i - 1];
        }else if(b[i - 1] < b[i]){
            if(m - b[i] + b[i - 1] <= mid) b[i] = b[i - 1];
        }
    }
    return true;
}

void solve(){
    cin >> n >> m;
    for(int i = 1; i <= n; i++) cin >> a[i];
    int l = 0, r = m - 1;
    while(l < r){
        int mid = (l + r) >> 1;
        if(check(mid)) r = mid;
        else l = mid + 1;
    }
    cout << l << endl;
}

signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T = 1;
    //cin >> T;
    while(T--) solve();
    return 0;
}
```

CF689C. Mike and Chocolate Thieves(反向构造 + 二分)

解析

我们知道：

- 一共有 **4 个小偷**。
- 每个小偷偷的数量是前一个的 `k` 倍（`k` 是未知的**大于 1 的整数**）。
- 第一个小偷偷的是 `a`，第二个 `a*k`，第三个 `a*k^2`，第四个 `a*k^3`。
- 要求所有数都不超过一个最大值 `n`。
- 题目给出某个 `m`，表示对于这个 `n`，有恰好 `m` 种这样的四元组序列 `(a, ak, ak², ak³)`。
  我们的任务是找出最小的 `n`，使得满足条件的方案数是 `m`。

我们反过来思考：

- 枚举所有可能的 `n` 太大，不现实。
- 但如果我们**知道某个 `n`**，我们可以**数一数有多少种合法的 `(a, k)`** 组合使得 `a*k^3 <= n`。

- 对于每个合法的 `k`，可以计算有多少个 `a` 满足 `a*k^3 <= n`。
- 而二分上界可以设定为 $8m$，因为 $i, 2i, 4i, 8i (1 \le i \le m)$ 一定是公比为整数的等比数列，这样最大值不超过 $8m$ 的等比数列个数一定不少于 m 个。
  于是，我们变成了一个典型的“**单调性+二分答案**”问题。

```cpp
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define int long long
#define vi vector<int>
const int N = 3e5 + 10, M = 2e5 + 10, inf = 0x3f3f3f3f, mod = 1e9 + 7;
int m;

int check(int mid){
    int ans = 0;
    for(int i = 2; mid / i / i / i > 0; i++)
        ans += mid / i / i / i;
    return ans;
}

void solve(){
    cin >> m;
    int l = 1, r = m << 3;
    while(l < r){
        int mid = (l + r) >> 1;
        if(check(mid) >= m) r = mid;
        else l = mid + 1;
    }
    cout << (check(l) == m ? l : -1);
}

signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T = 1;
    //cin >> T;
    while(T--) solve();
    return 0;
}
```

CF895B. XK Segments

解析

我们要使得区间 $[a_i, a_j]$ 中 x 的倍数恰好有 k 个。为此，我们考虑固定左端点看右端点应该满足什么要求。
首先，右端点不小于左端点，因此 $r \ge l$ 。
而同时，区间中 x 的倍数的个数如何保证呢?
我们先找到第一个不小于 l 的 x 的倍数，这个数是 $\left(\left\lfloor \frac{l-1}{x} \right\rfloor + 1\right) \times x$ ，而接下来我们要包含 k

个 x 的倍数，因此第 k 个倍数是 $\left(\lfloor\frac{l-1}{x}\rfloor+k\right)\times x$ 。同时右端点应当小于第 $k+1$ 个倍数。
因此，**右端点需要满足如下要求**：
$\max\left(\left(\lfloor\frac{l-1}{x}\rfloor+k\right)\times x,l\right)\le r<\left(\lfloor\frac{l-1}{x}\rfloor+k\right)\times x$
而找区间中元素的个数可以先对整个数组排序，再二分查找。

```
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define int long long
#define vi vector<int>
const int N = 3e5 + 10, M = 2e5 + 10, inf = 0x3f3f3f3f, mod = 1e9 + 7;
int n, x, k, ans;
vi nums;

void solve(){
    cin >> n >> x >> k;
    nums.resize(n);
    for(auto &v : nums) cin >> v;
    sort(nums.begin(), nums.end());
    for(auto &v : nums){
        int l = (v - 1) / x + k;
        ans += lower_bound(nums.begin(), nums.end(), (l + 1) * x) -
lower_bound(nums.begin(), nums.end(), max(v, l * x));
    }
    cout << ans << endl;
}

signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T = 1;
    //cin >> T;
    while(T--) solve();
    return 0;
}
```

CF479D. Long Jumps(其实不需要二分)

解析

- **最坏情况最多添加 2 个标记：**
  因为只需要测量两个固定长度 `x` 和 `y`，最坏的情况就是它们都无法测量，只能各打一个点。
- **我们按能否测出 `x` 和 `y` 分为 4 种情况：**

1. 已有标记能测出 `x` 和 `y`，输出 `0`
2. 能测出 `x`，但不能测出 `y`，在某个地方打 `y`
3. 能测出 `y`，但不能测出 `x`，在某个地方打 `x`

4. 都不能测出，我们就尝试：
   方式一：
   是否存在两个点之间的差值是 `x+y`？
   如果有，加上 `x` 就可以测出 `x` 和 `y`
   方式二：
   是否存在两个点差值是 `y - x`？
   如果有，在这两个点的基础上加一个点，可以测出两者
   比如：左点 `-x` 或者 左点 `+y`，但要在合法范围内 `[0, l]`
   否则：直接输出两个点 `[x, y]`

```
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define int long long
#define vi vector<int>
const int N = 3e5 + 10, M = 1e9, inf = 0x3f3f3f3f, mod = 1e9 + 7;
int n, l, x, y;
bool can_x, can_y;

void solve(){
    cin >> n >> l >> x >> y;
    vi nums(n);
    for(auto &v : nums) cin >> v;
    unordered_set<int> st(nums.begin(), nums.end());
    for(int i = 0; i < n; i++){
        if(st.count(nums[i] + x)) can_x = true;
        if(st.count(nums[i] + y)) can_y = true;
    }
    if(can_x && can_y) return void(cout << 0 << endl);
    if(can_x) return void(cout << 1 << endl << y << endl);
    if(can_y) return void(cout << 1 << endl << x << endl);
    for(int i = 0; i < n; i++){
        if(st.count(nums[i] + x + y)) return void(cout << 1 << endl <<
nums[i] + x << endl);
        if(st.count(nums[i] + (y - x))){
            int p1 = nums[i] + y, p2 = nums[i] - x;
            if(p1 >= 0 && p1 <= l) return void(cout << 1 << endl << p1 <<
endl);
            if(p2 >= 0 && p2 <= l) return void(cout << 1 << endl << p2 <<
endl);
        }
    }
    cout << 2 << endl << x << ' ' << y << endl;
}
signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T = 1;
    //cin >> T;
```

```
    while(T--) solve();
    return 0;
}
```

CF1044A. The Tower is Going Home

解析

我们去掉的竖线一定是前几条竖线。假设我们去掉了一部分竖线后，最少应该去掉哪些横线？我们应该去掉完全分离上下两部分的横线，即左端点在 1 右端点不小于最左侧还存在的竖线的横线。

而这些线一旦去掉，则可以到达终点。

于是，枚举去掉的竖线条数，对于每一个竖线条数可以二分找到满足条件的横线数量，最后二者加总取最小值即可。

这里也可以不使用二分，在升序 / 降序枚举去掉的竖线条数时，横线的去掉条数是单调变换的，可以用单指针移动实现。

时间复杂度为 $\mathcal{O}(n \log n + m \log m)$

```
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define int long long
#define vi vector<int>
const int N = 3e5 + 10, M = 1e9, inf = 0x3f3f3f3f, mod = 1e9 + 7;
int n, m, l, r, x, y, ans;

void solve(){
    cin >> n >> m;
    vi vert(n), hori; // 竖直/水平
    for(auto &v : vert) cin >> v;
    for(int i = 1; i <= m; i++){
        cin >> l >> r >> y;
        if(l == 1) hori.emplace_back(r);
    }
    sort(vert.begin(), vert.end());
    sort(hori.begin(), hori.end());
    vert.emplace_back(1e9); // 哨兵，添加边界防止越界
    m = hori.size();
    ans = n + m;
    for(int i = 0; i <= n; i++){ // 枚举清除多少个竖直阻碍
        //二分找到需要清除的水平阻碍的下标
        int pos = lower_bound(hori.begin(), hori.end(), vert[i]) -
hori.begin();
        // m - pos == 需要清除的水平阻碍数量
        ans = min(ans, m - pos + i);
    }
    cout << ans << endl;
}
```

```
signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T = 1;
    //cin >> T;
    while(T--) solve();
    return 0;
}
```

CF729C. Road to Cinema

解析

首先，容量越大越容易到终点。
而成本只跟初始租赁成本有关，因此，我们**只需算出一个最小容量**，再找到所有容量比最小容量大的车，取其成本最小值即可。
这里的**最小容量可以仅仅在已有的可选容量中二分**。
接下来考虑检查使用某个容量的车能否在 `t` 时间内到达终点。
假设某两站之间的距离为 s 。

- 如果 s 超过了容量，则无法到达。
- 先假定这段路都是按慢的速度走的，这样需要的时间是 `2s` ，多出来的油是 `v-s` 。而多出来的每单位油可以使得每单位的路减小一单位的用时，因此最小用时为 `2s-min(v-s,s)` 。
  将每一段的答案求和，即可得到某个容量的最小时间，也就完成了二分中的检查工作。
  时间复杂度为 $O((n+k)logn+klogk)$

```
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define int long long
#define vi vector<int>
#define pii pair<int, int>
const int N = 3e5 + 10, M = 1e9, inf = 0x3f3f3f3f, Inf = INT_MAX, mod =
1e9 + 7;
int n, m, s, t, l, r, ans = Inf;
vi sta;
vector<pii> cars; // 油箱容量/ 价格

bool check(int mid){
    int v = cars[mid].first, tot = 0;
    for(int i = 0; i <= m; i++){
        int dis = sta[i + 1] - sta[i];
        if(dis > v){
            tot = Inf;
            break;
        }else tot += 2 * dis - min(v - dis, dis);
    }
    return tot <= t;
```

```
}

void solve(){
    cin >> n >> m >> s >> t;
    cars.resize(n);
    for(auto &[x, y] : cars) cin >> y >> x;
    sort(cars.begin(), cars.end());
    sta.resize(m + 2);
    for(int i = 1; i <= m; i++) cin >> sta[i];
    sta[0] = 0, sta[m + 1] = s;
    sort(sta.begin(), sta.end());
    l = 0, r = n - 1;
    while(l < r){
        int mid = (l + r) >> 1;
        if(check(mid)) r = mid;
        else l = mid + 1;
    }
    if(check(l)) // 有可能所有车都容量都不行，需要特判
        for(int i = l; i < n; i++)
            ans = min(ans, cars[i].second);
    cout << (ans < Inf ? ans : -1) << endl;
}

signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T = 1;
    //cin >> T;
    while(T--) solve();
    return 0;
}
```

CF492D. Vanya and Computer Game(浮点数 * xy，按比例转化成整数)

解析

提示 1： 直接寻找最后一击是谁的并不容易，更容易找到最后一击是什么时候。

提示 2： 如果我们知道最后一击是什么时候发出的，我们也可以很容易找到最后一击的发出者。

提示 3： 我们不善于使用浮点数，如何将计算转换为整数计算？

正如**提示 1** 所说，我们并没有太好的办法直接判断最后一击的发出者，但是我们可以得到最后一击的时间。

怎么得到呢？我们注意到发出的总攻击次数和时间是正相关的，因此时间越长，发出攻击次数越多，因此可以考虑使用二分查找，找到第一个满足总攻击次数超过所需的时间。

由于此时刚好取到最小时间，因此此刻 **一定发生了攻击，即是最后一击的发生时间** 。

而如何判断此刻发生的攻击是谁发出的呢？我们只需要注意到每个人发起攻击的时间是循环的，即只需查看截止时间是否恰好是其攻击周期的倍数。

而**直接处理会涉及到浮点数**，我们可以将两个角色的攻击时间 **直接乘以 xy** ，那么任何一个攻

击时间都是整数，我们发现这时 `Vanya` 的攻击时间 变成了 `y`， `Vova` 的攻击时间变成了 `x`，所以我们不妨 `swap(x,y)`，这就能方便我们使用二分了。
注意，二分上界应至少为 $10^9 \times 10^6/2 = 10^{15}/2$ 数量级左右，即两人均以最慢攻击频率进行攻击时的总用时。

```
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define int long long
#define vi vector<int>
const int N = 3e5 + 10, M = 1e9, inf = 0x3f3f3f3f, Inf = INT_MAX, mod =
1e9 + 7;
int n, x, y;

void solve(){
    cin >> n;
    int l = 0, r = 1e15;
    while(l < r){
        int mid = (l + r) >> 1;
        if(mid / x + mid / y >= n) r = mid;
        else l = mid + 1;
    }
    if(l % x == 0 && l % y == 0) cout << "Both" << endl;
    else if(l % x == 0) cout << "Vanya" << endl;
    else cout << "Vova" << endl;
}

signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T = 1;
    cin >> T >> x >> y;
    swap(x, y);
    while(T--) solve();
    return 0;
}
```

CF645D. Robot Rapping Results Report(二分 + 拓扑排序)

解析

解法一: O(nlogn)

一旦从某一条边开始，前缀的所有边可以确定唯一的一组拓扑序，那么接下来新增的边不会改变这个拓扑序。

于是，能否确定唯一拓扑序这件事，**关于取的前缀边数是单调的** 。

因此考虑二分，看选取多少条前缀边，接下来只需判断对于一个图能否确定唯一的拓扑序即可。

而唯一的拓扑序可以看：每确定拓扑序中一个位置的节点时，**可选的新点的点集是否大小大**

于 1。如果存在，则可选点集中任选一个均可以作为拓扑序的下一项，因此无法唯一确定拓扑序，此时没有唯一拓扑序。时间复杂度为 $O((n+m)logm)$

```
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
//#define int long long
#define vi vector<int>
const int N = 1e5 + 10, M = 1e5 + 10, inf = 0x3f3f3f3f, mod = 1e9 + 7;
int n, m, a[N], b[N];
int h[N], e[N], ne[N], idx, deg[N], q[N];

void add(int a, int b){
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx++;
    deg[b]++;
}

bool check(int mid){
    memset(h, -1, sizeof h);
    memset(deg, 0, sizeof deg);
    idx = 0;
    for(int i = 1; i <= mid; i++) add(a[i], b[i]);
    int hh = 0, tt = -1;
    for(int i = 1; i <= n; i++) if(!deg[i]) q[++tt] = i;
    while(hh <= tt){
        if(tt - hh + 1 > 1) return false;
        int u = q[hh++];
        for(int i = h[u]; ~i; i = ne[i]){
            int v = e[i];
            if(--deg[v] == 0) q[++tt] = v;
        }
    }
    return tt == n - 1;
}

void solve(){
    cin >> n >> m;
    for(int i = 1; i <= m; i++) cin >> a[i] >> b[i];
    int l = 1, r = m;
    while(l < r){
        int mid = (l + r) >> 1;
        if(check(mid)) r = mid;
        else l = mid + 1;
    }
    cout << (check(l) ? l : -1) << endl;
}
```

```
signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T = 1;
    //cin >> T;
    while(T--) solve();
    return 0;
}
```

解法二: O(n), 松弛的思想

显然，要确定每个机器人的级别，我们要使用**拓扑排序**，不过由于要确定每个机器人的级别，就不能有多个拓扑序，而判断此点只需要查询队列中是否同时存在了 多于 1 个点就行了。

那么怎么确定“最少需要前几场比赛”才能确定呢？我们可以在拓扑排序时**记录确定这个点拓扑序的编号最小边**，使用类似 松弛 或 DP 的操作实现。

```
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
//#define int long long
#define vi vector<int>
const int N = 1e5 + 10, M = 1e5 + 10, inf = 0x3f3f3f3f, Inf = INT_MAX, mod
= 1e9 + 7;
int n, m, a, b, ans;
int h[N], e[N], ne[N], idx, deg[N], tp[N], ged[N];
// tp[N]: 拓扑序编号（第几名）
// ged[N]: 确定当前节点拓扑序的最小边编号
bool flag = true;

void add(int a, int b){
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx++;
    deg[b]++;
}

void topsort(){
    queue<int> q;
    int k = 0; // 目前排第几名
    for(int i = 1; i <= n; i++)
        if(!deg[i]) q.push(i);
    while(!q.empty()){
        if(q.size() > 1) return void(flag = false);
        int u = q.front(); q.pop();
        tp[u] = ++k;
        for(int i = h[u]; ~i; i = ne[i]){
            int v = e[i];
            if(tp[v] == tp[u] + 1) ged[v] = min(ged[v], i + 1);
            if(tp[v] < tp[u] + 1){
                tp[v] = tp[u] + 1;
```

```
                ged[v] = i + 1;
            }
            if(--deg[v] == 0) q.push(v);
        }
    }
}

void solve(){
    memset(h, -1, sizeof h);
    cin >> n >> m;
    for(int i = 1; i <= m; i++){
        cin >> a >> b;
        add(a, b);
    }
    topsort();
    if(!flag) cout << -1 << endl;
    else{
        for(int i = 1; i <= n; i++) ans = max(ans, ged[i]);
        cout << ans << endl;
    }
}

signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T = 1;
    //cin >> T;
    while(T--) solve();
    return 0;
}
```

CF1119D. Frets On Fire(前缀和 + 二分)

解析

有一个比较显然的结论：区间平移答案不变。因此一个 $[l_i, r_i]$ 的询问可以平移为 $[0, r_i - l_i]$ 。

先来看看单独一列的情况：答案显然是 `n` 。接下来我们增加一列，答案会增加多少呢?

1 3 4 5 9 → 1 2 3 4 5 6 9 10

答案增加了 `3` ，答案的增加归功于两部分：

1. 填补了之前数的空隙（比如 `1 3` 之间的空隙和 `5 9` 之间的空隙）。
2. 尾端添加了新数。

   第一部分的贡献可以这样计算：我们先找出最初的所有空隙（定义两个数 `l` , `r` 之间的空隙长度为 `r - l - 1` ），如果增加了 `x` 列（现在有 `x + 1` 列了），那么所有长度小于等于 `x` 的空隙都填平了，而长度大于 `x` 的空隙则长度减少了 `x` 。这个数据可以很轻松地通过二分和前缀和来计算。

   第二部分比较显然，如果**增加**了 `x` 列，则第二部分的增量就是 `x` 。

```cpp
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define int long long
#define vi vector<int>
const int N = 1e5 + 10, M = 1e5 + 10, inf = 0x3f3f3f3f, Inf = INT_MAX, mod
= 1e9 + 7;
int n, m, s[N], d[N], sum[N], l, r;

void solve(){
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> s[i];
    sort(s + 1, s + n + 1);
    for(int i = 1; i < n; i++) d[i] = s[i + 1] - s[i] - 1;
    sort(d + 1, d + n);
    for(int i = 1; i < n; i++) sum[i] = sum[i - 1] + d[i];
    cin >> m;
    while(m--){
        cin >> l >> r;
        r -= l;
        int pos = upper_bound(d + 1, d + n, r) - d;
        cout << sum[pos - 1] + (n - pos + 1) * r + n << " \n"[!m];
    }
}

signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T = 1;
    //cin >> T;
    while(T--) solve();
    return 0;
}
```

CF1073C. Vasya and Robot(前缀和 + 二分)(此题也可用前缀和 + 双指针)(1800)

```cpp
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define int long long
#define vi vector<int>
#define pii pair<int, int>
const int N = 1e5 + 10, M = 1e5 + 10, inf = 0x3f3f3f3f, Inf = INT_MAX, mod
= 1e9 + 7;
int n, x, y, ans = inf;
string s;
map<char, pii> mp;
vi cnt_x, cnt_y;
```

```
bool check(int l, int r){
    return abs(cnt_x[l] + cnt_x[n] - cnt_x[r] - x) + abs(cnt_y[l] +
cnt_y[n] - cnt_y[r] - y) <= r - l;
}

void solve(){
    cin >> n >> s >> x >> y;
    mp['U'] = {0, 1}, mp['D'] = {0, -1}, mp['L'] = {-1, 0}, mp['R'] = {1,
0};
    if(abs(x) + abs(y) - n > 0 || (x + y - n) & 1) return void(cout << -1
<< endl);
    cnt_x.resize(n + 1), cnt_y.resize(n + 1);
    for(int i = 1; i <= n; i++){
        auto &[dx, dy] = mp[s[i - 1]];
        cnt_x[i] = cnt_x[i - 1] + dx;
        cnt_y[i] = cnt_y[i - 1] + dy;
    }
    int l = 0, r = 0;
    while(l <= r){
        while(r < n && !check(l, r)) r++;
        if(!check(l, r)) break;
        ans = min(ans, r - l);
        l++;
    }
    cout << ans << endl;
}

signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T = 1;
    //cin >> T;
    while(T--) solve();
    return 0;
}
```

CF363D. Renting Bikes(1800)

解析

```
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define int long long
#define vi vector<int>
const int N = 1e5 + 10, M = 1e5 + 10, inf = 0x3f3f3f3f, mod = 1e9 + 7;
int n, m, a, b[N], p[N], sum[N], ans;
// 二分买的自行车数量
bool check(int mid){
```

```
    int cnt = 0;
    for(int i = 1; i <= mid; i++){
        int d = p[i] - b[n - mid + i];
        if(d > 0) cnt += d;
    }
    return cnt <= a;
}

void solve(){
    cin >> n >> m >> a;
    for(int i = 1; i <= n; i++) cin >> b[i];
    for(int i = 1; i <= m; i++) cin >> p[i];
    sort(b + 1, b + n + 1);
    sort(p + 1, p + m + 1);
    int l = 0, r = min(n, m);
    while(l < r){
        int mid = (l + r + 1) >> 1;
        if(check(mid)) l = mid;
        else r = mid - 1;
    }
    for(int i = 1; i <= l; i++) ans += p[i];
    ans -= a;
    // 注意可能不需要自己掏钱，所以 ans 要取 max(ans, 0)
    cout << l << ' ' << max(ans, 0ll) << endl;
}

signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T = 1;
    //cin >> T;
    while(T--) solve();
    return 0;
}
```

CF325B. Stadium and Games(1800)

解析

提示 1： 如果一开始有 k 个球员，如何计算总共需要的人数?

提示 2： 根据上面得出的式子，怎么快速地进行逆向计算?

首先，我们分析正向计算，即一开始有 k 个球员时，要比赛多少次。

如果 k 是偶数，则比 $k/2$ 轮剩下 $k/2$ 个人；否则，直接单循环赛，比赛 $\frac{k(k-1)}{2}$ 场。

于是我们只需要知道 k 除以几次 2 会变成奇数即可。

不妨设 $k=2^t\times m$ ，其中 m 为奇数。则在变成 m 个队伍之前进行的比赛数量是 $(2^t-1)\times m$ 场，因为每场比赛淘汰一个队伍，总淘汰的队伍数量是这么多（也可以使用等比数列来推）。

而后续单循环赛 $\frac{m(m-1)}{2}$ 场，总场数为 $(2^t-1)\times m+\frac{m(m-1)}{2}$ 。

而要让这个总场数等于目标 n ，如何找到可行的 m,t 呢?

由于 2^t 增长很快，因此直接枚举 t ，发现 $t\geq 60$ 时， 2^t-1 已经超过 10^{18} ，因此只需枚举 t

。而接下来的式子关于 m 是一个单调递增的二次函数，可以二分求解根（求根公式容易出现精度问题），于是可以找到对应的 m 。
注意，求解后需要验是否是方程的根，并验证 m 是奇数。
时间复杂度为 $\mathcal{O}(\log^2 n)$ 。
注意这里如果使用 C++ 写代码，很容易出现爆 `long long` 的问题，建议二分的过程中想清楚，或直接偷懒使用 `__int128_t` 等。

```
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define int long long
#define vi vector<int>
#define pii pair<int, int>
const int N = 1e5 + 10, M = 1e5 + 10, inf = 0x3f3f3f3f, mod = 1e9 + 7;
int n;
bool flag;

void solve(){
    cin >> n;
    for(int i = 0; i < 60; i++){
        int v = 1ll << i;
        int l = 0, r = min(1ll << 31, n / max(v - 1, 1ll));
        while(l < r){
            int mid = (l + r + 1) >> 1;
            if((v - 1) * mid + mid * (mid - 1) / 2 <= n) l = mid;
            else r = mid - 1;
        }
        if((v - 1) * l + l * (l - 1) / 2 == n && (l & 1)){
            cout << l * v << endl;
            flag = true;
        }
    }
    if(!flag) cout << -1 << endl;
}

signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T = 1;
    //cin >> T;
    while(T--) solve();
    return 0;
}
```

CF1227D2. Optimal Subsequences (Hard Version)(树状数组 + 二分)(1800)(需要复习！！！)

解析

提示 1： 长度为 k 和长度为 $k+1$ 的序列差别不大。

提示 2： 考虑离线查询，按照序列长度的升序进行解决。

首先，我们看选取长度为 k 的序列应该选取哪些元素。

我们只需选取最大的 k 个元素。在相等的情况下选最前面的几个。

这相当于按照从大到小的逻辑选取元素，且相等的情况下，初始下标越小的越优先选取。

于是只需将下标按照这个逻辑稳定排序，前 k 项即为长度为 k 的序列的元素。

因此，长度为 $k+1$ 的序列相当于长度为 k 的序列新增一项。

接下来考虑查询。我们可以离线查询，对于不同的长度 k 分别解决问题。如果按照 k 升序考虑，则我们相当于不断新增元素下标，再查找其中第 idx 个。

这件事可以使用树状数组解决。我们在目前选择的下标位置标记 1，其他位置标记 0，则我们只需要使用树状数组二分找到第一个不小于 idx 的位置即可。找到下标后，对应的数值也就得到确定了。

```cpp
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define int long long
#define vi vector<int>
#define arr3 array<int, 3>
#define lowbit(x) (x & -x)
const int N = 2e5 + 10, M = 1e5 + 10, inf = 0x3f3f3f3f, mod = 1e9 + 7;
int n, m, a[N], tr[N];
vector<arr3> queries;
vi order;

void add(int x, int k){
    for(int i = x; i < N; i += lowbit(i)) tr[i] += k;
}

int sum(int x){
    int ans = 0;
    for(int i = x; i; i -= lowbit(i)) ans += tr[i];
    return ans;
}
/*
int find(int k){ // 树状数组上跳跃二分，比较快
    int pos = 0;
    for(int i = 1 << 18; i; i >>= 1){
        if(pos + i <= n && tr[pos + i] < k){
            k -= tr[pos + i];
            pos += i;
        }
    }
    return pos + 1;
}
*/
```

```
int find(int k){
    int l = 1, r = n;
    while(l < r){
        int mid = (l + r) >> 1;
        if(sum(mid) >= k) r = mid;
        else l = mid + 1;
    }
    return l;
}

void solve(){
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i];
    cin >> m;
    queries.resize(m);
    for(int i = 0; i < m; i++){
        int k, pos; cin >> k >> pos;
        queries[i] = {k, pos, i};
    }
    order.resize(n);
    for(int i = 0; i < n; i++) order[i] = i + 1;
    sort(order.begin(), order.end(), [&](int i, int j){
        if(a[i] != a[j]) return a[i] > a[j];
        return i < j;
    });
    sort(queries.begin(), queries.end(), [&](const arr3 &q1, const arr3
&q2){
        return q1[0] < q2[0];
    });
    vi ans(m, 0);
    int added = 0;
    for(auto &q : queries){
        while(added < q[0]){
            int idx = order[added];
            add(idx, 1);
            added++;
        }
        int pos = find(q[1]);
        ans[q[2]] = a[pos];
    }
    for(auto &x : ans) cout << x << endl;
}

signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T = 1;
    //cin >> T;
    while(T--) solve();
    return 0;
}
```

CF847E. Packmen(贪心 + 二分答案)(1800)

解析

首先，左边的 asterisks 应该分给左边的 Packmen ，每个 Packmen 都应该顺序分配某个区间内的 asterisks 。因为如果不是连续的区间，可以将中间的 asterisk 也分给对应的 Packmen（他一定已经经过了对应的位置），而如果左右不对应，交换两者后所需的时间一定缩短。
于是我们从左到右考虑所有 Packmen 吃哪些 asterisks 。假设现在是从第 idx 个 asterisk 开始。
如果目前离它的距离太远，则无法吃掉任何一个。
否则，我们应当选择吃掉它。此时，这一个 Packmen 吃的 asterisks 越多越好，因此需要尽可能往右吃。我们能吃到的最靠右的 asterisk 是哪一个呢?
第 idx 个 asterisk 告诉了我们必须往左边走的步数。在这个基础上，我们相较于原位置最多向右多少步呢？只需分类讨论是先往左再往右还是先往右再往左即可。
先往左再往右时，则 $left \times 2 + right \leq t$ ，可得到对应的 $right$ 。
先往右再往左时，则 $right \times 2 + left \leq t$ ，也可得到对应的 $right$ 。
根据两位置的最大值即可得到目前能吃到的最多的 asterisks 。

```
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define int long long
#define vi vector<int>
#define arr3 array<int, 3>
#define lowbit(x) (x & -x)
const int N = 2e5 + 10, M = 1e5 + 10, inf = 0x3f3f3f3f, mod = 1e9 + 7;
int n;
string s;
vi ast, men;

bool check(int mid, int k){
    int cnt = 0; // 记录已被吃掉的星号的数量
    for(auto &p : men){
        if(abs(ast[cnt] - p) > mid) continue;
        int left = max(0ll, p - ast[cnt]); // 向左吃,将左边的全部吃了
        // 因为必须把左边的吃了，但是可能先往左边走，也可能先往右边走
        // 初始方向会影响能吃到的右边的距离，要取 max
        int right = p + max((mid - left) >> 1, mid - left * 2);
        while(cnt < k && ast[cnt] <= right) cnt++; // 更新已吃掉的星号数量
        if(cnt == k) return true;  // 如果所有星号都已吃完，返回 true
    }
    return false;
}

void solve(){
    cin >> n >> s;
    for(int i = 0; i < n; i++){
```

```
        if(s[i] == '*') ast.emplace_back(i);
        else if(s[i] == 'P') men.emplace_back(i);
    }
    int l = 0, r = 2 * n, k = ast.size();
    while(l < r){
        int mid = (l + r) >> 1;
        if(check(mid, k)) r = mid;
        else l = mid + 1;
    }
    cout << l << endl;
}

signed main(){
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    int T = 1;
    //cin >> T;
    while(T--) solve();
    return 0;
}
```