前言

~~因为深水区的题很重，没写完，明天之前一定写完。先发点写好的题解和std出来占坑。~~

23出了一场，24 qcjj又找我出，然后就出了。
题目难度，前期温暖，中后断崖~~（参考了算法竞赛入门经典的难度）~~。最后几题很重，折磨自己，反噬。

赛中，答疑子数组最多了，没料到，应该在题目说一下的......其它地方可能有不明显的题意（尽力写题目描述了），都尽量回复了。

已修好的锅：
C冬眠 样例描述错误

待修补的锅：
K方块掉落 没取模过了

A、柠檬可乐

代码：

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int a,b,k;
    cin>>a>>b>>k;
    if(a>=k*b)cout<<"good";
    else cout<<"bad";
    return 0;
}
```

B、左右互博

标题指，背景的人物是同一个人的两个马甲。

从最终状态考虑：最终每一堆都变成1。
因为 $2 \le y \le x$，所以 $1 \le \lfloor \frac{x}{y} \rfloor < x$。
所以只要有一堆大于 1，还没有结束，并且可以从这一堆，分出两堆，其中一堆是 1 个。

因此只要统计一下次数，并且看一下奇偶性即可。

代码：

```cpp
#include <bits/stdc++.h>
using namespace std;
using LL = long long;
const int N = 3 + 2e5;
int n, a[N];
int main() {
    cin >> n;
    for (int i = 1; i <= n; ++i) {
        cin >> a[i];
    }
    LL sum = 0;
    for (int i = 1; i <= n; ++i) {
        sum += a[i] - 1;
    }
    if (sum % 2) {
        cout << "gui" << endl;
    } else {
        cout << "sweet" << endl;
    }
    return 0;
}
```

C、冬眠

模拟题。

代码：

```cpp
#include <bits/stdc++.h>
using namespace std;
using LL = long long;
using PII = pair<int, int>;
const int N = 3 + 100;
int n, m, x, y, p, q;
char s[N][N];
int main() {
    cin >> n >> m >> x >> y;
    for (int i = 1; i <= n; ++i) {
        cin >> s[i] + 1;
    }
    cin >> p >> q;
    vector<PII> seq(q);
    for (auto& [op, z] : seq) {
        cin >> op >> z;
    }
    while (p--) {
        for (auto& [op, z] : seq) {
            if (op == 1) {
                char c = s[z][m];
                for (int i = m; i > 1; --i) {
                    s[z][i] = s[z][i - 1];
                }
                s[z][1] = c;
            } else {
                char c = s[n][z];
                for (int i = n; i > 1; --i) {
                    s[i][z] = s[i - 1][z];
                }
                s[1][z] = c;
            }
        }
    }
    cout << s[x][y] << endl;
    return 0;
}
```

D、守恒

知识点：最大公约数 gcd

标题指，+1-1操作后，数组的总和不变。

对于 $n=1$ 时，因为没法操作，所以答案是 1。可能需要特判。

现在考虑 $n \geq 2$ 的情况。
还是从最终情况考虑。假设数组的 gcd 是 g。那么数组里面至少有一个数是 g，并且最小是 g。其它数如果不是 g，那就是 g 的倍数。
因此我们想要知道一个数能不能成为数组的最大公约数，要满足以下条件（假设数组总和是 sum）：

1. sum 是 g 的倍数。因为所有数都是 g 的倍数。
2. $\frac{sum}{g} \geq n$。如果小于，那么最小就不是 g。

因此求出 sum，根号时间复杂度，找一下 sum 的因数判一下就行了。

代码：

```cpp
#include <bits/stdc++.h>
using namespace std;
using LL = long long;
const int N = 3 + 2e5;
int n, a[N];
int main() {
    cin >> n;
    for (int i = 1; i <= n; ++i) {
        cin >> a[i];
    }
    if (n == 1) { // n=1时以下代码不是输出1，需特判
        cout << 1 << endl;
        return 0;
    }
    LL sum = 0;
    for (int i = 1; i <= n; ++i) {
        sum += a[i];
    }
    auto check = [&](LL v) {
        // 每个数都是v的倍数，看看够不够n个数
        if (sum / v >= n) {
            return 1;
        }
        return 0;
    };
    int ans = 0;
    for (LL i = 1; i * i <= sum; ++i) {
        if (sum % i == 0) {
            if (check(i)) {
                ++ans;
            }
            if (i != sum / i && check(sum / i)) {
                ++ans;
            }
        }
    }
    cout << ans << endl;
    return 0;
}
```

E、漂亮数组

知识点：贪心/dp、set/map
预定是贪心题。但是好像验题人里面，只有一个写贪心，其他人都写的 dp。

求一个子数组的和，使用前缀和 $O(1)$ 求出来。例如子数组 L 到 R 的和是 $pre_R - pre_{L-1}$。

贪心做法：
枚举 i 从 1 到 n，想办法让 i 成右端点，在 i 的左边找到一个左端点 j 满足 $(pre_i - pre_{j-1})\%k = 0$。
用一个 set 存左边的 j，先往 set 插一个 $pre_{i-1}\%k$。
如果 set里面存在一个值是 pre_i 的元素，就说明存在一个 pre_{j-1} 使得 i 能成为右端点，然后把 set 清空。

清空 set 的感性理解：
set 里面的元素下标有大于 j 也有小于 j。但是这些元素，不管哪一个作为左端点，需要的右端点下标都比 i 大。如果选了这其中的元素作为左端点，那么必定不能选 $[j, i]$ 这个子数组，而且不知道啥时候才能找到合适的右端点。所以不如先选择 $[j, i]$ 这个子数组。

代码：

```cpp
#include <bits/stdc++.h>
using namespace std;
using LL = long long;
const int N = 3 + 2e5;
int n, k, a[N];
LL pre[N];
int main() {
    cin >> n >> k;
    for (int i = 1; i <= n; ++i) {
        cin >> a[i];
    }
    for (int i = 1; i <= n; ++i) {
        pre[i] = pre[i - 1] + a[i];
    }
    int ans = 0;
    set<LL> st;
    for (int i = 1; i <= n; ++i) {
        st.insert(pre[i - 1] % k);
        if (st.count(pre[i] % k)) {
            st.clear();
            st.insert(pre[i - 1]);
            ++ans;
        }
    }
    cout << ans << endl;
    return 0;
}
```

F、来点每日一题

知识点：dp

dp_i 表示恰好选择了 6 的倍数个数，并且强制选了第 i 个数，最大的分数是dp_i。

合法转移显然有下标 j 到下标 i 选 6 个数，然后 dp_{j-1} 和 6 个数的分数给 dp_i 取一个 max。这就 n^2 了，可以接受。

但是，6 个数的分数怎么办？
对于这 6 个数，前 5 个数每个数开两个 set，分别表示选某个数时最大值/最小值。最后只留下一个数，表示对应的最值。

- 为啥要最小值呢？因为有乘法，两个负数相乘就变正数，如果两个负数绝对值很大，那就结果很大。
- 最值只有一个数，为啥用set呢？为了避免一开始没得选、或没法选时，一个数都没有，感觉用 set 比较方便。

具体操作看下代码。

代码：

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 3 + 1e3;
int n, a[N], dp[N];
int main() {
    cin >> n;
    for (int i = 1; i <= n; ++i) {
        cin >> a[i];
    }
    auto fixMin = [&](set<int>& st) {
        while (st.size() > 1) {
            st.erase(--st.end());
        }
    };
    auto fixMax = [&](set<int>& st) {
        while (st.size() > 1) {
            st.erase(st.begin());
        }
    };
    for (int i = 1; i <= n; ++i) {
        set<int> mn1, mx1;
        set<int> mn2, mx2;
        set<int> mn3, mx3;
        set<int> mn4, mx4;
        set<int> mn5, mx5;
        for (int j = i; j <= n; ++j) {
            // 注意for的顺序不能变
            // 比如 mn5 依赖的是 i到j-1的 mn4和mx4
            // 如果先修改mn4和mx4，mn4和mx4就变成i到j的

            for (auto& k : mn5) {
                dp[j] = max(dp[j], dp[i - 1] + k - a[j]);
            }
            for (auto& k : mx5) {
                dp[j] = max(dp[j], dp[i - 1] + k - a[j]);
            }
            for (auto& k : mn4) {
                mn5.insert(k * a[j]);
                mx5.insert(k * a[j]);
```

```
            }
            for (auto& k : mx4) {
                mn5.insert(k * a[j]);
                mx5.insert(k * a[j]);
            }
            for (auto& k : mn3) {
                mn4.insert(k - a[j]);
                mx4.insert(k - a[j]);
            }
            for (auto& k : mx3) {
                mn4.insert(k - a[j]);
                mx4.insert(k - a[j]);
            }
            for (auto& k : mn2) {
                mn3.insert(k * a[j]);
                mx3.insert(k * a[j]);
            }
            for (auto& k : mx2) {
                mn3.insert(k * a[j]);
                mx3.insert(k * a[j]);
            }
            for (auto& k : mn1) {
                mn2.insert(k - a[j]);
                mx2.insert(k - a[j]);
            }
            for (auto& k : mx1) {
                mn2.insert(k - a[j]);
                mx2.insert(k - a[j]);
            }
            mn1.insert(a[j]);
            mx1.insert(a[j]);
            fixMin(mn1);
            fixMin(mn2);
            fixMin(mn3);
            fixMin(mn4);
            fixMin(mn5);
            fixMax(mx1);
            fixMax(mx2);
            fixMax(mx3);
```

```
            fixMax(mx4);
            fixMax(mx5);
        }
    }
    cout << *max_element(dp + 1, dp + n + 1);
    return 0;
}
```

G、数三角形（easy）

知识点：？？大概是批量思想吧

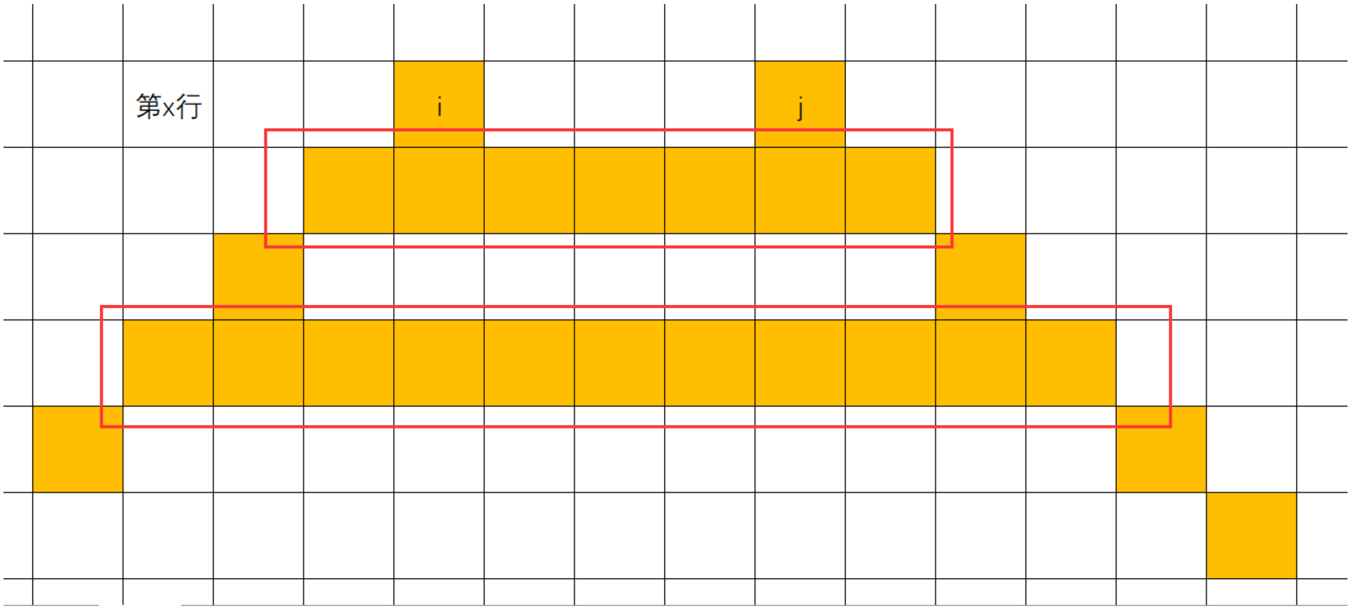
$O(n^5)$
$O(n^2)$ 枚举最头顶的那个点。$O(n)$ 去枚举左边和右边，没有星号就break。再 $O(n)$看看底边是不是都是星号。总共 $O(n^5)$。

$O(n^4)$
对每一行做一个前缀和，$O(1)$判一下底边星号的数量。底边 $O(n)$ 变成 $O(1)$。

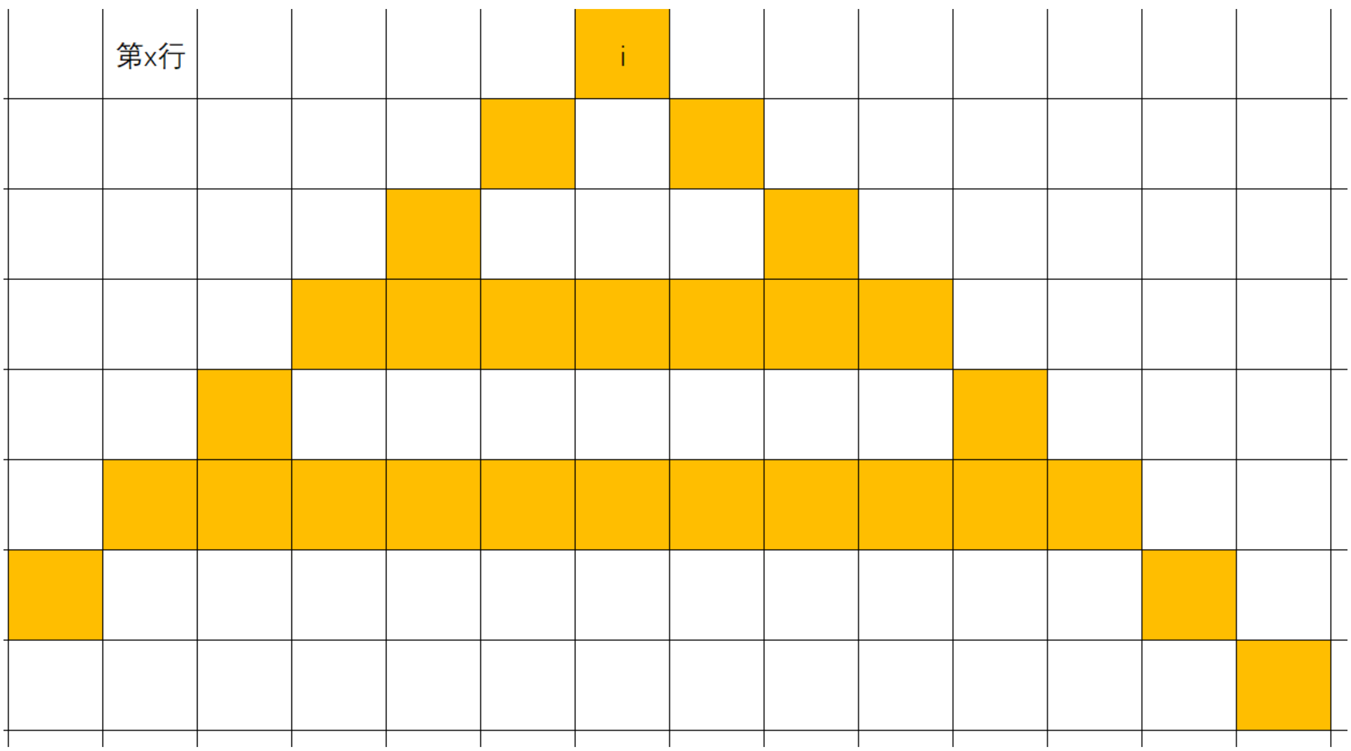
$O(n^3)$
对于本题就差不多了。

为了简单表示，(x, y) 意思是第 x 行第 y 列。

d 数组表示：$d_{x,i,j}$ 表示 (x, i) 和 (x, j) 都是星号，并且 (x, i) 和 (x, j) 能作为三角形左右两条边**目前**最上面的点，$d_{x,i,j}$就表示这个数目。
如下图，$d_{x,i,j}$ 就是 2，因为红色框中的两行能作为底边。（橙色格子代表星号，白色代表点）



到 $d_{x,i,i}$ 的时候就可以统计答案了。
如下图，$d_{x,i,i}$ 就是 2，可以看到图中 (x,i) 为顶点的两个满足要求的三角形，满足要求的底边和腰。



怎么计算？
显然上面的 $d_{x,i,j}$ 依赖下面的，得从下往上算，$O(n)$。
对于每一行，可以 $O(n^2)$ 搞出底边。$O(n^2)$ 算出下面行传给当前的 $d_{x,i,j}$。

```
#include <bits/stdc++.h>
using namespace std;
using LL = long long;
using PII = pair<int, int>;
const int N = 3 + 500;
int n, m;
char s[N][N];
int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; ++i) {
        cin >> s[i] + 1;
    }
    int ans = 0;

    // 500的3次方比较大，选择滚动数组
    vector<vector<int>> d0(m + 2, vector<int>(m + 2));

    for (int i = n; i >= 1; --i) {
        vector<vector<int>> d1(m + 2, vector<int>(m + 2));

        for (int j = 1; j <= m; ++j) {
            if (s[i][j] == '.') {
                // 不可能有答案了
                continue;
            }

            // 当前行的底边
            for (int k = j + 2; k <= m; k += 2) {
                if (s[i][k] == '.' || s[i][k - 1] == '.') {
                    // 被 点 中断了
                    break;
                }
                ++d1[j][k];
            }
            for (int k = j; k <= m; k += 2) {
                if (s[i][k] == '.') {
                    // 右边得是 星号
                    continue;
                }
```

```
                // 下面行传上来的
                d1[j][k] += d0[j - 1][k + 1];
            }
            ans += d1[j][j]; // 统计答案
        }
        swap(d0, d1); // O(1)交换


        // d1出作用域就被释放了
    }
    cout << ans << endl;
    return 0;
}
```

H、数三角形（hard）

知识点：树状数组
$O(n^2logn)$

可以考虑从上往下看，这一步就 $O(n^2)$ 了。剩下底边，就得想办法加速一下它了。

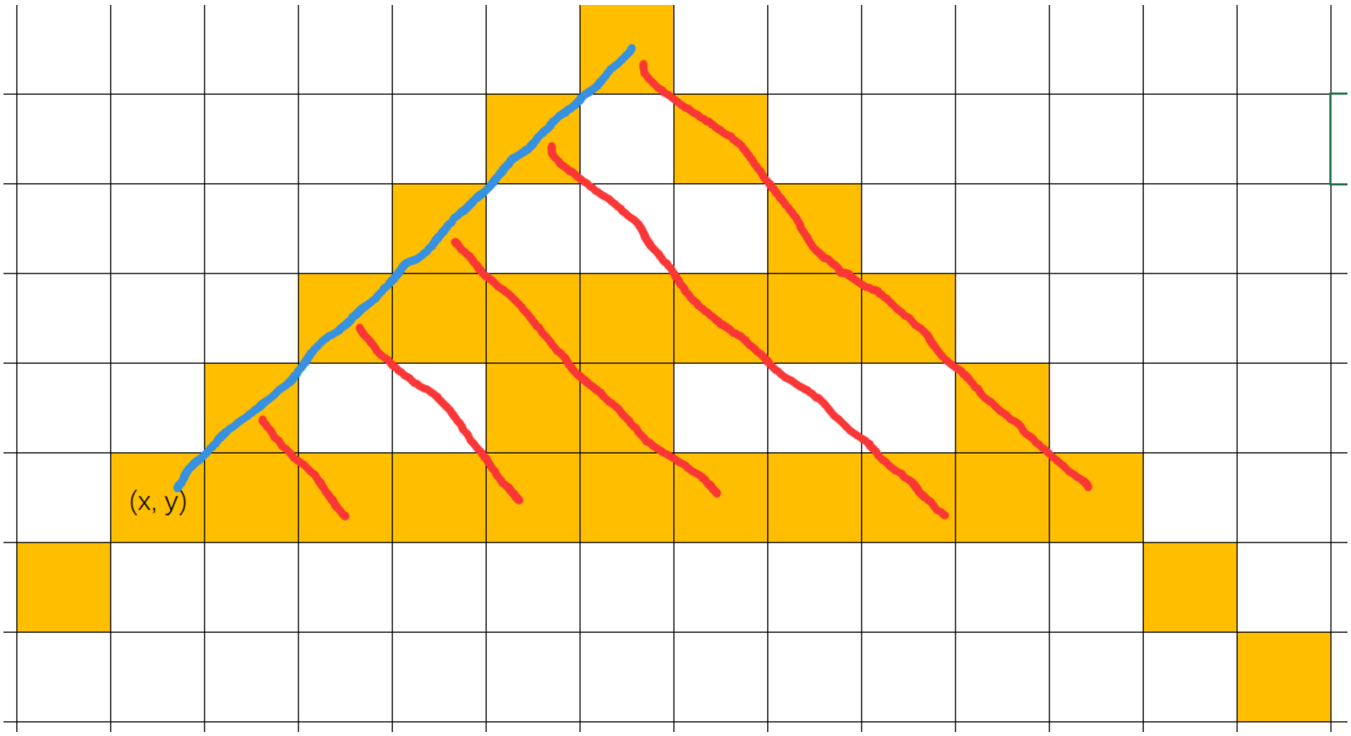
思考一下左边的腰，和右边的腰，它们的特点，
然后利用之，加速计算。

如下图，蓝色线划过的格子的左腰，以点 (x,y) 结束，它**可能**的右腰就是红色线。
这到底可能不可能，得想办法计算。
这个相对比较简单，可以找代码实现中的 $d1,d2$ 数组。

然后，这个蓝色的左腰，是可以算出**最大**长度的，它对应的右腰，形成**一段区间**（都隔着一个）。假设这左腰长度是 $maxD$，那么它最远的右腰就到了 $(x,y+maxD\times 2-2)$，通过两点的中点公式算出来。

对于 $[y,y+maxD\times 2-2]$ 这段区间不可能for过去处理，得用数据结构，下面就是用树状数组。
给 y 这个点加 1，到 $y+maxD\times 2-2$ 结束后 给 y 这个点 -1。

这样就考虑完了左腰的作用了。该该考虑下右腰了。
其实有点类似左腰，假设右腰最大长度是 $maxD$，最远的左腰就是 $(x, y - maxD \times 2 + 2)$
。
在树状数组 这段区间$[y - maxD \times 2 + 2, y]$ 就是满足要求的左腰的数量，求个和即可。



```cpp
#include <bits/stdc++.h>
using namespace std;
using LL = long long;
using PII = pair<int, int>;
const int N = 3 + 3000;
int n, m;
char s[N][N];
// d1[i][j] 表示 第i行第j列的最长左腰
// d2[i][j] 表示 第i行第j列的最长右腰
int d1[N][N], d2[N][N];
int tr[N];
LL ans;
vector<int> ve[N * 2];
void add(int p, int v) {
    for (; p <= m; p += p & -p) {
        tr[p] += v;
    }
}
int ask(int p) {
    if (p <= 0) {
        // 好像会传负数，负数会死循环
        return 0;
    }
    int ans = 0;
    for (; p >= 1; p -= p & -p) {
        ans += tr[p];
    }
    return ans;
}
void solve(int row, int left, int right) {
    // 处理 第row行 第left列 到 第right列
    int mx = 0, i;
    for (i = left; i <= right; i += 2) {
        // d1[row][i]就是 (row,i)的最长左腰
        int L = i, R = L + d1[row][i] * 2 - 2;
        mx = max(mx, R);
        ve[R].push_back(L); // 到最远的右腰，计算完就撤销这个+1
        add(i, 1);
        R = i;
```

```cpp
            L = R - (d2[row][i] * 2 - 2);
            ans += ask(i) - ask(L - 1);
            for (auto& j : ve[i]) { // 撤销
                add(j, -1);
            }
            ve[i].clear();
        }
        for (; i <= mx; i += 2) { // 出去了，没清空到
            for (auto& j : ve[i]) {
                add(j, -1);
            }
            ve[i].clear();
        }

        // 因为有间隔
        mx = 0;
        for (i = left + 1; i <= right; i += 2) {
            int L = i, R = L + d1[row][i] * 2 - 2;
            mx = max(mx, R);
            ve[R].push_back(L);
            add(i, 1);
            R = i;
            L = R - (d2[row][i] * 2 - 2);
            ans += ask(i) - ask(L - 1);
            for (auto& j : ve[i]) {
                add(j, -1);
            }
            ve[i].clear();
        }
        for (; i <= mx; i += 2) {
            for (auto& j : ve[i]) {
                add(j, -1);
            }
            ve[i].clear();
        }
    }
    int main() {
        ios::sync_with_stdio(false);
        cin.tie(0);
```

```cpp
cin >> n >> m;
for (int i = 1; i <= n; ++i) {
    cin >> s[i] + 1;
}
for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= m; ++j) {
        if (s[i][j] == '*') {
            // n^2预处理一下
            d1[i][j] = d1[i - 1][j + 1] + 1;
            d2[i][j] = d2[i - 1][j - 1] + 1;
        }
    }
}
for (int i = 1; i <= n; ++i) {
    // 处理第i行
    for (int j = 1; j <= m; ++j) {
        // 分段，跳过点
        if (s[i][j] == '.') {
            continue;
        }
        int k = j;
        while (k + 1 <= m && s[i][k + 1] == '*') {
            ++k;
        }

        // 第i行的j列到k列都是星号
        solve(i, j, k);
        j = k;
    }
}

// 会多算不合法的，减去
for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= m; ++j) {
        if (s[i][j] == '*') {
            ans -= 1;
        }
    }
}
```

```
    cout << ans << endl;
    return 0;
}
```

I、回头

知识点：最短路（dijkstra）

标题指，最短路不回头（没有负环的情况）。

从题目给的技能描述，是修改走到的点 y 的一条出边，因为都是正数边权，所以不可能回头走（回头最短路会增加，更亏）。所以 x 走到 y，尽量挑 y 比较小的出边。
x 走到 y，技能可以延后释放。就是在 x 点的时候，可以先尝试把费用（边权）欠着先，走到 y 点，要离开的 y 点的时候再把 y 的某一条出边改到 x 到 y 的边。

最短路数组 d 的定义：

1. $d_{x,0}$ 表示现在在点 x，没有欠着费用
2. $d_{x,1}$ 表示现在在点 x，欠着费用

预处理，对每个点的所有边排序。排序的是下标。代码中需要判断是不是同一条边，用的是比较下标。

对于点 x 要走到点 y，要分几种情况：

1. 来到 x 没有欠费，也就是 $d_{x,0}$。不欠费走到 y，就是用 $d_{x,0}+w$ 更新 $d_{y,0}$
2. 来到 x 没有欠费，欠费走到 y，就是用 $d_{x,0}$ 更新 $d_{y,1}$
3. 来到 x 欠费，就是 $d_{x,1}$，欠费走到 y。
4. 来到 x 欠费，不欠费走到 y。
   对于3、4点，更新需要看看从 x 走到 y 的是 x的哪一条出边。具体更新比较多，还请看一下代码。

最后，答案的可能有

1. $d_{n,0}$
2. $d_{n,1}$ 加上点 n 最短的出边。

```cpp
#include <bits/stdc++.h>
using namespace std;
using LL = long long;
using PII = pair<int, int>;
const int N = 3 + 2e5;
struct Node {
    LL d;
    int x, z;
    bool operator<(const Node& other) const { return d > other.d; }
};
int n, m;
vector<PII> edge[N];
vector<int> idx[N];
LL d[N][2];
int vis[N][2];
priority_queue<Node> q;
int main() {
    cin >> n >> m;
    while (m--) {
        int u, v, w;
        cin >> u >> v >> w;
        edge[u].push_back({v, w});
    }
    for (int i = 1; i <= n; ++i) {
        // 存下标
        idx[i].resize(edge[i].size());
        for (int j = 0; j < idx[i].size(); ++j) {
            idx[i][j] = j;
        }
        // 根据边权排序
        sort(idx[i].begin(), idx[i].end(), [&](const int& a, const int& b) {
            return edge[i][a].second < edge[i][b].second;
        });
        while (idx[i].size() > 2) {
            // 操作时，如果走的不是最短的，那肯定操作最短的边
            // 如果走的是最短的，那肯定操作次短的边
            // 因此可以只最短保留2条边
            idx[i].pop_back();
        }
```

```
}
memset(d, 0x3f, sizeof(d));
auto update = [&](int y, int z, LL v) {
    // 如果写成 if，v参数太长了，因此写成函数
    if (d[y][z] > v) {
        d[y][z] = v;
        q.push({d[y][z], y, z});
    }
};
update(1, 0, 0); // 在起点肯定没有欠
while (q.size()) {
    auto [dd, x, z] = q.top();
    /*
        和以下代码一样效果：
        auto node = q.top();
        LL dd = node.d;
        int x = node.x;
        int z = node.z;
    */
    q.pop();
    if (vis[x][z]) {
        continue;
    }
    vis[x][z] = 1;
    if (z == 0) {
        // 来到 x不欠费
        for (int i = 0; i < edge[x].size(); ++i) {
            auto [y, w] = edge[x][i];
            /*
                和以下代码一样效果：
                auto e = edge[x][i];
                int y = e.first, w = e.second;
            */
            update(y, 0, d[x][0] + w);
            update(y, 1, d[x][0]);
        }
    } else {
        // 来到 x 欠费
        if (idx[x].size() == 2) {
```

```
                // 前面操作成只保留最短的两条边

                // 没有idx[x].size() == 1，因为这个情况是走回去，没必要判
                // 而且对于下面代码会越界

                for (int i = 0; i < edge[x].size(); ++i) {
                    auto [y, w] = edge[x][i];
                    if (i != idx[x][0]) {
                        // 走去y不是idx[x][0]这最短边
                        // 拿idx[x][0]抵消欠费

                        // 不欠费走到 y
                        update(y, 0, d[x][1] + w + edge[x][idx[x][0]].second);
                        // 欠费走到 y
                        update(y, 1, d[x][1] + edge[x][idx[x][0]].second);
                    } else {
                        // 走去y是idx[x][0]这最短边
                        // 拿idx[x][1]次短边，抵消欠费

                        // 不欠费走到 y
                        update(y, 0, d[x][1] + w + edge[x][idx[x][1]].second);
                        // 欠费走到 y
                        update(y, 1, d[x][1] + edge[x][idx[x][1]].second);
                    }
                }
            }
        }
    }
    LL ans = d[n][0];
    if (idx[n].size() >= 1) {
        // 终点，该把费用清一下
        ans = min(ans, d[n][1] + edge[n][idx[n][0]].second);
    }
    if (ans > 1e18) { // 没有答案
        ans = -1;
    }
    cout << ans << endl;
    return 0;
}
```

J、画直线

知识点：状压dp，叉积

比较关键的点：从一个已经被修改颜色的点画直线，使得没被染色的点被染色，这样任意时候最多有一种颜色。 而不是像样例的做法，某个时候搞出两种颜色，最终变成一种。

对于染色过程，是个增量的过程。并且题目给的 n 只有 20，可以往状态压缩想去。

状压之前，首先得搞点有哪些直线？
两点就可以确定一条直线，因此可以用 $d_{i,j}$ 存：一定过 第 i 个点和第 j 个点，还有某些点，这些点在 第 i 个点和第 j 个点的直线上，$d_{i,j}$ 就表示这些点的二进制状态。

然后就可以不断加入直线，加到 n 个点就结束。这个加直线过程（加点过程）就是 dp 的过程。
dp_i 表示： i 的二进制下，第 j 位如果是 0，那么第 j 位就还没被染色；第 j 位如果是 1，那么第 j 位就被染色被。 dp_i 是状态 i 需要的最少直线数量。
转移，看一下代码。

```cpp
#include <bits/stdc++.h>
using namespace std;
using LL = long long;
using PLL = pair<LL, LL>;
const int N = 3 + 20;
PLL a[N];
int n;
int d[N][N];
int dp[(1 << 20) + 100];
LL cross(const PLL& a, const PLL& b) {
    return a.first * b.second - a.second * b.first;
}
PLL operator-(const PLL& a, const PLL& b) {
    return {a.first - b.first, a.second - b.second};
}
int main() {
    cin >> n;
    for (int i = 0; i < n; ++i) {
        cin >> a[i].first >> a[i].second;
    }

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (i == j) {
                // i=j时叉积算出来是0，显然不对
                continue;
            }
            for (int k = 0; k < n; ++k) {
                // 叉积判第 k 个点是不是在 i和j 的直线上
                if (cross(a[i] - a[j], a[j] - a[k]) == 0) {
                    d[i][j] |= 1 << k;
                }
            }
        }
    }
    memset(dp, 0x3f, sizeof(dp));
    for (int i = 0; i < n; ++i) {
        dp[1 << i] = 1; // 解决可能n=1的情况
        for (int j = 0; j < n; ++j) {
```

```
            if (i == j) {
                continue;
            }
            dp[d[i][j]] = 1;
        }
    }
    for (int i = 1; i < 1 << n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (i >> j & 1) {
                // 第 j 个点被染色
                for (int k = 0; k < n; ++k) {
                    // 第 k 个点没被染色
                    // j和k直线 如果被染色过，再染一遍会使答案更大，所以没判
                    if (j == k) {
                        continue;
                    }
                    // 选择 j和k之间的直线染色
                    dp[i | d[j][k]] = min(dp[i | d[j][k]], dp[i] + 1);
                }
            }
        }
    }
    cout << dp[(1 << n) - 1] << endl;
    return 0;
}
```

K、方块掉落

知识点：线段树

数据是随机数据，有某些解法没取模被放过去，将会加数据。

对于这种区间查询、可以往线段树想去。并且操作比较复杂，逆操作难求，线段树只有正着操作，就相对很是比较好的。

对于线段树来说，需要定义好区间信息。
比较显然的信息是，方块数量。

对于黄方块，方块数+1。
对于蓝方块，方块数+1。
对于红方块，方块数翻倍。会连续翻倍，所以还需要记录倍数。
但翻倍的是它脚底下的方块，得想办法找到它能翻倍的方块数量。
很显然，红方块最多翻倍到 字符串序列 它左边 最靠右的蓝方块，因为字符串再左一点就，就是红方块的前一列。

想到这里时，区间信息要划分一下，方块数量的含义了。以这个区间第一个蓝方块 $firstBlue$，最后一个蓝方块 $lastBlue$ 为分界。$lastBlue$ 及后面的方块是 $rData$，$firstBlue$ 前面的是 $lData$。$firstBlue$（包括它）到 $lastBlue$（不包括）的方块数量是 mid。
多记录一个 mid，是因为这一段既不能把别人翻倍，也不能被别人翻倍。

可能一个区间并没有蓝方块，再开一个变量 $data$，和一个 $blue$ 变量标记是否存在蓝方块。

具体合并操作，还请看代码两个operator+函数

```cpp
#include <bits/stdc++.h>
using namespace std;
using LL = long long;
#define lson (k << 1)
#define rson (k << 1 | 1)
const int mod = 7 + 1e9;
const int N = 3 + 2e5;
struct Data {
    LL a, b;   // 方块数，倍数(2的b次方)
};
struct TreeNode {
    int l, r;
    Data lData, rData, data;
    LL mid;
    int blue;

    // blue=1时表示这一段区间有蓝方块
    // lData、rData、mid、blue有意义

    // blue=0时表示这一段区间没有蓝方块
    // data有意义
} tr[N * 4];
int n, q, a[N];
char s[N];
LL pow2[N];
Data operator+(const Data& left, const Data& right) {
    Data ans = {};
    ans.a = (left.a * pow2[right.b] + right.a) % mod; // 右边将左边翻倍
    ans.b = left.b + right.b; // 2的b次方
    return ans;
}
TreeNode operator+(const TreeNode& left, const TreeNode& right) {
    TreeNode ans = {}; // 初始化里面全部成0
    ans.l = left.l;
    ans.r = right.r;

    // 分类讨论
    if (left.blue && right.blue) {
        // 左右都有蓝
```

```
        auto temp = left.rData + right.lData;  // 右边将左边翻倍
        ans.lData = left.lData;
        ans.rData = right.rData;
        //temp.a， 左边翻倍之后，因为被firstBlue和LastBlue夹着，不可能再被翻倍
        // 所以算到ans.mid 头上
        ans.mid = (left.mid + temp.a + right.mid) % mod;
        ans.blue = 1;
    } else if (left.blue) {
        // 左有蓝，右没蓝
        ans.lData = left.lData;
        ans.rData = left.rData + right.data;  // 右边将左边翻倍
        ans.mid = left.mid;
        ans.blue = 1;
    } else if (right.blue) {
        // 左没蓝，右有蓝
        ans.lData = left.data + right.lData;  // 右边将左边翻倍
        ans.rData = right.rData;
        ans.mid = right.mid;
        ans.blue = 1;
    } else {
        // 左没蓝，右没蓝
        ans.data = left.data + right.data;  // 右边将左边翻倍
    }
    return ans;
}
void build(int k, int l, int r) {
    tr[k].l = l;
    tr[k].r = r;
    if (l == r) {
        // 初始化
        if (s[r] == 'Y') {
            tr[k].data.a = 1;
        } else if (s[r] == 'R') {
            tr[k].data.a = 1;
            tr[k].data.b = 1;
        } else {
            tr[k].rData.a = 1;
            tr[k].blue = 1;
        }
```

```
        return;
    }
    int mid = l + r >> 1;
    build(lson, l, mid);
    build(rson, mid + 1, r);
    tr[k] = tr[lson] + tr[rson];
}
void update(int k, int p, char c) {
    if (tr[k].l == tr[k].r) {
        s[p] = c;
        memset(&tr[k], 0, sizeof(tr[k])); // 清零
        tr[k].l = p;
        tr[k].r = p;
        if (s[p] == 'Y') {
            tr[k].data.a = 1;
        } else if (s[p] == 'R') {
            tr[k].data.a = 1;
            tr[k].data.b = 1;
        } else {
            tr[k].rData.a = 1;
            tr[k].blue = 1;
        }
        return;
    }
    int mid = tr[k].l + tr[k].r >> 1;
    if (p <= mid) {
        update(lson, p, c);
    } else {
        update(rson, p, c);
    }
    tr[k] = tr[lson] + tr[rson];
}
TreeNode query(int k, int l, int r) {
    if (l <= tr[k].l && tr[k].r <= r) {
        return tr[k];
    }
    int mid = tr[k].l + tr[k].r >> 1;
    if (l <= mid && mid < r) {
        return query(lson, l, r) + query(rson, l, r);
```

```
    }
    if (l <= mid) {
        return query(lson, l, r);
    } else {
        return query(rson, l, r);
    }
}
int main() {
    cin >> n >> q;
    cin >> s + 1;
    pow2[0] = 1;
    for (int i = 1; i <= n; ++i) { // 预处理2的次方
        pow2[i] = pow2[i - 1] * 2 % mod;
    }
    build(1, 1, n);
    while (q--) {
        int op, l, r, p;
        char c;
        cin >> op;
        if (op == 1) {
            cin >> p >> c;
            update(1, p, c);
        } else {
            cin >> l >> r;
            auto ans = query(1, l, r);
            // 全加上。没有意义变量是0，所以全加上没问题
            cout << (ans.data.a + ans.lData.a + ans.rData.a + ans.mid) % mod
                 << '\n';
        }
    }
    return 0;
}
```