2023暑期CSP-S/NOIP模拟赛10 题解

T1 矩阵最大值

对于 subtask 1&2：

对于每一次修改，遍历整个矩阵来计算满足要求的点的个数。

时间复杂度 $O(nmq)$

对于 100% 的数据：

由题意可得，一行（列）中满足要求的点最多只有一个，可以使用 row_i 和 col_j 维护每行每列的最大值。很显然，满足要求的点 (i,j) 满足：$row_i = col_j$

考虑到每次修改，只会往更大值取进行修改，只会影响到一行和一列的最大值，并且最大值的位置只会保持不动或者变换到修改的位置。

对于初始矩阵计算答案数量，之后可以动态去维护答案个数，按照以下顺序考虑答案的变化：

1. 如果更改的位置是有效点，先将 $ans-1$，随后判断。
2. 如果新数大于所在的列的有效点，取消原有效点的资格, $ans-1$，随后判断。
3. 如果新数大于所在的行的有效点，取消原有效点的资格, $ans-1$，随后判断。注意，这里第 2 点并不矛盾。
4. 如果更改的行与列没有有效点，只需随后判断当前点是否满足条件。

时间复杂度：$O(nm+q)$

```
#include<bits/stdc++.h>
using namespace std;
const int N = 200005;
int n, m, q, ans;
int mx[N], my[N];
int w[N], r[N];
// w[i] 表示处于第 i 行的有效点对应的列
// r[j] 表示处于第 i 列的有效点对应的行
signed main() {
  std::ios::sync_with_stdio(false);
  cin.tie(NULL);
  cout.tie(NULL);
  register int i, j, x, y, t;
  cin >> n >> m >> q;
  for (i = 1; i <= n; i++) {
    for (j = 1; j <= m; j++) {
      cin >> t;
      mx[i] = max(mx[i], t);
      my[j] = max(my[j], t);
    }
  }
  for (i = 1; i <= n; i++)
    for (j = 1; j <= m; j++)
```

```
      if (mx[i] == my[j])
        ans++, w[i] = j, r[j] = i;
  while (q--) {
    cin >> x >> y >> t;
    if (w[x] == y) ans--; // Case 1
    if (t > mx[x]) { // Case 2
      mx[x] = t;
      if (w[x] != y && w[x] != 0)
        w[x] = r[w[x]] = 0, ans--;
    }
    if (t > my[y]) { // Case 3
      my[y] = t;
      if (r[y] != x && r[y] != 0)
        r[y] = w[r[y]] = 0, ans--;
    }
    if (mx[x] == my[y]) // Case 4
      ans++, w[x] = y, r[y] = x;
    cout << ans << '\n';
  }
  return 0;
}
```

T2 病毒感染

对于 subtask 1&2：

分析 p = 1 的部分，稳定的病毒所在的细胞一定是不会被其他病毒感染，否则就存在一种方法让这种病毒所在的初始细胞被感染，从而导致这种病毒不复存在。

因此，只需要对所有细胞，检查绝对安全的初始细胞就可以了（对自身易感程度最高的初始细胞）。

时间复杂度：$O(n^2)$

对于 subtask 1&3:

对于 $n \le 5$ ，我们可以考虑直接枚举所有的攻击顺序。

由于部分攻击是无效的，因此我们只考虑能够感染其他细胞的攻击。

可以使用 dfs(int a[]) 表示当前细胞感染病毒的状态，根据现有的病毒存在情况，获得每个细胞可能被攻击的情况，从中选出一个更新细胞状态继续递归，直到无法进行感染。

由于能够更新细胞感染情况的状态数较少，对于 $n \le 5$ 是能够通过的。

对于 80% 的数据（ subtask 1&2&3&4）：

现在我们来考虑 p=2 的部分，是否存在一种感染方式，能够让某种病毒存活下来。由于是存在某种方式即可，我们可以对每种病毒单独进行考虑。

使用 (i, j) 表示病毒 i 感染细胞 j ，如果病毒 i 感染了细胞 j 后能够存活下来存活下来，那么对于细胞 j 来说，所有易感染程度超过病毒 i 的病毒都被杀死了。

由于一个细胞可能会被多个病毒感染，考虑每个感染的情况会比较复杂。我们发现，细胞最后只会被某种病毒感染。所以我们只需要考虑细胞最后被哪一种病毒感染即可，忽略前面多次感染的过程。

对每一个 (i, j) ，将病毒分为两个集合，一个集合是对细胞 i 感染程度超过当前病毒的，也即 (i, j) 能够成功存活必须要干掉的（危险的集合），另一个集合是没有威胁的可以保留的（安全的集合）。而必须干掉的那个集合，只能通过保留的那个集合来进行消灭。

对此，我们可以使用 $O(n^2)$ 的枚举，检查是否能够通过保留的集合感染有威胁的集合。

总体时间复杂度: $O(n^4)$

如果使用 bitset 来进行维护，可以做到 $O(n^4/32)$ 。

```cpp
#include <bits/stdc++.h>
using namespace std;
int n;
const int maxn = 505;
int a[maxn][maxn], flag[maxn], endup[maxn], pos[maxn][maxn];
// endup 枚举的终点（只需要考虑已感染程度比初始病毒大的）
bitset<maxn> bis[maxn][maxn];
// bis[i][j]表示初始细胞i,易感染程度前j大的病毒有哪些（二进制位表示）
int main() {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
  cin >> n;
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= n; ++j)
            cin >> a[i][j], pos[i][a[i][j]] = j;
    }
  int tp;
  cin >> tp;
    if (tp == 1) {
        for (int j = 1; j <= n; ++j) {
            if (a[j][1] != j) {
                flag[j] = 1;
            }
        }
    }
  else {
        for (int i = 1; i <= n; ++i) {
            flag[a[i][1]] = -1;
            for (int j = 1; j <= n; ++j) {
                if (a[i][j] == i) {
                    endup[i] = j - 1;
                    break;
                }
                bis[i][j] = bis[i][j - 1];
                bis[i][j][a[i][j]] = 1;
            }
        }
```

```
        for (int i = 1; i <= n; ++i) { // 细胞i与原始病毒i
            if (flag[i] == 0) {
        // 考虑原始病毒能不能存活
                for (int j = 1; j <= n; ++j) { // 对细胞 i 危险的病毒 j
                    if (a[i][j] == i) break;
          // 能够威胁到病毒 j 的全是对细胞 i 来说危险的病毒
                    if (bis[a[i][j]][endup[a[i][j]]] == (bis[a[i][j]][endup[a[i][j]]]&bis[i][endup[i]])) {
                        flag[i] = 1;
                        break;
                    }
                }
        // 考虑感染其他细胞
                if (flag[i]) {
                    for (int j = 1; j <= n; ++j) { // 细胞j的(j,i)
                        if (pos[j][i] <= endup[j]) {
                            flag[i] = 0;
                            for (int k = 1; k <= n; ++k) {
                                if (a[j][k] == i) break;
                // 能够威胁到病毒 i 的全是对病毒 i 来说危险的病毒
                                if (bis[a[j][k]][endup[a[j][k]]] == (bis[a[j][k]][endup[a[j][k]]]&bis[j][pos[j][i] - 1])) {
                                    flag[i] = 1;
                                    break;
                                }
                            }
                            if (flag[i] == 0) break;
                        }
                    }
                }
            }
        }
    }
    int ans = 0;
    for (int i = 1; i <= n; ++i) if (flag[i] != 1) ++ans;
  cout << ans << '\n';
    for (int i = 1; i <= n; ++i) if (flag[i] != 1) cout << i << " ";
    return 0;
}
```

对于 100% 的数据：

考虑如何进行优化？

从细胞的角度来考虑问题：对于每个细胞 i，按易感染程度升序去考虑病毒 j 的 (i,j) ，并依次将之前考虑过的 $1...j-1$ 号病毒添加到安全的病毒的集合中去。对剩下的 n-1 个细胞，维护安全的集合中易感染程度最高的病毒所在的位置。

对每个危险的病毒， 检查在它的初始细胞中，是否有安全集合的细胞能够杀死他。如果存在某个危险的病毒不能被杀死，那么 (i,j) 就不成立。

时间复杂度: $O(n^3)$

如果使用 bitset 来进行维护，可以做到 $O(n^3/32)$ 。

```cpp
#include <bits/stdc++.h>
using namespace std;
int cnt[505], cnt1[505];
int n, a[505][505], p, Ans[505];
bitset<505> kil[505];
// kil[i]是 i 能杀死的集合

inline bool check(int x, bitset <505> d) {
    if (!d[x] && d.count() == n - 1) return true;
    if (d.count() == n) return true;
    return false;
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
  cin >> n;
    for (int i = 1; i <= n; ++i) for (int j = 1; j <= n; ++j) cin >> a[i][j];
  cin >> p;
    if (p & 1) {
        for (int i = 1; i <= n; ++i) if (a[i][1] == i) ++cnt1[i];
        int res = 0;
        for (int i = 1; i <= n; ++i) if (cnt1[i]) ++res;
    cout << res << '\n';
        for (int i = 1; i <= n; ++i) if (cnt1[i]) cout << i << " ";
    }
  else {
        for (int i = 1; i <= n; ++i) {
            int pos;
            for (int j = 1; j <= n; ++j) if (a[i][j] == i) {
        pos = j;
        break;
      }
            for (int j = 1; j <= pos; ++j) kil[a[i][j]][i] = 1;
        }
        for (int i = 1; i <= n; ++i) {
            int pos;
            for (int j = 1; j <= n; ++j) if (a[i][j] == i) {
        pos = j;
        break;
      }
            bitset <505> qwq;
            qwq.reset();
            for (int j = n; j > pos; --j) qwq |= kil[a[i][j]];
            for (int j = pos; j >= 1; --j) {
```

```
                if (check(i, qwq)) ++cnt1[a[i][j]];
                qwq |= kil[a[i][j]];
            }
        }
        int tot = 0;
        for (int i = 1; i <= n; ++i) if (cnt1[i]) Ans[++tot] = i;
    cout << tot << '\n';
        for (int i = 1; i <= tot; ++i) cout << Ans[i] << " ";
    }
    return 0;
}
```

T3 怪物猎人

对于 subtask1：

2^n 枚举每个怪物是否用魔咒消灭，直接计算代价。

时间复杂度：$O(n2^n)$

对于 subtask3：

树形 dp 退化成序列上的dp。

设 $dp[i][j][0/1][0/1]$ 表示以 i 为根的子树中，使用了 j 个魔咒，当前节点 i 是否使用了魔咒(0/1)，儿子节点是否使用魔咒(0/1)。

转移分类讨论直接由儿子转移过来。

对于 40%到 100% 的数据：

使用树形 dp 来解决问题。由于引发最低花费不同的是魔咒的使用，设 $dp[i][j][0/1]$ 表示以 i 为根的子树中，使用了 j 个魔咒，当前节点 i 是否使用了魔咒。

考虑转移方程，对于 u 为根的子树使用了 j 个魔咒，依次 u 的儿子节点 v ，考虑在 v 中使用 k 个魔咒的最小代价：

$dp[u][0][0] = hp[u],\ dp[u][1][1] = 0$

$dp[u][j+k][0] = min\{dp[u][j+k][0], dp[u][j][0] + min(dp[v][k][0] + hp[v], dp[v][k][1])\}$

$dp[u][j+k][1] = min\{dp[u][j+k][1], dp[u][j][1] + min(dp[v][k][0], dp[v][k][1])\}$

这里需要注意的是方程的转移，严格按照子树大小进行枚举的话，可以做到 $O(n^2)$ 的复杂度，而其他的枚举方式可能会退化到 $O(n^3)$ 的复杂度。同时需要注意考虑子树贡献时，要类似背包的处理方式或者使用临时数组来进行转移。

（这个是因为两个点x, y 他们只会在 $lca(x,y)$ 时候被统计）

```
#include <bits/stdc++.h>
#define inf 0x3f3f3f3f
#define inf64 0x3f3f3f3f3f3f3f3f
using namespace std;
```

```cpp
const int maxn = 2e3 + 10;
typedef long long LL;
int cnt, head[maxn], to[maxn << 1], nxt[maxn << 1];
LL hp[maxn], dp[maxn][maxn][2];

void add(int u, int v) {
    ++cnt, to[cnt] = v, nxt[cnt] = head[u], head[u] = cnt;
}

int siz[maxn];
LL tmp[maxn][2];

void dfs(int u, int fa) {
    siz[u] = 1;
    for (int i = head[u]; i; i = nxt[i]) {
        int v = to[i];
    if(v == fa) continue;
        dfs(v, u);
        memset(tmp, inf64, sizeof(tmp));
        for (int j = 0; j <= siz[u]; j++) {
            for (int k = 0; k <= siz[v]; k++) {
                tmp[j + k][0] = min(tmp[j + k][0], dp[u][j][0] + dp[v][k][0] + hp[v]);
                if (k > 0) tmp[j + k][0] = min(tmp[j + k][0], dp[u][j][0] + dp[v][k][1]);
                if (j > 0) tmp[j + k][1] = min(tmp[j + k][1], dp[u][j][1] + dp[v][k][0]);
                if (j > 0 && k > 0) tmp[j + k][1] = min(tmp[j + k][1], dp[u][j][1] + dp[v][k][1]);
            }
        }
        for (int j = 0; j <= siz[u] + siz[v]; j++) {
            dp[u][j][0] = tmp[j][0], dp[u][j][1] = tmp[j][1];
        }
        siz[u] += siz[v];
    }
    for (int i = 0; i <= siz[u]; i++) dp[u][i][0] += hp[u];
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
  int n;
  cin >> n;
  for (int i = 2, x, y; i <= n; i++) {
    cin >> x >> y;
    add(x, y), add(y, x);
  }
  for (int i = 1; i <= n; i++) cin >> hp[i];
```

```
  dfs(1, 0);
  for (int i = 0; i <= n; i++) {
    cout << min(dp[1][i][0], dp[1][i][1]);
    if (i == n) cout << '\n';
    else cout << ' ';
  }
}
```

T4 集卡游戏

对于 20% 的数据（subtask 1&2）:

枚举在德玛西亚城邦选择的卡片区间 $[l_1, r_1]$ 和艾欧尼亚城邦的卡片区间$[l_2, r_2]$，直接判断这种选择是否满足题意，并与之前的答案比较大小。

时间复杂度: $O(n^4)$

对于 60% 的数据 (subtask 1~6):

如果选择一个城邦的卡片，那么显然会选择所有的卡片。因此，在选择两个城邦的卡片时，至少有一个城邦的卡片选了超过自己城邦卡片总价值的一半以上。

假设现在第一个城邦选择的卡片价值一定要超过一半，那么设第一个城邦第一次前缀和超过总权值一半的位置为 p，则这个位置一定要被选择。（长度超过总权值一般的窗口始终覆盖 p）

那么答案就是在第二个的卡片中任意选择一段，然后将重复的类型第一个的卡片中标记出来，从 p 开始往两边尽可能拓展。

使用暴力枚举第二个城邦的卡片中的一段，直接计算第一个中包括 p 的那一段最长为多少（p 左边的最大值和右边的最小值）。

时间复杂度：$O(n^2)$

对于100%的数据:

考虑使用2个单调栈+线段树来优化这个过程。

枚举第二个城邦选择卡片的右端点 r，我们使用线段树来维护第二个城邦区间所有左端点l 对应的前缀和价值+第一个城邦卡片对应的所选区间价值和的最大值（$A[L...R] - B[l]$)。

对于第二个城邦左端点从小到大考虑的过程，他们对应的第一个城邦卡片的位置分布在 p 的两侧，可以使用两个单调栈来维护对应的 p 左边的最大值和 p 右边的最小值，两个单调栈中的每一个组合都是一个合理答案。使用线段树来寻找最大的答案是什么。

时间复杂度: $O(nlog_2n)$

```
#include <bits/stdc++.h>
#define maxn 500005
#define LL long long
using namespace std;
int n, m, a[maxn], b[maxn], c[maxn * 2];
int p[maxn], pn, al, ar, bl, br, pos[maxn], A[maxn], An, B[maxn], Bn; //A[0],B[0] are used as 0!
```

```cpp
LL ans, x[maxn], y[maxn];
LL mx[maxn << 2], tag[maxn << 2];

//维护Y中下标区间[l,r]所对应的 X[L...R]-Y[l]
void build(int i, int l, int r, LL x, LL *y) {
    tag[i] = 0;
    if (l == r) return void(mx[i] = x - y[p[l]]); // 初始X[0...N]-y[l]
    int mid = (l + r) >> 1;
    build(i << 1, l, mid, x, y), build(i << 1 | 1, mid + 1, r, x, y);
    mx[i] = max(mx[i << 1], mx[i << 1 | 1]);
}
//区间不会重叠，不需要给tag传递下去
void mdf(int i, int l, int r, int x, int y, LL v) {
    if (x <= l && r <= y) {
        mx[i] += v, tag[i] += v;
        return;
    }
    int mid = (l + r) >> 1;
    if (x <= mid) mdf(i << 1, l, mid, x, y, v);
    if (y > mid) mdf(i << 1 | 1, mid + 1, r, x, y, v);
    mx[i] = max(mx[i << 1], mx[i << 1 | 1]) + tag[i];
}

int find(int i, int l, int r) {
    if (l == r) return l;
    int mid = (l + r) >> 1;
    if (mx[i] == mx[i << 1] + tag[i])
        return find(i << 1, l, mid);
    else
        return find(i << 1 | 1, mid + 1, r);
}

inline void updans(LL x, int a, int b, int c, int d, bool flg) {
    if (flg) swap(a, c), swap(b, d);
    if (x > ans) ans = x, al = a, ar = b, bl = c, br = d;
}

void solve(int n, int m, int *a, int *b, LL *x, LL *y, bool flg) {
    // 一定会经过mid点
    int mid = lower_bound(x + 1, x + 1 + n, x[n] >> 1) - x;
    pn = 0;
    for (int i = 1; i <= m; i++) if (b[i]) p[++pn] = i, pos[pn] = b[i];
    p[pn + 1] = m + 1;
    // p: Y中与X相同的类型的点的位置
    build(1, 0, pn, x[n], y);
    for (int i = An = Bn = 0; i <= pn; i++) { // 枚举Y的右端点
        // 维护Y左端点
        if (i) {
            if (pos[i] <= mid) {
```

```
                mdf(1, 0, pn, A[An], i - 1, -x[pos[i]]); // 给Y中下标区间[A[An], i-1]添加对应的x的左端点 X[pos[i]]
                //维护 mid 左边的Max单调栈
                while (An && pos[i] > pos[A[An]])
                    mdf(1, 0, pn, A[An - 1], A[An] - 1, -x[pos[i]] + x[pos[A[An]]]), --An;

                A[++An] = i;
            } else {
                mdf(1, 0, pn, B[Bn], i - 1, -x[n] + x[pos[i] - 1]);
                //维护 mid 右边的Max单调栈
                while (Bn && pos[i] < pos[B[Bn]])
                    mdf(1, 0, pn, B[Bn - 1], B[Bn] - 1, -x[pos[B[Bn]] - 1] + x[pos[i] - 1]), --Bn;
                B[++Bn] = i;
            }
        }
    // 寻找答案
        if (y[p[i + 1] - 1] + mx[1] > ans) {
            int k = find(1, 0, pn); //寻找最大的左端点
            // 二分单调栈找对应X[l...r]
            int l = upper_bound(A + 1, A + 1 + An, k) - A;
            l = l <= An ? b[p[A[l]]] + 1 : 1;
            int r = upper_bound(B + 1, B + 1 + Bn, k) - B;
            r = r <= Bn ? b[p[B[r]]] - 1 : n;
            updans(y[p[i + 1] - 1] + mx[1], l, r, p[k] + 1, p[i + 1] - 1, flg);
        }
    }
}
int main() {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
  cin >> n >> m;
    for (int i = 1; i <= n; i++) cin >> a[i];
    for (int i = 1; i <= n; i++) cin >> x[i], x[i] += x[i - 1];
    for (int i = 1; i <= m; i++) cin >> b[i];
    for (int i = 1; i <= m; i++) cin >> y[i], y[i] += y[i - 1];
    updans(x[n], 1, n, 0, 0, 0), updans(y[m], 0, 0, 1, m, 0);
    for (int i = 1; i <= m; i++) c[b[i]] = i, b[i] = 0;
  // c[i]=j: i号课程在b的j的位置
  for (int i = 1; i <= n; i++) a[i] = c[a[i]], b[a[i]] = i;
  // 相同的课程中, a[i]=j: i号课程在b的j位置, b[i]=j: i号课程在a的j位置
  solve(n, m, a, b, x, y, 0), solve(m, n, b, a, y, x, 1);
    cout << ans << '\n';
  cout << al << " " << ar << '\n';
  cout << bl << " " << br << '\n';
}
```