2023暑期CSP-S/NOIP模拟赛 3 题解

T1 体育课

40% 的做法：

由于 $n <= 7$，按字典序枚举最开始时的排列，对每一个排列都计算最后得到的数的值，如果这个值和题目给出的值相等，那么我们就找到答案了。

40% - 80%的做法：

可以使用 C++ 的 `next_permutation` 函数，也可以在 dfs 枚举排列时，每一层都从小到大枚举，这样自然就是按 "字典序" 枚举了。

100%的做法：

类似第一场比赛地狱疣那道题，首先分析每一轮得到的数与初始序列的关系：

```
a       b      c       d       e        f        g       h
   a+b    b+c     c+d      d+e      e+f       f+g     g+h
      a+2b+c  b+2c+d  c+2d+e   d+2e+f    e+2f+g   f+2g+h
      a+3b+3c+d  b+3c+3d+e c+3d+3e+f  d+3e+3f+g  e+3f+3g+h
    a+4b+6c+4d+e b+4c+6d+4e+f c+4d+6e+4f+g d+4e+6f+4g+h
     a+5b+10c+10d+5e+f  b+5c+10d+10e+5f+g  c+5d+10e+10f+5g+h
      a+6b+15c+20d+15e+6f+g b+6c+15d+20e+15f+6g+h
                a+7b+21c+35d+35e+21f+7g+h
```

通过观察可以知道（大力打表找规律），每一次操作得到的数前面的系数都是一个组合数。

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

组合数的推导过程可以使用类似图中计数的方式，从 ai 到某一个具体的数有多少到达的方式。

当前问题：现在已知初始序列，可以直接计算最后的结果。可以通过搜索+剪枝的方式来得到字典序最小的答案。

可行性剪枝：如果一个排列前面的数的（带系数的）和就已经超过了sum，那它就不可能成为答案了。

```cpp
#include <bits/stdc++.h>
using namespace std;
int n, sum;
int C[13][13];
int ans[13];
bool used[13];
```

```cpp

void print() {
    for(int i = 1; i <= n; i++) cout << ans[i] << " ";
    exit(0);
}

void dfs(int step) {
    if(step > n) {
        if (!sum) print();
        return;
    }
    for(int i = 1, t = min(n, sum / C[n][step]); i <= t; i++)
        if(!used[i]) {
            sum -= i * C[n][step];
            used[i] = true;
            ans[step] = i;
            dfs(step + 1);
            used[i] = false;
            sum += i * C[n][step];
        }
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
  cin >> n >> sum;
    C[1][1] = 1;
    for(int i = 2; i <= n; i++)
        for(int j = 1; j <= i; j++)
            C[i][j] = C[i - 1][j - 1] + C[i - 1][j];
    dfs(1);
}
```

T2 宝石交易

首先可以发现的：

- 1 个七彩宝石价值 100 个无色宝石
- 1 个五色宝石价值 50 个无色宝石
- 1 个三色宝石价值 20 个无色宝石
- 1 个双色宝石价值 10 个无色宝石
- 1 个单色宝石价值 5 个无色宝石

我们不妨使用等值的无色宝石来代表每种宝石，分别用每个宝石的价值分别为 $v_1, v_2, \dots v_6 = 100, 50, \dots, 1$ 。

10% 的做法：

输出 "impossible"

60%-100%的做法：

本题是思维量较大，但代码量较小。

解决这道题的核心在于将三人的交换。首先得先理解，如何结算最高效?

第一步：ABC给所有宝石都拿出来放桌子上

第二步：各自拿走自己部分的宝石

假设他们原来分别有old_a, old_b, old_c的宝石，结算以后有new_a, new_b, new_c的宝石，其中：

$$new_a = old_a - x_1 + x_3$$
$$new_b = old_b - x_2 + x_1$$
$$new_c = old_c - x_3 + x_2$$

那么问题可以转化为：在六种宝石中，取两个集合$sum_1 = new_a$，$sum_2 = new_b$，这里C可以由A和B的情况计算得到。

考虑使用动态规划来处理：令 $f_{i,x,y}$ 表示考虑前 i 种宝石，构造出A拿了 x 的宝石，B拿了 y 的宝石情况下的最小交换次数。

令 $max_i = a[1][i] + a[2][i] + a[3][i]$ ，表示最多有多少个 i 种类的宝石, 。

$f_{i,x,y} = min_{0 \le p \le max_i,\ 0 \le q \le max_i - p} f_{i-1,x-p*v_i,y-q*v_i} + \frac{|p-a[1][i]|+|q-a[2][i]|+|max_i-p-q-a[3][i]|}{2}$

其中，如果A，B，C初始有一种宝石 a_1, b_1, c_1 个，最后宝石的数量分别为 a_2, b_2, c_2 个，那么他们最小的交易次数为 $\frac{|a_1-a_2|+|b_1-b_2|+|c_1-c_2|}{2}$ 。

由于无色宝石的总数不超过 1000 ，这种方法理论复杂度比较高，在开O2的情况下能够通过本题，当然你也可以采用记忆化的方法。

```cpp
#include <bits/stdc++.h>
using namespace std;
const int inf = 0x3f;
int ab, bc, ca;
int f[2][1001][1001];
int a[7], b[7], c[7], num[7];
int v[] = {0, 100, 50, 20, 10, 5, 1};

int main() {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
    cin >> ab >> bc >> ca;
    int Ta = 0, Tb = 0, Tc = 0;
    for (int i = 1; i <= 6; ++i) cin >> a[i], Ta += a[i] * v[i];
    for (int i = 1; i <= 6; ++i) cin >> b[i], Tb += b[i] * v[i];
    for (int i = 1; i <= 6; ++i) cin >> c[i], Tc += c[i] * v[i];
```

```
    memset(f, inf, sizeof(f));
  int tp = 1;
  f[tp][0][0] = 0;
    for (int i = 1; i <= 6; ++i) num[i] = a[i] + b[i] + c[i];
    Ta += ca - ab;
    Tb += ab - bc;
    Tc += bc - ca;
    for(int i = 1; i <= 6; ++i) {
    tp ^= 1;
    for(int x = 0; x <= Ta; ++x) for(int y = 0; y <= Tb; ++y) {
      if(f[tp ^ 1][x][y] == inf) continue;
      for(int p = 0; p <= num[i]; ++p) {
        if(x + p * v[i] > Ta)  break;
        for(int q = 0; p + q <= num[i]; ++q) {
          if(y + q * v[i] > Tb) break;
          int tmp = f[tp ^ 1][x][y] + (abs(p - a[i]) + abs(q - b[i]) + abs(a[i] + b[i] - p - q));
          if(f[tp][x + p * v[i]][y + q * v[i]] > tmp) f[tp][x + p * v[i]][y + q * v[i]] = tmp;
        }
      }
    }
     for(int x = Ta; x >= 0; x--) for(int y = Tb; y >= 0; y--) f[tp ^ 1][x][y] = inf;
  }
    if(f[tp][Ta][Tb] < inf) cout << f[tp][Ta][Tb] / 2;
    else cout << "impossible";
}
```

100%做法：

如果觉得有上述做法时间复杂度过高的话，也可以对dp的过程进行一些剪枝，将所有的能够达到的状态记录下来。

可以看到，类似A->B->C这样的结算方式是无意义的，不如A直接给C.

那么转移的时候只需要枚举6种情况：

A -> B、C　　B -> A、C　　C -> A、B

C、B -> A　　A、C - >B　　A、B -> C

同时：如果当前A的宝石与A在最终态的宝石的差不能被当前宝石种类与当前宝石种类后面所有钞票值的最大公约数整除，那么这个状态是无用的了，直接不考虑就可以了。

这样的话，有效的状态个数会很少，使用队列记录一下上一步有用的状态即可。

```
#include <bits/stdc++.h>
using namespace std;
const int val[6] = {1, 5, 10, 20, 50, 100};
const int gcd[6] = {1, 5, 10, 10, 50, 100};
const int e = 1005;
```

```
struct Rec {
    int a, b;
    void init(int p, int q) {
        a = p, b = q;
    }
} temp, S, T;
queue<Rec> seq[2];
int N, t, p, num[3][6], dp[2][e][e], sum, fin[3], x[3];

void Doit(int fr, int a, int b, int c) { //A->B/C,B->A/C,C->A/B
    Rec tmp;
    int m = t & 1;
    int p = (fr + 1) % 3, q = (fr + 2) % 3, r[3] = {a, b, c}, tt[3];
    for (int i = 1; i <= min(r[fr] / val[t], num[fr][t]); i++)
        for (int j = 0; j <= i; j++) {
            tt[p] = r[p] + val[t] * j;
            tt[q] = r[q] + val[t] * (i - j);
            tt[fr] = r[fr] - val[t] * i;
            if (dp[m][tt[0]][tt[1]] > dp[1 - m][a][b] + i) {
                dp[m][tt[0]][tt[1]] = dp[1 - m][a][b] + i;
                tmp.a = tt[0], tmp.b = tt[1];
                seq[1 - m].push(tmp);
            }
        }
}

void Stupid(int fr1, int fr2, int a, int b, int c) {
    Rec tmp;
    int m = t & 1;
    int p = fr1, q = fr2, o = 3 - fr1 - fr2, r[3] = {a, b, c}, tt[3];
    for (int i = 0; i <= min(r[p] / val[t], num[p][t]); i++)
        for (int j = 0; j <= min(r[q] / val[t], num[q][t]); j++) {
            tt[p] = r[p] - val[t] * i;
            tt[q] = r[q] - val[t] * j;
            tt[o] = r[o] + val[t] * (i + j);
            if (dp[m][tt[0]][tt[1]] > dp[1 - m][a][b] + i + j) {
                dp[m][tt[0]][tt[1]] = dp[1 - m][a][b] + i + j;
                tmp.a = tt[0], tmp.b = tt[1];
                seq[1 - m].push(tmp);
            }
        }
}

void Update(int q) {
    for (int i = 0; i < e; i++)
        for (int j = 0; j < e; j++)
            dp[q][i][j] = dp[1 - q][i][j];
}

```

```
int v[e][e];
void Dp() {
    memset(v, -1, sizeof(v));
    for (t = 0; t < 6; t++) {
        Update(t & 1);
        while (!seq[t & 1].empty()) {
            temp = seq[t & 1].front(), seq[t & 1].pop();
            int a = temp.a, b = temp.b, c = sum - a - b;
            if ((a - fin[0]) % gcd[t] || (b - fin[1]) % gcd[t] || v[a][b] == t)
continue;
            seq[!(t & 1)].push(temp);
            Doit(0, a, b, c);
            Doit(1, a, b, c);
            Doit(2, a, b, c);
            Stupid(0, 1, a, b, c);
            Stupid(1, 2, a, b, c);
            Stupid(0, 2, a, b, c);
            v[a][b] = t;
        }
    }
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
    for (int i = 0; i < 3; i++) cin >> x[i];
    for (int i = 0; i < 3; i++) {
        for (int j = 5; j >= 0; j--) {
            cin >> num[i][j];
            fin[i] += num[i][j] * val[j];
        }
        sum += fin[i];
    }
    memset(dp, 127, sizeof(dp));
    dp[1][fin[0]][fin[1]] = 0;
    S.init(fin[0], fin[1]);
    seq[0].push(S);
    for (int i = 0; i < 3; i++) fin[i] -= x[i], fin[(i + 1) % 3] += x[i];
    if (fin[0] < 0 || fin[1] < 0 || fin[2] < 0) {
        cout << "impossible" << '\n';
        return 0;
    }
    T.init(fin[0], fin[1]);
    Dp();
    if (dp[1][T.a][T.b] < 10 * e) cout << dp[1][T.a][T.b] << '\n';
    else cout << "impossible" << '\n';
    return 0;
}
```

其他做法：

发现题目约束三人宝石之和不会超过 1000，并且对每人价值小于10的宝石个数进行了约束，所以估计价无色宝石的个数比较多。

因为我们大力 DFS，枚举每个人最终最优情况中每种价值大于1的宝石的个数，最后判断可行性并用无色宝石调平。

给 DFS 加点可行性和最优性剪枝。

T3 桌面国王

30%的做法：

直接对每次查询，使用BFS即可。

时间复杂度 $O(n^2)$

100%的做法：

对每一次询问，可以用分块的思想来考虑，得到每个块内部的最短路情况，两个点之间的最短路可以通过块合并来得到。

区间问题可以考虑用莫队，ST表，或者线段树来维护。

这里可以将块设置为相邻的一个正方形为一块（4个城市），块的合并可以直接通过枚举两个块之间的路径得到。

对4个格子，使用$[0/1][0/1]$表示从左边的坐标$[l][0/1]$到右边的坐标$[r][0/1]$的最短路；初始化时，满足有$l=r$。

合并时：采用倍增的方式进行合并。

```cpp
#include <bits/stdc++.h>
#define MAXN 210000
using namespace std;
char s[2][MAXN * 4];
int n, m;
int tw[19];

int biu(int x, int y) {
    if (x == -1 || y == -1) return -2;
    return x + y;
}

void MIN(int &x, int y) {
    if (y < 0) return;
    if (x < 0) x = y;
    if (y < x) x = y;
}

struct info {
    int c[2][2];

```

```
    void init(int x) { // l==r
        if (s[0][x] == 'X' || s[1][x] == 'X') c[0][1] = c[1][0] = -1;
        else c[0][1] = c[1][0] = 1;

        if (s[0][x] == 'X') c[0][0] = -1;
        else c[0][0] = 0;

        if (s[1][x] == 'X') c[1][1] = -1;
        else c[1][1] = 0;
    }

    info operator +(const info &b)const {
        info reu;
        for (int k1 = 0; k1 <= 1; k1++)
            for (int k2 = 0; k2 <= 1; k2++) {
                reu.c[k1][k2] = -1;
                for (int p2 = 0; p2 <= 1; p2++)
                    MIN(reu.c[k1][k2], biu(c[k1][p2], b.c[p2][k2]) + 1);
            }
        return reu;
    }
} f[MAXN][19];

int TW(int x) {
    int reu = 0;
    while(x) x /= 2, reu++;
    return reu - 1;
}

info ask(int l, int r) {
    info reu = f[l][0];
    l++;
    int len = r - l + 1;
    while (len) {
        int p = len & -len;
        reu = reu + f[l][TW(p)];
        l += tw[TW(p)];
        len -= p;
    }
    return reu;
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
  cin >> n >> m;
    for (int i = 0; i <= 1; i++) cin >> (s[i] + 1);
    tw[0] = 1;
    for (int i = 1; i <= 18; i++) tw[i] = tw[i - 1] * 2;
```

```
    for (int i = 1; i <= n; i++) f[i][0].init(i);
    for (int j = 1; j <= 18; j++)
        for (int i = 1; i <= n; i++)
            if (i + tw[j] - 1 <= n)
                f[i][j] = f[i][j - 1] + f[i + tw[j - 1]][j - 1];
    for (int i = 1; i <= m; i++) {
        int l, r;
    cin >> l >> r;
        int p1 = (l > n), p2 = (r > n);
        if (l > n) l -= n;
        if (r > n) r -= n;
        if (l > r) {
            int tmp = l; l = r; r = tmp;
            tmp = p1; p1 = p2; p2 = tmp;
        }
        info q = ask(l, r);
    cout << q.c[p1][p2] << '\n';
    }
    return 0;
}
```

T4 乌鸦喝水

有一个面积为 m×n 的水池，每个位置水的深度是 d_{ij}。找到一个体积最大的长方体，高可以任意，长和宽不大于a,b 或b,a（题目保证 ab≤mn）。且满足在水池的某个位置**不倾斜**地放下这个长方体之后，长方体顶面高度严格低于水面高度。

思考：

推一下式子，假设长方体长宽高为xyz，以及xy地面覆盖的平面离水面最浅的为P，那么不露出水面的条件为立方体顶部离水面距离大于0：

$P+\frac{xyz}{nm}-z>0$

$P-z>-\frac{xyz}{nm}$

$mnP-nmz>-xyz$

$mnP>(nm-xy)z$

$\frac{mnP}{nm-xy}>z$

$z=[\frac{mnP-1}{nm-xy}]$

由此在已知底面，得出石头卡住的深度P后，可以通过计算直接得出石头最大的高度。

观察数据范围发现P很大，不能直接枚举，关键是求出深度P。

74%的做法：

枚举矩形左上端点（lx,ly），枚举长宽（x,y）的同时求高度

复杂度 $O(n^4)$

100%的做法：

通过枚举某一条底边界所在的行（m的维度），并枚举另一条底边界的长度（n的维度），可以将任务限制在一个剖面上（$m \times deep$ 的维度），可以发现变成了一个最大子矩形问题。

最大子矩阵通过单调栈，得到以每个deep作为矩阵的高时，矩阵最大的面积。对每个高度，递增时入栈，出栈时计算面积，直到将当前高度添加到单调栈中。

复杂度 $O(n^3)$

```cpp
#include <bits/stdc++.h>
using namespace std;
const int maxn = 505;
long long S;
int deep[maxn][maxn];
int now[maxn], st[maxn], len[maxn];
int top;
int tlen;

long long solve(int r, int c, int bottom) {
    long long h = (1ll * bottom * S - 1) / (S - 1ll * r * c);
    return h * r * c;
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(NULL);
  int a, b, n, m;
  cin >> a >> b >> m >> n;
  S = (long long)m * n;
  if (a > b) swap(a, b); //keep: a < b
  for (int i = 1; i <= m; ++i)
    for (int j = 1; j <= n; ++j)
      cin >> deep[i][j];
  long long ans = 0;
  for (int i = 1; i <= m; ++i) {
    memset(now, 0x7F, sizeof now);
    int mex;
    for (int j = i; j <= m && j < i + b; ++j) {
      if(j < i + a) mex = b;
      else mex = a;
      for (int k = 1; k <= n; ++k) now[k] = min(now[k], deep[j][k]);
      st[top = 0] = -1;
      len[0] = 0;
      for (int k = 1; k <= n; ++k) {
```

```
        tlen = 1;
        while (st[top] > now[k]) {
          ans = max(ans, solve(j - i + 1, min(len[top], mex), st[top]));
          if (now[k] < st[top - 1]) len[top - 1] += len[top];
          else tlen += len[top];
          --top;
        }
        st[++top] = now[k];
        len[top] = tlen;
      }
      while (top > 0) {
        ans = max(ans, solve(j - i + 1, min(len[top], mex), st[top]));
        len[top - 1] += len[top];
        --top;
      }
    }
  }
  cout << ans << "\n";
    return 0;
}
```

100%的做法2：

使用笛卡尔树来求最大子矩阵。