Mate en uno

https://jutge.org/problems/P57511_es

Dadas diversas posiciones de ajedrez, determinad si las blancas pueden dar mate en una sola jugada. Las posiciones dadas cumplen las propiedades siguientes:

- Es el turno de las blancas.
- Las negras sólo tienen el rey.
- Las blancas no tienen peones.
- Las blancas no pueden enrocarse.

Observación. Si no conocéis (o no estáis seguros) de las reglas del ajedrez, no dudéis en preguntar a los organizadores durante el concurso. En particular, recordad estas normas:

- Un rey nunca puede moverse a una posición atacada.
- En particular, los dos reyes nunca pueden estar en casillas adyacentes.
- Un rey se puede comer una pieza enemiga si está en una posición adyacente y no está defendida por ninguna otra pieza.
- Una posición es mate si el rey está amenazado y no tiene ningún movimiento válido.
- Un rey puede dar mate si moviéndose hace que otra pieza ataque al rey enemigo.

Entrada

La entrada consiste en diversos casos. Cada caso comienza con la posición del rey negro, seguida por el número n de piezas blancas, entre 2 y 20, seguido por la posición de las n piezas blancas. Las posiciones se indican con una letra con el tipo de pieza (**R**ey, **D**ama, **A**lfil, **C**aballo o **T**orre) seguido de una letra a-h para la columna, y un número 1-8 para la fila. Se os garantiza que no hay dos piezas ocupando la misma casilla del tablero, y que ninguna pieza está amenzando al rey negro.

Salida

Para cada posición dada, escribid información de la única jugada que da mate en uno al rey negro, con tres caracteres: el tipo de ficha, y la posición destino a la que se moverá. Si no es posible dar mate en uno, escribid "NO". Si hay más de una jugada que de mate en uno, escribid ">1".

Solución

La mayor dificultad de este problema es la implementación. Hay que hacerlo de manera ordenada y es recomendable usar las herramientas que nos proporciona el lenguaje de programación. En nuestro caso

programaremos la solución en c++.
La idea de la solución es para cada pieza miramos todos los movimientos posibles, para cada uno de estos movimientos marcamos todas las casillas amenazadas por una pieza blanca. Si las 9 casillas de la zona del rey negro están amenazadas este movimiento es un posible mate.
Hay que tener cosas en cuenta como las restricciones de movimientos de las piezas y hay que prestar atención al marcar las posiciones amenazadas, ya que una casilla con una pieza blanca puede estar amenazada por otra pieza blanca o que el rey negro no es un obstáculo en el momento de marcar las casillas amenazadas.

Antes de mirar la solución de este código es recomendable que os familiaricéis un poco con el concepto de **struct** y **class** de c++, ya que permita una solución del problema mucho más ordenada.

Código

C++

```cpp
#include <bits/stdc++.h>

using namespace std;

typedef vector<int> vi;
typedef vector<vi> vvi;

// Tablero de medida N x N
const int N = 8;

// Posiciones del rey negro
int rey_negro_x, rey_negro_y;

// Mira si la posicion esta en el tablero
inline bool inside_field(int x, int y) {
  return (x >= 0 and x < N and y >= 0 and y < N);
}

// distancia entre el la posicion y el rey negro
inline int dist(int x, int y) {
  return max(abs(x - rey_negro_x), abs(y - rey_negro_y));
}

// Este struct es para guardar la posicion que puede hacer mate
// El tipo es X si no hay mate
// El tipo es N si hay mas de un resultado
// En caso contrario el tipo es la pieza en (x,y)
struct Resultado {
  char tipo;
  int x, y;

  Resultado() {
    tipo = 'X';
  }

  void actualiza(char _t, int _x, int _y) {
    if (tipo != 'X' and _t != 'X') {
      tipo = 'N';
    }
```

```
40      else if (tipo == 'X') {
41        tipo = _t;
42        x = _x;
43        y = _y;
44      }
45    }
46  };
47
48  // Struct base de los diferentes tipos de piezas
49  // Todas las funcionas son las que tienen todos los tipos de pieza
50  struct Pieza {
51    int x_val, y_val;
52    char tipo;
53
54    Pieza(int _x, int _y) {
55      x_val = _x;
56      y_val = _y;
57    }
58
59    Pieza() {}
60
61    // Funcion para crear una nueva pieza del mismo tipo
62    virtual Pieza* nueva_pieza(int x, int y) = 0;
63
64    // Funcion para mover la pieza a todas la posiciones validas y mirar si hay mate
65    virtual void mueve(vvi& field, int ind, vector<Pieza*> blancas, Resultado&
      ↪  resultado) = 0;
66
67    // Marcar todas las posiciones que se pueden atacar
68    virtual void atacar(vvi& field) = 0;
69  };
70
71  // Mira si la posicion actual de las piezas es mate
72  bool es_mate(const vector<Pieza*>& blancas, vvi field) {
73    field[rey_negro_x][rey_negro_y] = 0;
74
75    // Marcamos la posiciones que se pueden atacar desde cada una de las piezas
76    for (auto pieza : blancas) {
77      pieza->atacar(field);
78    }
79
80    // Miramos si el rey tiene alguna opcion de movimiento
81    int valid_positions = 0;
82    for (int i = -1; i <= 1; ++i) {
83      for (int j = -1; j <= 1; ++j) {
84        int x = rey_negro_x + i;
85        int y = rey_negro_y + j;
86
87        if (inside_field(x,y)) {
88          ++valid_positions;
89
90          if (field[x][y]&2) {
91            --valid_positions;
92          }
93        }
94      }
95    }
```

```
96
97    // Si todas las posiciones posibles estan amenazadas es mate
98    return (valid_positions == 0);
99  }
100
101 // Struct de tipo Pieza que representa los movimientos y ataques del rey
102 struct Rey : public Pieza {
103
104   Rey(int _x, int _y) : Pieza(_x, _y) {
105     tipo = 'R';
106   }
107
108   Pieza* nueva_pieza(int x, int y) override {
109     return new Rey(x, y);
110   }
111
112   void mueve(vvi& field, int ind, vector<Pieza*> blancas, Resultado& resultado)
      ↪  override {
113     field[x_val][y_val] = 0;
114
115     for (int i = -1; i <= 1; ++i) {
116       for (int j = -1; j <= 1; ++j) {
117         if (i == 0 and j == 0) continue;
118
119         int x = x_val + i;
120         int y = y_val + j;
121
122         if (inside_field(x, y) and (field[x][y]&1) == 0 and dist(x, y) > 1) {
123           blancas[ind] = nueva_pieza(x, y);
124           field[x][y] = 1;
125
126           if (es_mate(blancas, field)) {
127             resultado.actualiza(tipo, x, y);
128           }
129
130           field[x][y] = 0;
131     }}}
132
133     field[x_val][y_val] = 1;
134   }
135
136   void atacar(vvi& field) override {
137     for (int i = -1; i <= 1; ++i) {
138       for (int j = -1; j <= 1; ++j) {
139         if (i == 0 and j == 0) continue;
140
141         int x = x_val + i;
142         int y = y_val + j;
143
144         if (inside_field(x,y)) {
145           field[x][y] |= 2;
146   }}}}
147 };
148
149 // Struct de tipo Pieza que representa los movimientos y ataques del alfil
150 struct Alfil : virtual public Pieza {
151
```

```
152   Alfil(int _x, int _y) : Pieza(_x, _y) {
153     tipo = 'A';
154   }
155
156   Alfil() {}
157
158   Pieza* nueva_pieza(int x, int y) override {
159     return new Alfil(x, y);
160   }
161
162   void mueve(vvi& field, int ind, vector<Pieza*> blancas, Resultado& resultado)
      ↪  override {
163     field[x_val][y_val] = 0;
164
165     for (int i = -1; i <= 1; i += 2) {
166       for (int j = -1; j <= 1; j += 2) {
167         int x = x_val + i;
168         int y = y_val + j;
169
170         while (inside_field(x, y) and (field[x][y]&1) == 0) {
171           blancas[ind] = nueva_pieza(x, y);
172           field[x][y] = 1;
173
174           if (es_mate(blancas, field)) {
175             resultado.actualiza(tipo, x, y);
176           }
177
178           field[x][y] = 0;
179
180           x += i;
181           y += j;
182     }}}
183
184     field[x_val][y_val] = 1;
185   }
186
187   void atacar(vvi& field) override {
188     for (int i = -1; i <= 1; i += 2) {
189       for (int j = -1; j <= 1; j += 2) {
190         int x = x_val + i;
191         int y = y_val + j;
192
193         while (inside_field(x,y)) {
194           field[x][y] |= 2;
195
196           if (field[x][y]&1) break;
197
198           x += i;
199           y += j;
200   }}}}
201 };
202
203 // Struct de tipo Pieza que representa los movimientos y ataques del caballo
204 struct Caballo : public Pieza {
205
206   Caballo(int _x, int _y) : Pieza(_x, _y) {
207     tipo = 'C';
```

```
208   }
209
210   Pieza* nueva_pieza(int x, int y) override {
211     return new Caballo(x, y);
212   }
213
214   void mueve(vvi& field, int ind, vector<Pieza*> blancas, Resultado& resultado)
      ↪  override {
215     field[x_val][y_val] = 0;
216
217     for (int i = -2; i <= 2; ++i) {
218       for (int j = -2; j <= 2; ++j) {
219         if (abs(i) + abs(j) != 3) continue;
220
221         int x = x_val + i;
222         int y = y_val + j;
223
224         if (inside_field(x, y) and (field[x][y]&1) == 0) {
225           blancas[ind] = nueva_pieza(x, y);
226           field[x][y] = 1;
227
228           if (es_mate(blancas, field)) {
229             resultado.actualiza(tipo, x, y);
230           }
231
232           field[x][y] = 0;
233     }}}
234
235     field[x_val][y_val] = 1;
236   }
237
238   void atacar(vvi& field) override {
239     for (int i = -2; i <= 2; ++i) {
240       for (int j = -2; j <= 2; ++j) {
241         if (abs(i) + abs(j) != 3) continue;
242
243         int x = x_val + i;
244         int y = y_val + j;
245
246         if (inside_field(x,y)) {
247           field[x][y] |= 2;
248   }}}}
249 };
250
251 // Struct de tipo Pieza que representa los movimientos y ataques de la torre
252 struct Torre : virtual public Pieza {
253
254   Torre(int _x, int _y) : Pieza(_x, _y) {
255     tipo = 'T';
256   }
257
258   Torre() {}
259
260   Pieza* nueva_pieza(int x, int y) override {
261     return new Torre(x, y);
262   }
263
```

```
264   void mueve(vvi& field, int ind, vector<Pieza*> blancas, Resultado& resultado)
      ↪  override {
265     field[x_val][y_val] = 0;
266
267     for (int i = -1; i <= 1; i++) {
268       for (int j = -1; j <= 1; j++) {
269         if (abs(i) + abs(j) != 1) continue;
270
271         int x = x_val + i;
272         int y = y_val + j;
273
274         while (inside_field(x, y) and (field[x][y]&1) == 0) {
275           blancas[ind] = nueva_pieza(x, y);
276           field[x][y] = 1;
277
278           if (es_mate(blancas, field)) {
279             resultado.actualiza(tipo, x, y);
280           }
281
282           field[x][y] = 0;
283
284           x += i;
285           y += j;
286     }}}
287
288     field[x_val][y_val] = 1;
289   }
290
291   void atacar(vvi& field) override {
292     for (int i = -1; i <= 1; i++) {
293       for (int j = -1; j <= 1; j++) {
294         if (abs(i) + abs(j) != 1) continue;
295
296         int x = x_val + i;
297         int y = y_val + j;
298
299         while (inside_field(x,y)) {
300           field[x][y] |= 2;
301
302           if (field[x][y]&1) break;
303
304           x += i;
305           y += j;
306   }}}}
307 };
308
309 // Struct de tipo Pieza que representa los movimientos y ataques de la dama
310 // Los movimientos y ataques son la suma de la torre y del alfil
311 struct Dama : public Torre, public Alfil {
312
313   Dama(int _x, int _y) {
314     tipo = 'D';
315     x_val = _x;
316     y_val = _y;
317   }
318
319   Pieza* nueva_pieza(int x, int y) override {
```

```
320       return new Dama(x, y);
321     }
322
323     void mueve(vvi& field, int ind, vector<Pieza*> blancas, Resultado& resultado)
        ↪  override {
324       Torre::mueve(field, ind, blancas, resultado);
325       Alfil::mueve(field, ind, blancas, resultado);
326     }
327
328     void atacar(vvi& field) override {
329       Torre::atacar(field);
330       Alfil::atacar(field);
331     }
332   };
333
334
335   // Funcion que que genera el tablero y miramos todos los movimientos
336   Resultado solution(const vector<Pieza*>& blancas) {
337     size_t n = blancas.size();
338     vvi field(N, vi(N, 0));
339
340     field[rey_negro_x][rey_negro_y] = 1;
341     for (auto piece : blancas) {
342       field[piece->x_val][piece->y_val] = 1;
343     }
344
345     Resultado resultado;
346     for (size_t i = 0; i < n; ++i) {
347       blancas[i]->mueve(field, i, blancas, resultado);
348     }
349
350     return resultado;
351   }
352
353   int main() {
354     char L, x_char, y_char;
355     while (cin >> L >> x_char >> y_char) {
356       rey_negro_x = int(x_char-'a');
357       rey_negro_y = int(y_char-'1');
358
359       int n;
360       cin >> n;
361       vector<Pieza*> blancas(n);
362       for (int i = 0; i < n; ++i) {
363         cin >> L >> x_char >> y_char;
364         int x = int(x_char-'a');
365         int y = int(y_char-'1');
366
367         switch(L) {
368           case 'R':
369             blancas[i] = new Rey(x, y);
370             break;
371           case 'C':
372             blancas[i] = new Caballo(x, y);
373             break;
374           case 'D':
375             blancas[i] = new Dama(x, y);
```

```
376             break;
377           case 'T':
378             blancas[i] = new Torre(x, y);
379             break;
380           case 'A':
381             blancas[i] = new Alfil(x, y);
382             break;
383           default:
384             blancas[i] = nullptr;
385         }
386       }
387
388       auto sol = solution(blancas);
389
390       if (sol.tipo == 'N') {
391         cout << ">1\n";
392       }
393       else if (sol.tipo == 'X') {
394         cout << "NO\n";
395       }
396       else {
397         cout << sol.tipo << char('a' + sol.x) << char('1' + sol.y) << '\n';
398       }
399     }
400   }
```