課題

地引　昌弘

2023.12.07

課題 1 — 驚きの手品

皆さんに嘘みたいな本当のお話を一つ紹介したいと思います。ある 2 人組の手品師のお話です (2 人の名前をそれぞれ甲 · 乙としましょう)。甲は 4 行 × 4 列の盤を用意し、観客から任意で選んだ被験者に 16 個のコインを表裏も含め自由に配置させます (図 1)。図では表を灰色、裏を黒としています。この間、乙は目隠しをされていて、どんな配置かを知ることはできません。次に被験者は全 16 マスのどれかを選びます。甲は被験者がマスを選んだ後、16 枚あるコインのうち一つだけを裏返します。そして最後に乙の目隠しが解かれ、乙が盤の配置を見ながら被験者の選んだマスを当てようというわけです。乙は盤の初期配置を知らないので、甲がどのコインを裏返したかも分かりません。そんな状況で、被験者の選んだマスを当てられるのでしょうか…。

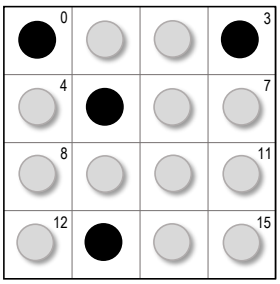

図 1: コインの配置例

x	y	$x \oplus y$
1	1	0
0	1	1
1	0	1
0	0	0

図 2: 排他的論理和の演算規則例

取り付く島もないように見えますが、実はこれを当てるための手助けとして**排他的論理和** (Exclusive Or) と呼ばれる演算があります (XOR 演算とも呼ばれます)。演算子には “$\oplus$” を使うことが多いです。$x \oplus y$ の値は、x == y の時は 0 に x != y の時は 1 になります (図 2)。これより、次のような興味深い結果が得られます (図 2 を参考に確認して下さい)。

i) $x \oplus x = 0$,　ii) $x \oplus 0 = 0 \oplus x = x$,　iii) $x \oplus y = y \oplus x$

では、いよいよ手品のタネを明かしましょう。甲がコインを裏返すマスは、被験者が選んだマスに応じて以下の手順で決定します。

1. 0 行 0 列 ∼ 3 行 3 列までの各マスに、0 ∼ 15 までの番号を割り当てます (図 1 に各マスへ割り当てられた番号をイメージする数字を入れました)。
2. 初期配置 (甲が裏返す前) にて裏向きのコインがあるマスの番号を変数 $\mathrm{T}_0, \ldots \mathrm{T}_n$ に代入します。この時、将来的に排他的論理和の演算を効果的に使うため、番号は十進数ではなく二進数として扱います。例えば、図 1 では 0, 3, 5, 13 番の 4 マスにあるコインが裏向きなので、これらを二進数に変換し、各 T の値は次のようになります。
   $\mathrm{T}_0 = 0000,\ \mathrm{T}_1 = 0011,\ \mathrm{T}_2 = 0101,\ \mathrm{T}_3 = 1101$
3. 被験者が選んだマスの番号を上と同様に二進数として変数 R に代入します。
4. 甲がコインを裏返したマスの番号を変数 X とします。
5. $\mathrm{S} = \mathrm{T}_0 \oplus \mathrm{T}_1 \oplus \cdots \oplus \mathrm{T}_n$ とした時、次の式を満たす X を求めます。
   $$\mathrm{X} = \mathrm{S} \oplus \mathrm{R} \qquad (1)$$

この手順で得られた X が示すコインを裏返すだけで、本当に被験者が選んだマスを発見できるのでしょうか。以下では、この手順で得られた X から、被験者が選んだ R を計算できる理由を説明します。

場合分け 1: 初期配置で X のマスにあるコインが表向きだった場合。甲によりこのコインは裏向きにされます。

式 (1) は、前ページで紹介した排他的論理和の特徴 i), ii) を用いて、次のように変形できます。

$$\mathrm{X} = \mathrm{S} \oplus \mathrm{R} \;\rightarrow\; \mathrm{S} \oplus \mathrm{X} = \mathrm{S} \oplus \mathrm{S} \oplus \mathrm{R} \;\rightarrow\; \mathrm{S} \oplus \mathrm{X} = 0 \oplus \mathrm{R} = \mathrm{R} \quad \cdots\cdots \quad (2)$$

乙は、甲がコインを裏返した後の盤を見ているので、初期配置の $\mathrm{T}_0, \ldots \mathrm{T}_n$ $(= \mathrm{S})$ と甲が裏返した X に裏向きのコインがあります。よって、これら全ての排他的論理和を計算することで、被験者が選んだ R を求めることができます。

場合分け 2: 初期配置で X のマスにあるコインが裏向きだった場合。甲によりこのコインは表向きにされます。

初期配置で裏向きだったコイン $\mathrm{T}_0, \ldots \mathrm{T}_n$ のうち、甲が表向きにしたコインを T_i とします。$\mathrm{X} = \mathrm{T}_i$ に注意して下さい。ここで、場合分け 1 と同様に式 (1) を式 (2) へ変形した後、さらに排他的論理和の特徴 iii) を用いて式 (2) を次のように変形できます。

$$\begin{aligned}
\mathrm{R} = \mathrm{S} \oplus \mathrm{X} \;&\rightarrow\; \mathrm{R} = \mathrm{T}_0 \oplus \cdots \oplus \mathrm{T}_{i-1} \oplus \mathrm{T}_i \oplus \mathrm{T}_{i+1} \oplus \cdots \oplus \mathrm{T}_n \oplus \mathrm{T}_i \\
&\rightarrow\; \mathrm{R} = \mathrm{T}_0 \oplus \cdots \oplus \mathrm{T}_{i-1} \oplus \mathrm{T}_{i+1} \oplus \cdots \oplus \mathrm{T}_n \oplus \mathrm{T}_i \oplus \mathrm{T}_i \\
&\rightarrow\; \mathrm{R} = \mathrm{T}_0 \oplus \cdots \oplus \mathrm{T}_{i-1} \oplus \mathrm{T}_{i+1} \oplus \cdots \oplus \mathrm{T}_n
\end{aligned}$$

この時も場合分け 1 と同じく、乙は甲がコインを裏返した後の盤を見ているので、初期配置の $\mathrm{T}_0, \ldots \mathrm{T}_n$ から甲が裏返した T_i を除く $\mathrm{T}_0, \ldots \mathrm{T}_{i-1}, \mathrm{T}_{i+1}, \ldots \mathrm{T}_n$ に裏向きのコインがあります。よって、これら全ての排他的論理和を計算することで、被験者が選んだ R を求めることができます。

にわかには信じられませんが、乙は目隠しを解かれた後、盤上で裏向きになっている全コインを対象に排他的論理和を計算するだけで、必ず被験者が選んだ R を発見できるというわけです。この議論では、“コインが表向きではなく裏向きである” ことを特に利用していないので、表向きのコインを使っても同じことができます。驚きですね。

課題 1 では、この手品を実行するプログラムを作ります。まずはその準備として、被験者がコインを配置した盤を作成する関数 `init_board1()` を用意しました (適当にコピーして下さい)。

```
def init_board1(k):

if (k == 1):
  _b = [[0, 1, 1, 0],
        [1, 0, 1, 1],
        [1, 1, 3, 1],
        [1, 0, 1, 1]]

elif (k == 2):
  size = 4
  _b = ita.array.make2d(size, size)
  for i in range(size):
    for j in range(size):
      _b[i][j] = int((10*random.random()) % 2)
  i = int(size*random.random())
  j = int(size*random.random())
  if (_b[i][j] == 0): _b[i][j] = 2
  else: _b[i][j] = 3

else:
  _b = None

return(_b)
```

`init_board1()` は、引数 `k` に 1 を渡すと、図 1 に対応した盤およびコインの初期位置を 2 次元の盤配列 `_b` として返します (`k` = 2 については、後述します)。盤配列 `_b` 内では、表 (or 裏) のコインが存在するマスを数値の 1 (or 0) で表わします (配列の各要素に対応する数値が格納されます)。また、被験者が選んだマスは、コインの表 (or 裏) に応じて数値の 3 (or 2) で表わします。これは、盤配列の要素が 3 (or 2) の場合、そのマスには表 (or 裏) のコインが存在していると考えて下さい。

以下では、i 行 · j 列のマスを座標に似せて < i, j > と表記することにします。(x-y 座標系と配列における i 行 · j 列とを混乱しないように)。`init_board1()` の動作イメージは下記の通り。

```
>>> b = init_board1(1)
>>> pprint.pprint(b, width=20)
[[0, 1, 1, 0],
 [1, 0, 1, 1],
 [1, 1, 3, 1],     # 被験者は<2,2>マスを選んでいる。値が 3 なので、ここには表のコインがあることに注意。
 [1, 0, 1, 1]]
```

課題 1-a: `init_board1(1)` により得られた盤配列 `b` に対し、次の仕様を満たす関数 `make_t()` を作成せよ。

1. `make_t()` は、盤配列 `b` を引数として受け取る。
2. 下記の手順で作成した 1 次元配列を要素とし、`b` 内の全裏向きコインについて格納した 2 次元配列 `T` を作成する。
   i. 図 1 のように、<0, 0> マス ～ <3, 3> マスへ 0 ～ 15 までの番号を割り当てる。
   ii. 裏向きのコインがあるマスの番号を二進数に変える。
   iii. ii で得られた二進数の 4 桁目を 0 番目、3 桁目を 1 番目 (以下同様)、とする 4 要素の 1 次元配列を作成する。
3. `T` を返す。
4. `b` 内に裏向きコインがない場合は `None` を返す。

2 次元配列 `T` は、1 ページで示した手順にある $T_0, \ldots T_n$ を格納した配列です。但し、排他的論理和の演算が簡略化できるように、T_i へは数値をそのまま格納するのではなく、二進数の各桁にある数値 0 or 1 を要素とする配列を格納しています (動作イメージを参照して下さい)。また、仕様 1-a-4 については、`make_t()` を課題 1-b で利用する際の準備です。`make_t()` の動作イメージは下記の通り。

```
>>> t = make_t(b)     # 盤配列 b については、上の動作イメージを参照のこと。
>>> pprint.pprint(t, width=20)
[[0, 0, 0, 0],        # 裏向きコインがある<0,0>マスの番号は 0 → 二進数 0000
 [0, 0, 1, 1],        # 同、<0,3>マスの番号は 3 → 二進数 0011
 [0, 1, 0, 1],        # 同、<1,1>マスの番号は 5 → 二進数 0101
 [1, 1, 0, 1]]
```

課題 1-b: `init_board1(1)` より得られた盤配列 `b` および `make_t()` より得られた 2 次元配列 `T` に対し、次の仕様を満たす関数 `find_x()` を作成せよ。

1. `find_x()` は、盤配列 `b` と 2 次元配列 `T` を引数として受け取る。
2. 下記の手順で作成した 1 次元配列 `X` を作成する。
   i. 図 1 のように、<0, 0> マス ～ <3, 3> マスへ 0 ～ 15 までの番号を割り当てる。
   ii. 被験者が選んだマスの番号を二進数に変える。
   iii. ii で得られた二進数の 4 桁目を 0 番目、3 桁目を 1 番目 (以下同様)、とする 4 要素の 1 次元配列 `R` を作成する。
   iv. `T` の各要素と `R` に対し、1 ページの式 (1) で示した排他的論理和の演算を行ない、その結果を iii と同様な形式で格納した 4 要素の 1 次元配列を作成する。
3. `X` を返す。

動作イメージは下記の通り。

```
>>> x = find_x(b, t)     # 盤配列 b および 2 次元配列 t については、上の動作イメージを参照のこと。
>>> print(x)
[0, 0, 0, 1]             # 二進数 0001 → 番号:1 → 甲が裏返すコインは<0,1>マス
```

乙は、甲が <0, 1> マスのコインを裏返した後に盤を見るので、排他的論理和は `T` の各要素に [0, 0, 0, 1] を加えた計算となります (場合分け 1)。その結果、下記の式より [1, 0, 1, 0] が得られます。同じ位置にある成分同士が計算対象となります (それほど面倒な計算ではないので、上の動作イメージにある 2 次元配列 `T` を対象に、各成分毎に縦方向の排他的論理和を計算してみて下さい)。二進数 1010 は十進数の 10 であり、番号 10 に該当するマスは <2, 2> です。これは確かに被験者が選んだマス (盤配列 `b` 内で数値 3 のマス) ですね。

$$(0,0,0,0) \oplus (0,0,1,1) \oplus (0,1,0,1) \oplus (1,1,0,1) \oplus (0,0,0,1) = (1,0,1,0)$$

最後に、一般化しておきましょう。

課題 1-c: `init_board1()` は引数 `k` に 2 を渡すと、被験者が選んだマスおよびコインの向きをランダムに決定した盤配列を返す。この任意な盤配列に対しても、`make_t()` および `find_t()` が課題 1-a, 1-b と同じ仕様を満たすように、必要があればそれぞれ修正せよ。

課題 1-b で一般化できていれば、特に課題 1-c で行なう作業はありません。一応、動作イメージを示しておきます。

```
>>> b = init_board1(2)
>>> pprint.pprint(b, width=20)
[[0, 0, 1, 0],
 [0, 0, 0, 0],
 [1, 0, 0, 2],     # 被験者は<2,3>マスを選んでいる。値が 2 なので、ここには裏のコインがあることに注意。
 [0, 0, 1, 1]]
>>> t = make_t(b)
>>> pprint.pprint(t, width=20)
[[0, 0, 0, 0],
 [0, 0, 0, 1],
 [0, 0, 1, 1],
 [0, 1, 0, 0],
 [0, 1, 0, 1],
 [0, 1, 1, 0],
 [0, 1, 1, 1],
 [1, 0, 0, 1],
 [1, 0, 1, 0],
 [1, 0, 1, 1],     # 被験者の選んだマス (番号:11, 二進数 1011) だが、コインは裏なので T に入れる。
 [1, 1, 0, 0],
 [1, 1, 0, 1]]
>>> x = find_x(b, t)
>>> print(x)
[0, 0, 0, 0]       # 二進数 0000 → 番号:0 → 甲が裏返すコインは<0,0>マス
```

初期配置で <0, 0> マスにあるコインは裏向き (数値 0) なので、この実行例は場合分け 2 に該当します。乙が盤を見る時は、<0, 0> マスにあるコインは表向きなので、上の 2 次元配列 `T` に対し、最初の要素を除いて各成分毎に縦方向の排他的論理和を計算してみます。すると今度は [1, 0, 1, 1] が得られました (これも各自で確認してみて下さい)。二進数の 1011 は十進数の 11 であり、番号 11 に該当するマスは <2, 3> です。今回も確かに被験者が選んだマス (盤配列 `b` 内で数値 2 のマス) になっていますね。

ちょっとワクワクしませんか?

追加要件:

- 十進数 → 二進数の変換は、Python が提供する関数 (例えば、`bin` 関数, `format` 関数など) を使わず、変換用コードを作成して下さい。
- 排他的論理和の計算については、Python の演算子 “^” を利用してもよいです。

課題 2 — ナイトの交換

次は、**チェス** (Chess) を題材としたプログラムを考えてみましょう。チェスは、古代インドで発明された**チャトランガ** (Chaturanga) を起源とするボード ゲームです。日本では将棋として派生し、各文化に応じてルールに多少の違いはあるものの、現代でも世界中に残っています。チェスは、8 行 × 8 列の計 64 マスからなる盤上に 6 種類の駒を並べ、自分と相手プレーヤとで交互に一駒ずつ動かして行きます。駒は、その種類に応じて動けるマスが決まっており、**キング** (King) と呼ばれる駒を取った方が勝ちとなります (相手キングが存在するマスに、自分駒のどれか (但し自分キング以外) が到達した時点で自分の勝ち)[1]。ところで、チェスで使われる駒の一つに、**ナイト** (Knight) と呼ばれる駒があります。この駒は騎士を表わしていると言われ、他の駒を飛び越えられる変わった動きをします (図 3)。ナイトの駒が図 3 のマス N にある場合、このナイトは 1 回の移動で赤丸マスのどれか一つに動くことができます。

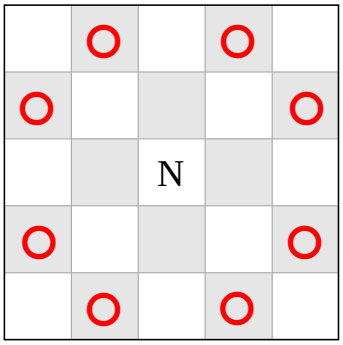

図 3: ナイトが 1 回の移動で到達できるマス

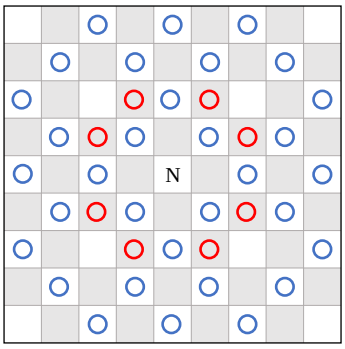

図 4: ナイトが 2 回の移動で到達できるマス

図 4 は、ナイトを 2 回動かした時に到達できるマスを示しています。N のマスにあるナイトは、1 回目で図 4 にある赤丸マス (のどれか) へ移動し、そこから 2 回目の移動で (最初の移動先から届くどれかの) 青丸マスへ到達できます。チェスの盤は白と黒の市松模様になっており、図 3 と図 4 を見比べてみると、各回では元の色と違う色のマスへ移動していることが分かります。ナイトの動きはこのように独特であり、数理科学の世界において、これまで様々なモチベーションを与えて来ました (ある意味、課題 1 で取り上げた排他的論理和なども、その一つと言えるかも知れません)。以下では、チェスの勝負から少し離れ、このナイトの動きに纏わるアルゴリズムを考えてみましょう。

さて、今回はナイトを使った図 5 のパズルを解いてみます。このパズルは、ナイトの移動規則に従って 3 個の白ナイト (図内では灰色で表記) と黒ナイトを順番に移動させ、その位置を入れ替えます。但し、相手の駒を取ってはいけません (つまり、駒が存在するマスへは移動できない)。このパズルは、**ナイトの交換** (Knight Exchage, Knight Swap) と呼ばれており、以下では便宜上、“K.E.” と呼ぶことにします。

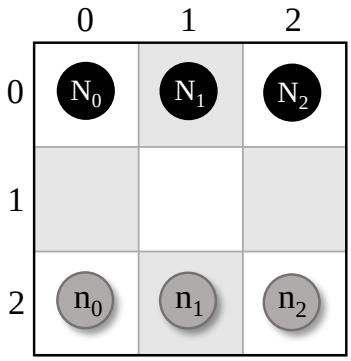

図 5: ナイトの交換 (3 行 × 3 列盤)

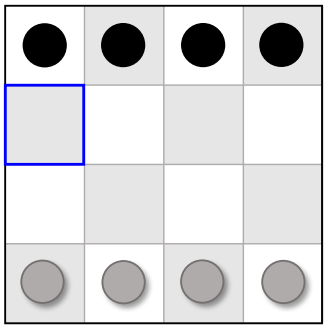

図 6: ナイトの交換 (4 行 × 4 列盤)

[1] とは言え、将棋やチェスでは、(一度に動けるマスは決まっていますが) 王将 or キングの駒は盤上のどこにでも自由に動けるので、相手の駒が迫って来ると当然逃げます。つまり、そう簡単には相手キングのマスに自分の駒を到達させることはできません。参考ですが、中国にもチャトランガを由来とする**シャンチー** (象棋) と呼ばれるゲームがあります。シャンチーでは、盤の中央に盤を二分する河と呼ばれるマス行があり (河を渡れる駒は制限される)、また、王将 or キングに該当する駒 (帥将と呼ばれる) が初期位置からあまり動けない (つまり、皇帝は王宮から外に出られない) といった、中国の歴史を反映したルールになっています。

実際に解いてみると、K.E.はあまり難しくないことが分かります。以下では、課題 1 と同じく i 行 · j 列のマスを座標に似せて < i, j > と表わした解の一つを示します (再掲しますが、x-y 座標系と配列における i 行 · j 列とを混乱しないように)。この例解では、8 手で白ナイト/黒ナイトの位置を入れ替えています。

1. $n_0 :< 2, 0 > \rightarrow < 1, 2 >$ **2.** $N_1 :< 0, 1 > \rightarrow < 2, 0 >$ **3.** $n_2 :< 2, 2 > \rightarrow < 0, 1 >$ **4.** $N_2 :< 0, 2 > \rightarrow < 1, 0 >$
5. $n_1 :< 2, 1 > \rightarrow < 0, 2 >$ **6.** $N_0 :< 0, 0 > \rightarrow < 2, 1 >$ **7.** $n_0 :< 1, 2 > \rightarrow < 0, 0 >$ **8.** $N_2 :< 1, 0 > \rightarrow < 2, 2 >$

図 7: 3 行 × 3 列盤におけるナイトの交換手順

課題 2 では、K.E.を解く (上記のような解を出力する) プログラムを作ります。まずはその準備として、今回もパズルを作成する関数 `init_board2()` を用意しました (適当にコピーして下さい)。

```
def init_board2(k):
  _b1 = [[2, 2, 2],
         [0, 0, 0],
         [1, 1, 1]]
  _b2 = [[2, 2, 2, 2],
         [9, 0, 0, 0],
         [0, 0, 0, 0],
         [1, 1, 1, 1]]
  _b = None
  if (k == 1):
    _b = _b1
  elif (k == 2):
    _b = _b2
  return(_b)
```

`init_board2()` は、引数 `k` に 1 を渡すと、図 5 に対応した盤およびナイトの初期位置を 2 次元の盤配列 `_b` として返します ($k = 2$ については、後述します)。盤配列 `_b` 内では、白 (or 黒) ナイトが存在するマスを数値の 1 (or 2) で表わし、駒が存在しないマスを 0 で表わします (配列の各要素に対応する数値が格納されます)。動作イメージは下記の通り。

```
>>> b = init_board2(1)
>>> pprint.pprint(b, width=20)
[[2, 2, 2],
 [0, 0, 0],
 [1, 1, 1]]
```

課題 2-a: `init_board2(1)` により得られた盤配列 `b` に対し、次の仕様を満たす関数 `can_knt_mv()` を作成せよ。

1. `can_knt_mv()` は、盤配列 `b` および `b` 内の座標 <i 行, j 列> と座標 <p 行, q 列> を引数として受け取る (引数: 5 個)。
2. <i, j> マスにあるナイトが移動規則に従い <p, q> マスへ移動できる場合は、`True` を返す。
3. 移動できない場合は、`False` を返す。

動作イメージは下記の通り。移動できない指定は他にも考えられるので、全てのパターンを踏まえて**抜けがないように**作って下さい。

```
>>> b = init_board2(1)
>>> pprint.pprint(b, width=20)
[[2, 2, 2],
 [0, 0, 0],
 [1, 1, 1]]
>>> can_knt_mv(b, 2, 0, 1, 2)    # 左下隅の<2,0>マスにある白 (1) ナイトは、<1,2>へ移動できる。
True
>>> can_knt_mv(b, 2, 0, 0, 1)    # 同、<0,1>には黒 (2) ナイトがあるので移動できない (駒は取れない)。
False
>>> can_knt_mv(b, 2, 0, 1, 1)    # <2,0>→<1,1>の移動は、ナイトの移動規則に反する。
False
```

では、K.E.の解を求めるプログラムの作成に取り組んでみましょう。表記の煩雑さを避けるため、以後は黒ナイトの初期位置を “白ゴール”、白ナイトの初期位置を “黒ゴール” と呼ぶことします。K.E.を解く基本的な手順 (疑似コード) は、次のようになります。

1: 白ゴール上にない白ナイトが存在する or 黒ゴール上にない黒ナイトが存在する間、以下を繰り返す。
   1-A: 白ゴール以外に存在する白ナイトのうち、ナイトの移動規則に従い移動可能な白ナイトを探す。
   1-B: 発見した白ナイトを移動可能なマスへ移動する。
   1-C: 黒ゴール以外に存在する黒ナイトのうち、ナイトの移動規則に従い移動可能な黒ナイトを探す。
   1-D: 発見した黒ナイトを移動可能なマスへ移動する。
   1-E: (繰り返しはここまで)

2: 仕様として指定された値を返して終了

ナイトは図 3 (5 ページ) で示したように、8 方向へ移動できます。しかし、毎回 8 方向を調べるのでは計算量が増大する可能性があるため、取り敢えず今回はゴールに近付く 4 方向だけを調べることにします (以後、探索の効率化を目指すこのような方針を方針 I と呼ぶことにします)。但し、これは探索範囲を狭める行為なので、一般には “計算が終わるかどうか/解が得られるかどうか” のバランスを見ながら、調整して行く必要があります (付録を参照)。

課題 2-b: `init_board2(1)` により得られた盤配列 `b` に対し、次の仕様を満たす関数 `ex_knt1()` を作成せよ(10 ページの注意事項を確認のこと)。

1. `ex_knt1()` は、盤配列 `b` を引数として受け取る (`init_board2(1)` の引数に注意)。
2. ナイトの移動に際しては、課題 2-a で作成した `can_knt_mv()` を利用してその可否を調べる。
3. 最終的な盤配列および得られた解の解配列 `s` を返す。
4. 解配列 `s` は、“移動回数 × 4” の 2 次元配列で、各要素には移動した順番毎に、移動元のマス <i, j> および移動先のマス <p, q> が格納される (動作イメージを参照)。

`can_knt_mv()` は、その仕様に合った意味のある利用をして下さい (無意味な利用は、仕様未達とします)。動作イメージは下記の通り。表記の便宜上、`init_board2()` が返す盤配列は `b0` で受け、`ex_knt1()` が返す盤配列は `b1` で受けています。

```
>>> b0 = init_board2(1)
>>> pprint.pprint(b0, width=20)     # 白 (1)/黒 (2) ナイトの初期位置
[[2, 2, 2],
 [0, 0, 0],
 [1, 1, 1]]
>>> b1, s = ex_knt1(b0)
>>> pprint.pprint(b1, width=20)     # パズル終了時の配置: 両ナイトの位置が入れ替わっている。
[[1, 1, 1],
 [0, 0, 0],
 [2, 2, 2]]
>>> pprint.pprint(s, width=20)      # 最終的な移動回数は不明なので、解配列の長さは適宜調整すること。
[[2, 0, 1, 2],                      # 1 手目: 白ナイトの移動<2,0>→<1,2>を意味
 [0, 1, 2, 0],                      # 2 手目: 黒ナイトの移動<0,1>→<2,0>を意味
 [2, 2, 0, 1],                      # 3 手目以降: 意味は上二つと同じ。
 [0, 2, 1, 0],
 [2, 1, 0, 2],
 [0, 0, 2, 1],
 [1, 2, 0, 0],
 [1, 0, 2, 2]]
```

続いて、K.E.を 4 行 × 4 列盤 (図 6, 5 ページ) に拡張した場合を考えてみます。交換するナイトの数が増えるので、一見難しそうに見えますが、移動可能なマス (つまり空きマス) の数はそれ以上に増えるので、実は 3 行 × 3 列盤より簡単に解くことができます。そこで、空きマスの数を減らすため、そのうち一つは利用不可という制約を入れてみましょう。

例えば、図 6 のパズルにおいて青マスを使わないとしたら、どうなるでしょうか。以下に実例をお見せしますが、この制約は大変厳しく、パズルが途端に難しくなります。よって最後は、制約付きの 4 行 × 4 列盤 K.E. を解くプログラムにチャレンジしてみます。`init_board2()` は、引数 `k` に 2 を渡すと、利用不可のマスを追加した 4 行 × 4 列盤の盤配列を返します。盤配列では、利用不可のマスを 9 で表わしています。今回はこれを利用することにしましょう。

さて、前ページの疑似コードで示した手順により、制約付き 4 行 × 4 列盤を解いてみると、途中でどの白ナイトも移動できない手詰まりになってしまいました (7 ページの下線で示した方針 I の扱いに注意して下さい[2])。その時点までの解配列も表示しておきます。

```
>>> pprint.pprint(b, width=20)    # 手詰まりになった時点における盤の状況
[[1, 1, 0, 1],
 [9, 0, 1, 0],
 [0, 0, 0, 2],
 [2, 0, 2, 2]]
>>> pprint.pprint(s, width=20)    # 同、その時点までの手順
[[3, 0, 1, 1],
 [0, 0, 2, 1],
 [3, 1, 1, 2],
 [0, 1, 2, 0],
 [3, 2, 1, 3],
 [0, 2, 2, 3],
 [1, 2, 0, 0],
 [0, 3, 2, 2],
 [3, 3, 1, 2],
 [2, 0, 3, 2],
 [1, 1, 0, 3],
 [2, 1, 3, 3],
 [1, 3, 0, 1],
 [2, 2, 3, 0]]
```

解行列より、最後の移動である黒ナイト:<2, 2>→<3, 0>を戻し (図 8-(a))、別の黒ナイトを<2, 3>から<3, 1>へ移動させてみても (図 8-(b))、白ナイトを移動させることができません。つまり、2 手前にある白ナイト:<1, 3>→<0, 1>が良くなかったわけで (この移動をされてしまうと、以後は黒ナイトがどこに移動しても手詰まりとなる)、これも戻す必要があります (図 8-(c))。しかし、図 8-(c) では、白ナイトは<1, 3>から<0, 1>への移動しかできません。よってこれよりさらに遡り、黒ナイト:<2, 1>→<3, 3>が良くなかったと言えます。以後はキリがありませんが、黒ナイト:<2, 1>→<3, 3>を戻した局面では (図 8-(d))、黒ナイトがどこに移動しようが、やはり白ナイトの手番では<1, 3>から<0, 1>への移動しかできません。これは、4 手前となる白ナイト:<1, 1>→<0, 3>の時点で、将来的な手詰まりが決まっていたわけです。実は、ここで例示した手順について神様視点から話をすれば、初手を白ナイト:<3, 0>→<1, 1>にした場合、次に黒ナイトを<0, 0>から<2, 1>へ動かしてしまうと解がありません。なんと最初の 2 手目で既に誤っていたのです。

```
[[1, 1, 0, 1],      [[1, 1, 0, 1],      [[1, 0, 0, 1],      [[1, 0, 0, 1],
 [9, 0, 1, 0],       [9, 0, 1, 0],       [9, 0, 1, 1],       [9, 0, 1, 1],
 [0, 0, 2, 2],       [0, 0, 2, 0],       [0, 0, 2, 2],       [0, 2, 2, 2],
 [0, 0, 2, 2]]       [0, 2, 2, 2]]       [0, 0, 2, 2]]       [0, 0, 2, 0]]

      (a)                 (b)                 (c)                 (d)
```

図 8: 1 手ずつ後戻りさせた局面

とは言え、通常は神様視点などできないので、どれが最善手か (手詰まりにならないか) は分かりません。そこで、取り敢えず動かしてみて、手詰まりとなった場合は別の移動先を探る (要は、別の候補を採用してやり直す) という方針を考えてみることにします。これを方針 II と呼ぶことにしましょう。次ページに、手詰まりの解消を目指す方針 II の手順を示します。方針 II には手順の後戻りが入っていることに注意して下さい。

[2] これは、ナイトの移動を制約することで、探索の効率化を図る方針でした (8 方向探索 → 4 方向探索)。ここでは、ナイトをゴールから遠ざかる方向へ移動させません。一見、制約を緩和して 8 方向探索にすることで解が得られるように見えますが、手詰まりの局面で例えば<0, 3>にある白ナイトを<2, 2>へ後戻りさせても、次の手番ではこの白ナイトを再び<0, 3>へ移動するしかなく (他の白ナイトは動けない)、同じ手順の繰り返しに陥ってしまいます (<0, 1>の白ナイトも同じ)。つまり、問題の本質は、何手か先を読み、手詰まり or 繰り返しにならない移動先を選択しなければならない点にあります。要は、この下線部分をどう作るかということです。

i. 最後に動かしたナイトを、まだ試していない別方向へ移動できるならば、その方向へ動かして探索を続ける。
ii. どの方向へも移動できない or どの方向へ移動しても最終的な手詰まりが分かった場合は、そのナイトの移動はあきらめ、まだ動かしていない別のナイトを選んで探索を続ける。
iii. 全てのナイトを試しても解を得られなかった場合は手順を 1 手戻し、1 手戻した局面でまだ動かしていないナイトを選んで探索を続ける (1 手戻すの具体的な処理は下の疑似コードを参照のこと)。

今回の課題で導入した方針 I · 方針 II はそれぞれ、**ヒューリスティク法** (Heuristic Method) および**バックトラック法** (Backtrack Method) と呼ばれています。計算機で扱う問題の一つに、大量の選択肢から条件に合致する解を探し出すという問題があります (文字通り探索問題と呼ばれています)。数値解析のように、効率的に解を求める一般化されたアルゴリズム (例えばオイラー法など) が存在すれば、それを実行すればよいのですが、探索問題の多くにはそのようなアルゴリズムが存在しません。よって、解の候補を総当たりで調べることになります。しかし、これでは計算量の観点から実用的ではないため、問題の構造や制約の特徴により少しでも効率が良さそうなアルゴリズムを発見して行く (heuristic) わけです。但し、あくまで "良さそうな" というレベルなので、必ず解を探せるわけではありません。そこで、これ以上探索を進めても (手順を進めても) 解は存在しないことが判明したら、その探索を終了して別の解候補を選択できるまで戻り (backtrack)、新たな探索を開始する必要があります。バックトラックを行なう場合、正しく後戻りができるように、必要な記録を残しておく必要があります。どのタイミングでどんな記録を残すべきかの判断は相当に複雑であり (そして、不要になったら消去する必要もある)、一般にループを用いたアルゴリズムの作成は難しくなります。これに対し再起呼び出しでは、呼び出し先から戻って来た段階で (状況に応じて副作用を修正する必要はあるものの) 呼び出し前のデータに戻っているため、タイミング · 記録は形式的 (or 自動的) に決まります。よって、バックトラックが入る処理を作成する際の見通しは良くなります。再起呼び出しを用いたヒューリスティク法 + バックトラック法による K.E.の解を探す手順の一例 (疑似コード) は、以下のようになります。但し、白ナイトと黒ナイトを交互に移動させる必要があるので、下請けルーチンをうまく作成して、見通し良く作って下さい。

1: 白/黒全てのナイトがゴールに到達したら**探索成功**を返す。
2: 手番に応じて盤上のナイトを一つずつ調べて行く (探索ループ)。
  2-A: ゴールへ 2 行近付く+自マスより左方向 (‡1) へ移動できるか。
    2-A-a: ‡1 へ移動させ (盤配列/解配列を修正し)、それ以降を再起呼び出しにより探索
    2-A-b: 再起呼び出しから**探索成功**を返された場合、解を発見できたので探索ループを抜ける。
    2-A-c: 再起呼び出しから**探索失敗**を返された場合、‡1 への移動では最終的に解を発見できなかったので、別方向を調べるため盤配列/解配列を原状復帰
  2-B: ゴールへ 2 行近付く + 自マスより右方向へ移動できるか。
    2-B-□: 手順 2-A-a/b/c と同様な処理を実行
  2-C: ゴールへ 1 行近付く + 自マスより左方向へ移動できるか。
    2-C-□: 手順 2-A-a/b/c と同様な処理を実行
  2-D: ゴールへ 1 行近付く + 自マスより右方向へ移動できるか。
    2-D-□: 手順 2-A-a/b/c と同様な処理を実行
  2-E: (繰り返しはここまで)
3: 探索結果を返す。

(手順 2 の探索ループを**探索失敗**で抜けて来た場合は、再帰の呼び出し元へ**探索失敗**が返る。
その後、再帰の呼び出し元では以下の二つが行なわれる → これが 1 手戻すの処理に該当
- 盤配列/解配列の原状復帰 (手順 2-A-c)
- 別方向 or 別ナイトの探索 (手順 2-A/B/C/D or これら全てが失敗となり手順 2 で別ナイトを選択))

課題 2-c: `init_board2(2)` により得られた盤配列 `b` に対し、課題 2-b と同じ仕様を満たす関数 `ex_knt2()` を作成せよ (10 ページの注意事項を確認のこと)。必要があれば `can_knt_mv()` を修正してもよい。

動作イメージは次ページの通り。表記の便宜上、`init_board2()` が返す盤配列は `b0` で受け、`ex_knt2()` が返す盤配列は `b1` で受けています。

```
>>> b0 = init_board2(1)
>>> pprint.pprint(b0, width=20)    # 白 (1)/黒 (2) ナイトの初期位置
[[2, 2, 2, 2],
 [9, 0, 0, 0],
 [0, 0, 0, 0],
 [1, 1, 1, 1]]
>>> b1, s = ex_knt2(b0)
>>> pprint.pprint(b1, width=20)    # パズル終了時の配置: 両ナイトの位置が入れ替わっている。
[[1, 1, 1, 1],
 [9, 0, 0, 0],
 [0, 0, 0, 0],
 [2, 2, 2, 2]]
>>> pprint.pprint(s, width=20)     # 最終的な移動回数は不明なので、解配列の長さは適宜調整すること。
[[3, 0, 1, 1],                     # 1 手目: 白ナイトの移動<3,0>→<1,1>を意味
 [0, 0, 1, 2],                     # 2 手目: 黒ナイトの移動<0,0>→<1,2>を意味
 [3, 1, 2, 3],                     # 3 手目以降: 意味は上二つと同じ。
 [0, 1, 2, 0],
 [3, 2, 1, 3],
 [0, 2, 2, 1],
 [2, 3, 0, 2],
 [0, 3, 2, 2],
 [1, 1, 0, 3],
 [1, 2, 3, 1],
 [3, 3, 1, 2],
 [2, 0, 3, 2],
 [1, 2, 0, 0],
 [2, 1, 3, 3],
 [1, 3, 0, 1],
 [2, 2, 3, 0]]
```

＊注意事項

- 課題 2-c で作成する ex_knt2() が課題 2-b の仕様も満たしている場合は、ex_knt1() の兼用として ex_knt2() だけを提出してもよいです (その旨の説明を添付して下さい)。
- 課題 1/2 共に、各仕様を完備していない場合でも提出は受理します。

付録: 探索範囲と計算の効率化について (理解を深めるために)

上でお見せした ex_knt2() の実行例では、370,883 回のバックトラック (後戻り) が発生しました。チェスや将棋のようなボード ゲームでは、駒が盤の端より中央のマスにある方が動ける範囲は広いので、重要になります。そこで、制約を右のように中央のマスへ変えた (制約を厳しくした) 盤配列に対して ex_knt2() を実行してみると、なんとバックトラックが 10,000,000 回発生しても解を得られません。これは、再帰呼び出しが深くなっても、それに合わせて対象が小さくならない (指数的に探索範囲が狭くならない) ことに原因があります。よって以下では、発見的 (ヒューリスティク) な効率化を探ってみることにします。まずは、前ページの手順 2-A/B/C/D を if 文の並列 (‡2) ではなく、if - elif (‡3) に代えてみます。‡2 では、再帰呼び出しから探索失敗を返された場合、例えば手順 2-A-c の次に手順 2-B を実行しますが、‡3 では、手順 2-B へ移らず手順 2 (探索ループ) を抜けるので、探索範囲が狭まっていると言えます (無駄な探索を減らせれば嬉しい)。‡3 をそのまま実行してみると、残念ながら解はありませんでした (459,991 回のバックトラックを行ない、可能な範囲を全て調べたが解はなかった)。そこで探索範囲を少し広げるため、移動方向を増やすことにします。新たに、“i) ゴールから 1 行遠ざかる + 自マスより左方向” と “ii) ゴールから 1 行遠ざかる + 自マスより右方向” の二つを追加し、6 方向の移動に変えてみます (手順が 2-A ～ G に増えるわけです)。但し、後戻りの移動を許すと、5 ページ脚注のような振動が発生する恐れが出て来ます。その理由ですが、n 手目にナイトをマス a から b へ移動させ、n+2 手目に b から a へ戻したとします。その後、n+4 手目の再帰呼び出しは n 手目の再帰呼び出しと違うので、そこでは b はまだ移動したことのないマスとして扱われるからです。試しに実行してみると、今度はバックトラックを 480 回行なうだけで解が得られました。これもまた、ちょっと驚きですが、このように探索可能な範囲の調整は一筋縄では行かないところがあります (ここが腕の見せ所とも言えますが)。

```
>>> pprint.pprint (b0,  width=20)
[[ 2,  2,  2,  2 ],
 [ 0,  9,  0,  0 ],
 [ 0,  0,  0,  0 ],
 [ 1,  1,  1,  1 ]]
```