課題

地引　昌弘

2024.12.05

課題 1 — ハノイの塔

再帰の考え方を用いることで、見通しが良くなる問題の一つに**ハノイの塔** (Tower of Hanoi) があります。これは、次のようなパズルです (御存知の方も多いとは思いますが)。

- パズルは、3 本の塔 (A ～ C) と中央に穴の開いた大きさの異なる複数の円盤 (1 ～ n) から構成されます。
- パズルの初期状態では、塔 A に全ての円盤が小さいものを上として順に積み重ねられています。
- 次にこの円盤を 1 枚ずつ別の塔へ移します。但し、小さな円盤の上に大きな円盤を乗せてはいけません。 . . . . ‡1
- 上の規則に従い円盤を移して行き、最終的に塔 A にある全円盤を塔 C に移すことができれば、パズルは完成です。

ハノイの塔の概要を図 1 に、‡1 の規則を図 2 に示します。

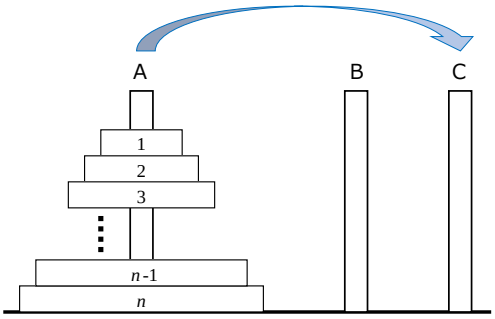

図 1: ハノイの塔の目標

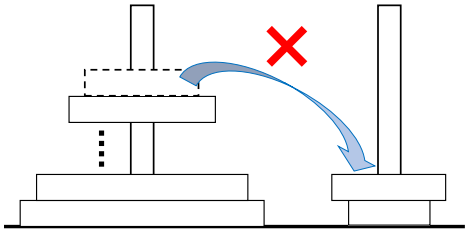

図 2: 円盤を移す際の制約

このパズルを解く手順 (円盤の動かし方) は数多く存在しますが、多くの人の関心を惹いたのは最短手順の探求でした。例えば、円盤の総数が n 枚の時、最短何手で完成できるかという問題です。一見して考えられる手順の種類があまりに多く、取り付く島もないようにも見えますが、上の規則 ‡1 について少し慎重に考えてみると、一番大きな円盤 n を塔 C に移す場合、塔 C には他の円盤があってはならないことが分かります。なぜならば、円盤 n は一番大きな円盤であり、もし塔 C に円盤があればそれがどんな円盤であっても円盤 n より小さいため、‡1 により円盤 n を塔 C に移せないからです。これより、次のことが言えます。

“塔 A にある円盤 n を塔 C に移す直前では、塔 B に円盤 $1 \sim n-1$ が置かれていなければならない。” . . . . . ‡2

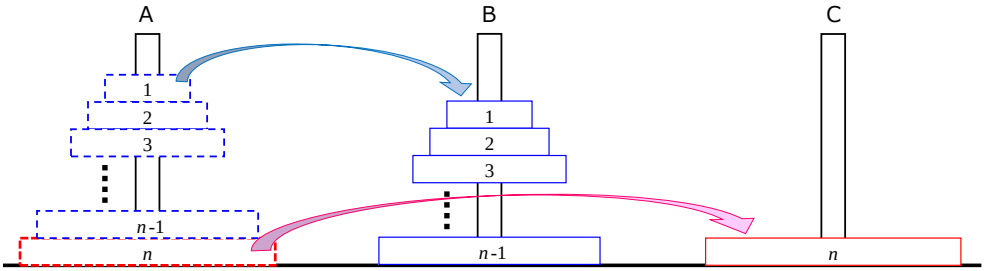

図 3: ハノイの塔を解くヒント

さて、ここで問題を整理してみます。まずは、n 枚の円盤をある塔 (例えば塔 A) から別の塔 (例えば塔 C) へ移す手数を $M_3(n)$ とします。‡2 より、この手順では、まず $n-1$ 枚の円盤をある塔 (例えば塔 B) へ移し (手数: $M_3(n-1)$)、次に

円盤 n を別の塔 (例えば塔 C) へ移して (手数: +1)、最後に $n-1$ 枚の円盤を円盤 n のある塔 (例えば塔 C) へ移します (手数: $M_3(n-1)$)。つまり、次の漸化式が成り立つわけです (再帰の考え方が利用されていますね)。

$$M_3(n) = 2M_3(n-1) + 1 \quad \cdots\cdots \ddagger 3$$

以上の議論より、円盤の総数が n 枚の時にハノイの塔を完成する最短の手数 $M_3(n)$ を求めることができました。では、最短の手順 `S(n)` を求めるにはどうしたらよいでしょうか。これも上の議論を参考にすると、次のように考えることができます。

1. まずは、$n-1$ 枚の円盤を最短の手順 `S(n-1)` により、ある塔 (例えば塔 B) へ移す (図 3 の青移動)。
2. 次に円盤 n を別の塔 (例えば塔 C) へ移す (図 3 の赤移動)。
3. 最後に、$n-1$ 枚の円盤を最短の手順 `S(n-1)` により、円盤 n のある塔 (例えば塔 C) へ移す。

$\cdots\cdots \ddagger 4$

これより、円盤数が n 枚の時にハノイの塔を完成する最短の手順 `S(n)` では、円盤数が $n-1$ 枚における最短の手順 `S(n-1)` を利用していることが分かります。つまり、あるサイズの問題を対象とした手順の中で、自身より小さいサイズの問題を対象とした同じ手順を呼び出しているわけで、これはまさに再帰処理と言えます。

今回の課題では、このハノイの塔を題材としたプログラムの作成に取り組むこととします。まずはその準備として、ハノイの塔の初期状態を作成する関数 `init_tower3()` を用意しました (適当にコピーして下さい)。

```
import ita
def init_tower3(size):
  src = ita.array.make1d(size+1); src[0] = 'src'
  dst = ['dst']
  wrk = ['wrk']
  for i in range(size):
    src[i+1] = i+1
  return(src, dst, wrk)
```

`init_tower3()` では、円盤の枚数を引数 `size` として渡されるとハノイの塔の初期状態を以下のようにモデル化します。

- 三つの塔を一次元配列 `src`, `dst`, `wrk` として作成する。
- 各塔配列の先頭には、塔を識別するための名前'src', 'dst', 'wrk' が入る。 $\cdots\cdots \ddagger 5$
- `size` 枚の円盤は 1 ～ size の数値で表わす。数値の小さい方が小さい円盤となる。
- ある塔 T の上から i 枚目に円盤 j がある場合は、塔配列およびその要素を用いて `T[i] = j` と扱う (添字番号に注意)。

初期状態では、塔配列 `src` に全ての円盤が置かれています (図 3 の塔 A に該当)。パズルでは、これらの円盤を塔配列 `wrk` を作業用の塔 (同、塔 B) として利用しながら、塔配列 `dst` に移す (同、塔 C) ことを目指します。`init_tower3()` の動作イメージは下記の通り。

```
>>> src, dst, wrk = init_tower3(3)
>>> print(src, dst, wrk)
['src', 1, 2, 3] ['dst'] ['wrk']
```

この例では円盤の枚数を 3 枚としました。塔配列 `src` には、上から順に円盤 1, 2, 3 が入っています。塔配列 `dst`, `wrk` には円盤は 1 枚もありません。ここで一つ重要な注意があります。塔 T_1 にある円盤を塔 T_2 に移すとは、塔 T_1 の一番上にある円盤を塔 T_2 の一番上に移すことを意味します。円盤の移動では、各塔配列の添字番号に十分注意して要素の挿入/削除を行なって下さい。

課題 1-a: `init_tower3()` により得られた塔配列 `a`, `b` に対し、次の仕様を満たす関数 `chk_mv()` を作成せよ。

1. `chk_mv()` は、塔配列 `a`, `b` を引数として受け取る。
2. ハノイの塔の規則に従い塔配列 `a` から `b` へ円盤を移動できる場合は、`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。円盤を 5 枚として、ある程度手順が進んだ状態を示しています。移動できない場合は他にも考えられるので、全ての場合を踏まえて**抜けがないように**作って下さい。

```
>>> print(src, dst, wrk)
['src', 4, 5] ['dst', 3] ['wrk', 1, 2]
>>> chk_mv(src, dst)    # 塔 src の頂上円盤は塔 dst の頂上円盤より大きいので、src から dst へは移動不可
False
>>> chk_mv(wrk, dst)    # 同、塔 wrk は塔 dst より小さいので、wrk から dst へは移動可能
True
```

では、いよいよハノイの塔を解くプログラムを作ることにします。ハノイの塔を解く手順は、‡4 (2 ページ) で示した通り、再帰処理としてモデル化することができます。以下では、この再帰処理を関数 r_hanoi3() と表わし、r_hanoi3() の疑似コードを考えてみることにします。r_hanoi3() の基本動作は、次のようになります。

- 塔配列 src, dst, wrk を引数として受け取る。
- wrk を作業用の塔として利用し、src から dst へ n 枚の円盤を移す。

n 枚の円盤を移す r_hanoi3() では、その内部で n-1 枚の円盤を移すために r_hanoi3() を再帰的に呼び出すことになります。この時、‡4 より、最初は src から wrk へ n-1 枚の円盤を移し、その後 wrk から src へ n-1 枚の円盤を移すことに注意して下さい。また、再帰処理における重要部分は、再帰呼び出しを終了する条件の定義および再帰元に戻った際の扱いです。例えば、再帰元に戻った際に副作用はあるのか or 次に実行すべき処理 (更に戻るのか、別の処理を行なうのか) などの検討です。以上より、ハノイの塔を解く再帰処理部分の疑似コードは、次のようになります。この疑似コードでは、手順 2 および 4 が再帰呼び出しとなります。

r_hanoi3(src, n, dst, wrk): wrk を作業用として利用し、src から dst へ n 枚の円盤を移動する。

1: 移動する円盤がない時は、何もしないで戻る。
2: dst を作業用として利用し、src から wrk へ n-1 枚の円盤を移す。
3: src から dst へ 1 枚の円盤を移す。
4: src を作業用として利用し、wrk から dst へ n-1 枚の円盤を移す。
5: 指定された n 枚の円盤を src から dst へ移したので戻る。

課題 1-b: init_tower3() より得られた各塔配列に対し、次の仕様を満たす関数 honoi3() を作成せよ。

1. hanoi3() は、塔配列 src, dst, wrk を引数として受け取る (引数: 4 個)。
2. ハノイの塔の規則に従い、src にある全ての円盤を dst へ移す。
3. 円盤の移動に際しては、関数 chk_mv() を用いて移動の可否を確認すること。
4. パズルの正解手順を解配列 ans として返す。
5. 解配列 ans は、“要素数 = 移動回数” の 2 次元配列で、各要素には移動した順番毎に、[移動元の塔名, 移動する円盤の番号, 移動先の塔名] が格納される (塔名は ‡5,2 ページを利用/動作イメージを参照)。

hanoi3() の作成では、内部で r_hanoi3() の手順やその考え方をうまく利用して下さい。また、chk_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。この例では円盤の枚数を 3 枚とし、円盤を 7 回移動させることでパズルを完成させています。この移動回数は、漸化式 ‡3 (2 ページ) より得られる $M_3(n) = 2^n - 1$ $(n = 3)$ を満たしています。

```
>>> src, dst, wrk =init_tower3(3)    # 円盤 3 枚における塔の初期状態は以前の動作イメージを参照のこと。
>>> ans = hanoi3(src, dst, wrk)
>>> print(src, dst, wrk)
['src'] ['dst', 1, 2, 3] ['wrk']     # パズル終了後の状態
>>> pprint.pprint(ans, 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。
[['src', 1, 'dst'],                  # 1 手目: 塔 src から塔 dst へ円盤 1 を移動
 ['src', 2, 'wrk'],                  # 2 手目: 塔 src から塔 wrk へ円盤 2 を移動
 ['dst', 1, 'wrk'],                  # 3 手目以降: 意味は上二つと同じ。
 ['src', 3, 'dst'],
 ['wrk', 1, 'src'],
 ['wrk', 2, 'dst'],
 ['src', 1, 'dst']]
```

課題 2 — ハノイの塔の拡張

課題 1 で紹介したハノイの塔は塔が 3 本でしたが、塔の本数を増やしたらどうなるでしょうか。直観的には、円盤を移動させる状況が緩和される (移動の可能性を持つ塔が増える) ので、パズルを完成させる手順が短くなると予想できます。では、具体的な最短手順はどうなるでしょうか。

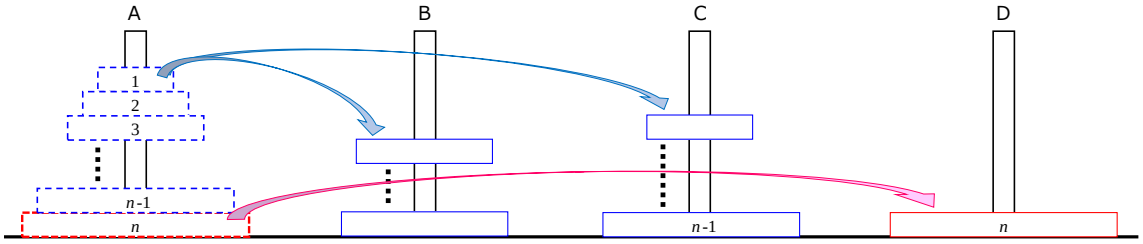

図 4: 塔が 4 本の場合

塔が 3 本の場合と同様に考えると、図 4 で示すように、塔 A にある円盤 n を塔 D に移す直前では、塔 B および塔 C に円盤 $1 \sim n-1$ が置かれていなければなりません。但し、塔 B·C における円盤 $1 \sim n-1$ の置き方については、特に条件はありません。ハノイの塔の規則に従っていればよいだけです。しかし、このままでは移動の自由度が高く、漸化式 ‡3 (2 ページ) を導いたような規則性を見出すことは少々難しく思えます。そこで、もう少し考えてみましょう。

まずは、円盤 n を置く前、円盤 $n-1$ を置いた時点を考えてみます。これは、図 4 において塔 D に円盤 $n-1$ だけが置かれている状態 (‡6) です。 その後、パズルを完成させるには、塔 D で円盤 n の上に円盤 $n-1$ を置かなければならないため、次の手順が必要になります。

1. 円盤 $n-1$ を塔 D に移す (これが ‡6)。
2. 円盤 $n-1$ を塔 B or C (の一番下) に移す。
3. 円盤 n を塔 D に移す。
4. 円盤 $n-1$ を塔 D に移す。

ここで着目するのは手順 2 です。仮に円盤 $n-1$ を塔 C に移したとします[1]。この時点では、塔 B $\sim$ D にある円盤では円盤 $n-1$ が一番大きいので、塔 C には円盤 $n-1$ だけが置かれることになります。その後、手順 3 で円盤 n を塔 D に移しますが、この時は図 4 のように塔 D には円盤 n だけが置かれることになります。つまり、手順 2 では塔 C に円盤 $n-1$ だけ、手順 3 では塔 D に円盤 n だけが置かれるわけです。これより手順 1 の直前では、塔 B に円盤 $1 \sim n-2$ を移して塔 C および塔 D を空けておけば (図 5 ①)、以後は “塔 C に円盤 $n-1$ (同 ②)” $\rightarrow$ “塔 D に円盤 n (同 ③)” $\rightarrow$ “塔 D に円盤 $n-1$ (同 ④)” と移動できるので、都合が良さそうです (以後、“**古典的アルゴリズム**” と呼ぶことにします)。

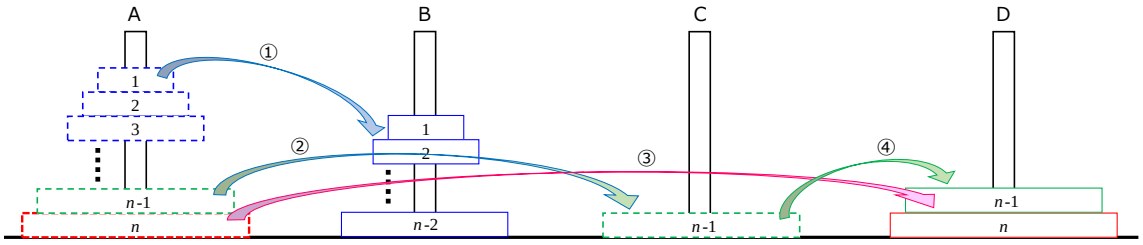

図 5: 4 本の塔を用いた円盤の移動

この方法でパズルを完成させる手数を $M_4(n)$ とすると、漸化式 ‡3 (2 ページ) と同様に、次の漸化式が成り立ちます。塔 A からは円盤を 2 枚ずつ移動させるので、漸化式にある n は 2 ずつ変化することに注意して下さい。

$$M_4(n) = 2M_4(n-2) + 3 \quad \cdots\cdots \ddagger 7$$

簡略化のため n を偶数とすると、この漸化式の一般項は $M_4(n) = 3(2^{\frac{n}{2}} - 1)$ となり、$M_3(n)$ の平方根で済むことが分かります。確かに塔を増やすと、劇的に手数は減りますね。

次の課題では、塔を 4 本に拡張したハノイの塔を古典的アルゴリズムで解くプログラムを作成します。塔を 3 本にした場合と同じく、ハノイの塔の初期状態を作成する関数 `init_tower4()` を用意しました (適当にコピーして下さい)。

[1] 円盤 $n-1$ を塔 D に置いたまま円盤 n を塔 C へ置くとした場合も、最後は円盤 n の上に円盤 $n-1$ を置くことになるので同じ議論ができます。

init_tower4() では、作業用の塔配列として wrk1, wrk2 の二つを作成します。他の仕様については、init_tower3() と同じです。

```
import ita

def init_tower4(size):
  src = ita.array.make1d(size+1); src[0] = 'src'
  dst = ['dst']
  wrk1 = ['wrk1']
  wrk2 = ['wrk2']
  for i in range(size):
    src[i+1] = i+1
  return(src, dst, wrk1, wrk2)
```

塔を 4 本に拡張したハノイの塔を解く古典的アルゴリズムの手順は、塔を 3 本とした場合と同様な再帰処理としてモデル化できます。この再帰処理を関数 r_hanoi4a() と定義すると、r_hanoi4a() の疑似コードは次のようになります。この疑似コードでは、手順 2 および 6 が再帰呼び出しとなります。

r_hanoi4a(src, n, dst, wrk1, wrk2): wrk1,wrk2 を作業用として利用し、src から dst へ n 枚の円盤を移動する。

1: 移動する円盤がない時は、何もしないで戻る。
2: dst, wrk2 を作業用として利用し、src から wrk1 へ n-2 枚の円盤を移す。
3: src から wrk2 へ 1 枚の円盤を移す。
4: src から dst へ 1 枚の円盤を移す。
5: wrk2 から dst へ 1 枚の円盤を移す。
6: src, wrk2 を作業用として利用し、wrk1 から dst へ n-2 枚の円盤を移す。
7: 指定された n 枚の円盤を src から dst へ移したので戻る。

但し、ここで一つ重要な考慮が必要です。4 本の塔では src から dst へ円盤を 2 枚ずつ移動するため、その前処理として $n-2$ 枚の円盤を作業用の塔に移します。つまり、1 回の再帰呼び出しで n を 2 ずつ減らすわけです。この時、n の偶奇によって手順 1 にある “移動する円盤がない” 場合の判断をどう扱えばよいか、よく考えて下さい。また、手順 3·4 では src から 2 枚の円盤を移動させていますが、“src から必ず 2 枚の円盤を移動できるか” どうか、これも n の偶奇に応じてよく考えて下さい。

課題 2-a: init_tower4() より得られた各塔配列に対し、次の仕様を満たす関数 honoi4a() を作成せよ。

1. hanoi4a() は、塔配列 src, dst, wrk1, wrk2 を引数として受け取る (引数: 4 個)。
2. ハノイの塔の規則に従い、古典的アルゴリズムを利用して src にある全ての円盤を dst へ移す。
3. 円盤の移動に際しては、関数 chk_mv() を用いて移動の可否を確認すること (利用上の注意は課題 1-b と同じ)。
4. パズルの正解手順を解配列 ans として返す (解配列の仕様は課題 1-b と同じ)。

hanoi4a() の作成では、内部で r_hanoi4a() の手順やその考え方をうまく利用して下さい。動作イメージは下記の通り。この例では円盤の枚数を 6 枚としました。

```
>>> src, dst, wrk1, wrk2 = init_tower4(6)
>>> print(src, dst, wrk1, wrk2)
['src', 1, 2, 3, 4, 5, 6] ['dst'] ['wrk1'] ['wrk2']
>>> ans = hanoi4a(src, dst, wrk1, wrk2)
>>> pprint.pprint(ans,width=25)
[['src', 1, 'wrk2'],
 ['src', 2, 'dst'],
 ['wrk2', 1, 'dst'],
 ['src', 3, 'wrk2'],
 ['src', 4, 'wrk1'],
 ['wrk2', 3, 'wrk1'],
 ['dst', 1, 'wrk2'],
 ['dst', 2, 'wrk1'],
 ['wrk2', 1, 'wrk1'],
 ['src', 5, 'wrk2'],
 ['src', 6, 'dst'],
 ['wrk2', 5, 'dst'],
 ['wrk1', 1, 'wrk2'],
 ['wrk1', 2, 'src'],
 ['wrk2', 1, 'src'],
 ['wrk1', 3, 'wrk2'],
 ['wrk1', 4, 'dst'],
 ['wrk2', 3, 'dst'],
 ['src', 1, 'wrk2'],
 ['src', 2, 'dst'],
 ['wrk2', 1, 'dst']]
>>> len(ans)            # 手数
21
```

最短手順の追求

塔を 4 本に拡張したハノイの塔を解く古典的アルゴリズムの手順は、円盤 n のある塔 A と円盤 $1 \sim n-2$ を一時的に避難させる作業用の塔一つを除き (つまり、2 本の塔を除き)、塔 A から “塔の総数 - 2” 枚ずつの円盤を移動させる手順と考えることができます。これを一般化すれば、塔を p 本に拡張したハノイの塔を効率的に解く手順とは、円盤 n のある塔 A と円盤 $1 \sim n-x$ を一時的に避難させる作業用の塔一つを除き、塔 A から “$p-2\ (=x)$” 枚ずつの円盤を移動させる手順 (‡8) と言うことができそうです。 この考え方は長く支持され、‡8 が最短手順だと考えられて来ました。

しかし、1941 年に反例が見つかります。この反例は、J.S.Frame と B.M.Stewart により示されました (以後、“**FS アルゴリズム**” と呼ぶことにします)。それは次のような手順です。

1. 塔 A にある上から k 枚の円盤を、塔 4 本を用いて作業用の塔 (例えば塔 B) へ移す (図 6 ①)。
2. k 枚の円盤を塔 B へ残したまま、塔 A にある残りの $n-k$ 枚を、塔 3 本 (塔 A,C,D) を用いて塔 D へ移す (図 6 ②)。
3. 塔 B にある k 枚の円盤を、塔 4 本を用いて塔 D へ移す (図 6 ③)。

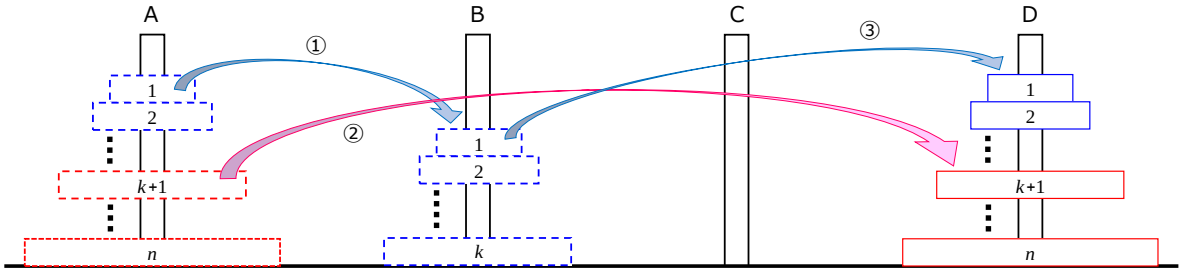

図 6: Frame-Stewart アルゴリズム

以下では、FS アルゴリズムによりどのくらい手順を減らせるのか、実際に作成したプログラムを用いて評価してみましょう。

まずは簡略版を作ってみます。FS アルゴリズムの簡略版では、塔 4 本に古典的アルゴリズムを用いた手順と塔 3 本の手順を組み合わせます。図 6 で示した手順に対応した疑似コードは次のようになります。

簡略版 FS アルゴリズム (‡9):

1: 塔 A にある全ての円盤 (ここでは n 枚) を、4 本の塔を用いて塔 D に移動させる。
2: 手順 1 により得られた解配列を最短手順として記録する。
3: 4 本版ハノイの塔を利用する円盤数 (ここでは 0 枚) を記録する。
4: 4 本版ハノイの塔を利用する円盤数 k を増やしながら、最短手数を調べて行く (探索ループ)。
  4-A: 各塔配列および解配列を初期化
  4-B: 塔 A にある k 枚の円盤を、4 本の塔を用いて作業用の塔 (例えば塔 B) に移動させる。
  4-C: 塔 A にある $n-k$ 枚の円盤を、3 本の塔を用いて塔 D に移動させる。
  4-D: 塔 B にある k 枚の円盤を、4 本の塔を用いて塔 D に移動させる。
  4-E: 手順 4-B ~ 4-D により得られた解配列の長さを調べる。
    4-E-a: これまでの最短手順より短ければ、新たに発見された最短手順として解配列を記録する。
    4-E-b: 4 本版ハノイの塔を利用する円盤数 k を記録する。
  4-F: (繰り返しはここまで)
5: 結果を返す。

上の疑似コードでは、塔 4 本を利用した移動に古典的アルゴリズムを利用しますが、幾つか注意することがあります。まずは手順 4-A です。プログラムの作り方によっては、手順 1 や手順 4-B ~ 4-D でパズルを完成させた後、塔配列が最終状態 (パズルが成功した状態) になっている場合があるので、これらを初期化しないと条件を変えて新たにパズルを解くことができません。また、手順 4-E で調べる解配列は、手順 4-B ~ 4-D を通じた解配列なので、これもプログラムの作り方によっては、その扱いに配慮が必要です。

課題 2-b: `init_tower4()` より得られた各塔配列に対し、次の仕様を満たす関数 `honoi4b()` を作成せよ。

1. `hanoi4b()` は、塔配列 `src, dst, wrk1, wrk2` を引数として受け取る (引数: 4 個)。
2. ハノイの塔の規則に従い、簡略版 FS アルゴリズムを利用して `src` にある全ての円盤を `dst` へ移す。
3. 円盤の移動に際しては、関数 `chk_mv()` を用いて移動の可否を確認すること (利用上の注意は課題 1-b と同じ)。
4. 塔 4 本を利用する枚数 `k` およびを解配列 `ans` を返す (解配列の仕様は課題 1-b と同じ)。

`hanoi4b()` の作成では、内部で `r_hanoi3()/r_hanoi4a()` の手順やその考え方をうまく利用して下さい。動作イメージは下記の通り。この例では円盤の枚数を 6 枚としました。簡略版 FS アルゴリズムでは、まずは 3 枚の円盤を 4 本の塔を用いて移動し、その後残りの 3 枚を 3 本版ハノイの塔として移動する手順が最短でした。円盤 6 枚/塔 4 本を完成させる場合、古典的アルゴリズム (`hanoi4a()`) では 21 手必要でしたが、簡略版 FS アルゴリズム (`hanoi4b()`) では 17 手で完成できます。

```
>>> src, dst, wrk1, wrk2 = init_tower4(6)
>>> print(src, dst, wrk1, wrk2)
['src', 1, 2, 3, 4, 5, 6] ['dst'] ['wrk1'] ['wrk2']
>>> k, ans = hanoi4b(src, dst, wrk1, wrk2)
>>> pprint.pprint(ans,width=25)
[['src', 1, 'dst'],
 ['src', 2, 'wrk2'],
 ['src', 3, 'wrk1'],
 ['wrk2', 2, 'wrk1'],
 ['dst', 1, 'wrk1'],
 ['src', 4, 'dst'],
 ['src', 5, 'wrk2'],
 ['dst', 4, 'wrk2'],
 ['src', 6, 'dst'],
 ['wrk2', 4, 'src'],
 ['wrk2', 5, 'dst'],
 ['src', 4, 'dst'],
 ['wrk1', 1, 'src'],
 ['wrk1', 2, 'wrk2'],
 ['wrk1', 3, 'dst'],
 ['wrk2', 2, 'dst'],
 ['src', 1, 'dst']]
>>> print(k, len(ans))  # 初期移動数/手数
3 17
```

究極の手順

課題 2-b で作成した簡略版 FS アルゴリズムは、確かに古典的アルゴリズムより手順は減っていますが、最短手順の追求という観点ではまだ改良の余地があります。(既にお気付きかも知れませんが) FS アルゴリズムでは、最初に k 枚の円盤を 4 本の塔を用いて移動しますが (図 6 ①)、簡略版 FS アルゴリズムではここに古典的アルゴリズムを用いています。課題 2-b で見たように、簡略版とは言え古典的アルゴリズムより FS アルゴリズムの方が手順を減らせるのであれば、最初に k 枚の円盤を動かす場合も、FS アルゴリズムを利用すればさらに手順を減らせると考えられます (対称性より、最初に移動した k 枚を塔 D に移す手順も同じことが言えます)。これを完全版 FS アルゴリズムと呼ぶことにします。

完全版 FS アルゴリズムの基本的な手順は、疑似コード ‡9 と同じですが、大きな違いは手順 1, 4-B, 4-D が再帰呼び出しとなる点です。これより、疑似コード ‡9 を、`r_hanoi3()` や `r_hanoi4a()` のように再帰処理 `r_hanoi4c()` として修正する必要があります (`r_hanoi3()/r_hanoi4a()` の手順やその考え方をうまく利用して下さい)。その際、注意の必要な部分は疑似コード ‡9 と同様、手順 4-A や 4-E に関連した処理ですが、再帰処理であることに十分な考慮が必要です。例えば、円盤 k 枚/各塔の状態 S_i を対象とした G_i 世代の `r_hanoi4c()`$^{(i)}$ から、$k = \sum k_j$ を満たす k_p に対して G_{i+1} 世代の `r_hanoi4c()`$^{(i+1)}$ を呼び出したとします (‡10)。 `r_hanoi4c()`$^{(i)}$ では、`r_hanoi4c()`$^{(i+1)}$ から戻った後、次は k_q について調べる (k_q を対象に `r_hanoi4c()`$^{(i+1)}$ を呼び出す) ことになるのですが (‡11)、 ‡10 と ‡11 では、各塔の状態 S_i や解配列がどうなっている必要があるのか、(プログラムの作り方によりますが) よく考えて下さい (情報をどう受け渡すのか/退避や初期化がいるのかなど)。

課題 2-c: `init_tower4()` より得られた各塔配列に対し、次の仕様を満たす関数 `honoi4c()` を作成せよ。

1. `hanoi4c()` は、塔配列 `src, dst, wrk1, wrk2` を引数として受け取る (引数: 4 個)。
2. ハノイの塔の規則に従い、完全版 FS アルゴリズムを利用して `src` にある全ての円盤を `dst` へ移す。
3. 円盤の移動に際しては、関数 `chk_mv()` を用いて移動の可否を確認すること (利用上の注意は課題 1-b と同じ)。
4. 戦略配列 `kl` およびを解配列 `ans` を返す (解配列の仕様は課題 1-b と同じ)。
5. 戦略配列は、塔 4 本/塔 3 本のどちらの方法で何枚の円盤を移動させたかを記録する 2 次元配列で、選択した順番毎に [移動元の塔名, 移動先の塔名, 戦略, 移動枚数] が格納される (動作イメージを参照)。戦略については、4 本の塔を用いて移動する場合は `@4`、3 本の塔を用いて移動する場合は `@3` と表記する。

動作イメージは下記の通り。この例では円盤の枚数を 10 枚としました。完全版 FS アルゴリズムでは、塔 4 本/塔 3 本の切り替えを複数回行なっていることが分かりますね (完全版 FS アルゴリズムでも、塔 4 本を用いて円盤を移動する際は古典的アルゴリズムを利用している点に注意して下さい)。解配列は大きいので、手数のみ (49 手) を表示しました。

```
>>> src, dst, wrk1, wrk2 = init_tower4(10)
>>> print(src, dst, wrk1, wrk2)
['src', 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] ['dst'] ['wrk1'] ['wrk2']
>>> kl, ans = hanoi4c(src, dst, wrk1, wrk2)
>>> pprint.pprint(kl, width=50)
[['src', 'dst', '@4', 3],      # 塔 4 本を利用して src から dst へ 3 枚を移動
 ['src', 'wrk1', '@3', 3],     # 塔 3 本を利用して src から wrk1 へ 3 枚を移動 (src は残り 4 枚)
 ['dst', 'wrk1', '@4', 3],     # 塔 4 本を利用して dst から wrk1 へ 3 枚を移動 (wrk1 上に 6 枚)
 ['src', 'dst', '@3', 4],      # 塔 3 本を利用して src から dst へ 4 枚を移動
 ['wrk1', 'src', '@4', 3],     # 塔 4 本を利用して wrk1 から src へ 3 枚を移動
 ['wrk1', 'dst', '@3', 3],     # 塔 3 本を利用して wrk1 から dst へ 3 枚を移動
 ['src', 'dst', '@4', 3]]      # 塔 4 本を利用して src から dst へ 3 枚を移動 (パズル完成)
>>> print(len(ans))
49
```

では最後に、三つの手法を比較してみましょう。以下では、円盤の枚数を 12 枚として、各手法の所要時間/手数/付加情報を表示させています。

```
>>> import time
>>> src, dst, wrk1, wrk2 = init_tower4(12)
>>> print(src, dst, wrk1, wrk2)
>>> start = time.process_time()
>>> ans = hanoi4a(src, dst, wrk1, wrk2)
>>> finish = time.process_time()
>>> print("C: %g %d" % ((finish-start), len(ans)))
['src', 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] ['dst'] ['wrk1'] ['wrk2']
C: 0 189                                    # 古典的アルゴリズムの結果
>>> src, dst, wrk1, wrk2 = init_tower4(12)
>>> start = time.process_time()
>>> k, ans = hanoi4b(src, dst, wrk1, wrk2)
>>> finish = time.process_time()
>>> print("FS1: %g %d k=%d" % ((finish-start), len(ans), k))
FS1: 0.015625 89 k=7                        # 簡略版 FS アルゴリズムの結果
>>> src, dst, wrk1, wrk2 = init_tower4(12)
>>> start = time.process_time()
>>> kl, ans = hanoi4c(src, dst, wrk1, wrk2)
>>> finish = time.process_time()
>>> print("FS2: %g %d len(kl)=%d" % ((finish-start), len(ans), len(kl)))
FS2: 32.3906 81 len(kl)=7                   # 完全版 FS アルゴリズムの結果
```

この結果を見ると、FS アルゴリズムは古典的アルゴリズムに比べて手数を大きく減らせることが分かります。また、興味深いのは FS アルゴリズムの簡略版と完全版との比較です。パズル完成までの手数は、完全版で 81 手、簡略版で 89 手でした。但し、完全版では 7 種類の戦略を駆使しているものの (再帰を何度も繰り返して最適な戦略の組合せを探索した)、簡略版では最初に 7 枚を古典的アルゴリズムで移動させるだけです。これにより、完全版では 32.4 秒もの時間を要しますが、簡略版は 16 ミリ秒しか掛かりません。つまり、この問題では性能を 1 割向上するために、コストが 2,000 倍ほど必要というわけです。円盤の枚数が増えれば、その差はもっと大きくなるでしょう。システムの実用性という観点から見ると、考えさせられますよね。

＊留意事項

- ハノイの塔を完成させる手順には様々な方法がありますが、今回の課題では再帰処理を利用して解いて下さい。
- 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(どこまで作ったかを説明して下さい)。

後記

最後に、ハノイの塔を完成させる最短手順およびその手数について、少し紹介しておきます。塔が 3 本の場合は、図 3 (1 ページ) で示す手順が絶対に必要なので、これが最短手順であり、また漸化式 ‡3 より得られる $M_3(n) = 2^n - 1$ が最短手数であることが分かります。では、塔を 4 本にした場合、完全 FS アルゴリズムは最短手順なのでしょうか。塔 4 本では、図 3 (1 ページ) のような絶対に必要な配置を特定できません。また、枚数に応じて戦略の組合せも変わります。よって、完全 FS アルゴリズムが最短手順であることの証明については、議論があります。

円盤の枚数を n 枚、塔の本数を r 本とした場合、完全 FS アルゴリズムを用いた最短手数の漸化式 $M(n,r)$ は、次のように考えることができます。

1. k 枚の円盤を古典的アルゴリズムで移動する手数: $M(k,r)$
2. $n-k$ 枚の円盤を塔を一つ減らした状態で移動する手数: $M(n-k,r-1)$
3. $n-k$ 枚の円盤を塔を古典的アルゴリズムで移動する手数: $M(k,r)$
4. $k' = \underset{1 \leq k < n}{\operatorname{argmin}} \big(M(n,r) = 2M(k,r) + M(n-k,r-1)\big) \Rightarrow M(n,r) = 2M(k',r) + M(n-k',r-1)$

k' は、$M(n,r) = 2M(k,r) + M(n-k,r-1)$ を最小にする k を意味します。この k' に対して、最終的な完全 FS アルゴリズムによる最短手数 $M(n,r)$ が決まるわけです。$M(n-k,r-1)$ を求めるには、$M(\square, r-2)$ の計算が必要な点に注意して下さい。つまり、塔の本数が r 本の場合、$r = \sum r_i$ を満たす全 r_i の組み合わせに応じた塔の組み合わせを調べる必要があり、これが完全 FS アルゴリズムの一般化となります。k' は n, r に応じて変わり、残念ながら $M(n,r)$ はあまり綺麗な式で表わせません。円盤の枚数を $n = {}_{r-3+x}C_{r-2} + a,\ 0 < a \leq {}_{r-3+x}C_{r-3}$ とすると、$M(n,r) = M(n-a,r) + a2^x$ であることが示されています (円盤を $n-a$ 枚から n 枚に増やすと、手数は $a2^x$ だけ増える)。参考までに、塔の本数 r を 5 本にした時の、完全 FS アルゴリズムの手数を以下に示します (簡略化のため $M(n) = M(n,5)$ と表記)。

$x = 0$:
$n = {}_{2+x}C_3 + a = {}_2C_3 + a = 0 + a,\ 0 < a = {}_{2+x}C_2 = {}_2C_2 = 1,\ (n,a) = (1,1)$
$M(1) = M(n-a) + a2^x = M(1-1) + 1 * 2^0 = 1$

$x = 1$:
$n = {}_{2+x}C_3 + a = {}_3C_3 + a = 1 + a,\ 0 < a = {}_{2+x}C_2 = {}_3C_2 = 3,\ (n,a) = (2,1), (3,2), (4,3)$
$M(2) = M(n-a) + a2^x = M(2-1) + 1 * 2^1 = M(1) + 1 * 2^1$
$M(3) = M(n-a) + a2^x = M(3-2) + 2 * 2^1 = M(1) + 2 * 2^1$
$M(4) = M(n-a) + a2^x = M(4-3) + 3 * 2^1 = M(1) + 3 * 2^1$

$x = 2$:
$n = {}_{2+x}C_3 + a = {}_4C_3 + a = 4 + a,\ 0 < a = {}_{2+x}C_2 = {}_4C_2 = 6,\ (n,a) = (5,1), (6,2), (7,3), \cdots, (10,6)$
$M(5) = M(n-a) + a2^x = M(5-1) + 1 * 2^2 = M(4) + 1 * 2^2$
$\vdots$
$M(10) = M(n-a) + a2^x = M(10-6) + 6 * 2^2 = M(4) + 6 * 2^2$

最後に肝心な、塔を 4 本以上にしたハノイの塔における最短手順について、少し触れておきます。計算機による全探索により、完全 FS アルゴリズムを用いた手順が最短であることが分かっています。但し、全ての n や r について探索することはできないので、完全 FS アルゴリズムは最短手順であると認識されていますが、証明はされていません (計算機で探索できる範囲では、反例は見つかっていないという状況)。