コンピュータとプログラミング C言語編

6. 再帰呼び出し

阿部洋丈 / ABE Hirotake
habe@cs.tsukuba.ac.jp

再帰呼び出し (recursive call)

- 関数定義の中から自分自身を呼び出すこと
  - 繰り返し(iteration)を表現する方法の一種
- for や while などの繰り返しを使って書くのに比べ、再帰で書くと記述が簡単になることがある
  - 再帰的に定義された関数やアルゴリズムを直接的に表現
  - データ構造の探索（次週詳しく説明）
- そのかわり、繰り返しを使って書くよりも時間やメモリを消費する場合がある

最初の例：フィボナッチ数列

$$F(n) = \begin{cases} 1 & (n = 1) \\ 1 & (n = 2) \\ F(n-1) + F(n-2) & (n > 2) \end{cases}$$

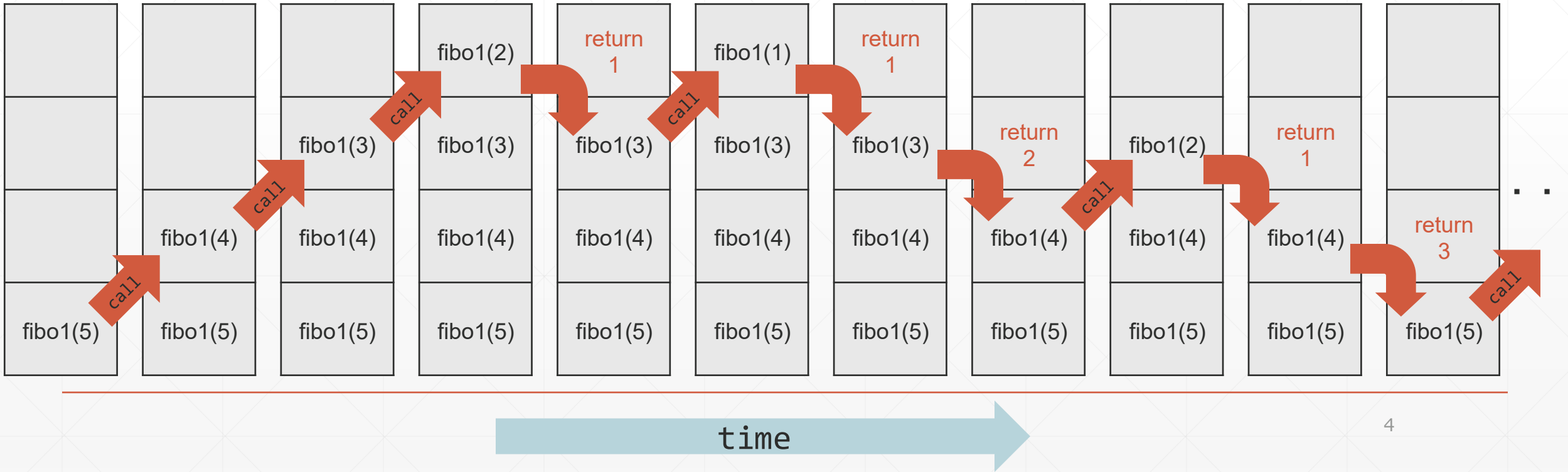
```
int fibo1(int n) {
  if (n == 1 || n == 2) {
    return 1;
  } else {
    return fibo1(n-1) + fibo1(n-2);
}
```

関数定義の中に自分自身への呼び出しを含んでいる

fibo1 の実行の様子

```
int fibo1(int n) {
  if (n == 1 || n == 2) {
    return 1;
  } else {
    return fibo1(n-1) + fibo1(n-2);
}
```

- 実行時スタックが伸び縮みしながら計算が進む
  - それぞれのスタックが引数や途中経過を保持している
- fibo1(5) の場合：



再帰呼び出しを使わない場合

```
int fibo2(int n) {
  int a = 1, b = 1, c, i;
  for (i = 3; i <= n; i++) {
    c = a + b;
    b = a;
    a = c;
  }
  return a;
}
```

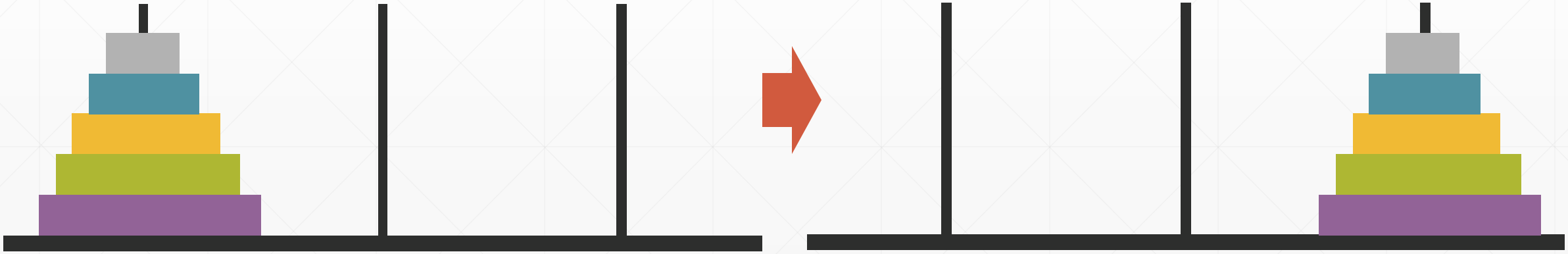
- 再帰呼び出しを使った関数は繰り返しでも実現出来る
- 繰り返しで書いた方が効率が良くなる場合がある
  - フィボナッチは効率が格段に良くなる

再帰呼び出しを使う場合の注意

- 呼び出しの連鎖が永遠には続かないようにする
  - フィボナッチ数列の場合は、「nが徐々に減少する」という性質と「nが2以下であればそれ以上呼び出さない」というルールのおかげで停止した
- 実行時スタックが深くなり過ぎないようにする
  - スタックのための領域は有限であり、そこの限界を超えるとプログラムは停止する (stack overflow)

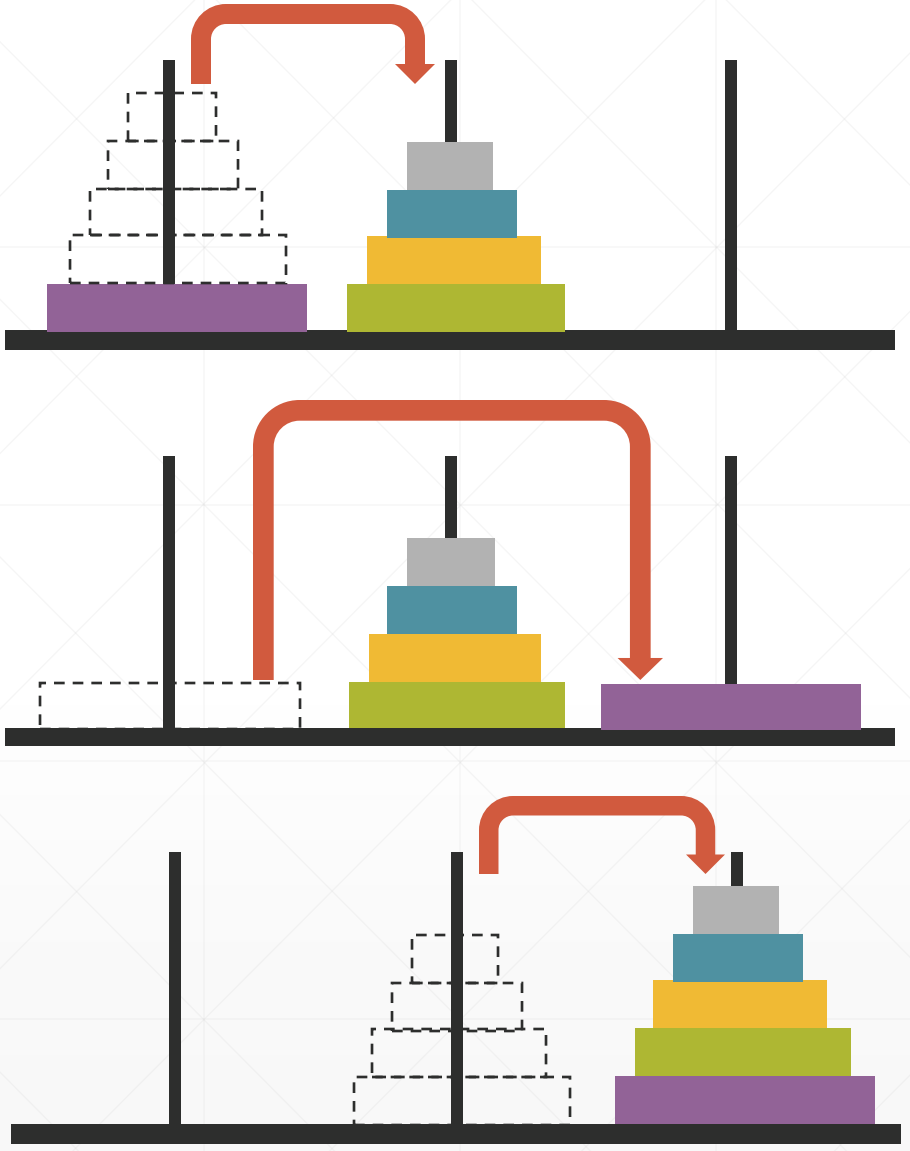
もう一つの有名な例題：ハノイの塔

- 円盤が重なって出来ている塔の位置を移動させるゲーム
  - 円盤を置ける箇所は三箇所（円盤を棒に挿す）
  - 一度に動かせる円盤は一枚だけ
  - 円盤が重なる場合は小さい円盤の方が上でなくてはならない



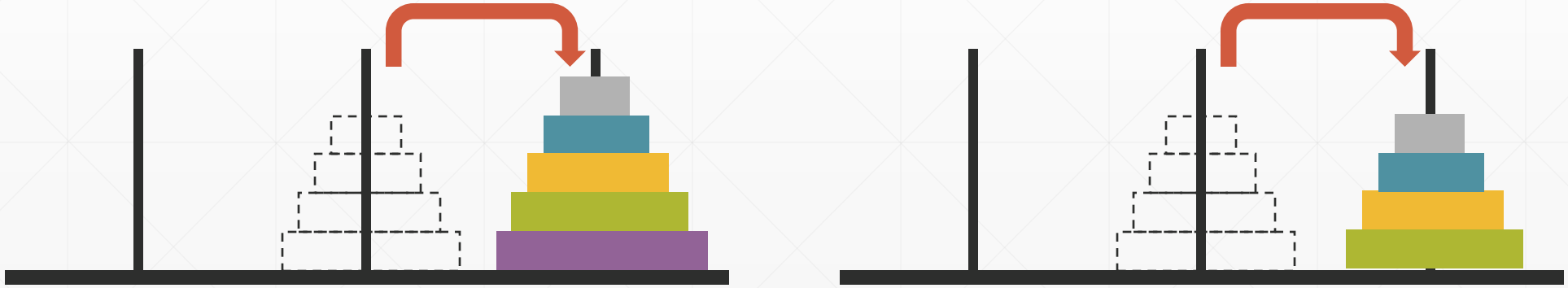
ハノイの塔の解法

- step 0: 高さが 1 の時は自明
- step 1: 最大の円盤の上に乗っている高さn-1の塔を**「何らかの方法」**で別の棒に動かす
- step 2: 最大の円盤をさらに別の棒に動かす
- step 3: 再び**「何らかの方法」**を使って、高さ n-1 の塔を最大の円盤の上に動かす



「何らかの方法」って？

- 実は、前ページの説明が「何らかの方法」そのもの
  - 高さ n-1 の塔を移動させる際は、それ以外の円盤の上には何でも乗せられる。つまり、それらは「地面」だと思って良い
  - 下図の右の場合も左の場合も取るべき手順は変わらない
  - 部分問題を利用した再帰的定義



ループを再帰で表現する

- for や while で書けるコードは、再帰でも書ける
  - 興味のある人は「チューリング完全」について調べよ
- 再帰ですっきり書けそうなら、まずは再帰で書くと良い（性能等の問題が発生したらループで書き直す）

```
char *s = "Hello world!";
char *p = s;
while (*p != '\0') {
  putchar(*(p++));
}
do {
  putchar(*(--p));
} while (p != s);
```

```
void print_both(char *s) {
  if (*s == '\0') { return; }
  putchar(*s);
  print_both(s + 1);
  putchar(*s);
}
...
  print_both("Hello world!");
```

相互再帰 (mutual recursion)

- 複数の関数が相互に参照し合う形で定義される関数
- 例： Hofstadter male-female sequences (in GEB book)

```
int M(int n) {
  return (n == 0) ? 0 : n - F(M(n - 1));
}

int F(int n) {
  return (n == 0) ? 1 : n - M(F(n - 1));
}
```

ほぼ線形に増加するが、不規則な動きが含まれる。
そして、n=150 あたりから計算量が膨大になる。

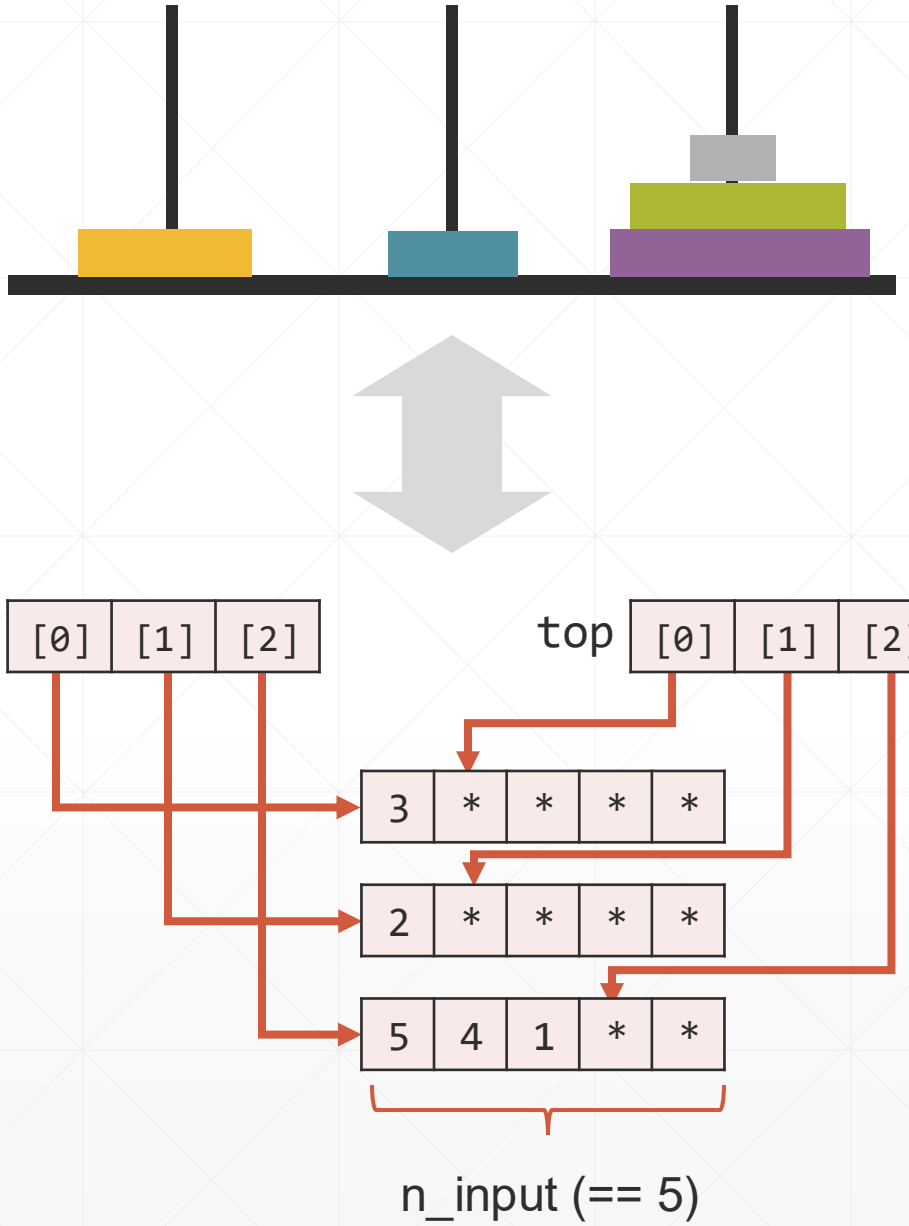
```
F: 1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6, 7, 8, 8, 9, 9, 10, 11, 11, 12, 13, ...
M: 0, 0, 1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 7, 8, 9, 9, 10, 11, 11, 12, 12, ...
```

今日の演習

- 以下の課題のいずれか一方を選択して行うこと
  - 課題A: hanoi.c の内容を読み、足りない部分を補ってプログラムを完成させよ。また、tower および top の内容に整合性があるかどうかを検査する関数 verify_towers を再帰で作成せよ
  - 課題B（難易度：高）: Hofstadter male-female sequences を、実行速度が早くなるように改良し、元のバージョンからどのように変化したかを考察せよ（nと実行時間の関係、nと関数呼出回数の関係、メモリ使用量etc.）

課題Aの補足(1)

- hanoi.c における塔の表現
  - 塔の情報は塔ごとに malloc で確保した unsigned の配列に格納
  - tower は各塔の底を指す
    - 途中で書き変わることは無い
  - top は各塔の最も上の円盤の次のマスを指す
    - 円盤が移動する度に書き換わる
    - top[x]の指す先、および、それより後ろのマスに入っている値には意味は無い



課題Aの補足(2)

- verify_towers が確認すべき性質：
  - 1からnまでのすべての円盤がいずれかの塔に一回だけ含まれている
  - 積み重なった円盤の大小関係が逆転していない
  - top の各ポインタが適切な箇所を指している
  - ヒント：最も大きい円盤はいずれかの塔の一番下にある
- verify_towers の返戻値と引数の例：
  - int verify_towers(unsigned i, unsigned *t0, unsigned *t1, unsigned *t2);
  - i 以下の円盤について、与えられた3つの部分塔 t0, t1, t2 が上記の性質を満たしている場合に 1 を返す。そうでない場合は 0 を返す
  - 引数は、本当は配列にしたいところだが、それだと記述がやや煩雑になるので今回はサボっても良い
  - ヒント：i > 0 の場合は、サイズ i の円盤についてチェックし、i より小さい円盤のチェックは再帰呼び出しで行えば良い。i == 0 の時は top が正しいかをチェックする。

完成版の hanoi.c の実行例 (n = 5)

- move 関数が呼ばれる度に、move の内容と、処理後の塔の状態を表示 (print_towers 関数)
- hanoi 関数を抜ける直前に verify_towers 関数を実行
- move の回数をカウントし、最後に総数を表示

```
### start
0|5]4]3]2]1]
1|
2|
### move 0 to 2
0|5]4]3]2]
1|
2|1]
### verify: OK
### move 0 to 1
0|5]4]3]
1|2]
2|1]
```

⋮

⋮

```
### move 1 to 2
0|1]
1|
2|5]4]3]2]
### move 0 to 2
0|
1|
2|5]4]3]2]1]
### verify: OK
### verify: OK
### verify: OK
### verify: OK
### verify: OK
### 31 steps to complete
```