再帰

計算機アルゴリズム特論：2017年度

只木進一

再帰
recursion

- 関数や手続きが、その関数・手続きそのもので記述される
- 参考：数学的帰納法
  - 自然数nに関する命題$S(n)$
  - $S(1)$は正しい（多くの場合自明）
  - $S(n)$を仮定して$S(n+1)$が成立を導出

- 関数$f(n)$の値は$f(n-1)$が分かると計算できる
- 注意：停止条件
- 利点：記述が簡潔になる

例：階乗：非再帰

$$n! = \prod_{k=1}^{n} k$$

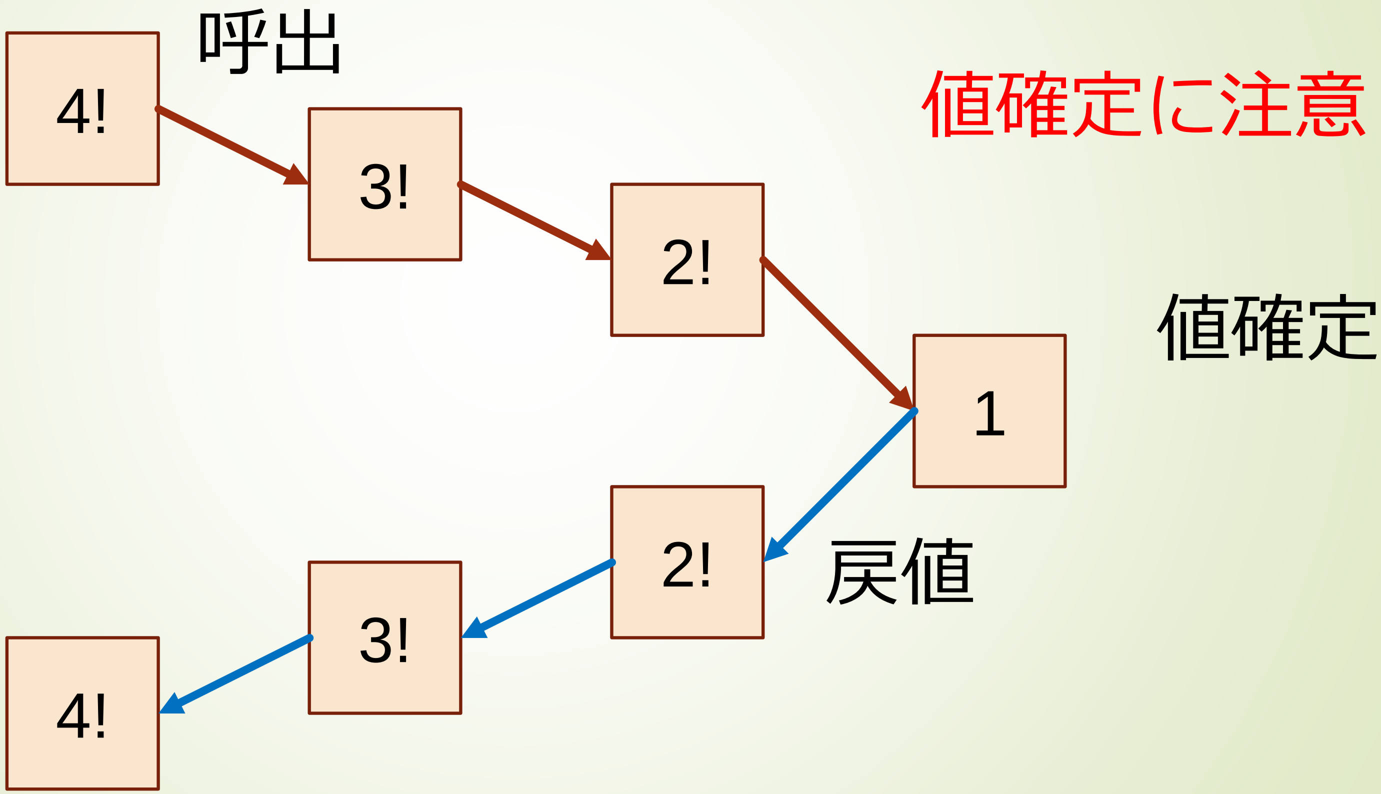
```
function factorial(n){
  int k=1;
  for(int i=1;i<=n;i++){
    k *= i;
  }
  return k;
}
```

例：階乗：再帰

$$n! = n \times (n-1)!$$

```
function factorial(n){
  if ( n==1) return 1;
  return n * factorial (n-1);
}
```

再帰の動作



値確定に注意

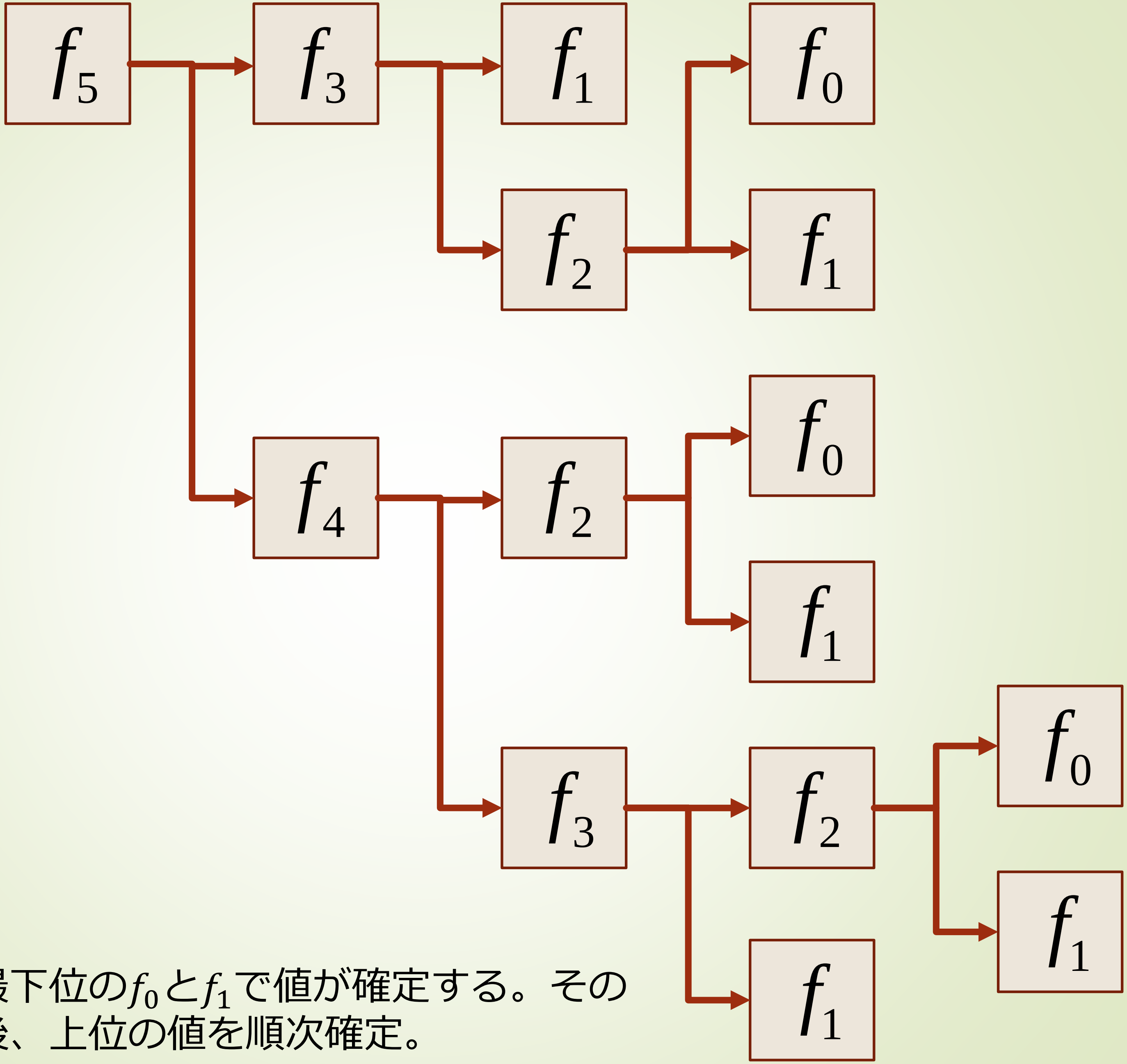
戻りの部分に留意

例：Fibonacci数

$$f_0 = 0$$

$$f_1 = 1$$

$$f_n = f_{n-2} + f_{n-1}$$



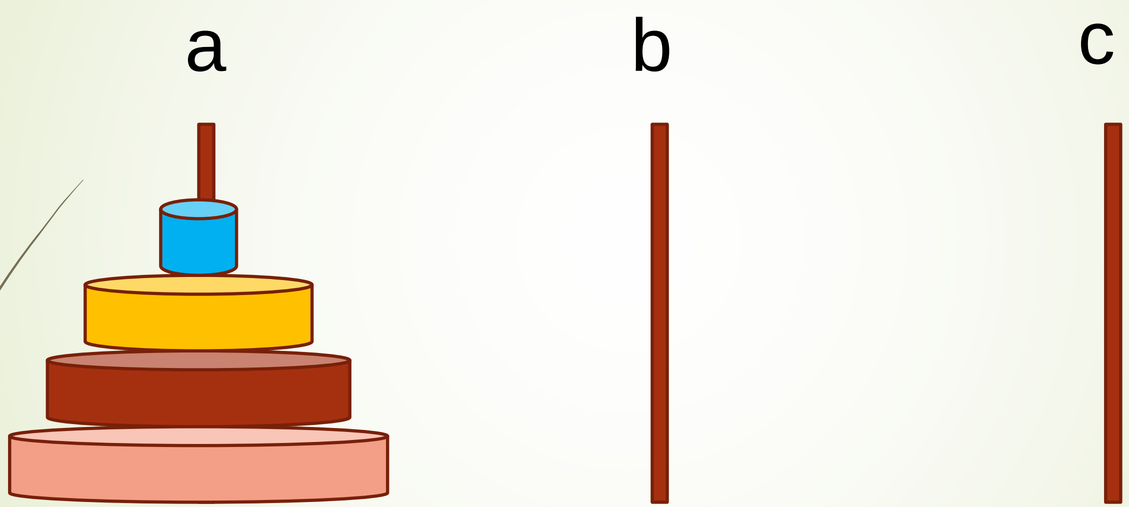
最下位のf_0とf_1で値が確定する。その後、上位の値を順次確定。

- 単純な再帰では、下位の二つの値が必要
  - f_nの計算には、f_{n-1}とf_{n-2}の二つが必要
  - 一つの再帰毎に、必要な計算が倍になってしまう
  - 同じ値を複数回計算
- しかし、下位の値さへ分かれば良いはず。
  - f_nの計算には、$f_m (m < n)$だけが必要。

再帰と非再帰

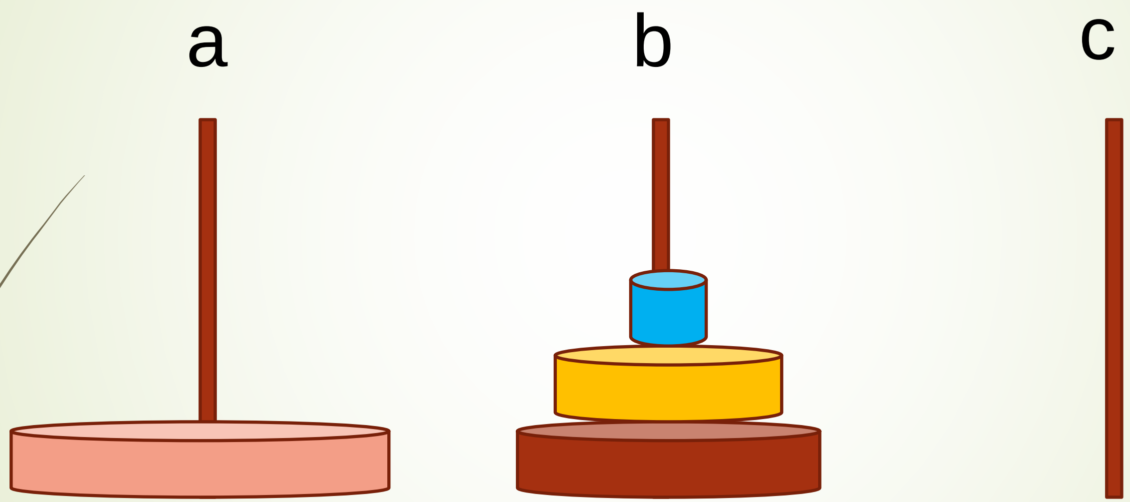
- 再帰アルゴリズム
  - 表記が簡便
  - 停止条件を忘れると暴走
  - デバッグ困難
- 実行する計算装置は非再帰的動き
  - かならず非再帰で記述できる
- 速度と見やすさのトレードオフ

例：ハノイの塔



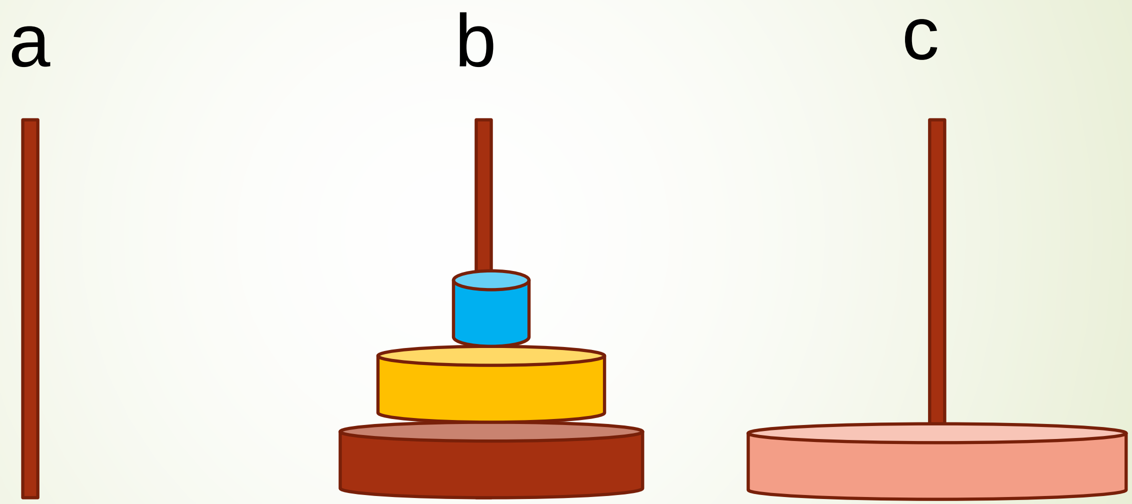
- 円盤をaからcへ移動させる
- 一度に一つの円盤を移動
- 小さい円盤が常に上になければならない

仮に3枚を動かすことが可能ならば



- 3枚をbへ
- 一番大きな円盤をcへ
  - 一枚動かすのは制約なし

仮に3枚を動かすことが可能ならば



- 残り3枚をcへ

ハノイの塔の再帰表現

- N枚の円盤をaからcへ移動させる
  - $N-1$枚の円盤をaからbへ移動させる
  - 最後の一枚をcへ移動させる
  - $N-1$枚の円盤をbからcへ移動させる
- 始点と終点は変数であることに注意

```
moveDisks(始点、終点、枚数) {
    if (枚数== 1) {
        始点から終点に1枚移動;
        return;
    }
    o = 空いている棒;
   moveDisks(始点, o, 枚数 - 1);
   moveDisks(始点,終点, 1);
   moveDisks(o, 終点, 枚数 - 1);
}
```

```
void moveDisks(int from, int to, int number) {
    if (number == 1) {
        moveSingleDisk(from, to);
        return;
    }
    int o = 3 - (from + to); //other pillars
    moveDisks(from, o, number - 1);
    moveDisks(from, to, 1);
    moveDisks(o, to, number - 1);
}
```

クラス構成

- Pillar : 一つの柱。push/pop操作。
  - Diskを置くスタック
  - pushの際に、Diskの順序を確認
  - moveSingleDisk() でのみ操作
- Disk : 円盤 : 大きさを定義
- Hanoi
  - 3本のPillar
  - 指定された円盤数を準備

計算量
円盤の移動回数

- n枚の円盤の移動回数$N(n)$

$$N(n) = 2N(n-1)+1$$

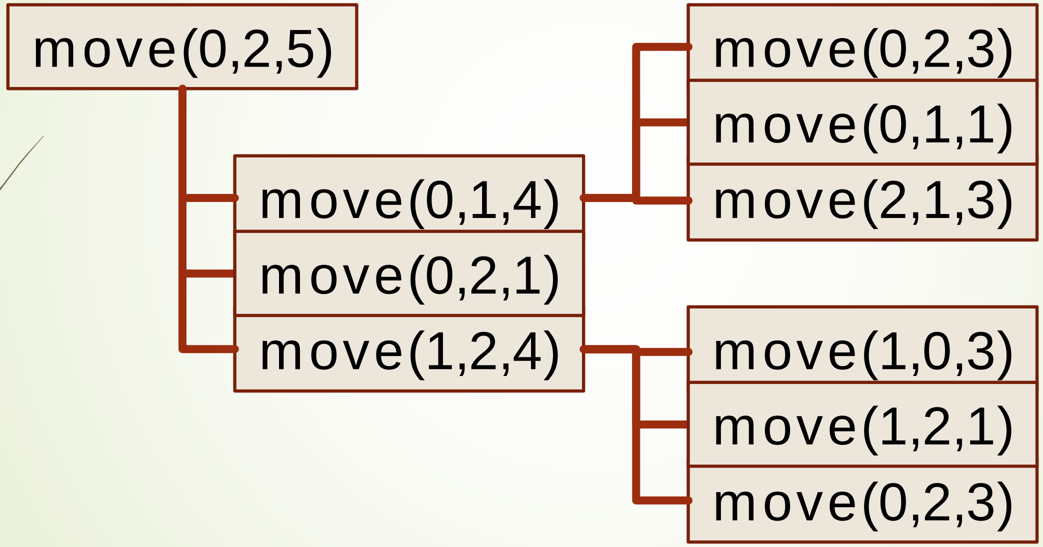
- 解

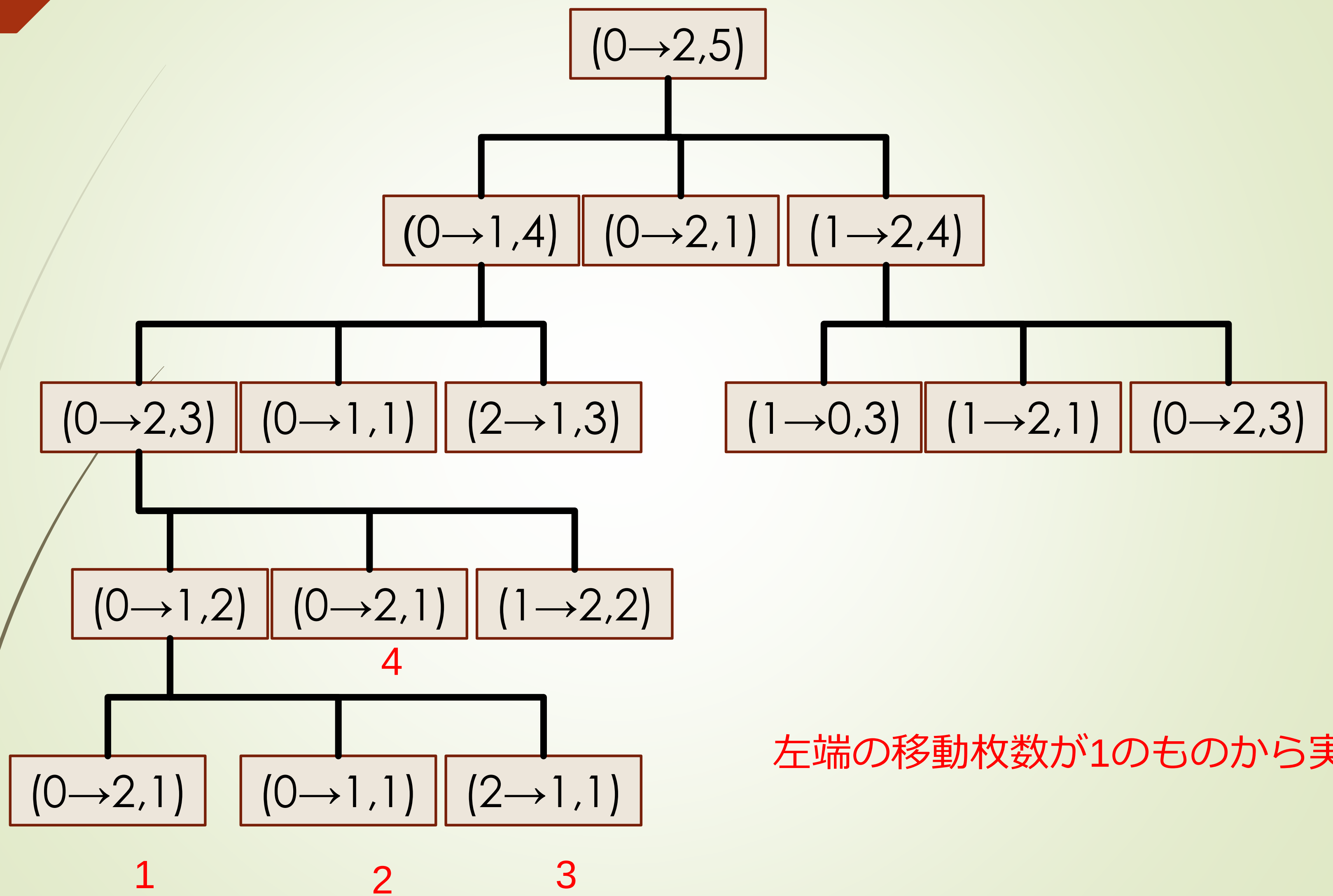
$$N(n) = 2^n - 1$$

数学的帰納法により証明せよ

- $O(2^n)$の計算時間を要する

計算量
なぜ、そんなに時間がかかる





左端の移動枚数が1のものから実行

再帰から非再帰へ

- moveDisks(from, to, num)の場合
  - moveDisks(from, o, num-1)の処理が終わるまで、その後の処理を待たせる
- 待たせる処理をスタックに積み、非再帰化可能
  - 処理タスクのクラスを作成
  - 処理タスク（始点、終点、枚数）

スタックを使った非再帰化

- 処理タスク$(f \rightarrow t, m)$をスタックに

$$[(0\rightarrow 2,4)] \Rightarrow [(0\rightarrow 1,3),(0\rightarrow 2,1),(1\rightarrow 2,3)]$$
$$\Rightarrow [(0\rightarrow 2,2),(0\rightarrow 1,1),(2\rightarrow 1,2),(0\rightarrow 2,1),(1\rightarrow 2,3)]$$
$$\Rightarrow [\boxed{(0\rightarrow 1,1)},(0\rightarrow 2,1),(1\rightarrow 2,1),(0\rightarrow 1,1),(2\rightarrow 1,2),(0\rightarrow 2,1),(1\rightarrow 2,3)]$$
$$\Rightarrow [\boxed{(0\rightarrow 2,1)},(1\rightarrow 2,1),(0\rightarrow 1,1),(2\rightarrow 1,2),(0\rightarrow 2,1),(1\rightarrow 2,3)]$$
$$\Rightarrow [\boxed{(1\rightarrow 2,1)},(0\rightarrow 1,1),(2\rightarrow 1,2),(0\rightarrow 2,1),(1\rightarrow 2,3)]$$
$$\Rightarrow [\boxed{(0\rightarrow 1,1)},(2\rightarrow 1,2),(0\rightarrow 2,1),(1\rightarrow 2,3)]$$
$$\Rightarrow [(2\rightarrow 1,2),(0\rightarrow 2,1),(1\rightarrow 2,3)]$$

枚数1のものを実行してタスクを削除
2枚以上のタスクは、サブタスクをスタックへ

```
void start() {
    Stack<Task> tasks = new Stack<>();
    tasks.push(new Task(0, 2, n));
    while (!tasks.empty()) {
        Task task = tasks.pop();
        int from = task.from;
        int to = task.to;
        int number = task.number;
        if (number == 1) {
            moveSingleDisk(from, to);
        } else {
            int o = 3 - (from + to); //other pillar
            tasks.push(new Task(o, to, number - 1));
            tasks.push(new Task(from,to,1));
            tasks.push(new Task(from, o, number - 1));
        }
    }
}
```