アルゴリズム講座

第3回 〜深さ優先探索編〜

LeadHACK
@utsubo_21 高澤 栄一

Lesson 3

今回の内容

- 下記のアルゴリズム・データ構造を10回程度に分けて１つずつ解説

Lesson1
- 全探索

Lesson2
- 二分探索

Lesson3
- 深さ優先探索（+メモ化再帰）
- 幅優先探索
- 動的計画法
- 累積和
- グラフ・木
- ダイクストラ法
- ワーシャルフロイド法
- クラスカル法
- Union-Find

参考：レッドコーダーが教える、競プロ・AtCoder上達のガイドライン【中級編：目指せ水色コーダー！】
https://qiita.com/e869120/items/eb50fdaece12be418faa

深さ優先探索
＆
再帰を使いこなす

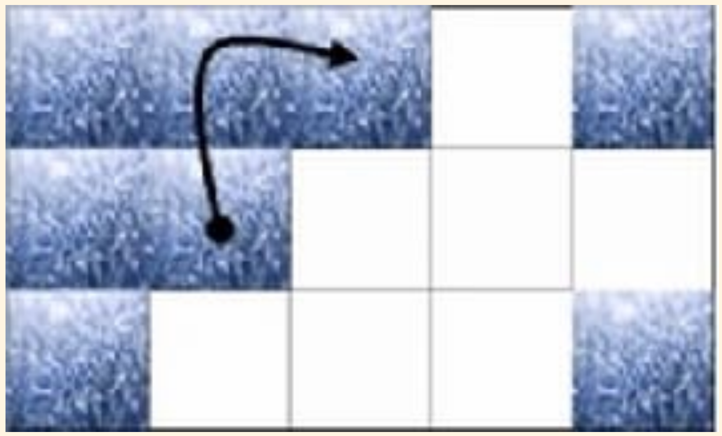
シンプルな深さ優先探索

早速ですが問題です

問題

- 概要
  - n×mのマスを表す2次元配列がある
    - 移動可能なマス: 1
    - 移動不可能なマス: 0
  - 任意のマスから移動を始める
  - 一度移動したマスにもう一度移動できない
  - 最大何マス移動できますか？

m



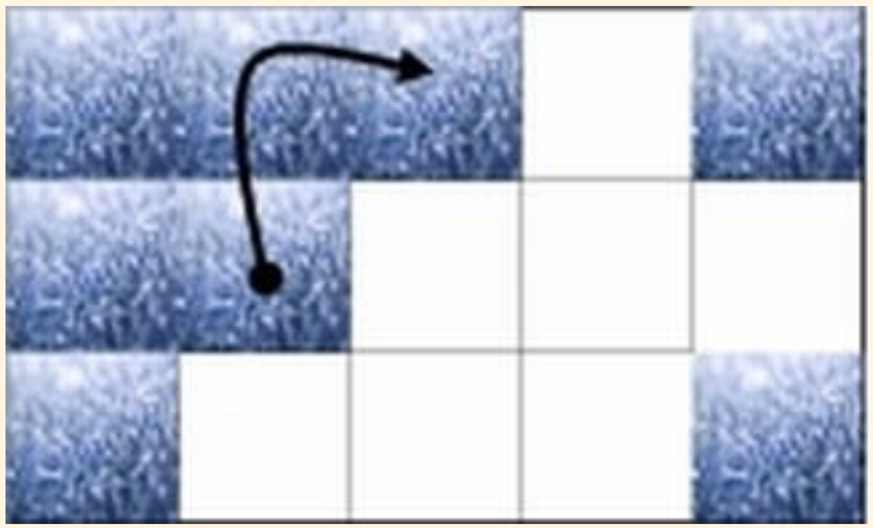
n

11101
11000
10001

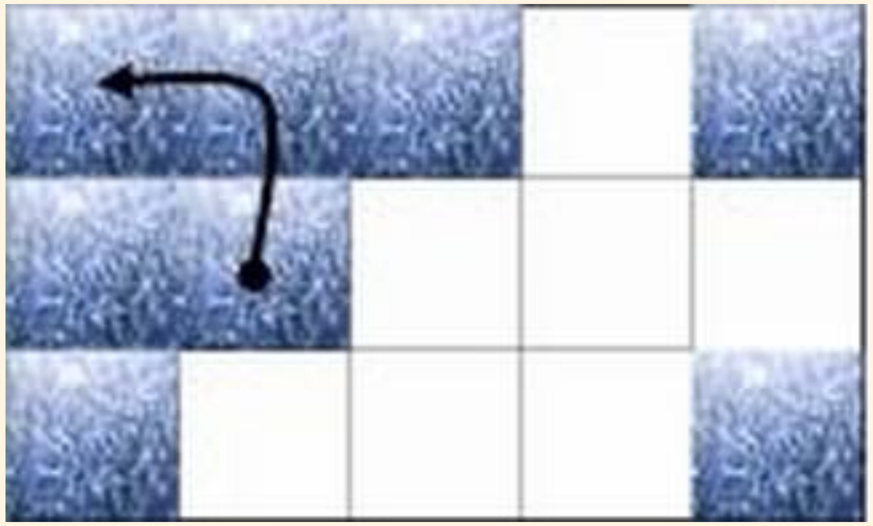
- 制約
  - 移動方法は20万通りを超えない
  - m, n ≦ 90

例

移動数：1マス



移動数：3マス



移動数：3マス

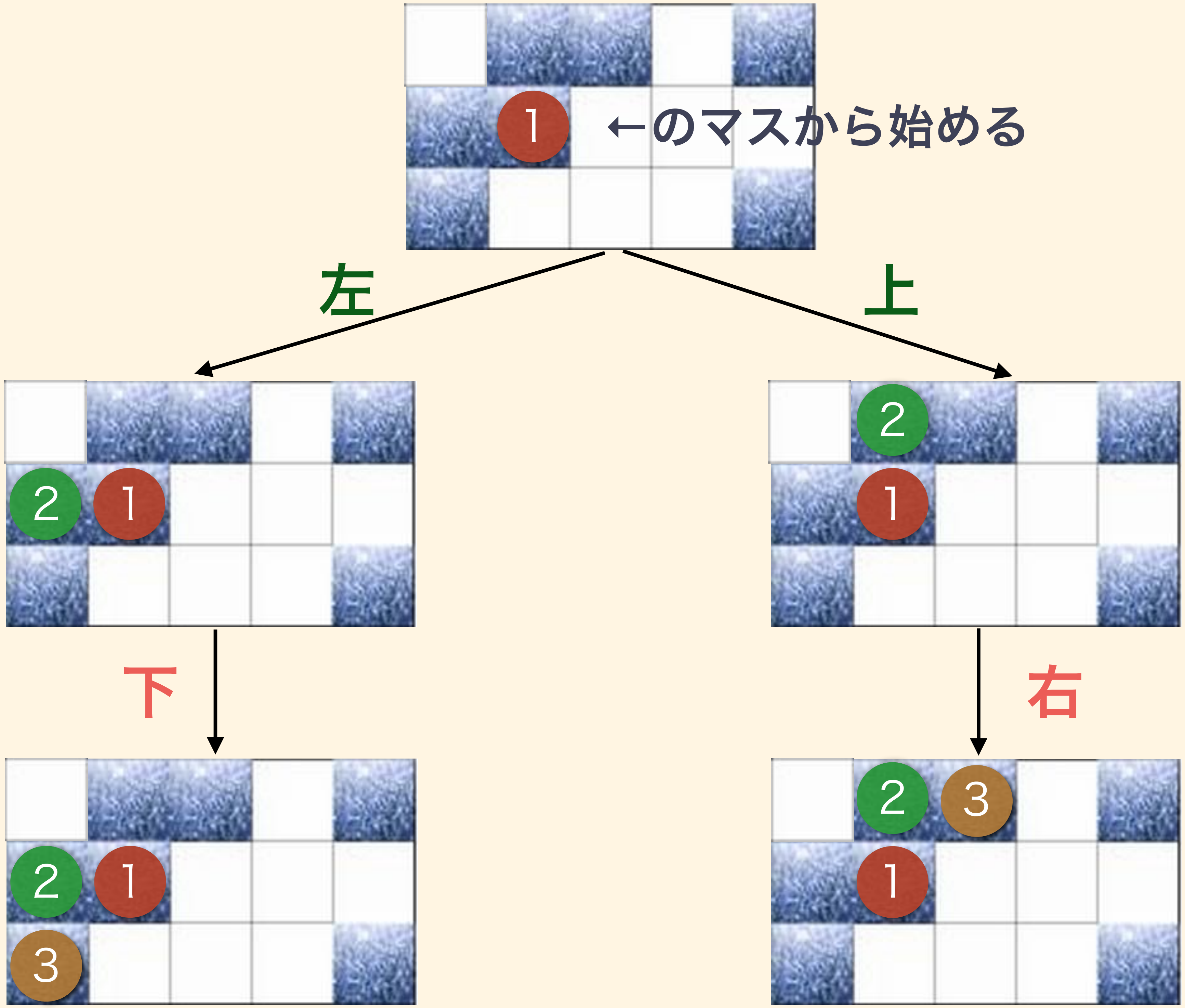
移動数：5マス

どのように解くか？

- 移動方法は20万通り以下であることが保証
- 移動方法を全列挙して,
  最大移動マス数を調べれば良い
- 一旦移動を始めるマスを固定して考える

どう全列挙するか？

* 1スライドに収まらないので
グリッドの左上1マス消しました



➠ forやwhileで実装するのは少し厳しそうな雰囲気

実装例

```
grid[N][M] := 移動可能マスが1, それ以外は0
visited[N][M] := 0初期化
maxCnts = 0
bool ok(x, y) {
  return grid[y][x] && !visited[y][x] // 移動可能, 未訪問
}

void dfs(x, y, step) {
  maxCnts = max(maxCnts, step) // 最大値更新
  visited[y][x] = true         // 訪問済みにする
  if (ok(x-1,  y)) dfs(x-1,   y, step + 1) // 左へ
  if (ok(x,  y+1)) dfs(  x, y+1, step + 1) // 下へ
  if (ok(x+1,  y)) dfs(x+1,   y, step + 1) // 右へ
  if (ok(x,  y-1)) dfs(  x, y-1, step + 1) // 上へ
  visited[y][x] = false        // 1歩戻るので訪問済みを解除
}
```

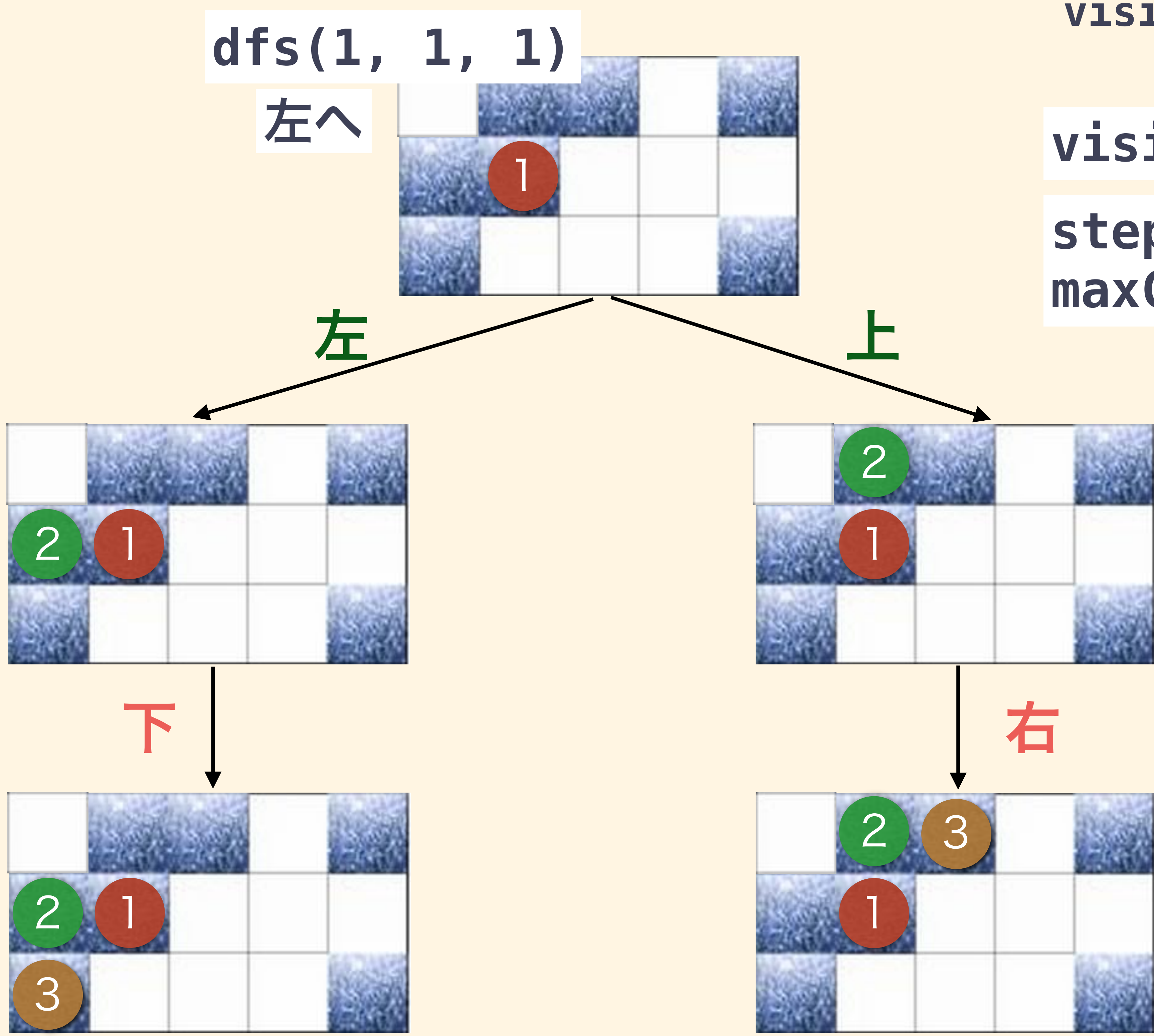
実際に流れを追って処理を確認しよう

流れを追う

```
maxCnts  =  1
            00000
visited  =  01000
            00000
```

visited更新

step=1
maxCntsも更新

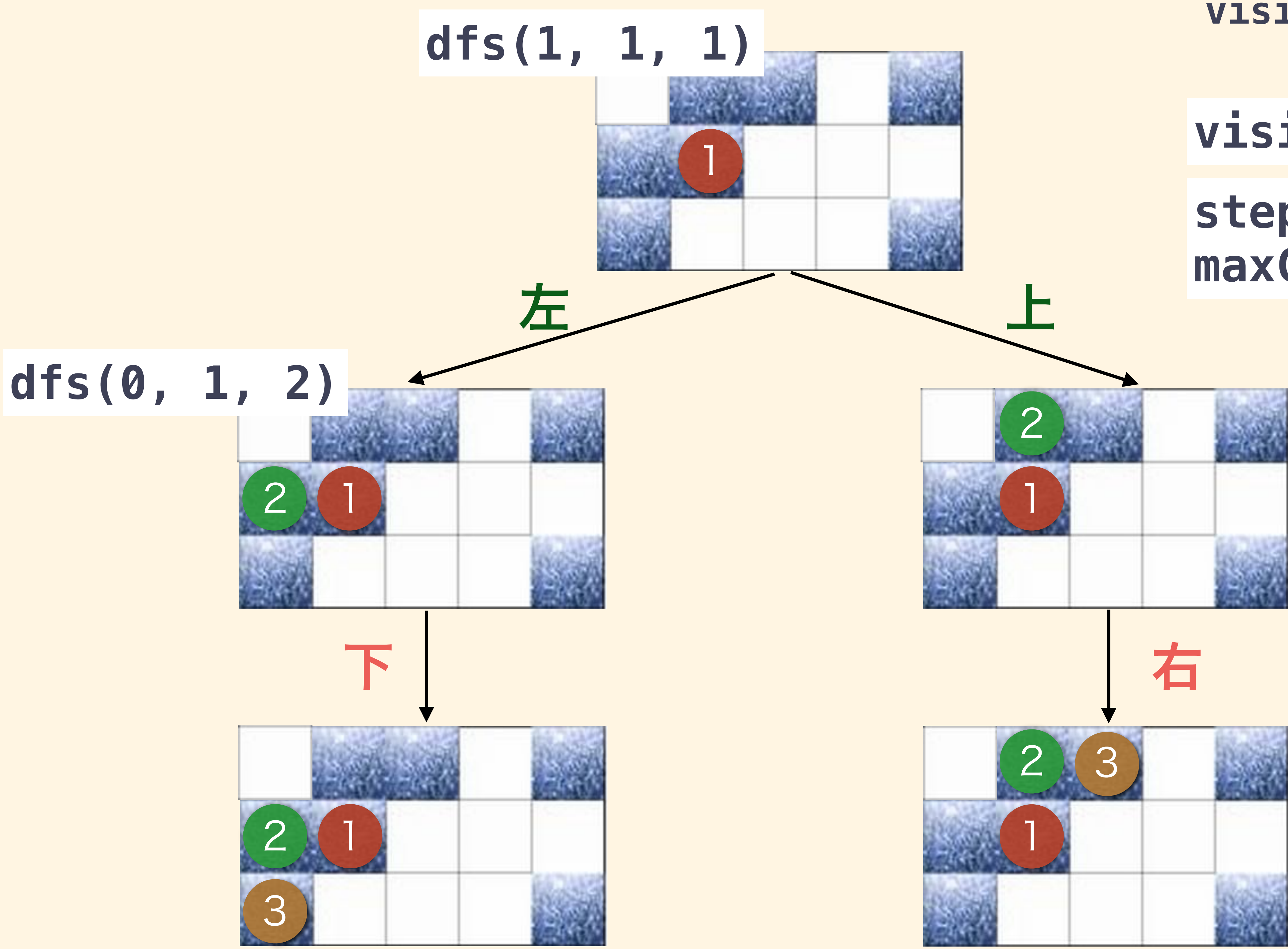


流れを追う

```
maxCnts  =  2
            00000
visited  =  11000
            00000
```

visited更新

step=2
maxCntsも更新

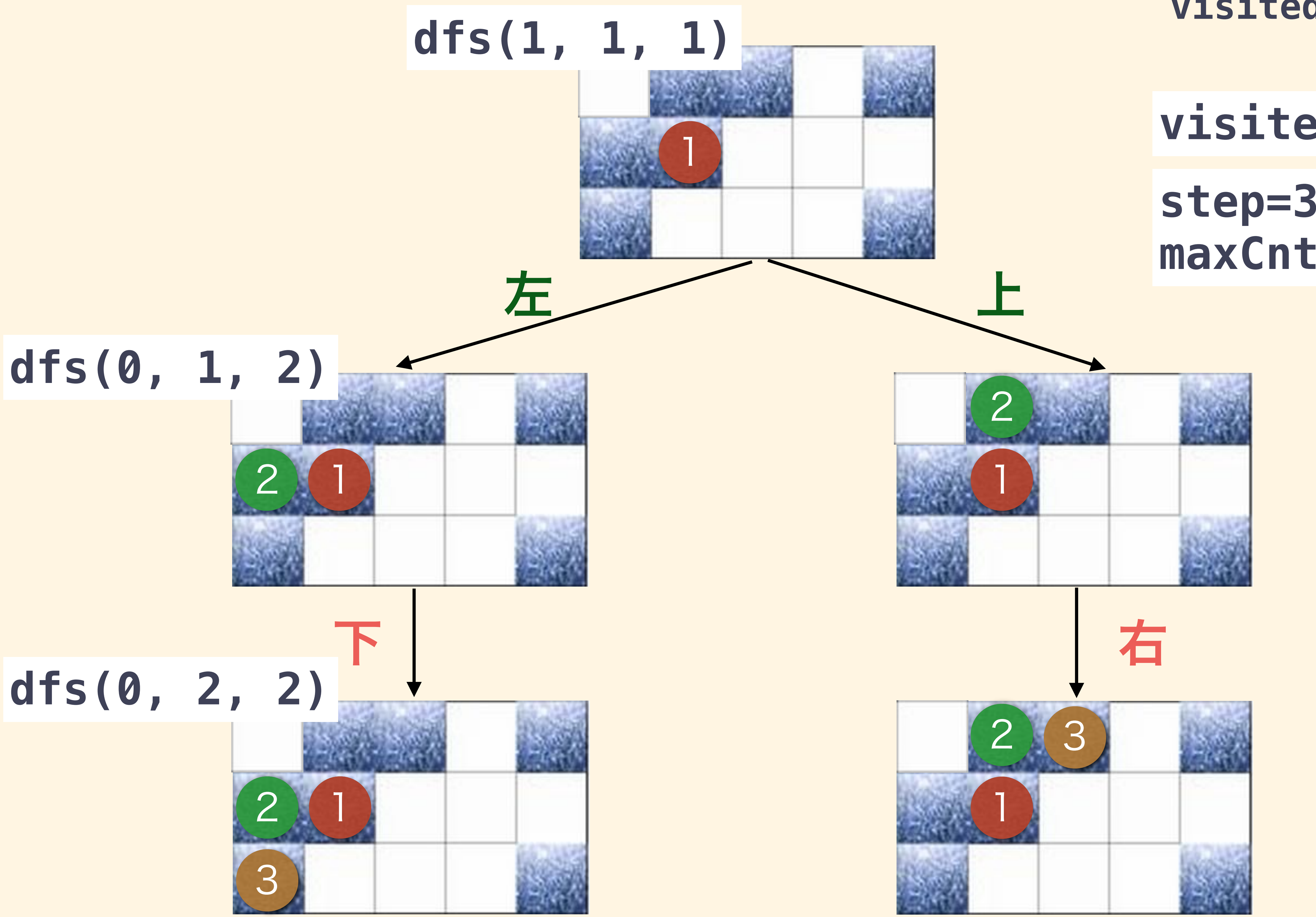


流れを追う

```
maxCnts  =  3
            00000
visited  =  11000
            10000
```

visited更新

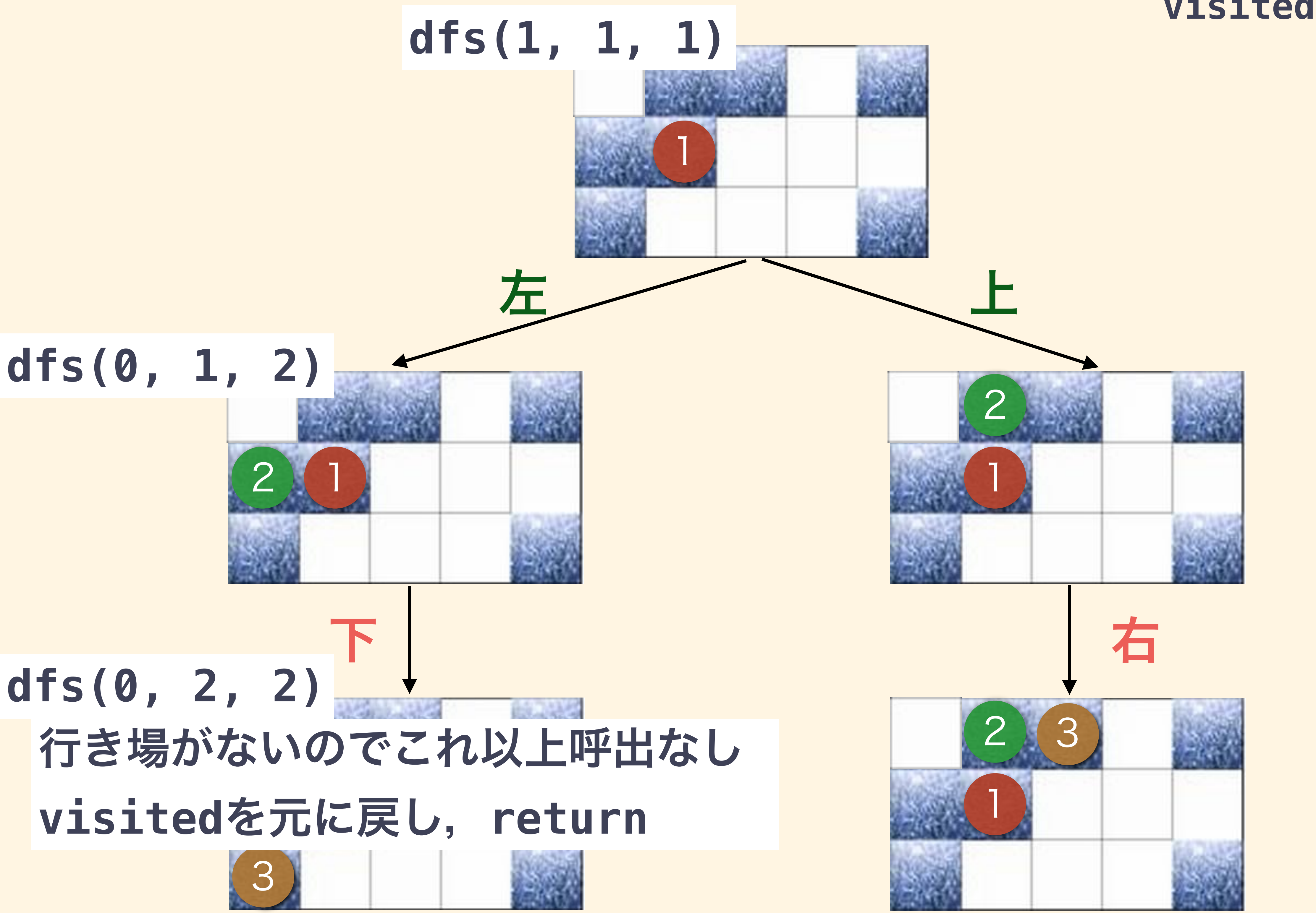
step=3
maxCntsも更新



流れを追う

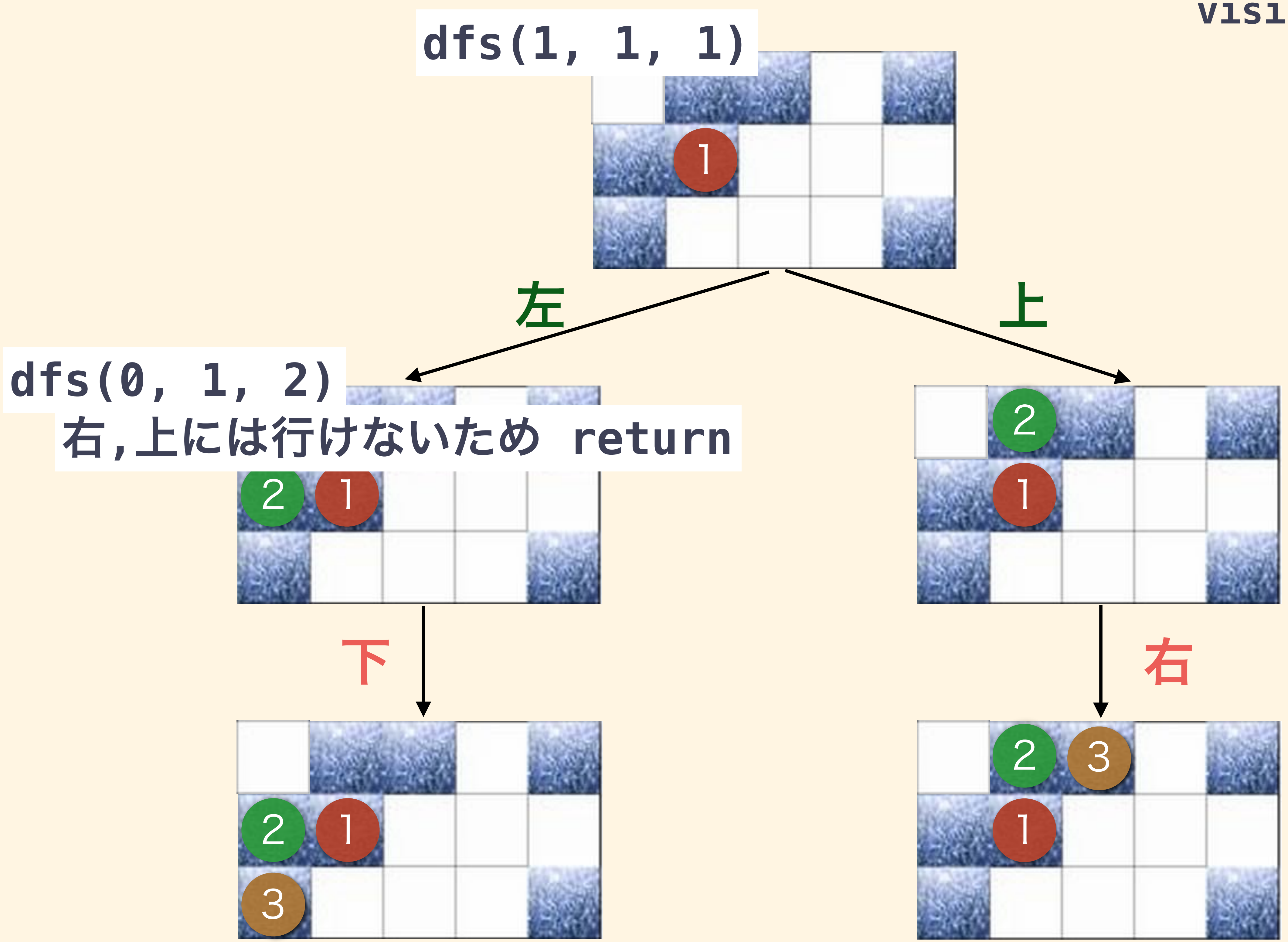
maxCnts = 3

visited =
00000
11000
00000



流れを追う

```
maxCnts  =  3

            00000
visited  =  01000
            00000
```

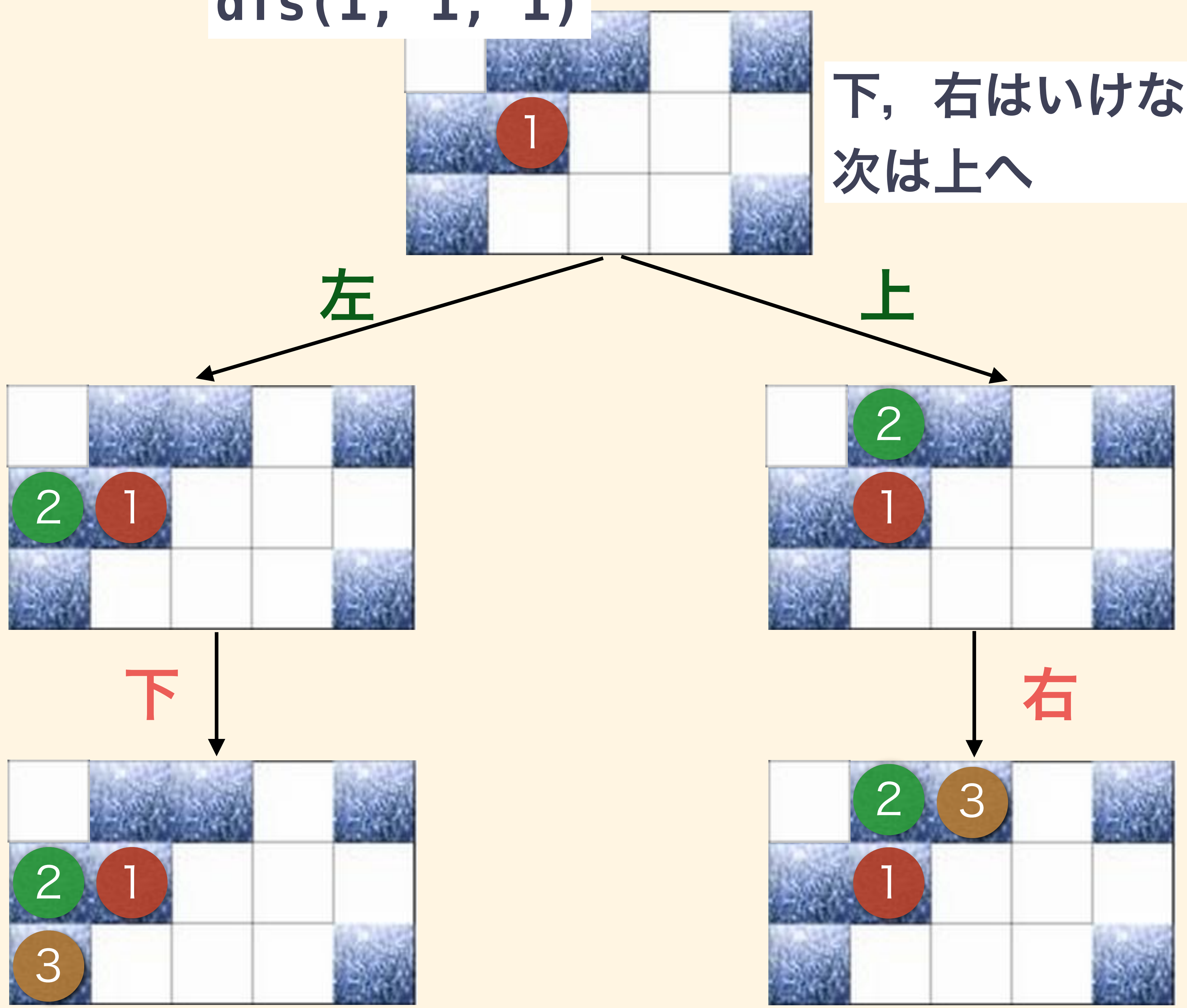


流れを追う

```
maxCnts  =  3

            00000
visited  =  01000
            00000
```

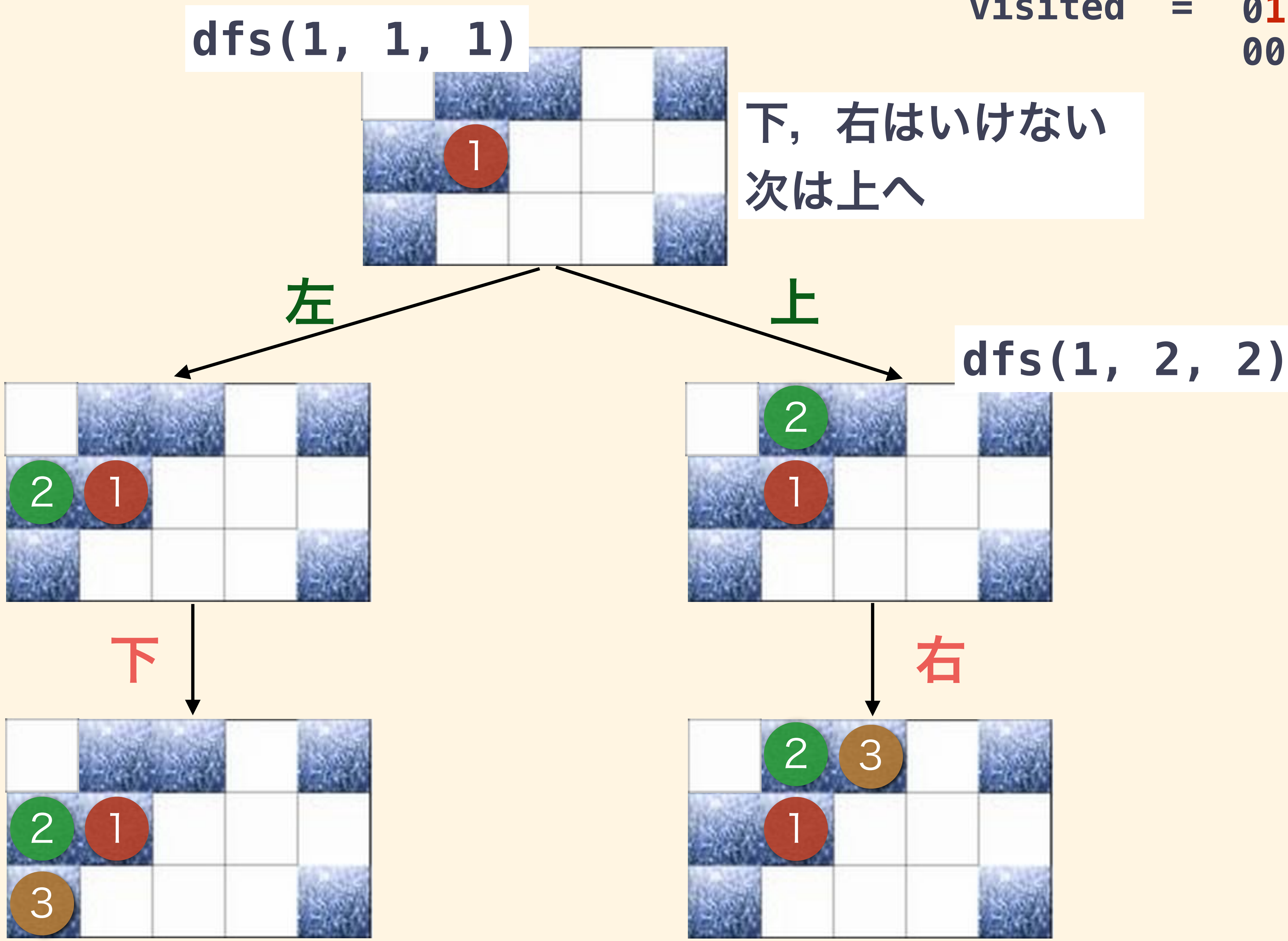
`dfs(1, 1, 1)`

下，右はいけない
次は上へ



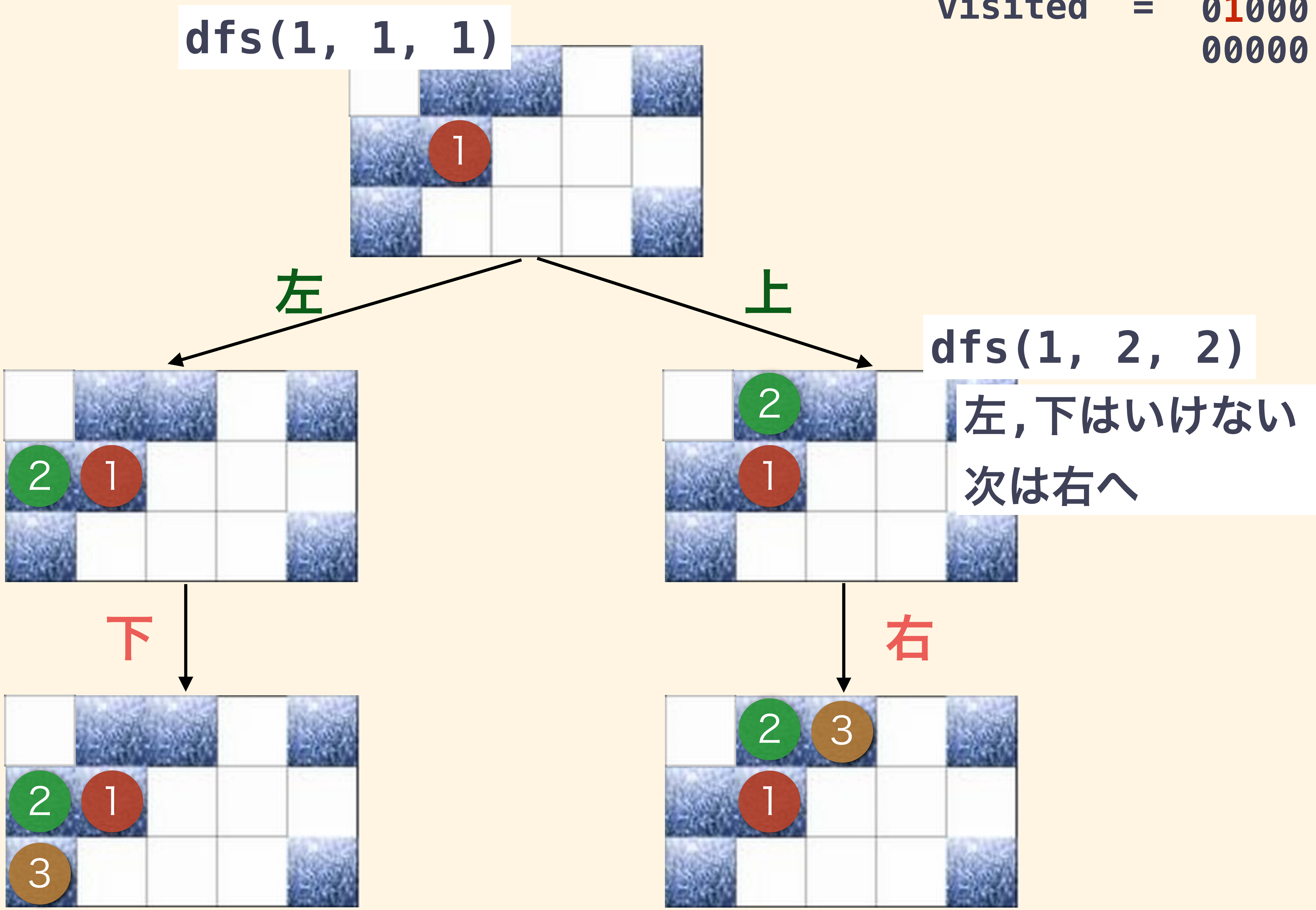
流れを追う

```
maxCnts  =  3

            01000
visited  =  01000
            00000
```



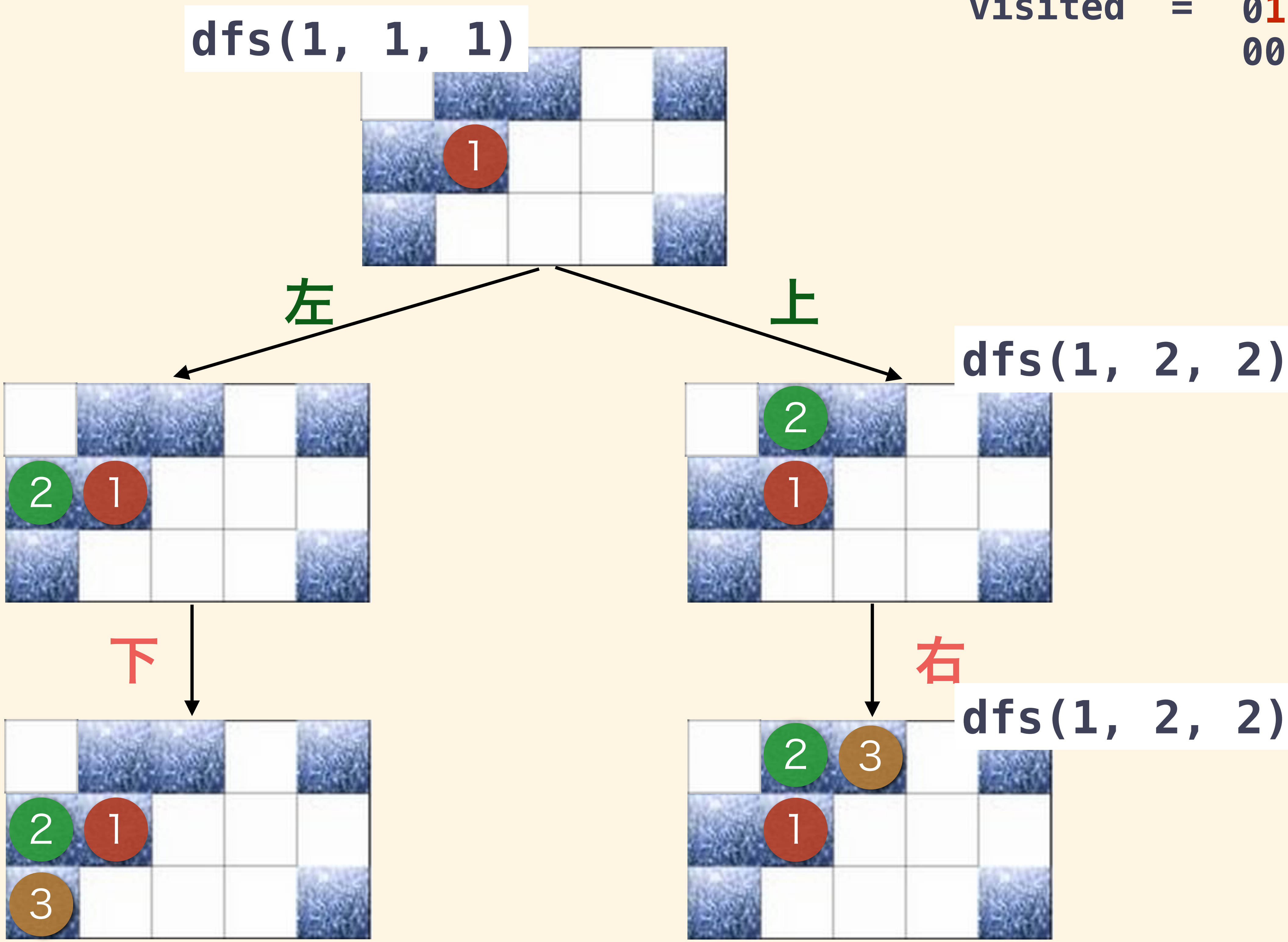
流れを追う

```
maxCnts  =  3

            01000
visited  =  01000
            00000
```



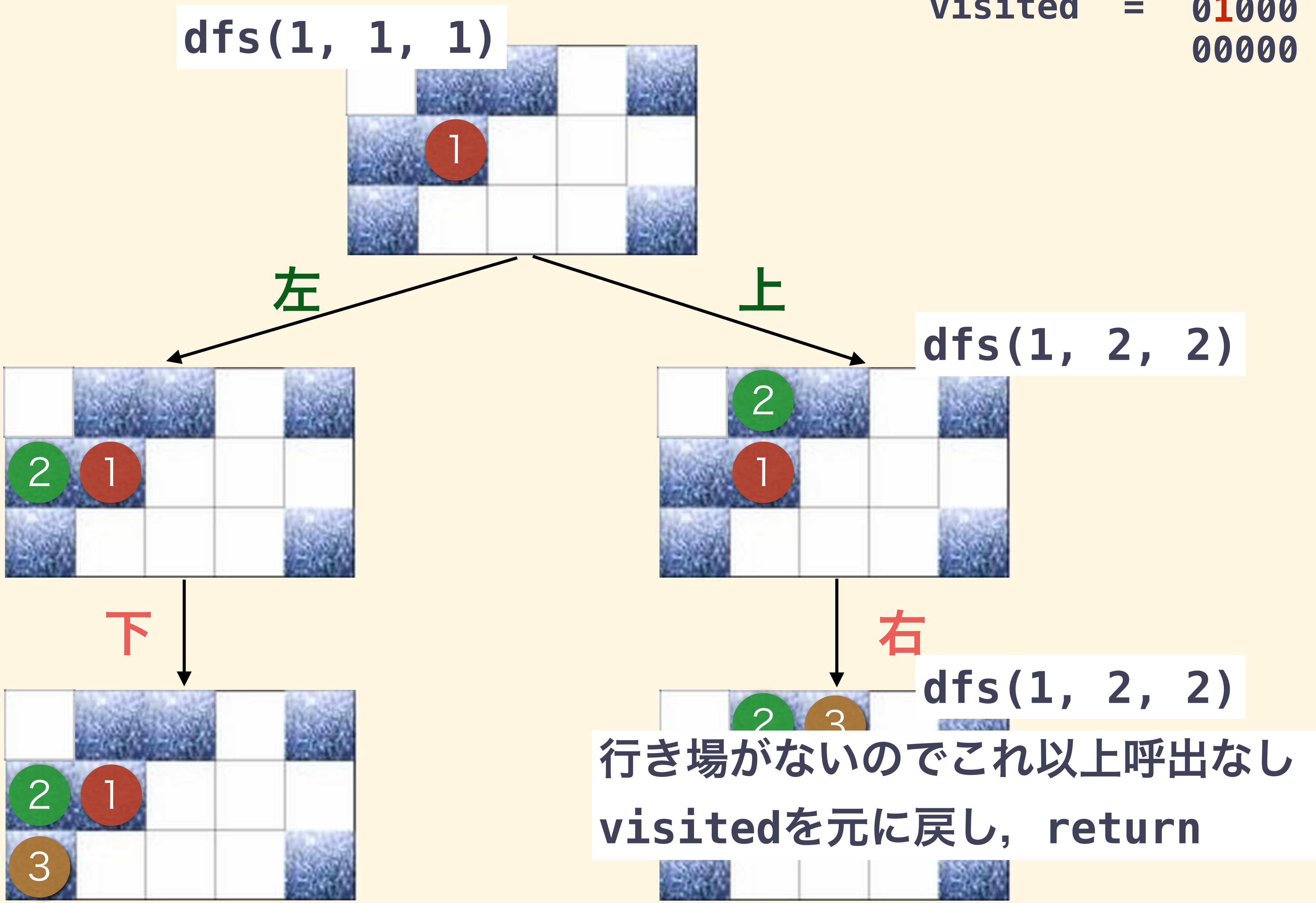
流れを追う

```
maxCnts  =  3

            01100
visited  =  01000
            00000
```



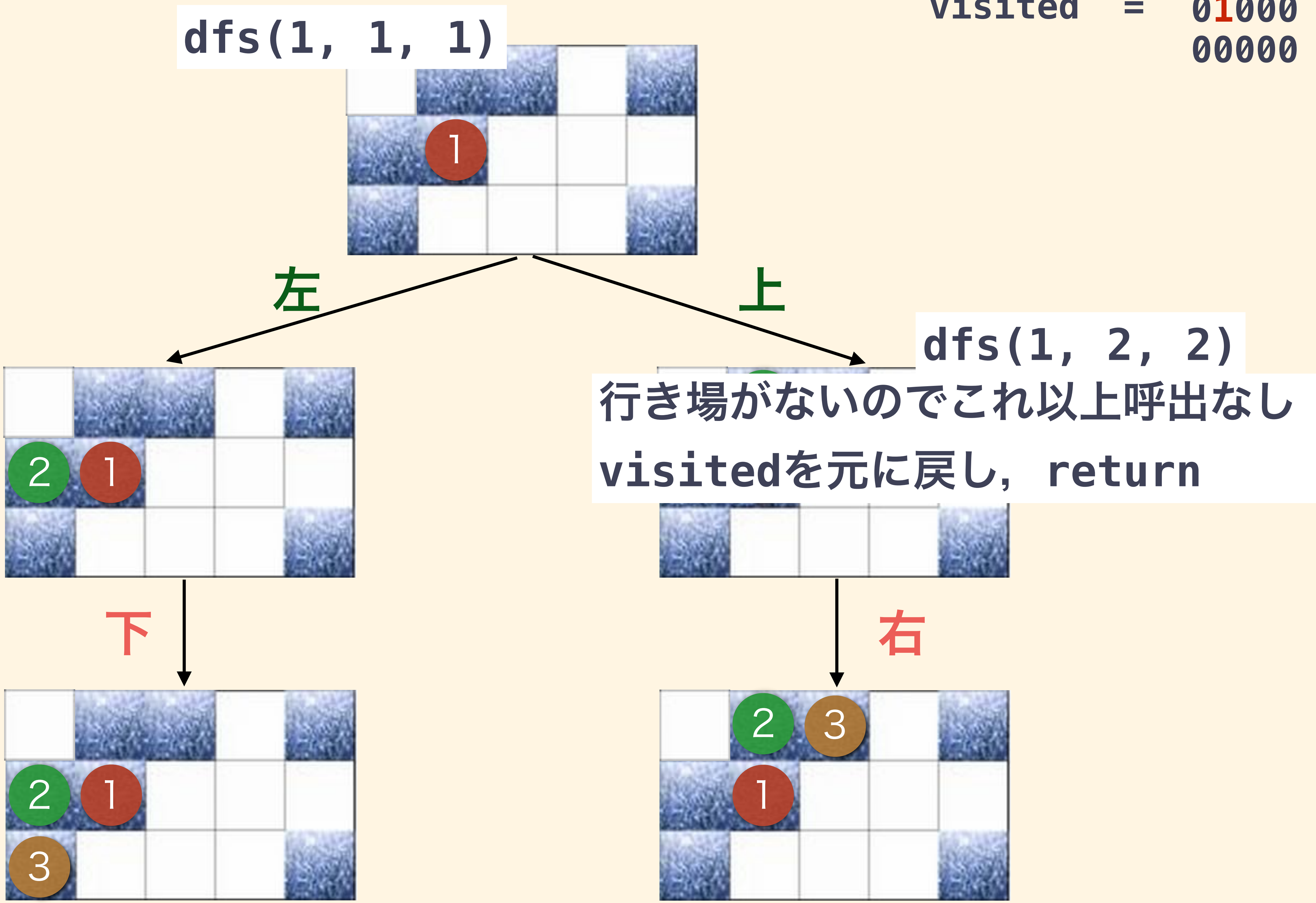
流れを追う

```
maxCnts  =  3

            01000
visited  =  01000
            00000
```



流れを追う

```
maxCnts  =  3

            00000
visited  =  01000
            00000
```

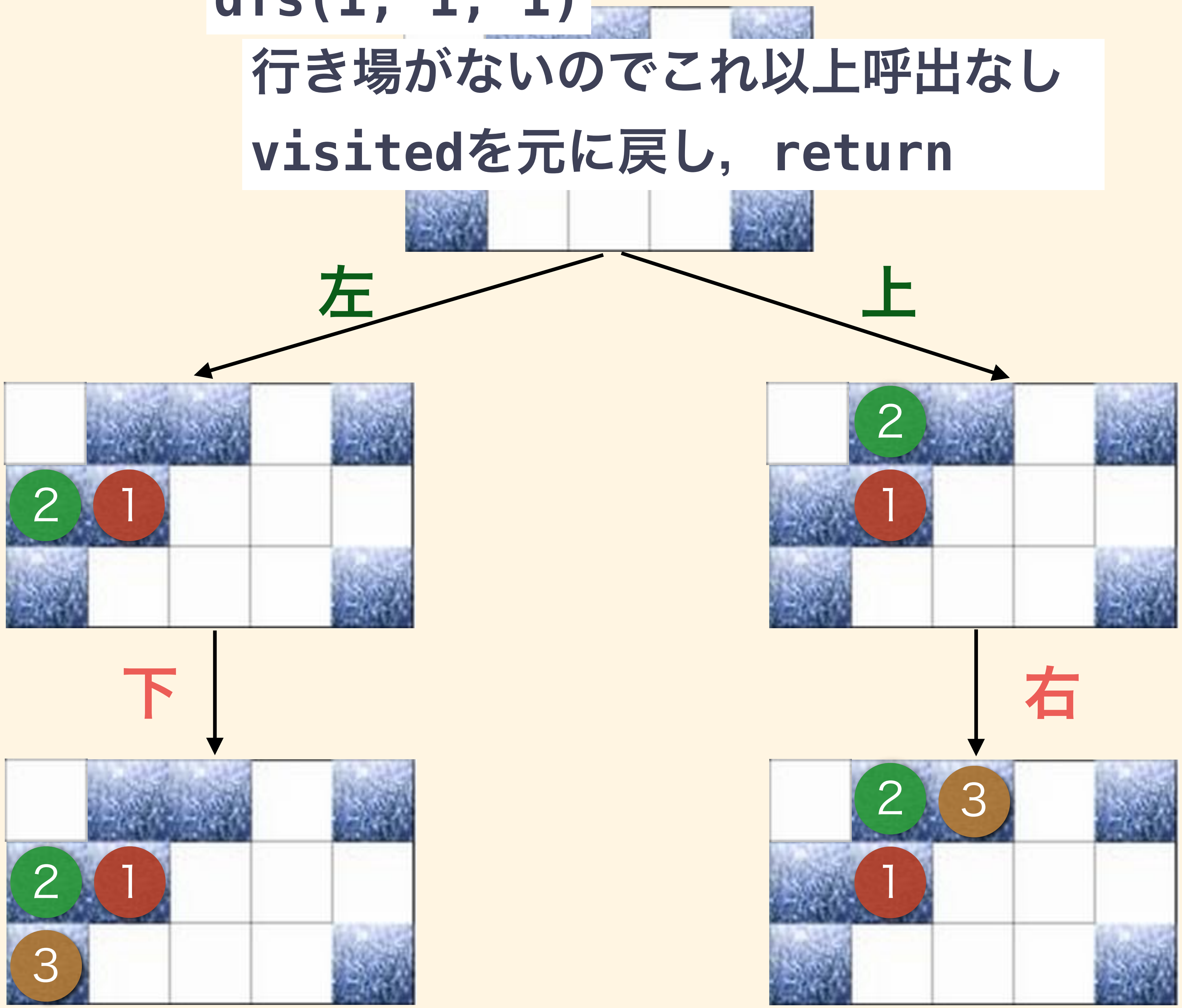


流れを追う

```
maxCnts  =  3
            00000
visited  =  00000
            00000
```

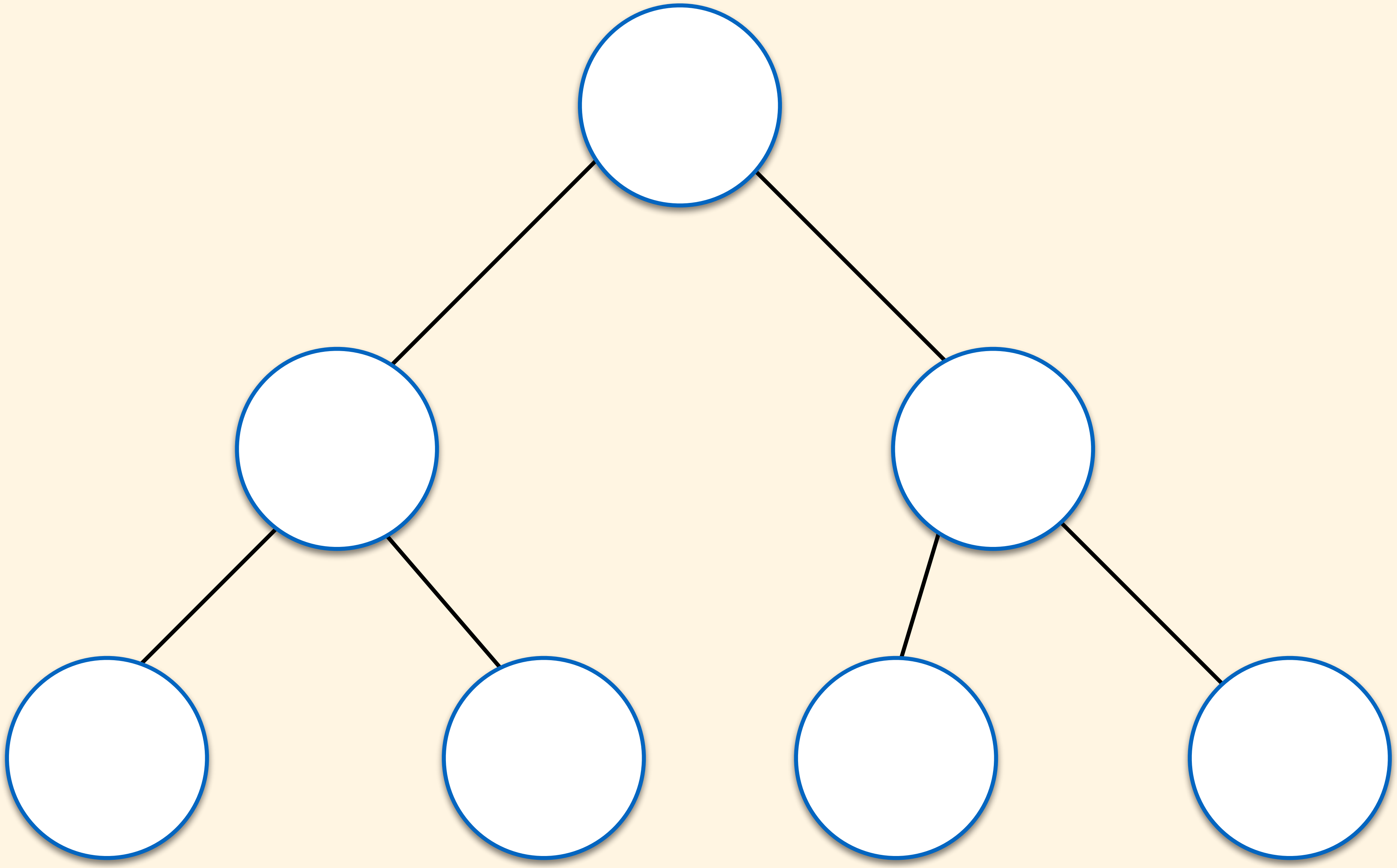
dfs(1, 1, 1)

行き場がないのでこれ以上呼出なし
visitedを元に戻し，return



探索完了後，maxCntsに答えが入っている

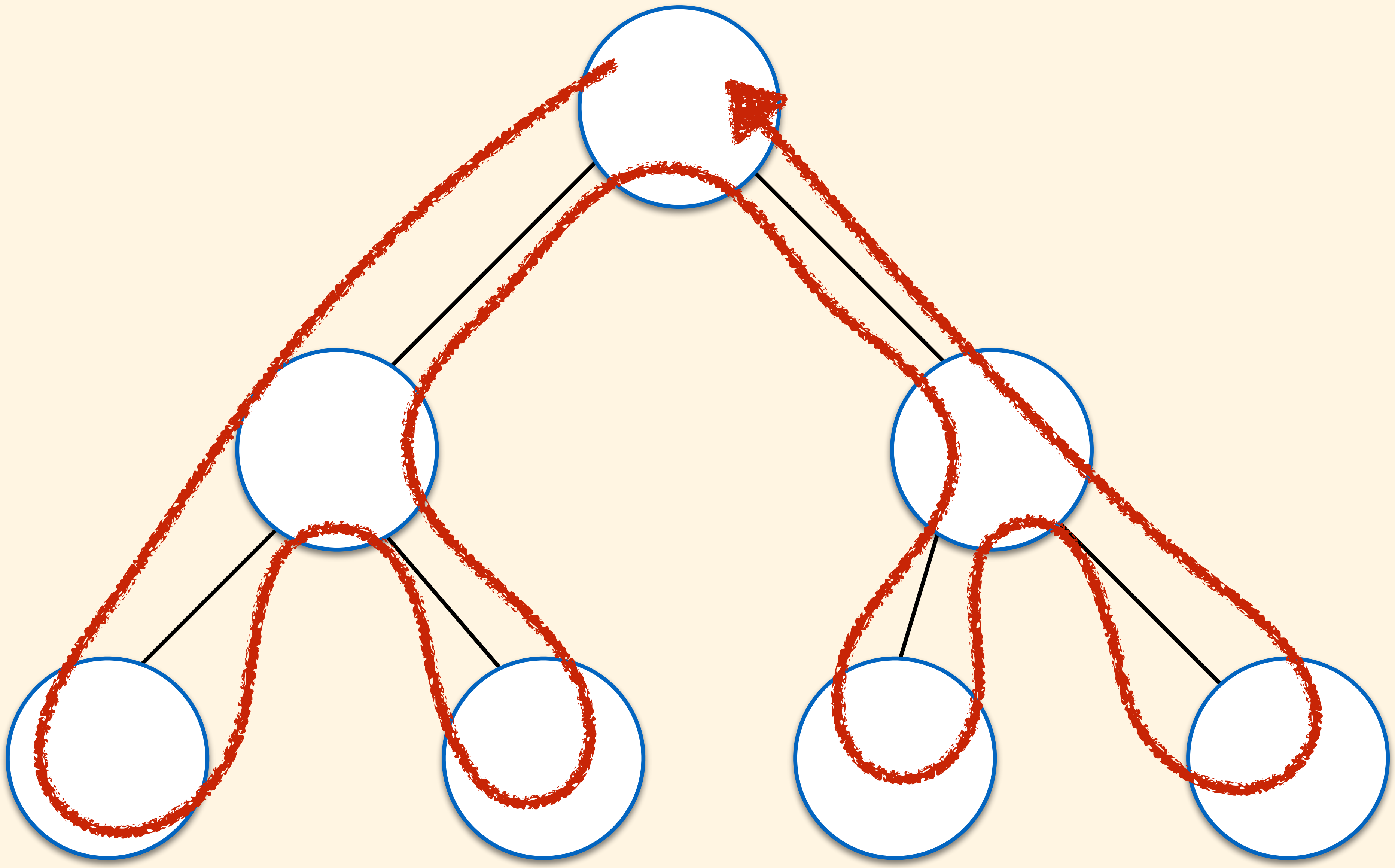
深さ優先探索 (バックトラック法)



- こういう構造を全部舐めたい時に

各丸は状態を表す (先の例でいうどのように移動してどこにいるか)

深さ優先探索 (バックトラック法)



- 行けるところまで進み，ダメなら戻る探索方法を「深さ優先探索 (バックトラック法)」と呼ぶ

DFS実装テンプレート

```
void dfs(state) {
  if (終了条件) {
    答えを更新してreturn
  }
  visited[state] = true   // 訪問済みにする
  dfs(nextstate)          // 次の状態に遷移(複数あれば複数個)
  visited[state] = false  // 1歩戻るので訪問済みを解除
}
```

私が意識しているポイントは

- 終了条件
- 今の状態から次に遷移する状態

ちょっぴり応用

```
void dfs(x, y, step) {
  if (step == 5) {
    visitedが1に立っている座標が今までの経路(step=5なので長さ5の経路)
    return
  }
  visited[state] = true   // 訪問済みにする
  // 上下左右にdfs呼び出し
  visited[state] = false  // 1歩戻るので訪問済みを解除
}
```

先の例を少し応用すると,

長さkの経路の列挙とかも出来る

メモ化再帰の前に
再帰処理入門

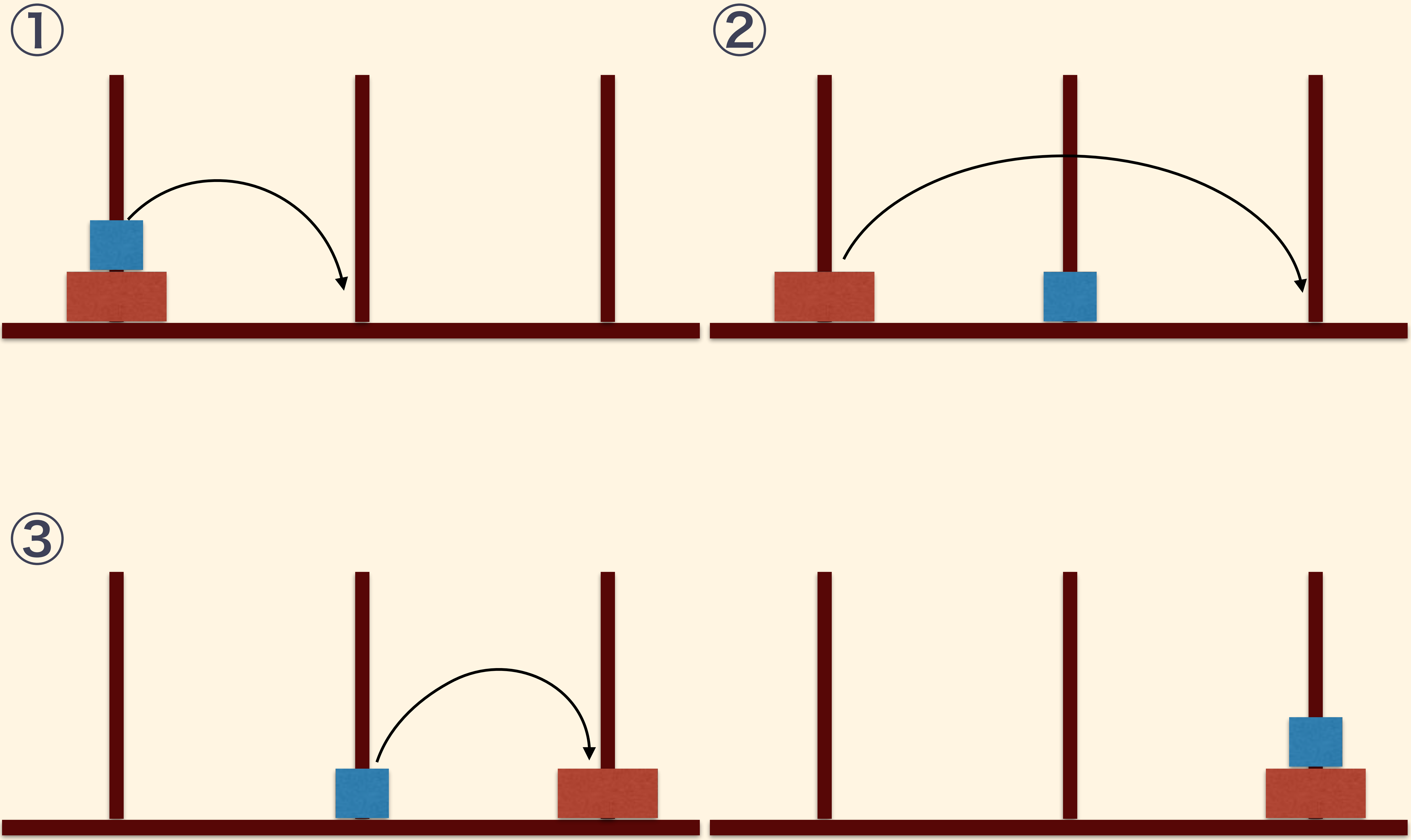
ハノイの塔

ハノイの塔は最短で何手かかるか？ https://www.youtube.com/watch?v=NJhqH9aYP4Y

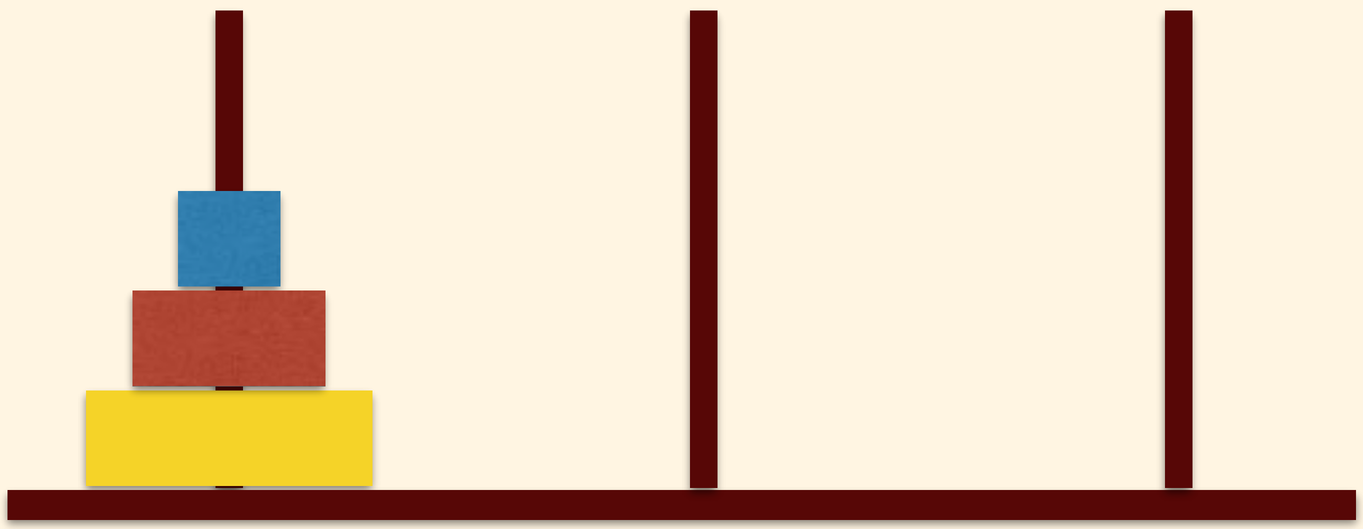
ハノイの塔

ハノイの塔は最短で何手かかるか？ https://www.youtube.com/watch?v=NJhqH9aYP4Y

2個は3手掛かる



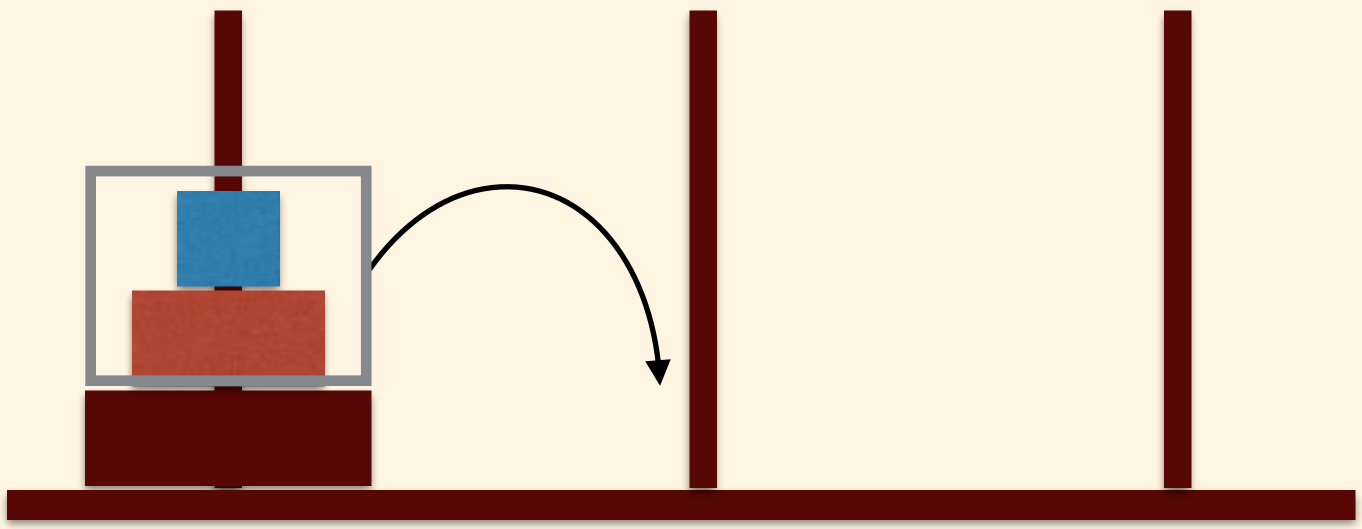
3個は？



- 黄を右に持っていく必要がある
- そのためには上の2個(青,赤)が邪魔

➠ 上の2個をまず真ん中の棒に退避させる

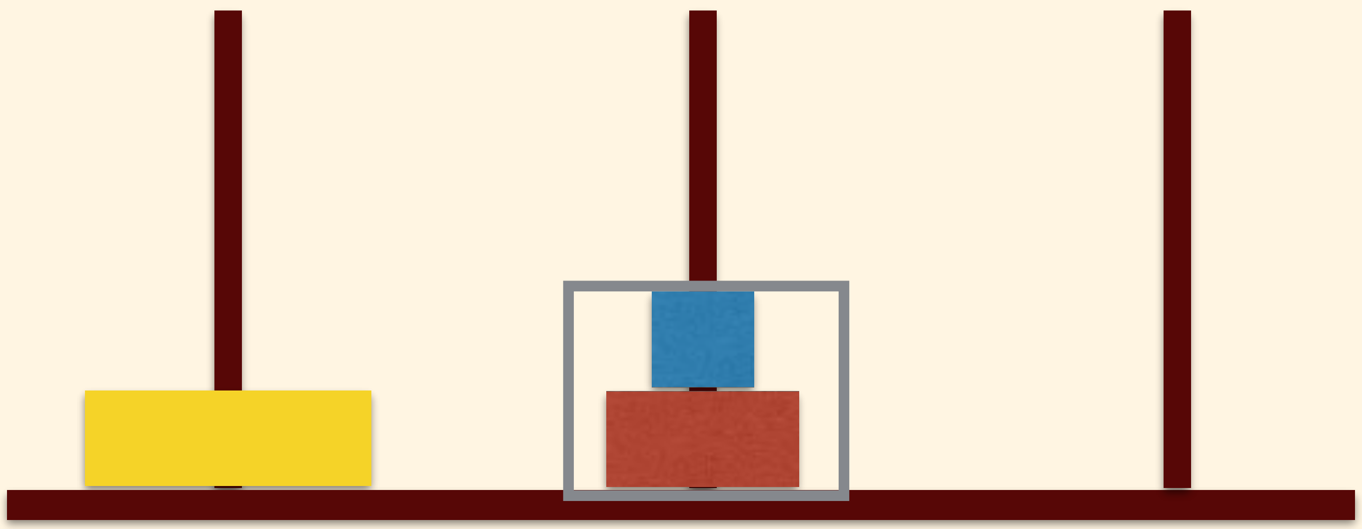
3個は？



- 黄も床として考えれば，2個のハノイの塔と同じ
- 移動する場所が真ん中になっただけ

➠ さっきの議論から3手で真ん中に移動できる

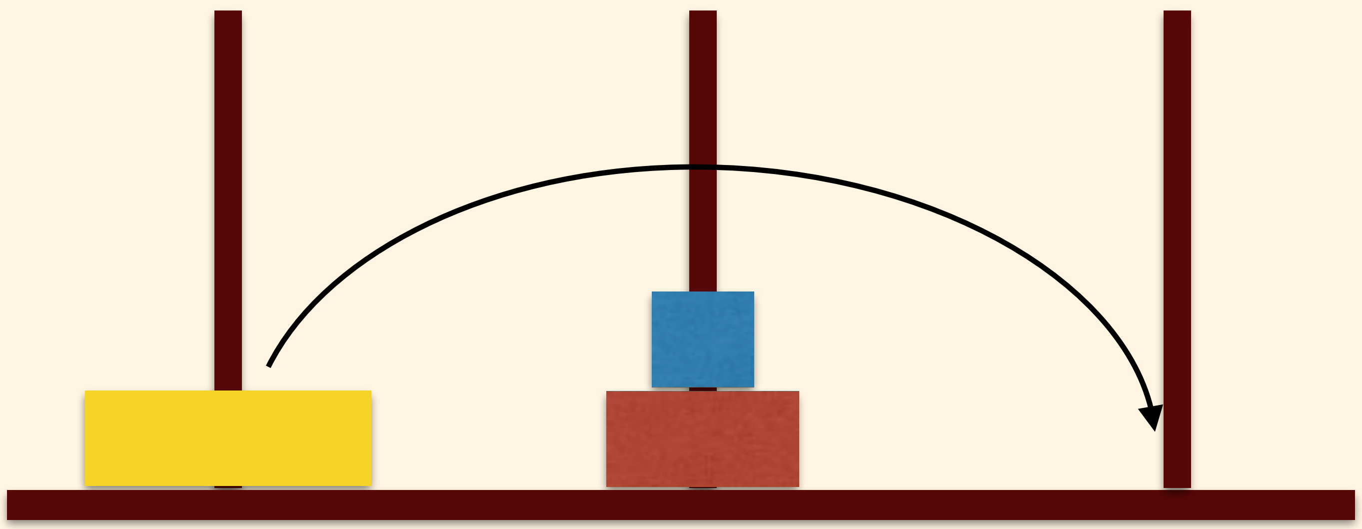
3個は？



- 黄も床として考えれば，2個のハノイの塔と同じ
- 移動する場所が真ん中になっただけ

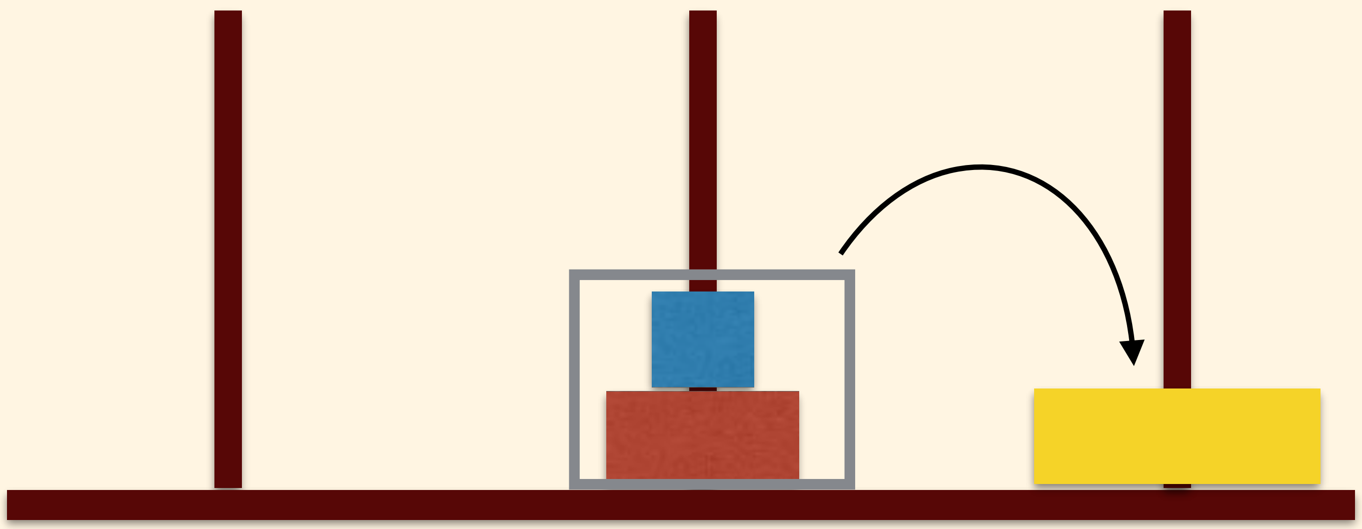
➠ さっきの議論から3手で真ん中に移動できる

3個は？



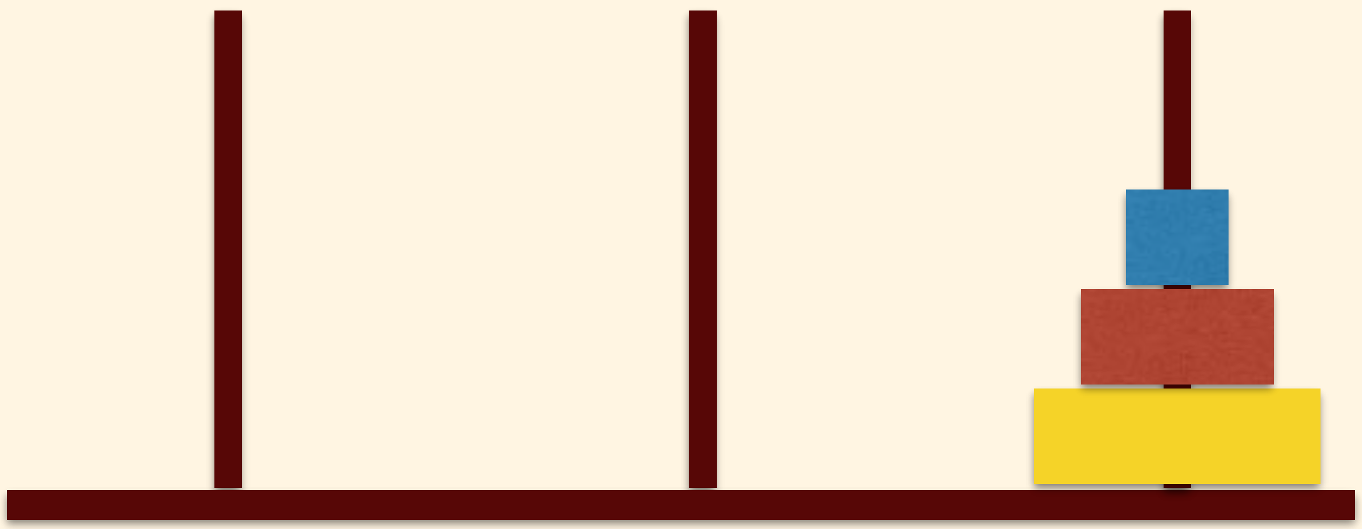
- さらに1手で黄色を右に動かし

3個は？



- 最後にもう一度2個のハノイの塔をやれば良い

3個は？



- 最後にもう一度2個のハノイの塔をやれば良い

➠ 3手 + 1手 + 3手で7手！

N個のハノイの塔を解く関数を書く

- まず1個, 2個はそのまま書いてしまおう

```
int solveHanoi(n) {
  if (n == 1) return 1
  if (n == 2) return 3
}
```

n=1 の時に 1手

n=2 の時に 3手

を返す関数がかけた

N個のハノイの塔を解く関数を書く

- 3個のハノイの塔は下記の手順で解けた
  - 2個のハノイの塔を解く
  - 一番大きな盤を1個動かす
  - 2個のハノイの塔を解く

つまり，3個のハノイの塔を解く最短手数は？

2個のハノイの塔を解く手数 × 2 + 1

➠ **そのまま関数に書くと**

N個のハノイの塔を解く関数を書く

つまり，3個のハノイの塔を解く最短手数は？

2個のハノイの塔を解く手数 × 2 + 1

➠ そのまま関数に書くと

```
int solveHanoi(n) {
  if (n == 1) return 1
  if (n == 2) return 3
  if (n == 3) {
    return solveHanoi(2) + solveHanoi(2) + 1
  }
}
```

* solveHanoi(2)で2個のハノイの塔は解けるのであった

N個のハノイの塔を解く関数を書く

もう少し一般化出来て，N>1に対し，

N個のハノイの塔を解く最短手数は？

N-1個のハノイの塔を解く手数 × 2 + 1

```
int solveHanoi(n) {
  if (n == 1) return 1
  return solveHanoi(n-1) + solveHanoi(n-1) + 1
}
```

* solveHanoi(n-1)でn-1個のハノイの塔は解ける

N個のハノイの塔を解く関数を書く

もう少し一般化出来て，N>1に対し，

N個のハノイの塔を解く最短手数は？

N-1個のハノイの塔を解く手数 × 2 + 1

```
int solveHanoi(n) {
  if (n == 1) return 1
  return solveHanoi(n-1) + solveHanoi(n-1) + 1
}
```

★ 関数を「ある問題を解くもの*」と認識すれば

自然に再帰が書ける(はず*)

* 正確にはある問題のインスタンス
* 個人の感想です

N個のハノイの塔を解く関数を書く

もう少し一般化出来て，N>1に対し，

N個のハノイの塔を解く最短手数は？

N-1個のハノイの塔を解く手数 × 2 + 1

```
int solveHanoi(n) {
  if (n == 1) return 1
  return solveHanoi(n-1) + solveHanoi(n-1) + 1
}
```

★再帰処理を使えば

「大きな問題を小さい問題に分割して解く」

を簡単に実装出来る

メモ化再帰

フィボナッチ数列を再帰で書く

- フィボナッチ数列のn番目の数 F(n)は？

$$F(n) = F(n-1) + F(n-2) \quad n>1$$

$$F(0) = 0$$
$$F(1) = 1$$

0, 1番目の数は0, 1と定義

フィボナッチ数列：0, 1, 1, 2, 3, 5, 8, 13, 21,...

フィボナッチ数列を再帰で書く

- フィボナッチ数列のn番目の数 F(n)は？

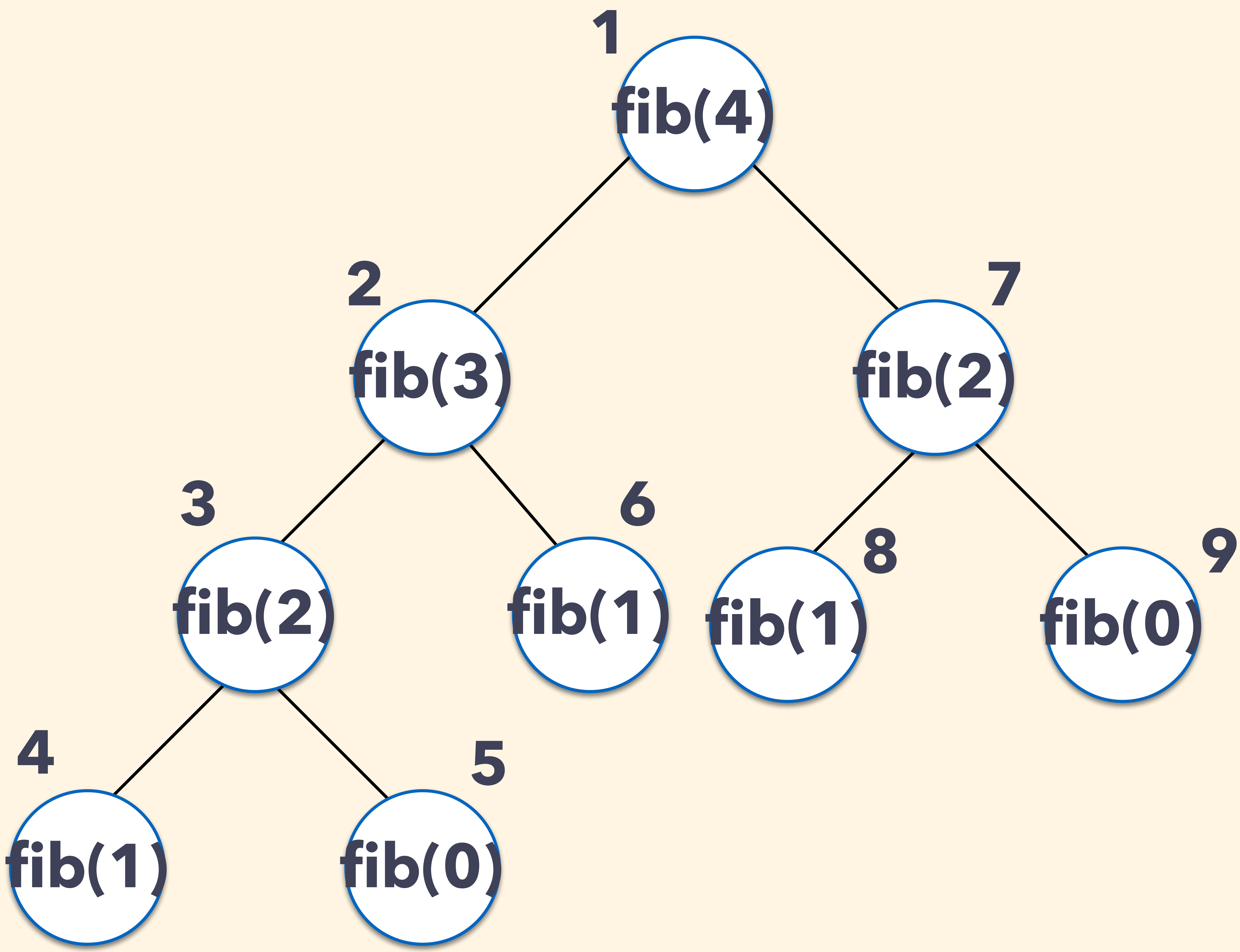
$$F(n) = F(n-1) + F(n-2) \quad n>1$$

$$F(0) = 0$$
$$F(1) = 1$$

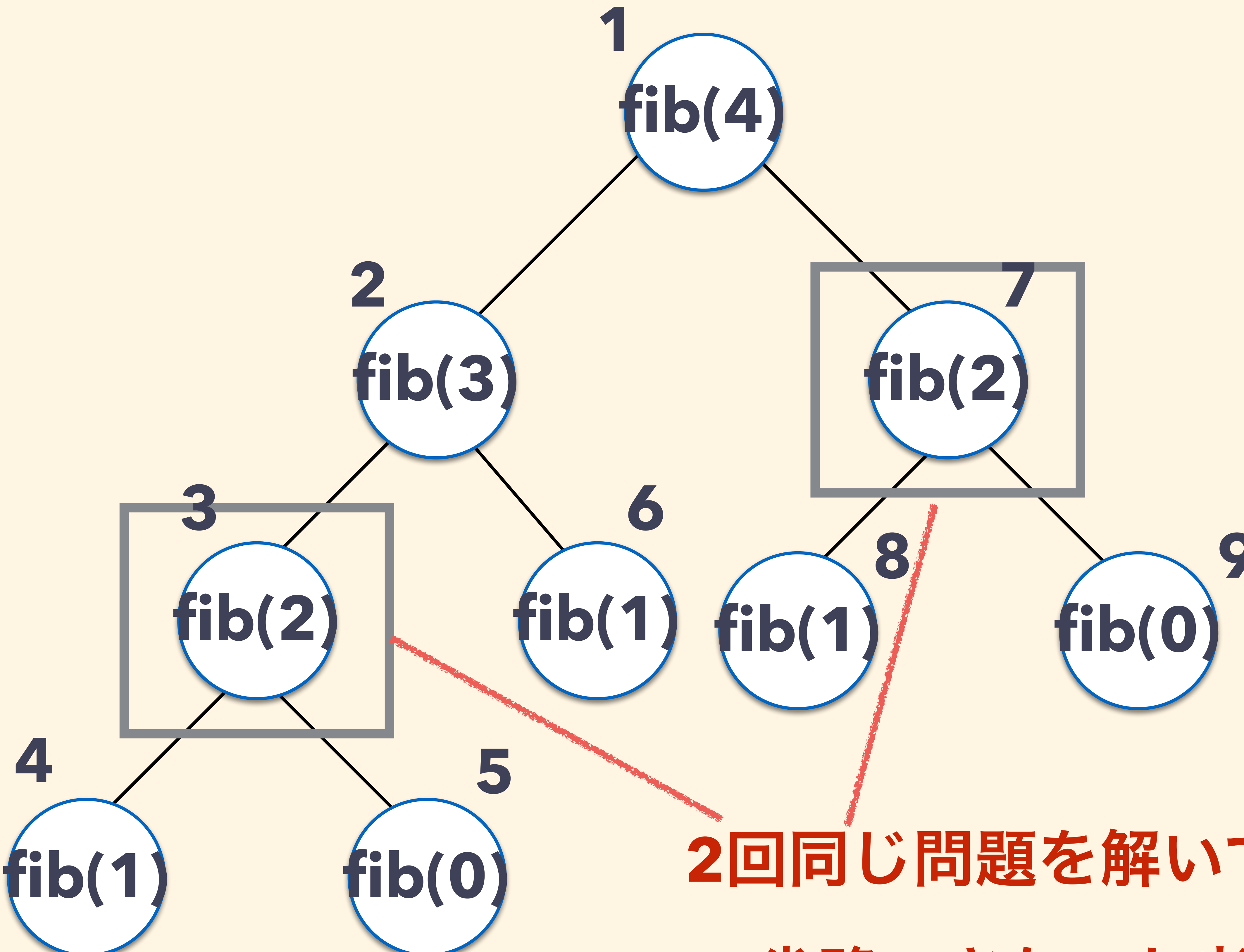
0, 1番目の数は0, 1と定義

```
// fibをn番目の数を返す関数として定義

int fib(n) {
  if (n == 0) return 0
  if (n == 1) return 1
  return fib(n-1) + fib(n-2)
}
```

fib(4)の呼出しを図で書くと



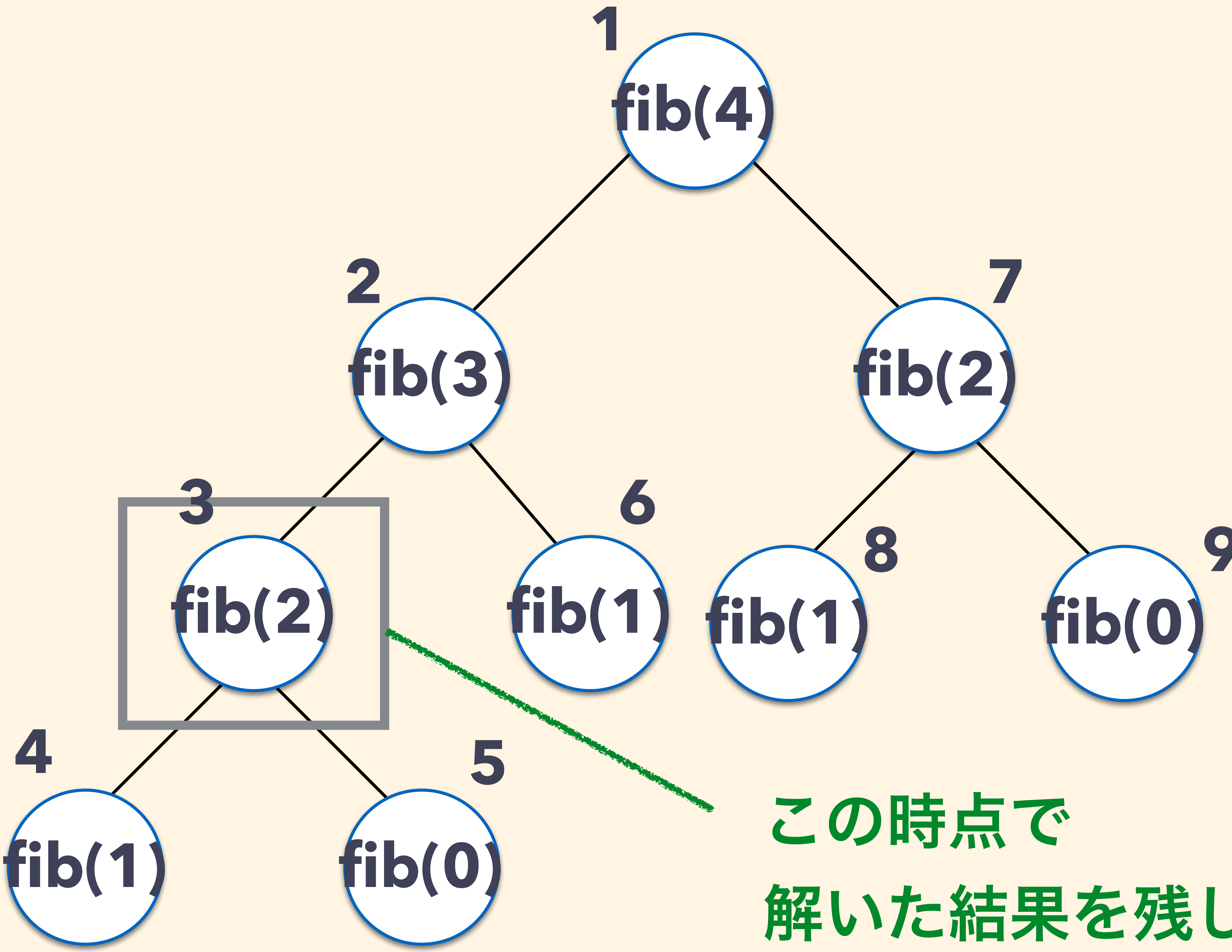
fib(4)の呼出しを図で書くと



2回同じ問題を解いている

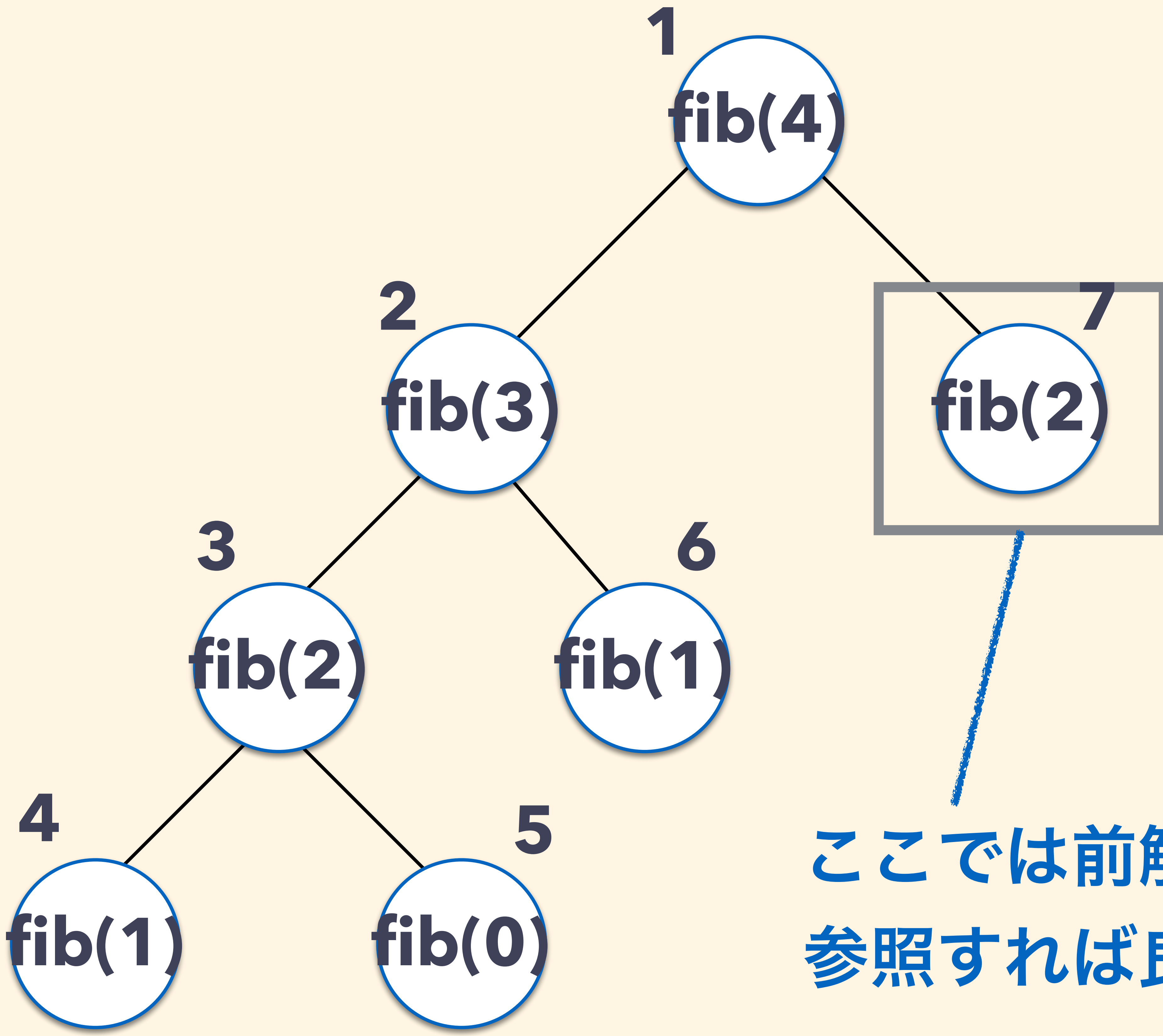
➠ 省略できないか考える

fib(4)の呼出しを図で書くと



この時点で
解いた結果を残しておく

fib(4)の呼出しを図で書くと

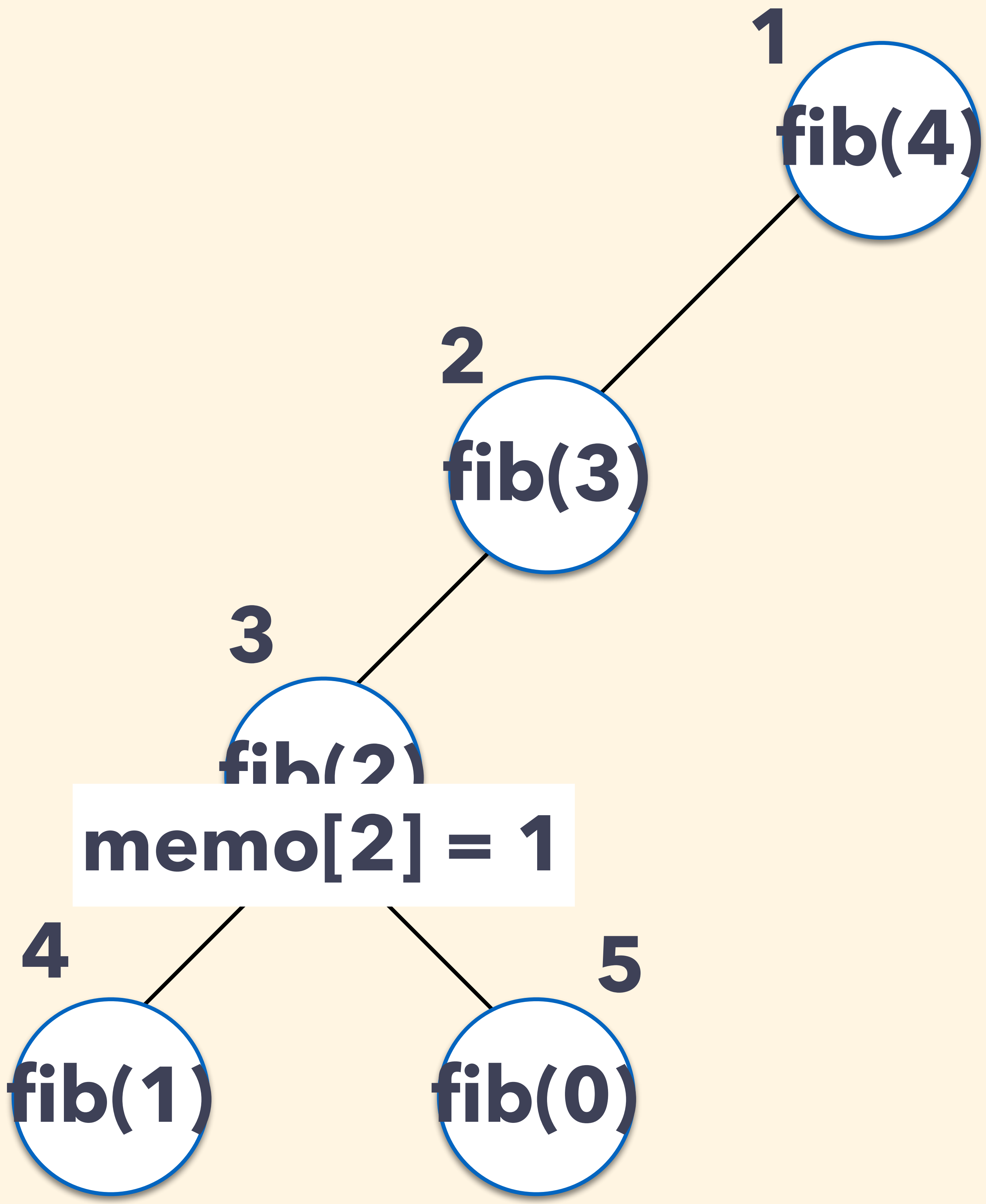


ここでは前解いた結果を
参照すれば良い

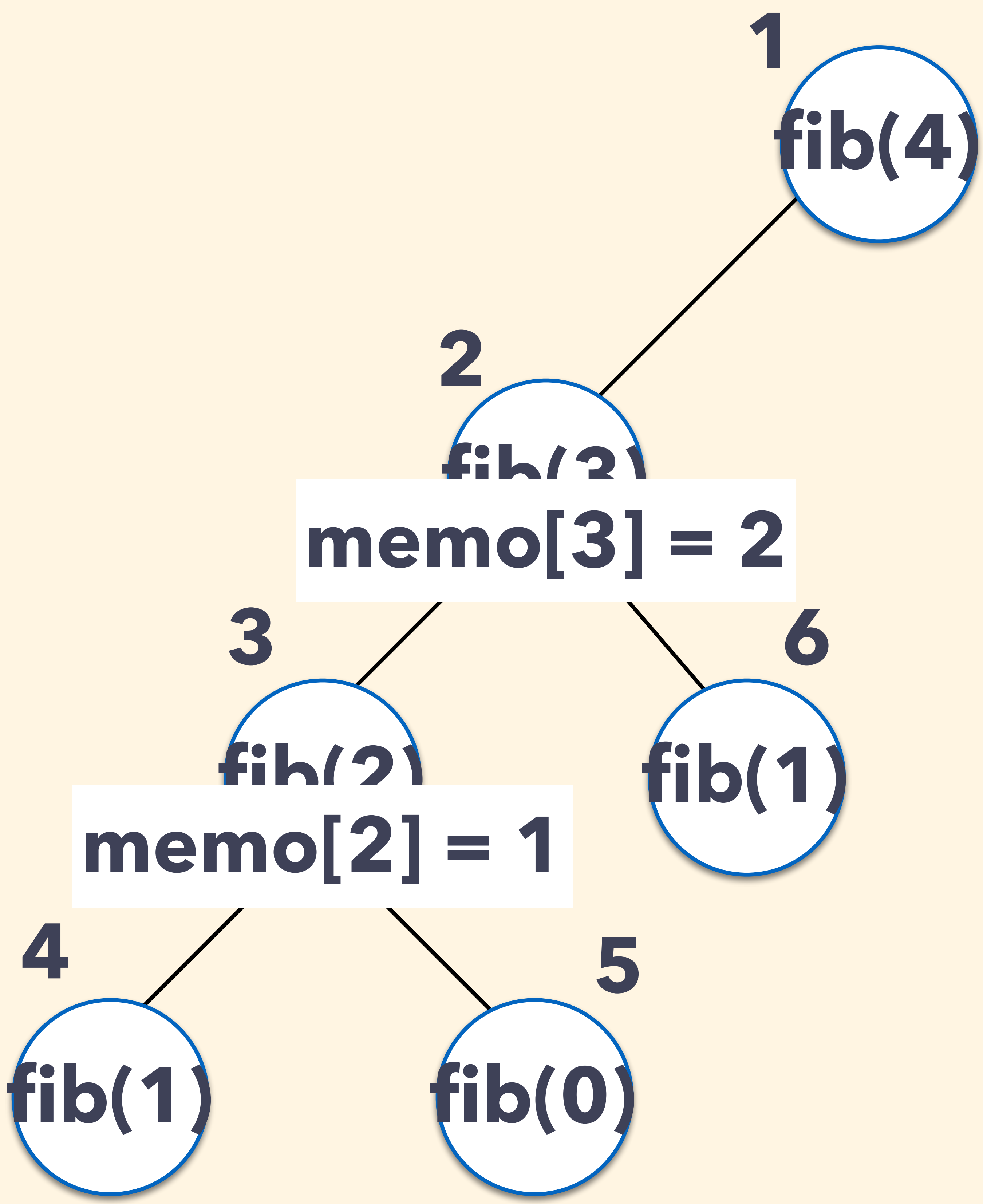
コードで書くと？

```
memo[MAX_N] := fib(n)を解いた結果を保存
               -1で初期化しておく
// fibをn番目の数を返す関数として定義
int fib(n) {
  if (n == 0) return 0
  if (n == 1) return 1
  if (memo[n] != -1) {
    return memo[n] // 計算済みであればそれを利用
  }
  memo[n] = fib(n-1) + fib(n-2) // 計算結果保存
  return memo[n]
}
```

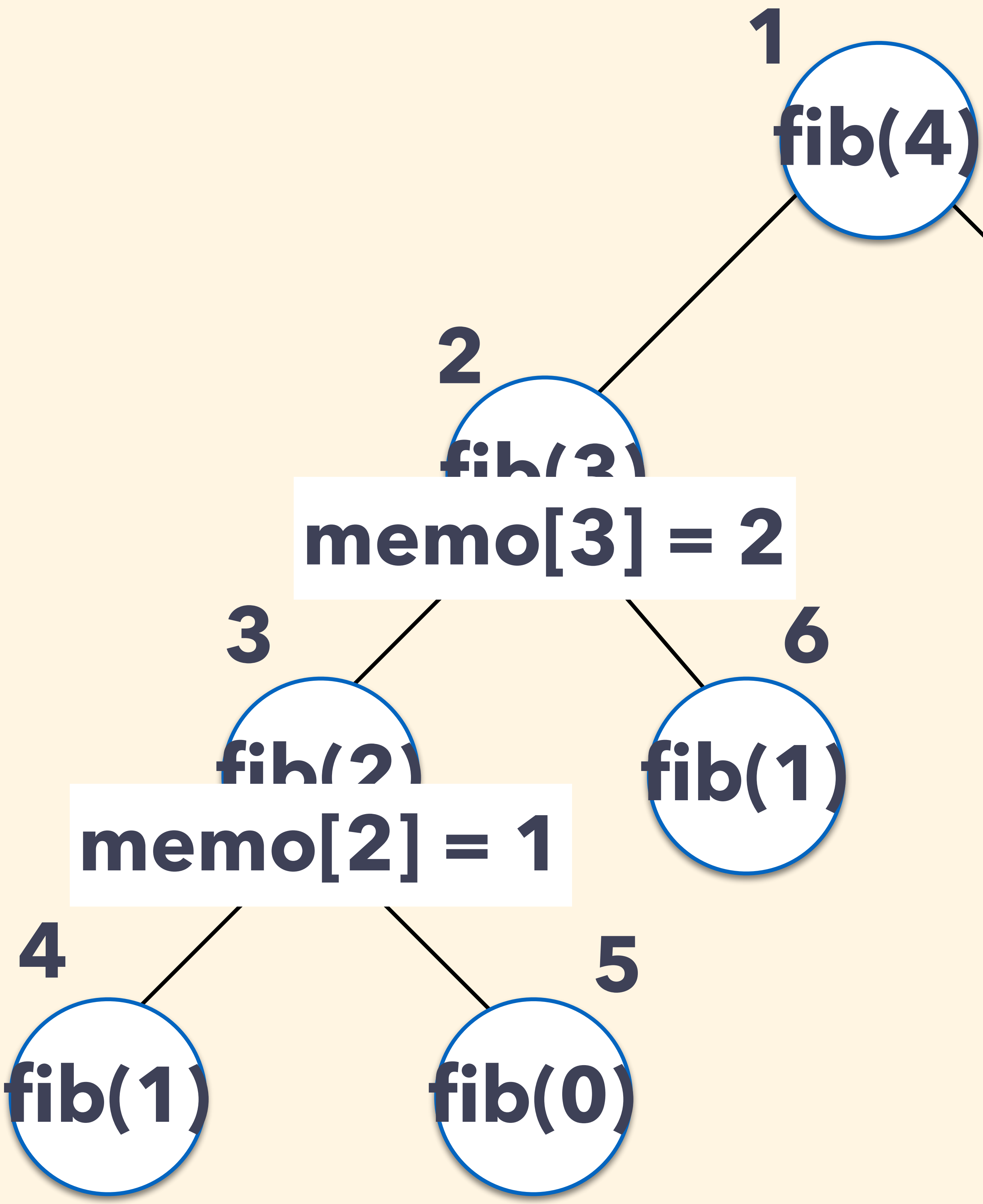
fib(4)の呼出しを図で書くと



fib(4)の呼出しを図で書くと



fib(4)の呼出しを図で書くと



3の時点でfib(2)は計算済み

memo[2]に答えが入っている

計算量について

- メモ化なしフィボナッチ
    - O(2^n)
        - 毎回2股に分岐する

- メモ化ありフィボナッチ
    - O(n)
        - fib(i)はそれぞれ1回しか計算されず,
          1回の計算は定数回の演算で済む

まとめ

- 深さ優先探索(バックトラック法)で,
  移動経路等の複雑なパターンの列挙が出来る
- 再帰は関数をある問題を解くものとして
  考えると書きやすい(?)
- メモ化によりグッと計算量を小さくできる
  - メモ化って競プロ用語だと思っていたのですが,
    そんなこともないらしい

難問に挑戦

- AtCoder Beginner Contest 008 金塊ゲーム：https://atcoder.jp/contests/abc008/tasks/abc008_4
    - 部分点の99点まで
    - かなり難しめのABC D
    - 結構好きな問題なので是非挑戦して下さい
    - 気持ち実装重めなのでアルゴリズムだけ考えて, 解説を見て答え合わせでも良いと思います