

# プログラミング第一同演習

---

慶應義塾大学 理工学部 情報工学科

講義担当：河野 健二

演習担当：杉浦 裕太

- 再帰呼び出し (*recursive call*)
  - 高度なプログラミング技術のひとつ
  - 難しい問題は再帰呼び出しを使うことが多い
  - 「アルゴリズム」の講義でさまざまなものを学ぶ
  
- 慣れるには相当な訓練が必要
  - これから、いろいろな事例に出会う
  - 少しずつ、慣れていけばよい
  - 関数型言語を習得するとよい
    - ◆ O'Caml, Haskell など

# 再帰呼び出し (recursive call)

- 関数の中で自分自身を呼び出すこと

- 例1: 関数 f() の中で関数 f() を呼び出す

```
int f()  
{  
    ...  
    f();  
    ...  
}
```

- 例2: 関数 f() と g() が互いに呼び合っている. 相互再帰

|   |   |
|---|---|
| <pre>int f()<br/>{<br/>    ...<br/>    g();<br/>    ...<br/>}</pre> | <pre>int g()<br/>{<br/>    ...<br/>    f();<br/>    ...<br/>}</pre> |
|---|---|

## 例題：階乗の計算

- 整数  $n$  の階乗  $n!$  を計算する関数 `int factorial(int n)` を考える

- $n$  の階乗は以下のように表すことができる

$n! = 1$                       if  $n == 0$   
 $n! = n \times (n - 1)!$  otherwise

- $!$  の代わりに `factorial()` を使うと...

`factorial(n) = 1`                      if  $n == 0$   
`factorial(n) = n × factorial(n - 1)` otherwise

- 関数 `factorial` の定義に関数 `factorial` が使われている
- 関数 `factorial` が再帰的に定義されている

# 定義に従って C の関数を定義する

## ■ 関数 factorial の（数学的な）定義

$$\begin{array}{ll} \text{factorial}(n) = 1 & \text{if } n == 0 \\ \text{factorial}(n) = n \times \text{factorial}(n - 1) & \text{otherwise} \end{array}$$

## ■ そのまま下のように書ける

```
int factorial(int n)
{
    if (n == 0) ← 停止条件
        return 1;
    return n * factorial(n-1);
}
```

再帰呼び出し

### 停止条件:

この条件が成り立つときは、再帰的に関数を呼び出さない  
停止条件がないと、永遠に関数が呼び出されてしまう

### 再帰呼び出し:

自分自身を呼び出すこと

## 再帰の様子：呼び出しのとき (1/4)

```
r = factorial(3);
```

(1)

n == 3 として呼び出す

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n-1);
}
```

n の値は 0 ではないので,  
return n \* factorial(n - 1);  
を実行する

したがって,  
factorial(2);  
を呼び出す

## 再帰の様子：呼び出しのとき (2/4)

n == 3 として呼び出されている

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n-1);
}
```

factorial(2) を呼び出す

n == 2 として呼び出す

n の値は 0 ではないので、  
factorial( 1 )  
を呼び出す

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n-1);
}
```

## 再帰の様子：呼び出しのとき (3/4)

n == 2 として呼び出されている

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n-1);
}
```

factorial(1) を呼び出す

n == 1 として呼び出す

n の値は 0 ではないので、  
factorial(0)  
を呼び出す

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n-1);
}
```



## 再帰の様子：呼び出しのとき (4/4)

n == 1 として呼び出されている

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n-1);
}
```

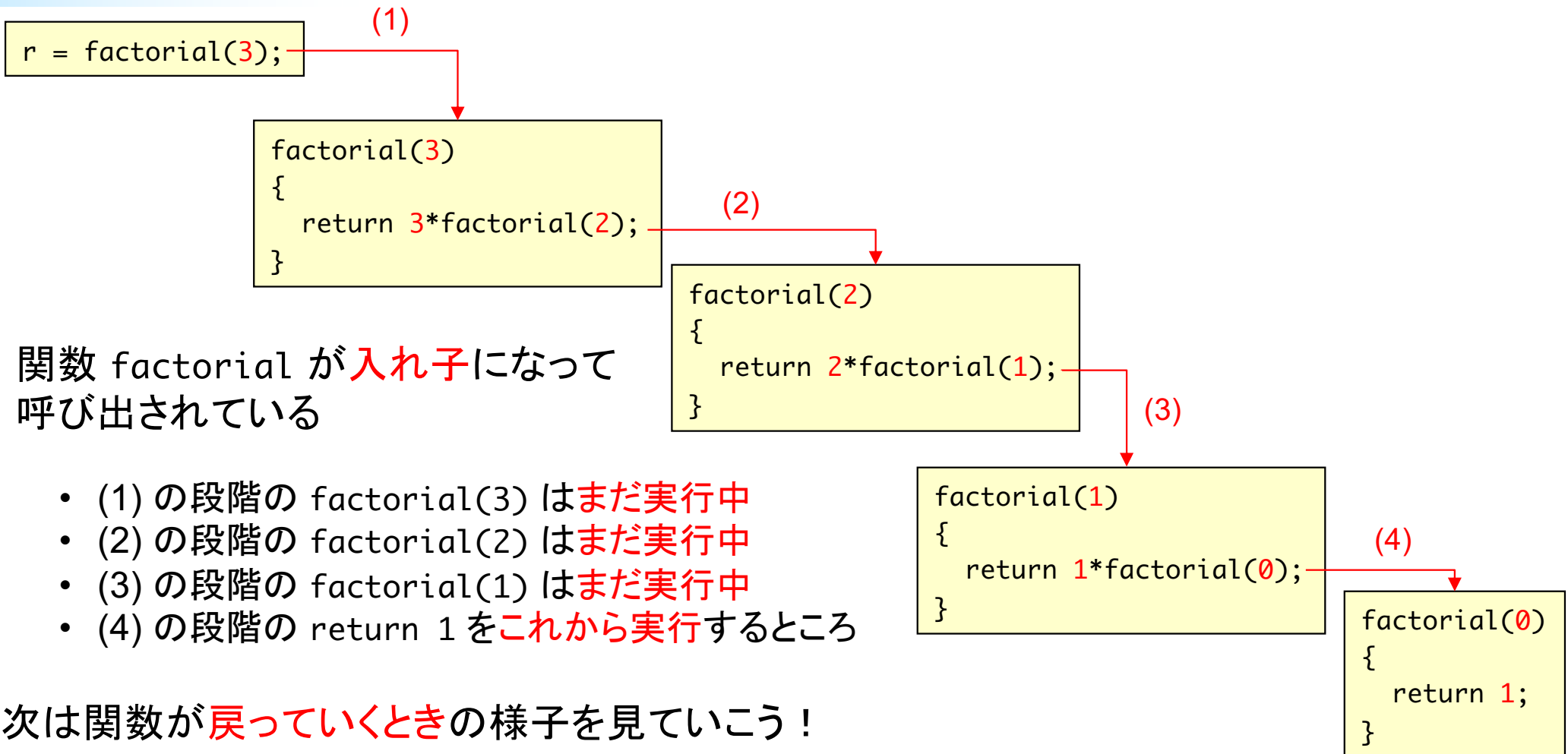
factorial(0) を呼び出す

n == 0 として呼び出す

n の値は 0 なので,  
return 1  
を実行する

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n-1);
}
```

## 再帰の様子: ここまでの状態



## 再帰の様子：戻るとき (1/4)

n == 1 として呼び出されている

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n-1);
}
```

n は 1 のまま

呼び出し中の factorial(0) が 1 を返す  
n の値は 1 なので、  
return n \* factorial(n - 1) は  
return 1 \* 1  
となる。すなわち 1 を返す

factorial(0) は 1 を返す

n == 0 として呼び出す

n の値は 0 なので、  
return 1  
を実行する

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n-1);
}
```

## 再帰の様子：戻るとき (2/4)

$n == 2$  として呼び出されている

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n-1);
}
```

factorial(1) は 1 を返す

$n == 1$  として呼び出す

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n-1);
}
```

呼び出し中の factorial(1) が 1 を返す  
 $n$  の値は 2 なので、  
 $\text{return } n * \text{factorial}(n - 1)$  は  
 $\text{return } 2 * 1$   
 となる。すなわち 2 を返す

$\text{return } n * \text{factorial}(n - 1)$  は 1 を返す

## 再帰の様子：戻るとき (3/4)

n == 3 として呼び出されている

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n-1);
}
```

n は 3 のまま

factorial(2) は 2 を返す

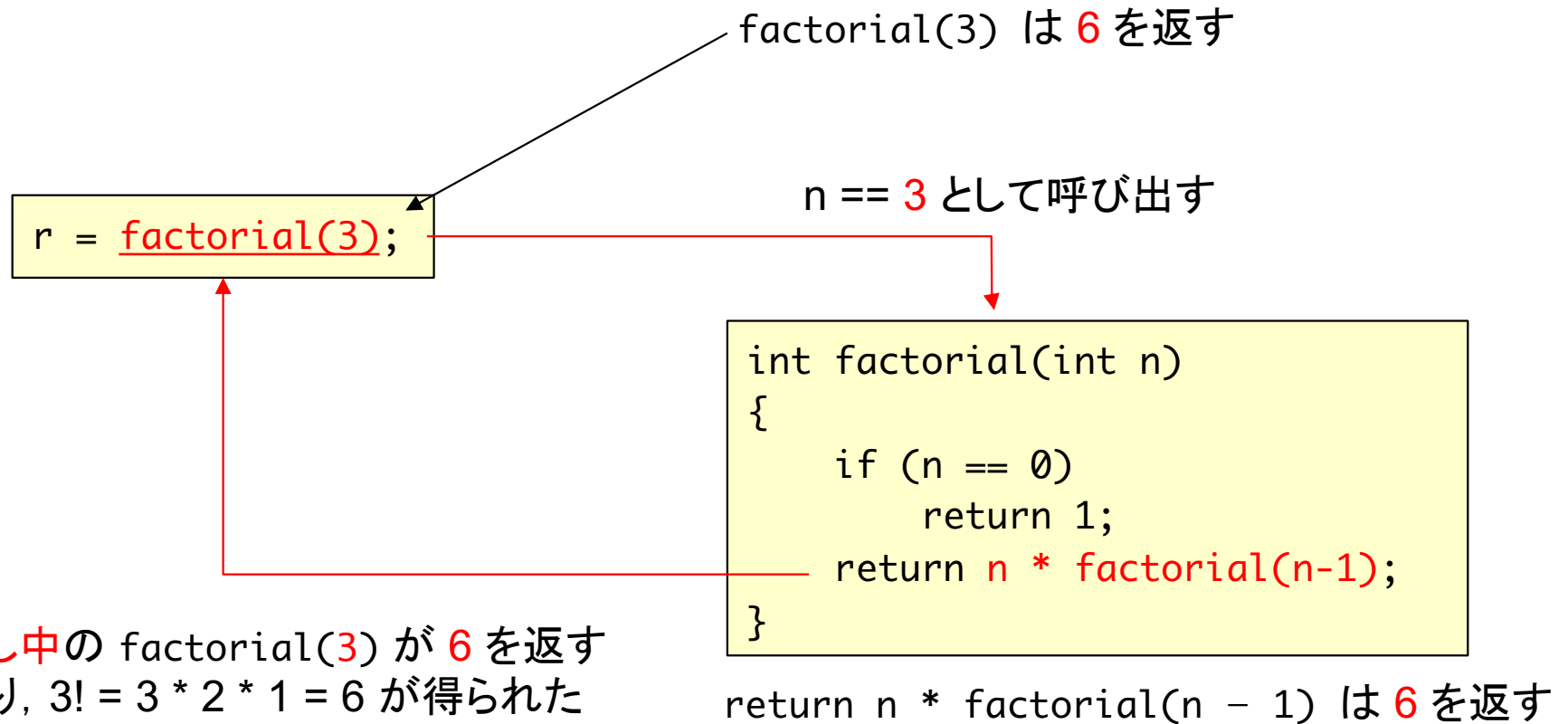
n == 2 として呼び出す

呼び出し中の factorial(2) が 2 を返す  
n の値は 3 なので、  
return n \* factorial(n - 1) は  
return 3 \* 2  
となる。すなわち 6 を返す

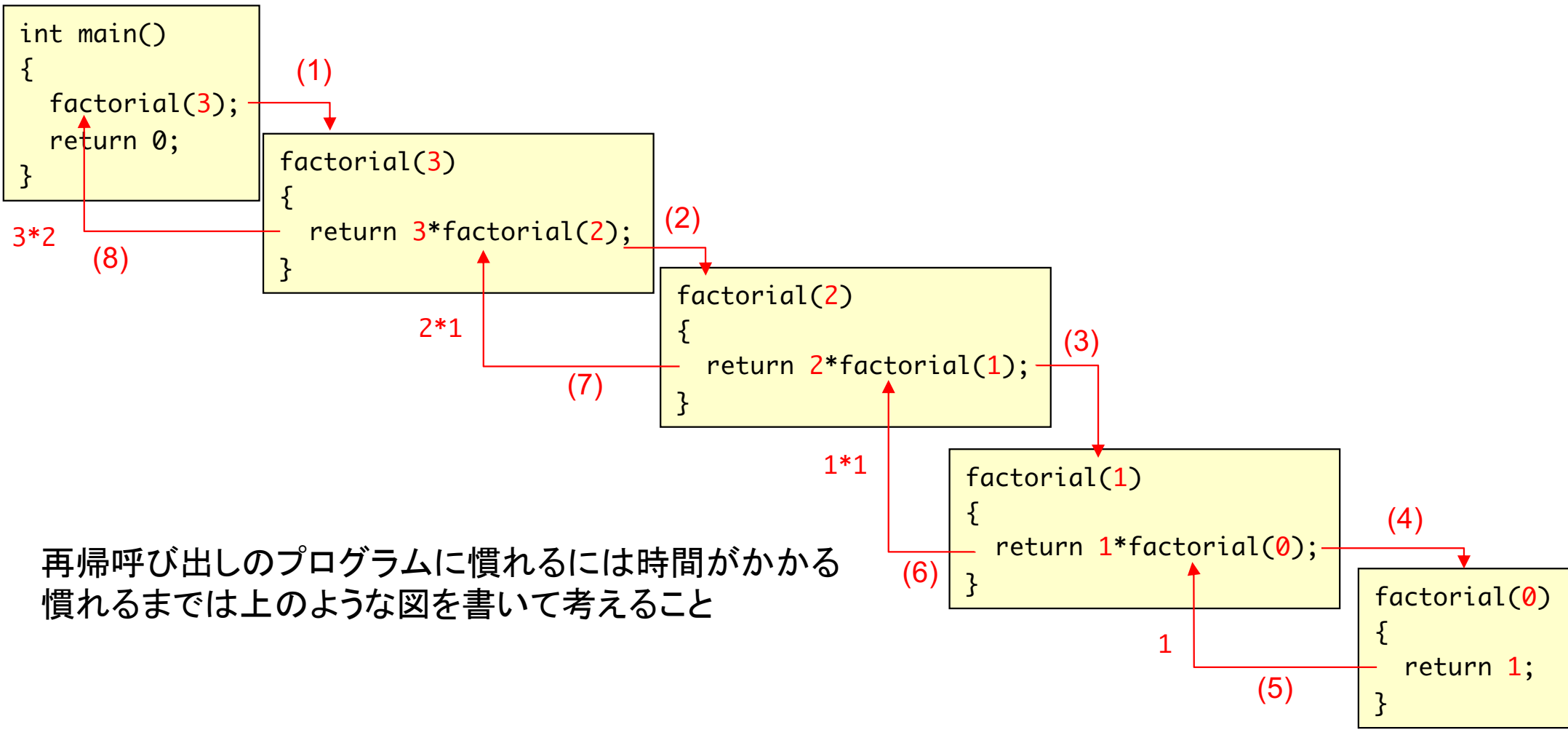
```
int factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n-1);
}
```

return n \* factorial(n - 1) は 2 を返す

## 再帰の様子：戻るとき (4/4)



# 再帰の様子：開始から終了まで



# 再帰の動作の様子

- 再帰の動作を理解するため, 簡単な例を調べよう

```
void f(int n)
{
    printf("f(%d) called¥n", n);

    if (n <= 0) {
        printf("recursive call finished¥n");
        return;
    }

    f(n - 1);
    printf("after recursive call: n = %d", n);
}
```



- f(2) として呼び出してみると...

```
f(2) is called
  f(1) is called
    f(0) is called
    recursive call finished
  after recursive call: n = 1
after recursive call: n = 2
```

- わかりやすくするため、インデントと色は変えてある

# 再帰の様子：開始から終了まで

f(2);

f(2) is called

```
f(n == 2 で呼び出し)
{
    printf("f(%d) called¥n", n);
    f(n - 1);
    printf("after recursive call: n = %d", n);
}
```

f(1) is called

after recursive call: n = 2

```
f(n == 1 で呼び出し)
{
    printf("f(%d) called¥n", n);
    f(n - 1);
    printf("after recursive call: n = %d", n);
}
```

recursive call finished

after recursive call: n = 1

```
f(n == 0 で呼び出し)
{
    printf("recursive call finished¥n");
}
```

# フィボナッチ数列を再帰で解く例

- $\text{Fib}(n) = 1$  if  $n \leq 2$   
 $\text{Fib}(n) = \text{Fib}(n - 1) + \text{Fib}(n - 2)$  otherwise

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

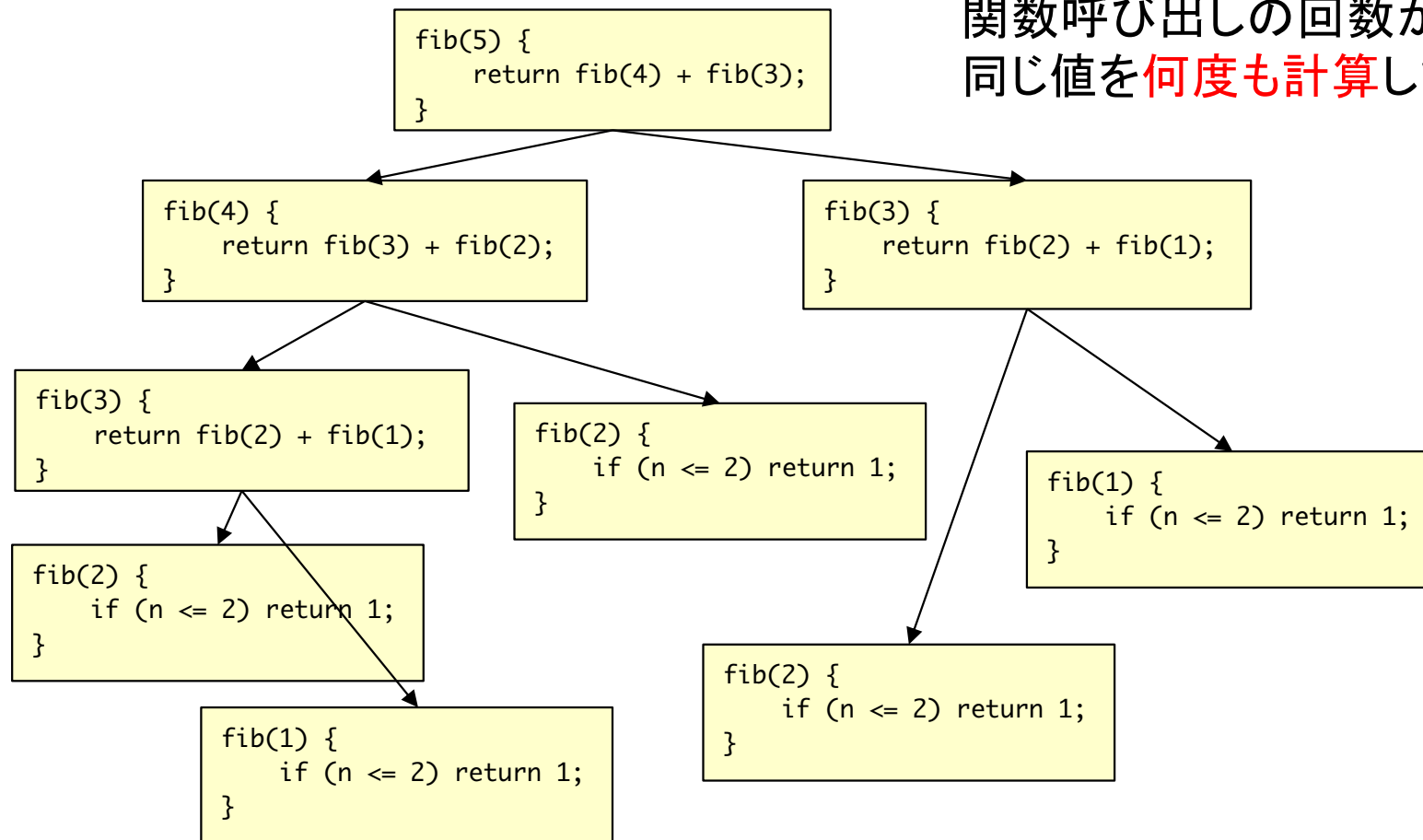
停止条件

再帰呼び出し

再帰呼び出し

# 再帰の様子

関数呼び出しの回数が**多い**  
同じ値を**何度も計算**している



# なんで再帰なんてやるの？

- 階乗の例:
  - 再帰を使わなくとも解くことができる
- フィボナッチの例:
  - 再帰を使わなくとも解くことができる
  - 再帰を使うと, 無駄な計算が多い

## 反復型の n! の関数定義

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

# 再帰型プログラムと反復型プログラム



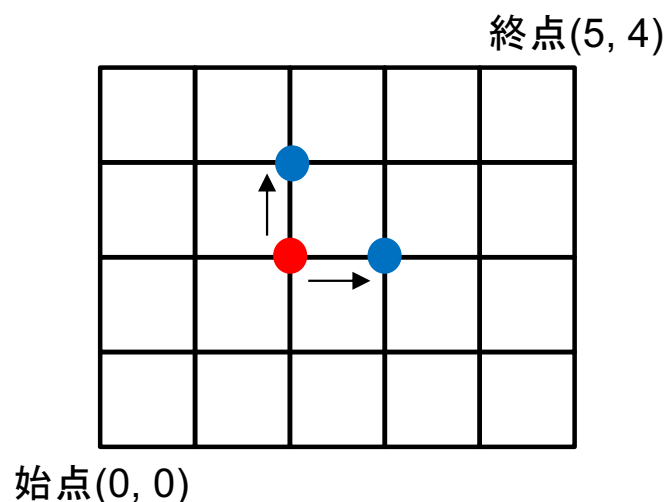
- 再帰型の方が反復型より簡潔になることが多い
  - $n!$  のように, 定義自身やデータ構造の性質から自然に導き出されるものが多い
  
- 再帰型でなければ解けない問題も多い
  - 「アルゴリズム」の講義で多くの実例に接するはず
  - 難しい問題の多くは, 再帰型で解くことが多い
    - ◆ N-Queen 問題, 最適化問題の多く, 他にも数多い

# どちらを使うべき？

- 反復型 vs 再帰型のどちらがいいのか？
  - ケース・バイ・ケース
  
- 反復型で自然に書けるなら反復型でよい
  - 実行効率は反復型の方が良い
    - ◆ 再帰型は何度も関数呼び出しを行うため、遅くなりがち
  
  - 階乗を解くのに再帰は不要
    - ◆ 再帰でも解けるというだけ
  
  - だからといって、再帰を理解しなくてよいことにはならない

## 再帰に適した例題: 経路数

- 碁盤目状の道路がある. 始点から終点までの経路の数を求めなさい.  
ただし, 右または上方向にだけ進めて, その逆方向に進むことは考えないものとする



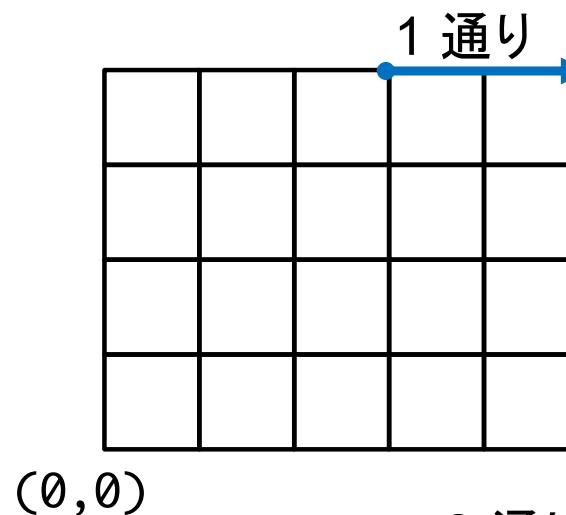
### 考え方

点  $(x, y)$  から終点までの経路数は,  
 $(x+1, y)$  からの経路数 +  $(x, y+1)$  からの経路数  
である

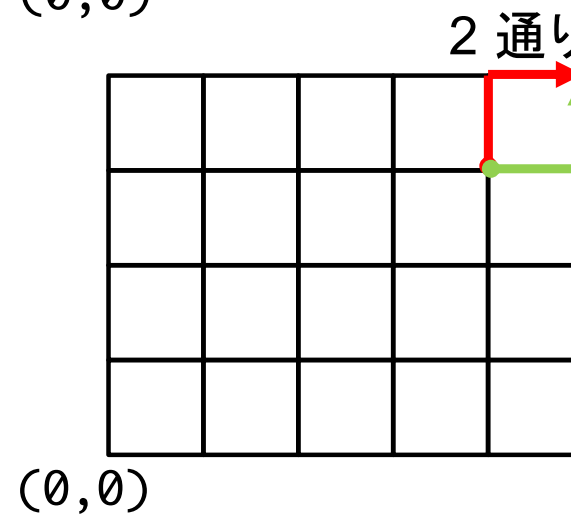


## 考え方 (1/4)

- 点 (3, 4) からの行き方は 1 通り

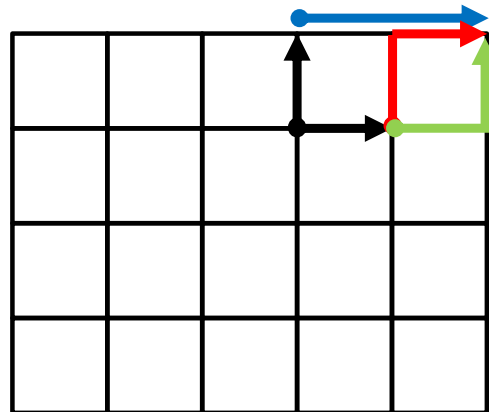


- 点 (4, 3) からの行き方は 2 通り



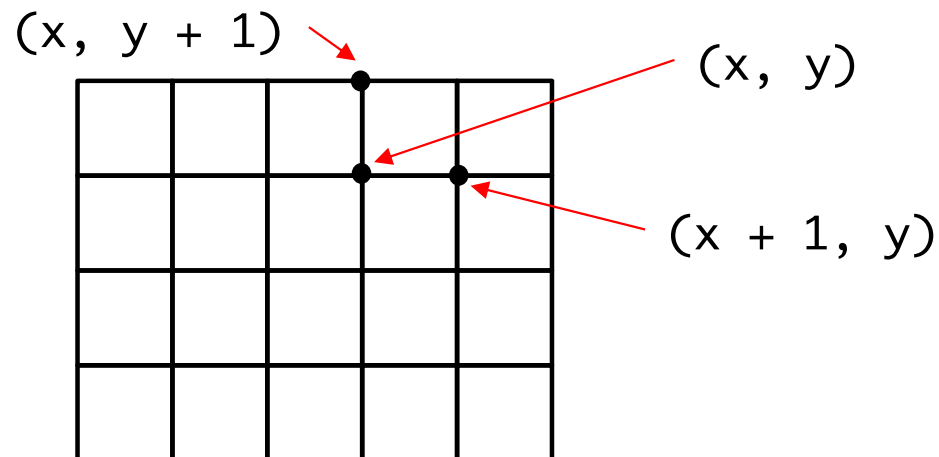
## 考え方 (2/4)

- 点 (3, 3) からの行き方は何通り？
  - 点 (3, 3) から次に進めるのは、点 (3, 4) と点 (4, 3) の二つだけ
  - 点 (3, 3) から終点までの行き方は 1 通り
  - 点 (4, 3) から終点までの行き方は 2 通り
  - よって、 $1 + 2 = 3$  通り



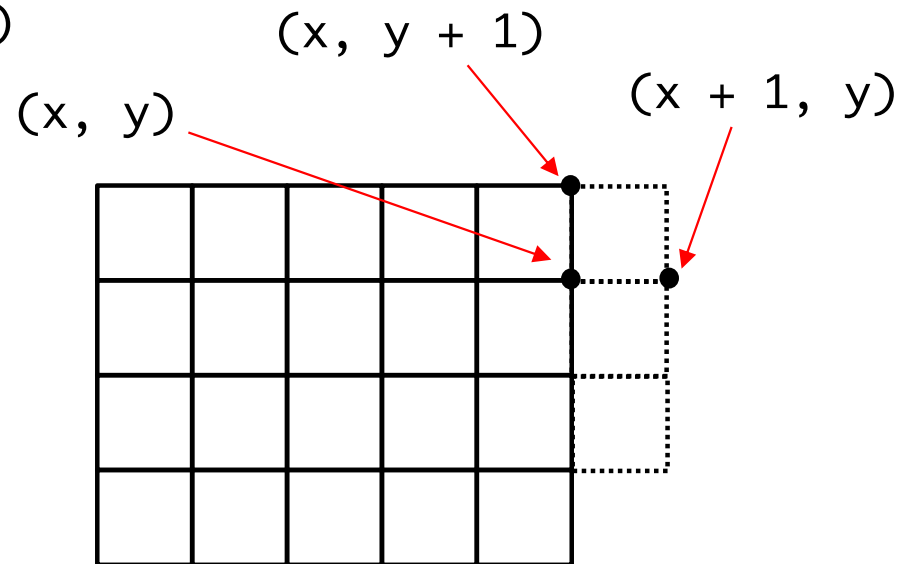
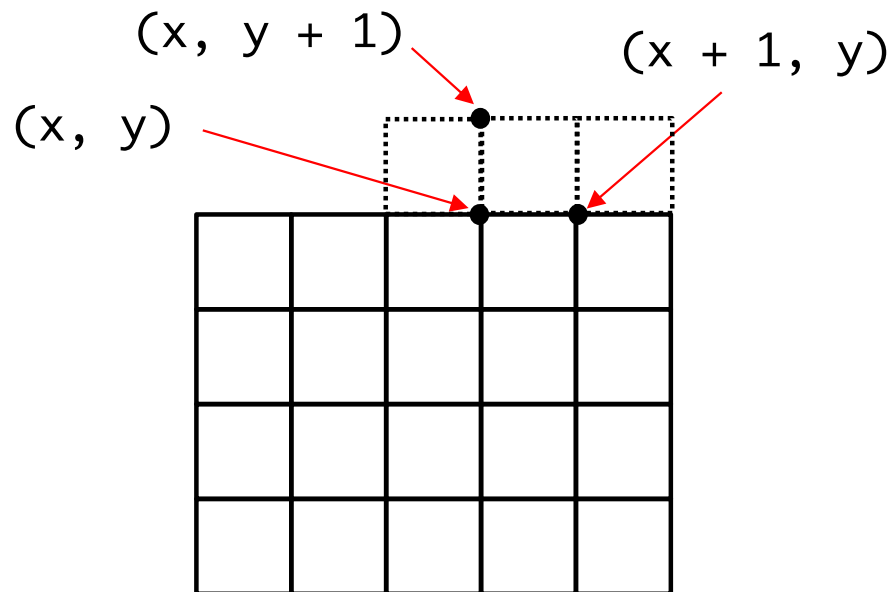
## 考え方 (3/4)

- 点  $(x, y)$  から終点までの行き方を  $S(x, y)$  通りとする
- $S(x, y)$  を求めるプログラムを書けばよい
  - 点  $(x, y)$  から進めるのは、点  $(x, y + 1)$  と点  $(x + 1, y)$  のみ
  - よって、 $S(x, y) = S(x, y + 1) + S(x + 1, y)$  となる
    - ◆  $S$  の定義に  $S$  が出てくる再帰の形となっている



### ■ 境界のところに注意

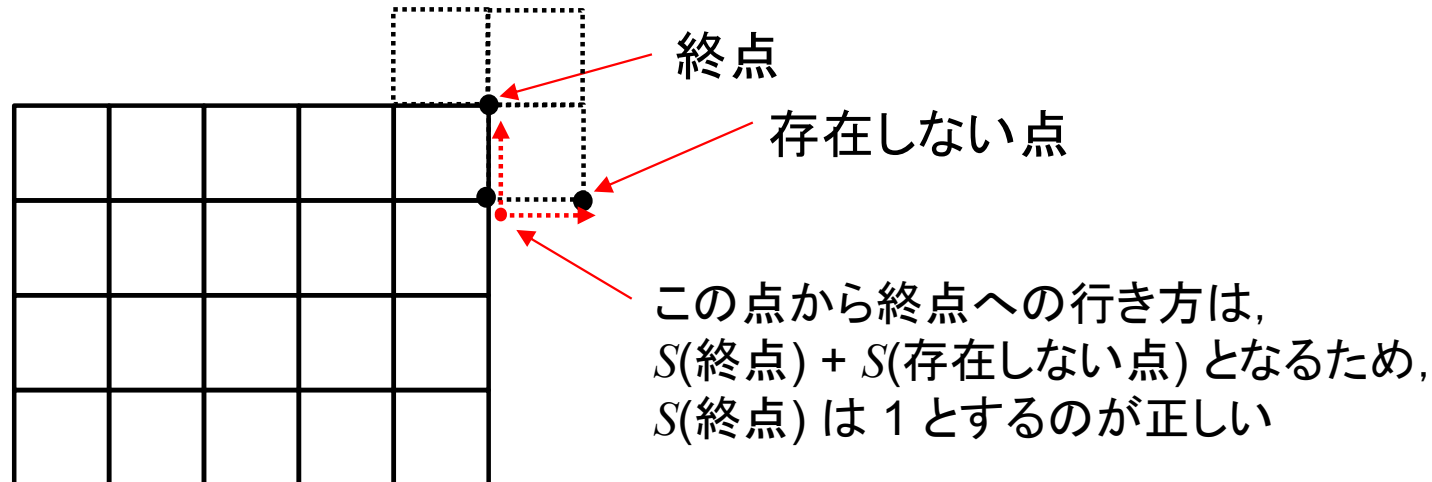
- 下の左図では  $(x, y + 1)$  には行けない.  $S(x, y + 1) = 0$  と考えればよい
- 下の右図では  $(x + 1, y)$  には行けない.  $S(x + 1, y) = 0$  と考えればよい



## まとめと...

- $S(x, y) = 0$  if 点 $(x, y)$  が存在しない  
     $= 1$  if 点 $(x, y)$  が終点  
     $= S(x + 1, y) + S(x, y + 1)$  otherwise

- 終点の場合, そこへの行きは 1 通りと数える



## 例題: プログラムと実行例

```
#include <stdio.h>

int xmax, ymax;

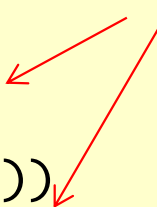
int search(int, int);

int main()
{
    printf("Input X-max & Y-max: ");
    scanf("%d %d", &xmax, &ymax);
    printf("Number of routes: %d\n", search(0, 0));
    return 0;
}
```

## 例題: プログラムと実行例

```
int
search(int x, int y)
{
    if ((x > xmax) || (y > ymax))
        return 0;
    if ((x == xmax) && (y == ymax))
        return 1;
    return search(x+1, y) + search(x, y+1);
}
```

停止条件



```
% ./a.out
Input X-max & Y-max: 5 4
Number of routes: 126
%
```

- この問題のように,

一般項を求めるのは難しいが,  
漸化式のようなものは求まる

というケースは多い

- こういう時に活躍するのが再帰型の解法



- 再帰によるプログラミング
  - 自分自身を呼び出す関数
  
- 再帰の基本的な考え方を説明した
  - 多くの有用な実例はアルゴリズム等で学ぶ
  
- 問題解決のための強力な武器
  - 直面した問題を再帰的に解決できる力を身につけよう
  - それなりの訓練と時間がかかる