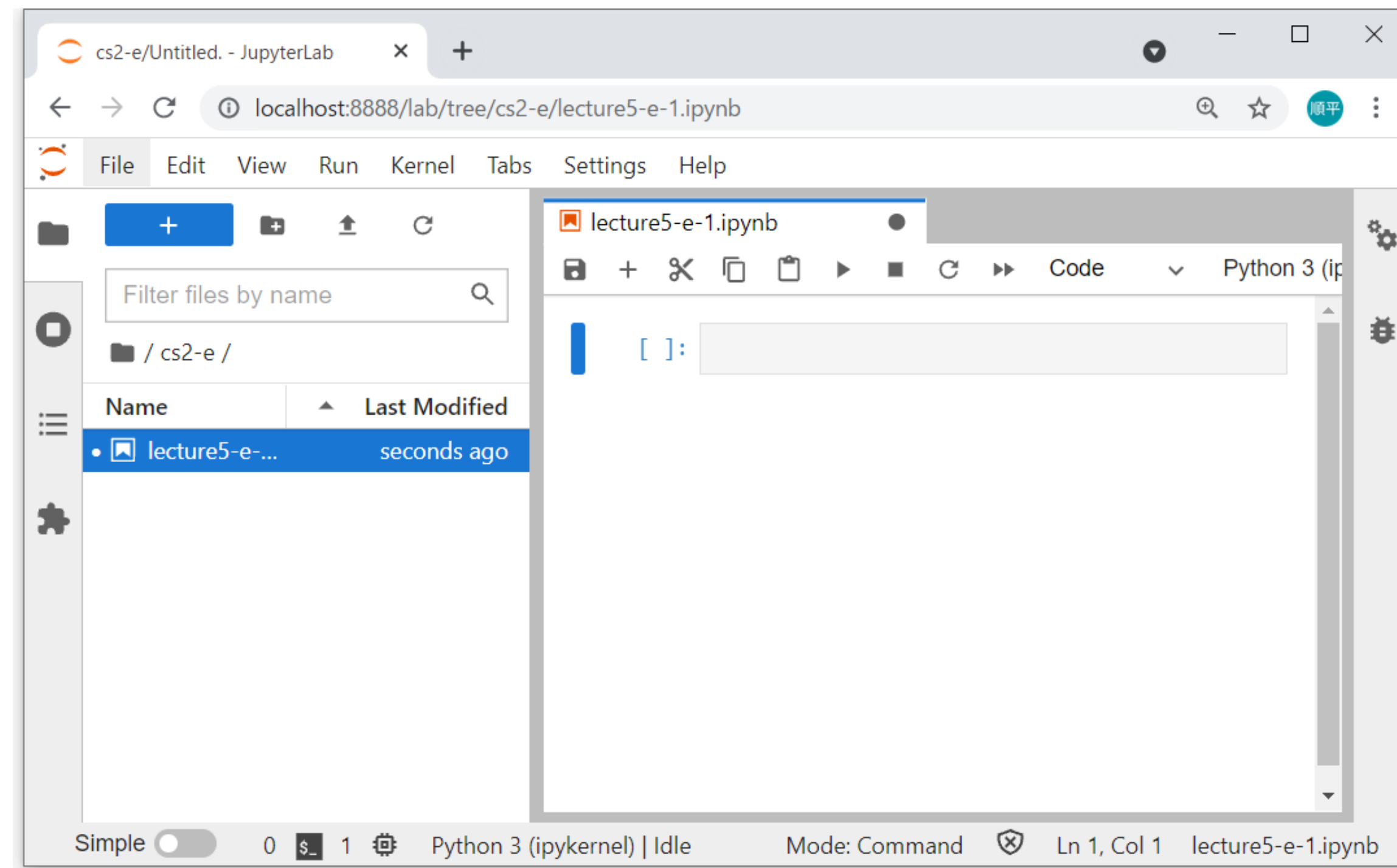


1. 再帰の考え方

関数の定義の中でその関数自身を呼び出すことを
「再帰(呼び出し)」といいます。
再帰はとても重要なので、これを機に使えるようになりましょう。

jupyter labを開いてください

- cs2-e フォルダの中に、**lecture7-e-1.ipynb** を作成してください。



- 以下、出てくるコードは問題ごとに各セルに打ち込んで実行してみてください。

例： $n!$ を計算したい

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1$$

$$(n-1)!$$

よって

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

もし $(n-1)!$ が計算できたとするならば

$(n-1)!$ に n をかけるだけで簡単に $n!$ が計算できる！（問題を分割する）

fact(n) 関数を作る(未完成)

※ 英語で階乗は factorial といいます。だから fact(n)。

```
# n!を計算する関数(未完成)
```

```
def fact(n):
```

```
    return n * fact(n - 1)
```



n-1 の階乗 fact(n-1) が計算できたとするならば

n の階乗 fact(n) が計算できる

$n \leq 1$ のときの処理

```
# n!を計算する関数(未完成)
def fact(n):

    return n * fact(n - 1)
```

$\text{fact}(1) \rightarrow 1 * \text{fact}(0) \rightarrow 1 * 0 * \text{fact}(-1) \rightarrow$

..
となってしまう、計算がいつまでたっても終わらない

➡ $n \leq 1$ のときは、 $\text{fact}(n) = 1$ とする

$n \leq 1$ のときの処理

$n!$ を計算する関数(完成品)

```
def fact(n):
```

```
    if  $n \leq 1$ :
```

```
        return 1
```

```
    return  $n * \text{fact}(n - 1)$ 
```

$n \leq 1$ のときの
処理を追加して完
成！

例題

階乗と同じ要領で、自分でプログラムを書いてみよう！

- **例題1:** $1+2+3+\cdots+n$ を計算する関数 $\text{sum}(n)$ を再帰を使って書いてみよう
- **例題2:** フィボナッチ数列を計算する関数 $\text{fib}(n)$ を再帰を使って書いてみよう
- **例題3:** クイックソートを計算する関数 $\text{qsort}(\text{lst})$ を再帰を使って書いてみよう

例題1: $1+2+3+\dots+n$ を計算する関数 $\text{sum}(n)$

- 問題を分割する

$$\begin{aligned}\text{sum}(n) &= 1 + 2 + 3 + \dots + (n-1) + n \\ &= \text{sum}(n-1) + n\end{aligned}$$

もし $\text{sum}(n-1)$ が計算できれば、 $\text{sum}(n)$ も計算できる！

例題1: $1+2+3+\dots+n$ を計算する関数 $\text{sum}(n)$

- 関数で表現する

```
# 1+2+3+...+n を計算する関数(未完成)  
def sum(n):
```

```
    return sum(n-1) + n
```

↑
 $\text{sum}(n-1)$ が計算できたとするならば
 $\text{sum}(n)$ も計算できる

例題1: $1+2+3+\dots+n$ を計算する関数 $\text{sum}(n)$

- 例外ケースを対処する

$1+2+3+\dots+n$ を計算する関数

```
def sum(n):
```

```
    if n <= 0:
```

```
        return 0
```

```
    return sum(n-1) + n
```

$n \leq 0$ のときは
 $\text{sum}(n) = 0$ と定義

完
成！

例題2: フィボナッチ数列を計算する関数 $\text{fib}(n)$

- 問題を分割する

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$$

もし $\text{fib}(n-1)$ と $\text{fib}(n-2)$ が計算できれば、 $\text{fib}(n)$ も計算できる！

例題2: フィボナッチ数列を計算する関数 $\text{fib}(n)$

- 関数で表現する

```
# フィボナッチ数列を計算する関数(未完成)
def fib(n):
```

```
    return fib(n-1) + fib(n-2)
```



$\text{fib}(n-1)$ と $\text{fib}(n-2)$ が計算できたとするならば
 $\text{fib}(n)$ も計算できる

例題2: フィボナッチ数列を計算する関数 fib(n)

- 例外ケースを対処する

フィボナッチ数列を計算する関数

```
def fib(n):
```

```
    if n <= 0:
```

```
        return 0
```

```
    if n == 1:
```

```
        return 1
```

```
    return fib(n-1) + fib(n-2)
```

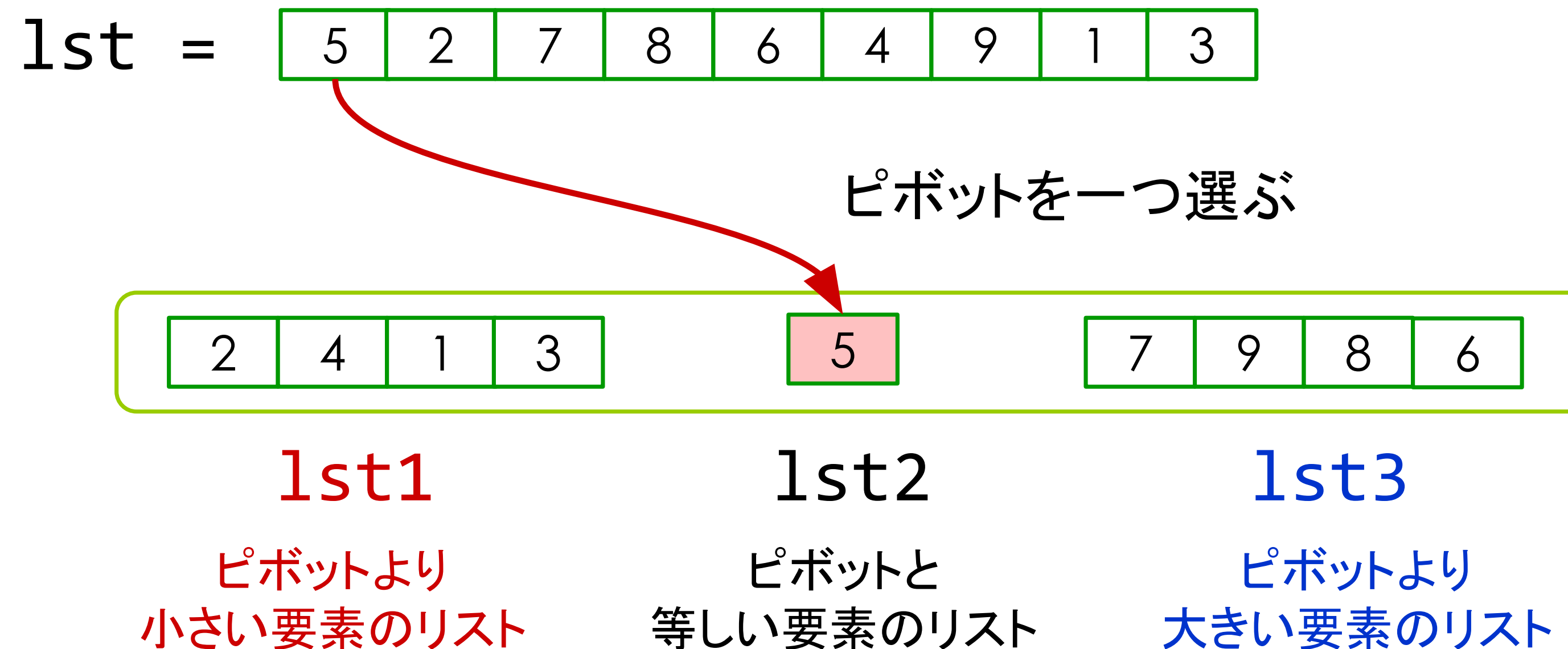
fib(0) = 0

fib(1) = 1

完
成！

例題3: クイックソート $qsort(lst)$

- 問題を分割する



もし $qsort(lst1)$ と $qsort(lst3)$ が計算できれば、 $qsort(lst)$ も計算できる！

例題3: クイックソート `qsort(lst)`

- 関数で表現する

```
# クイックソートを計算する関数(未完成)
```

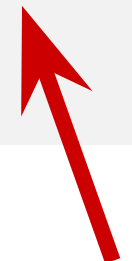
```
def qsort(lst):  
    pivot = lst[0]
```

```
    lst1 = [x for x in lst if x < pivot]
```

```
    lst2 = [x for x in lst if x == pivot]
```

```
    lst3 = [x for x in lst if x > pivot]
```

```
    return qsort(lst1) + lst2 + qsort(lst3)
```

 `qsort(lst1)` と  `qsort(lst3)` が計算できればOK

例題3: クイックソート `qsort(lst)`

- 例外ケースを対処する

クイックソートを計算する関数

```
def qsort(lst):
```

```
    if len(lst) <= 1:  
        return lst
```

要素数1個以下のリストは
分割できないのでそのまま
返す

```
    pivot = lst[0]
```

```
    lst1 = [x for x in lst if x < pivot]
```

```
    lst2 = [x for x in lst if x == pivot]
```

```
    lst3 = [x for x in lst if x > pivot]
```

```
    return qsort(lst1) + lst2 + qsort(lst3)
```

完
成！

再帰の考え方まとめ

問題を分割する($n-1$ のケースはできたものとする)という考え方が重要。

以下のように考えれば、再帰の問題は怖くない！

- 問題を分割する
- 関数で表現する
- 例外ケースを対処する

階乗もフィボナッチもクイックソートも全部同じ！