第4章 設計技法（2）

再帰と分割統治法

畠野 和裕
PAS社 車載システムズ事業部 SSBU6-2

目次

4.1 再帰とは

- 手続きの中で自分自身を呼び出すこと
  - 再帰呼び出しとも言う
- 再帰関数
  - 再帰呼び出しを行う関数のこと
- メリット
  - 簡潔にアルゴリズムを記述できる
- デメリット
  - 呼び出し回数が大きすぎるとスタックオーバーフローが起きる可能性がある

4.1 再帰とは

- code 4.1
  - 1からNまでの総和を計算する再帰関数
  - func() の中で、自分自身func() を呼び出している

```
1
2   def func(n):
3       if n==0:
4           return 0
5       else:
6           return n + func(n-1)
7
```

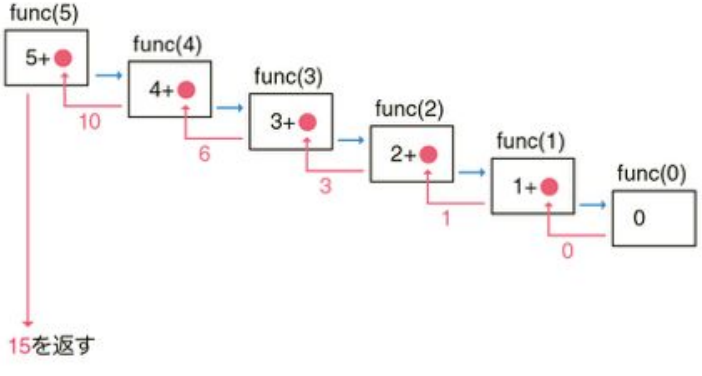


図 4.1　再帰関数の概念図

例としてn=5のとき、

func(5) = 5 + func(4)
= 5 + 4 + func(3)
= 5 + 4 + 3 + func(2)
…
= 5 + 4 + 3 + 2 + 1 + 0

4.2 再帰の例1：ユークリッドの互除法

- ある2つの整数m, nの最大公約数を求めるアルゴリズム
- 流れ
  a. mをnで割った時の余りをrとする
  b. r = 0であれば、この時のmが最大公約数であり、mを出力して終了
  c. r ≠ 0 であれば、m←n、n←rとして、a.に戻る
- code 4.4

```
8   def gcd(m, n):
9       if n == 0:
10          return m
11
12      return gcd(n, m%n)
13
14  print(gcd(51, 15)) # 3が出力される
15  print(gcd(15, 51)) # 3が出力される
```

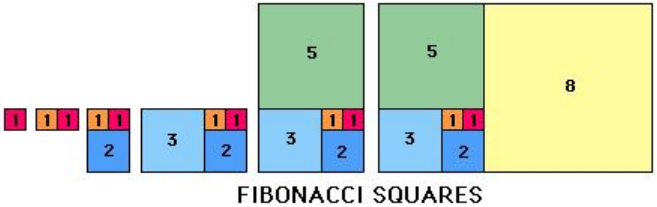
例として、(m,n) = (51, 15)のとき、

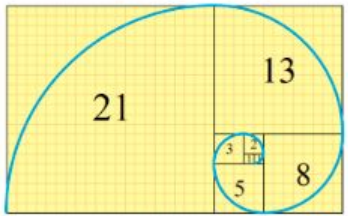
- r=51%15=6より、(m,n) = (15, 6)に更新
- r=15%6=3より、(m,n) = (6, 3)に更新
- r=6%3=0より、(m,n) = (3, 0)に更新
- r=0より、3が最大公約数

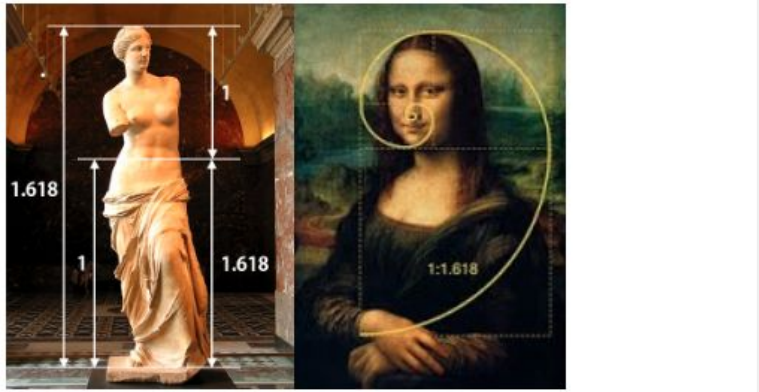
4.3 再帰の例2：フィボナッチ数列

- どの数字も前２つの数字を足した数字という規則の数列（漸化式）
  - 螺旋構造、黄金比など、自然界に多くみられる数列
  - 再帰関数内で再帰関数を複数回呼び出す例

$$F_0 = 0,$$
$$F_1 = 1,$$
$$F_{n+2} = F_n + F_{n+1} \ (n \geq 0)$$







4.3 再帰の例2：フィボナッチ数列

- code 4.5
  - フィボナッチ数列も再帰関数で表現すると簡潔にアルゴリズムを記述できる

```
17  def fibo(n):
18      if n == 0:
19          return 0
20
21      elif n == 1:
22          return 1
23
24      return fibo(n-1) + fibo(n-2)
25
26  print(fibo(6)) # 8が出力される
```

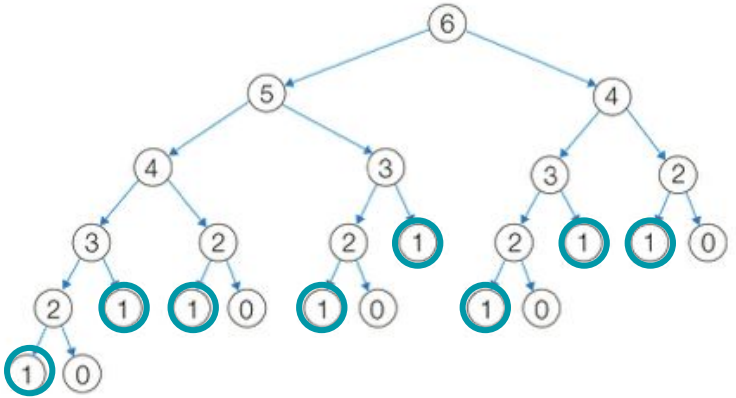


図 4.2　フィボナッチ数列を求める再帰呼び出し

4.4 メモ化して動的計画法へ

- 再帰関数の問題点
  - 同じ計算を何度も実行して効率が悪い

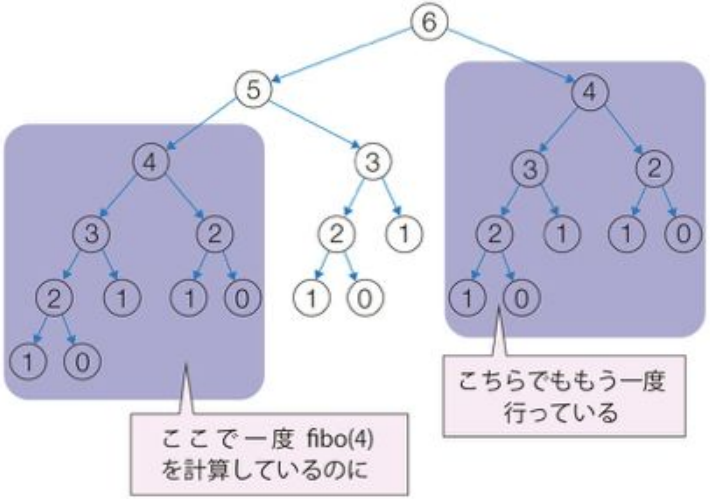


図 4.3　フィボナッチ数列を求める再帰関数における無駄

計算量：約 O(1.6^N)

指数関数的に計算量が増加してしまう

code 4.7

```
29  n = 6
30  f = [0]*(n+1)
31  f[0] = 0
32  f[1] = 1
33
34  for i in range(2, n+1):
35      f[i] = f[i-1] + f[i-2]
36
37  print(f[n]) # 8が出力される
```

実はfor文の反復でも求められる

計算量：O(N)

4.4 メモ化して動的計画法へ

- 再帰関数の無駄を省くためには？
  - 引数に対する答えをメモ化する
  - 具体的には、**memo[v]←fibo[v] の答えを格納する配列を用意する**

code 4.8

```
#include <iostream>
#include <vector>
using namespace std;

// fibo(N) の答えをメモ化する配列
vector<long long> memo;

long long fibo(int N) {
    // ベースケース
    if (N == 0) return 0;
    else if (N == 1) return 1;

    // メモをチェック（すでに計算済みならば答えをリターンする）
    if (memo[N] != -1) return memo[N];

    // 答えをメモ化しながら，再帰呼び出し
    return memo[N] = fibo(N - 1) + fibo(N - 2);
}
```

無駄な再計算がないので
高速化できる

```

int main() {
    // メモ化用配列を -1 で初期化する
    memo.assign(50, -1);

    // fibo(49) をよびだす
    fibo(49);

    // memo[0], ..., memo[49] に答えが格納されている
    for (int N = 2; N < 50; ++N) {
        cout << N << " 項目: " << memo[N] << endl;
    }
}
```

4.4 メモ化して動的計画法へ

- 再帰関数の無駄を省くためには？
  - 引数に対する答えをメモ化する
  - 具体的には、**memo[v]←fibo[v] の答えを格納する配列を用意する**

code 4.8 Python実装例

```python
class Fib_memo:
    def __init__(self):
        self.fib_memo = {}

    def cal_fib(self, n):
        if n == 0:
            self.fib_memo[n] = 0
            return 0

        elif n == 1:
            self.fib_memo[n] = 1
            return 1

        #既に計算した項なら保存した情報を使う
        if n in self.fib_memo.keys():
            return self.fib_memo[n]

        self.fib_memo[n] = self.cal_fib(n-1) + self.cal_fib(n-2)
        return self.fib_memo[n]
```

```python
n = 6
fib_memo = Fib_memo()
ans = fib_memo.cal_fib(n)
print(ans) # 8が出力される
```

参考元：https://qiita.com/chi-na/items/b903bd7cd7433e3a8a3f

4.5 再帰の例3：再起関数を用いる全探索

- 再帰関数を用いて部分和問題に対する全探索アルゴを設計
- 部分和問題
  - N 個の正の整数$a_0, a_1, \cdots, a_{N-1}$ と正の整数W が与えられます
  - $a_0, a_1, \cdots, a_{N-1}$ の中から何個かの整数を選んで総和をW とすることができるかを判定

表 3.1　部分集合を整数の二進法表現に対応付ける

部分集合	二進法での値	十進法での値
$\emptyset$	000	0
$\{a_0\}$	001	1
$\{a_1\}$	010	2
$\{a_0, a_1\}$	011	3
$\{a_2\}$	100	4
$\{a_0, a_2\}$	101	5
$\{a_1, a_2\}$	110	6
$\{a_0, a_1, a_2\}$	111	7

表 3.2　部分集合 $\{a_0, a_2, a_3, a_6\}$ に i 番目の要素 a_i が含まれるかどうかを判定する

i	$1 << i$	bit & $(1 << i)$
0	00000001	01001101 & 00000001 = 00000001 (true)
1	00000010	01001101 & 00000010 = 00000000 (false)
2	00000100	01001101 & 00000100 = 00000100 (true)
3	00001000	01001101 & 00001000 = 00001000 (true)
4	00010000	01001101 & 00010000 = 00000000 (false)
5	00100000	01001101 & 00100000 = 00000000 (false)
6	01000000	01001101 & 01000000 = 01000000 (true)
7	10000000	01001101 & 10000000 = 00000000 (false)

3.5節で全bit探索で設計したものを再帰関数で設計する

4.5 再帰の例3：再起関数を用いる全探索

- 考え方：2つの小問題に分けられる
  - $a_0, a_1, \cdots, a_{N-1}$ の中から、a_{N-1} を選ばないとき
  - $a_0, a_1, \cdots, a_{N-1}$ の中から、a_{N-1} を選ぶとき

- a_{N-1} を選ばないとき
  - $a_0, a_1, \cdots, a_{N-2}$ の中から何個かの整数を選んで総和を W とすることができるか
- a_{N-1} を選ぶとき
  - $a_0, a_1, \cdots, a_{N-2}$ の中から何個かの整数を選んで総和を $W-a_{N-1}$ とすることができるか

4.5 再帰の例3：再起関数を用いる全探索

- a_{N-1} を選ばないとき
  - $a_0, a_1, \cdots, a_{N-2}$ の中から何個かの整数を選んで総和を W とすることができるか
- a_{N-1} を選ぶとき
  - $a_0, a_1, \cdots, a_{N-2}$ の中から何個かの整数を選んで総和を $W-a_{N-1}$ とすることができるか

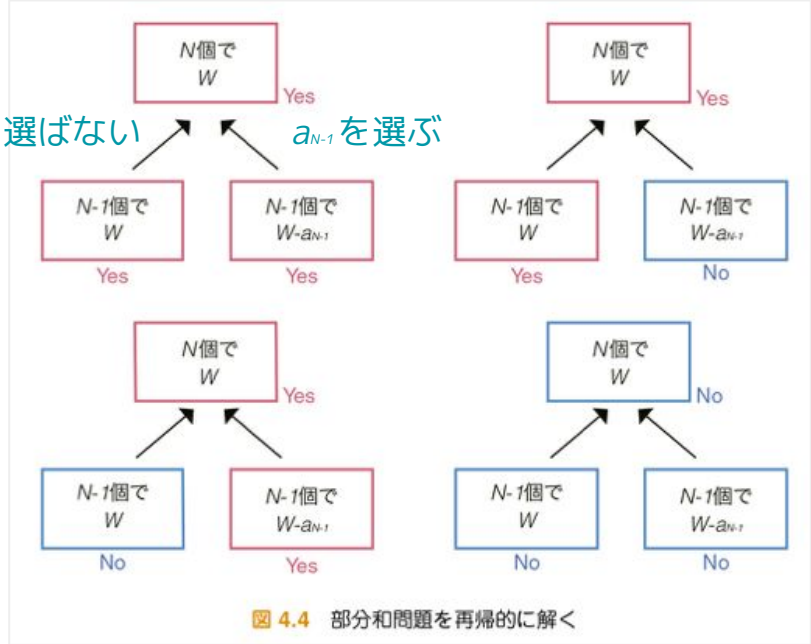


図 4.4　部分和問題を再帰的に解く

a_{N-1} を選ばない or a_{N-1} を選ぶ のうち、

少なくとも一方がYesとなれば、

元の問題もYesとなる

4.5 再帰の例3：再起関数を用いる全探索

- a_{N-1} を選ばないとき
  - $a_0, a_1, \cdots, a_{N-2}$ の中から何個かの整数を選んで総和を W とすることができるか
- a_{N-1} を選ぶとき
  - $a_0, a_1, \cdots, a_{N-2}$ の中から何個かの整数を選んで総和を $W-a_{N-1}$ とすることができるか

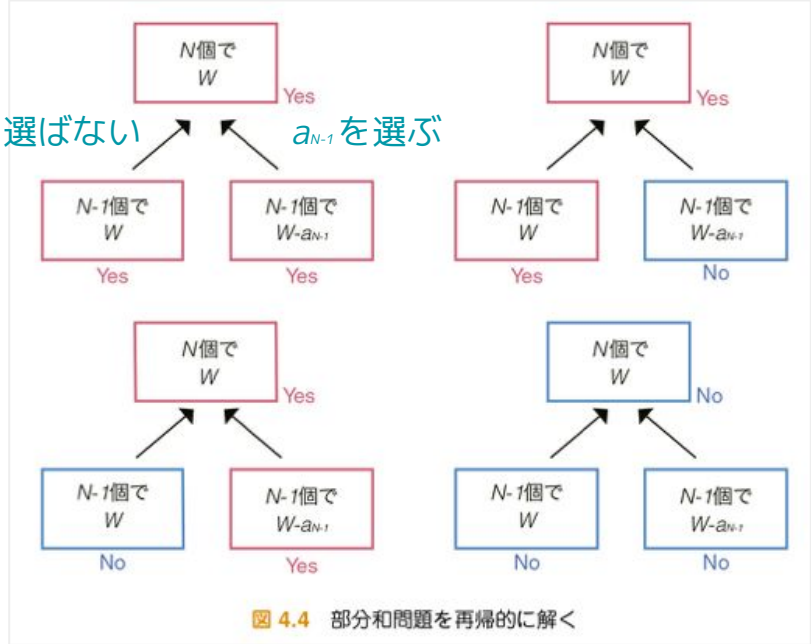


図 4.4　部分和問題を再帰的に解く

N個の整数についての問題

→N-1個の整数についての問題

→N-2個の整数についての問題

…

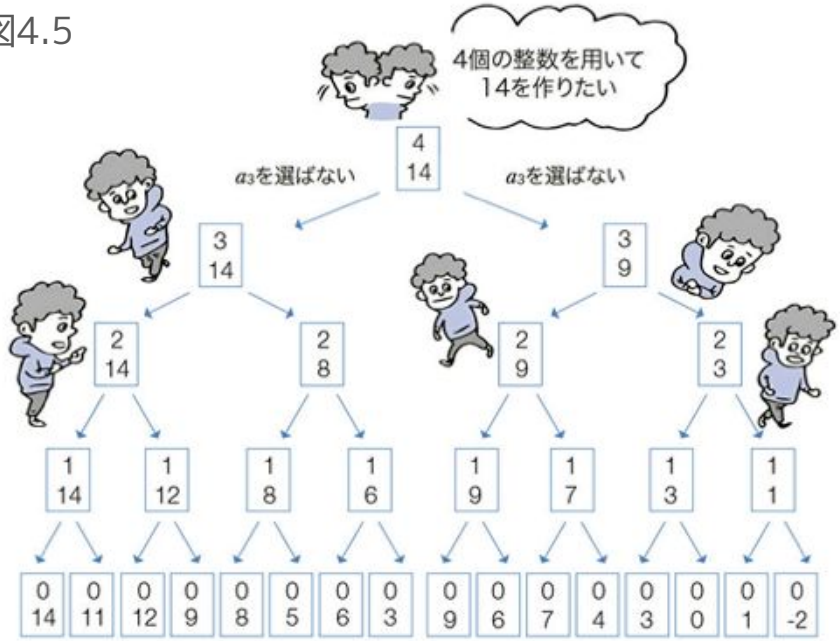
→0個の整数についての問題

つまり、再帰的に解ける

4.5 再帰の例3：再起関数を用いる全探索

- 部分和問題の例
  - n = 4, W=14
  - a = [3, 2, 6, 5]

図4.5



a = [3, 2, 6, 5]　a[3]を選ぶ？

a = [3, 2, 6]　a[2]を選ぶ？

a = [3, 2]　a[1]を選ぶ？

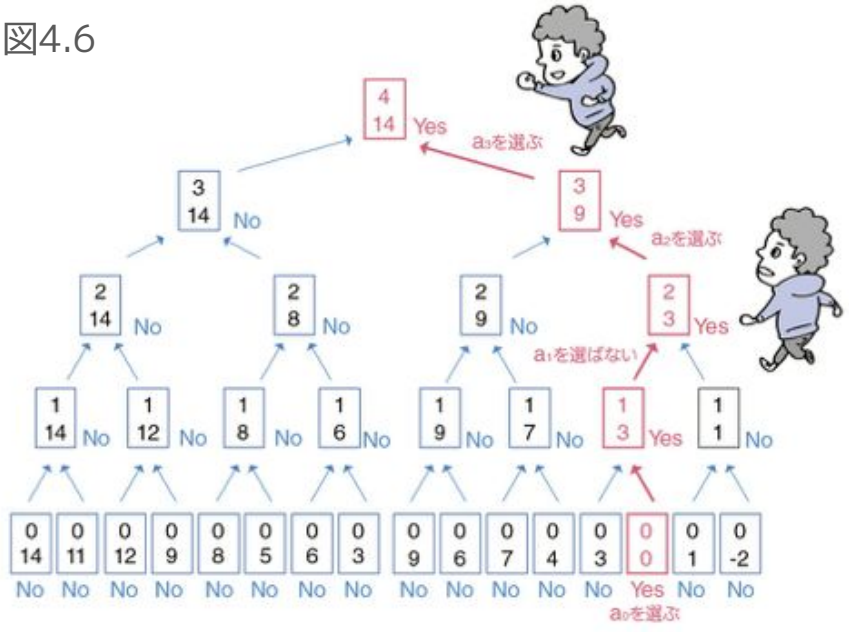
a = [3]　a[0]を選ぶ？

a = []　→sum(a)=0のはず

4.5 再帰の例3：再起関数を用いる全探索

- 部分和問題の例
  - n = 4, W=14
  - a = [3, 2, 6, 5]

図4.6



0個の整数の総和は常に0なので、

0個の整数についての問題で、

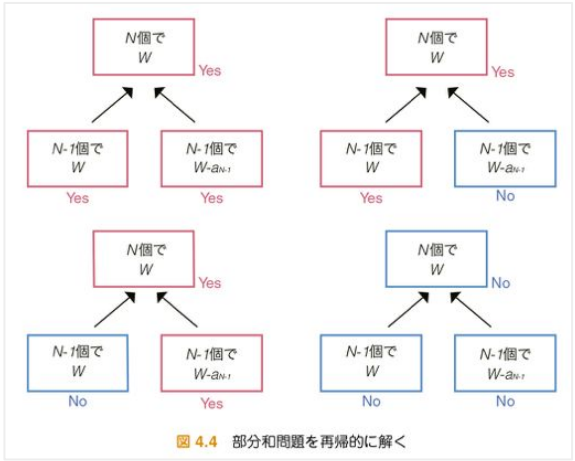
値が0の要素があれば、

元の問題の答えもYesになる

4.5 再帰の例3：再起関数を用いる全探索

- 部分和問題

code 4.9 Python実装例



func()で上記の処理を表現

```python
def func(i, w, a):
    # ベースケース
    if i==0:
        if w==0:
            return True
        else:
            return False

    # a[i-1]を選ばない場合
    if func(i-1, w, a):
        return True

    # a[i-1]を選ぶ場合
    if func(i-1, w-a[i-1], a):
        return True

    # どちらもFalseの場合
    return False
```

```python

n = 4
w = 14
a = [3, 2, 6, 5]

if func(n, w, a):
    print("Yes")
else:
    print("No")
```

Q：計算量はいくつ？

4.6 分割統治法

- 部分和問題
  - 整数N個の問題を整数N-1の小問題に、整数N-1個の問題を整数N-2の小問題に、・・・
  - という感じで、再帰的に繰り返した

- 分割統治法
  - 与えられた問題をいくつかの部分問題に分解し、各部分を再帰的に解いて、それらの解を組み合わせて元の問題の解を構成するアルゴリズム
  - 活用例：マージソート（12章）
    - 単純なソート：O(N^2)
    - マージソート：O(N lon N)

4.7 まとめ

- 再帰
  - 手続きの中で自分自身を呼び出すこと
  - アルゴリズムを簡潔に記述できる
  - リソース量を使いすぎたり、計算量が指数的に増える場合がある
  - 無駄な計算を省くためのメモ化
    - →動的計画法（5章）
  - 問題をより小さな問題に分割して解くという考え方
    - →分割統治法（12章）