Python勉強会@HACHINONE

第18章

動的計画法

お知らせ

Python勉強会@HACHINOHEでは、ジョン・V・グッターグ『Python言語による
プログラミングイントロダクション』近代科学社、2014年をみんなで勉強しています。

この本は自分で読んで考えて調べると力が付くように書かれています。

自分で読んで考えて調べる前に、このスライドを見るのは、いわばネタバレを
聞かされるようなものでもったいないです。

是非、本を読んでからご覧ください。

動的計画法

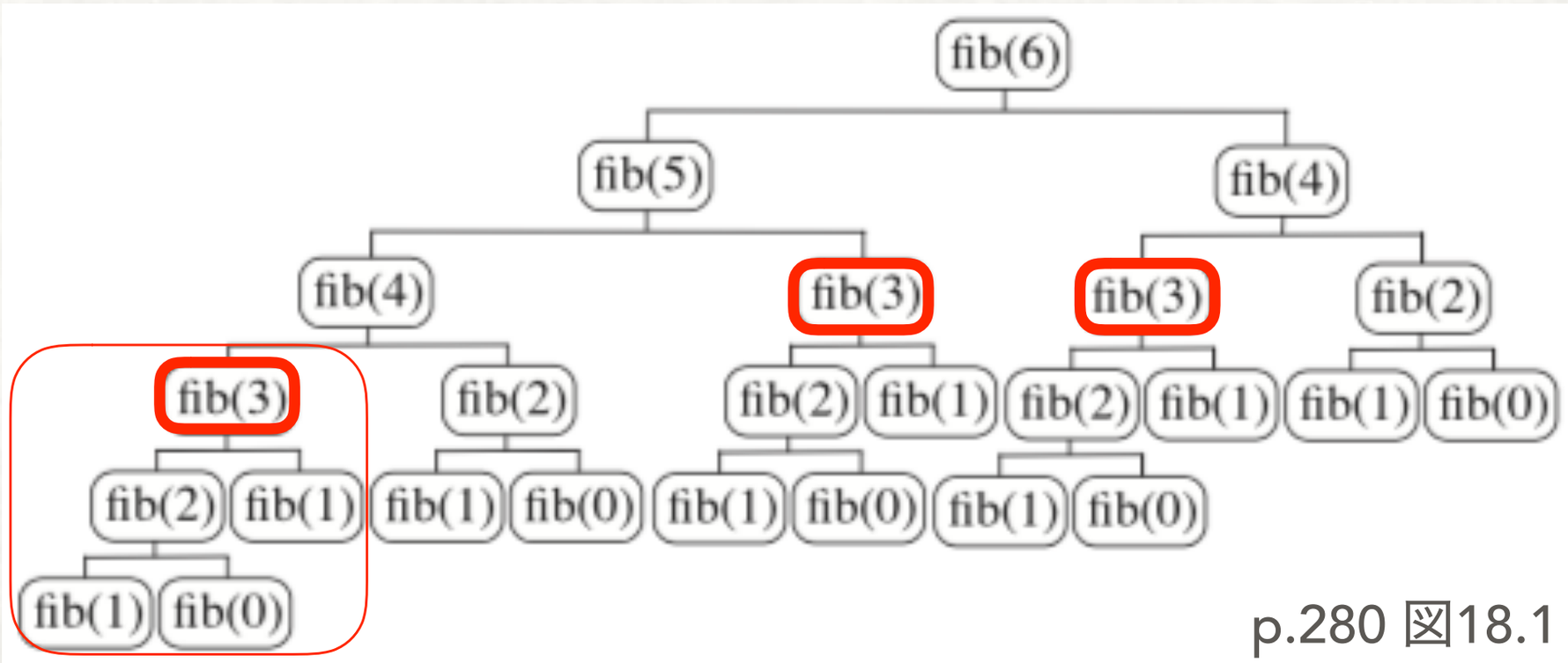
Python勉強会@HACHINOHE

- リチャード・E・ベルマンが命名
- 次の手法で計算を減らす方法の総称
  - 問題を部分問題に分割
  - 部分問題の計算結果を記録して使い回す

フィボナッチ数の例

Python勉強会@HACHINOHE

- フィボナッチ数の計算量はO(fib(n))で大変大きい ※p.280のfib(120)の例
- fib(6)の計算の例
  - 実にたくさんの回数、関数が再帰的に呼び出されている
  - 例えばfib(3)は3回呼びされ、それぞれ4回関数を呼び出す
- 一回計算した結果を保存しておくと計算量O(n)で高速に計算できる



p.280 図18.1

フィボナッチ数の動的計画法プログラム

Python勉強会@HACHINOHE

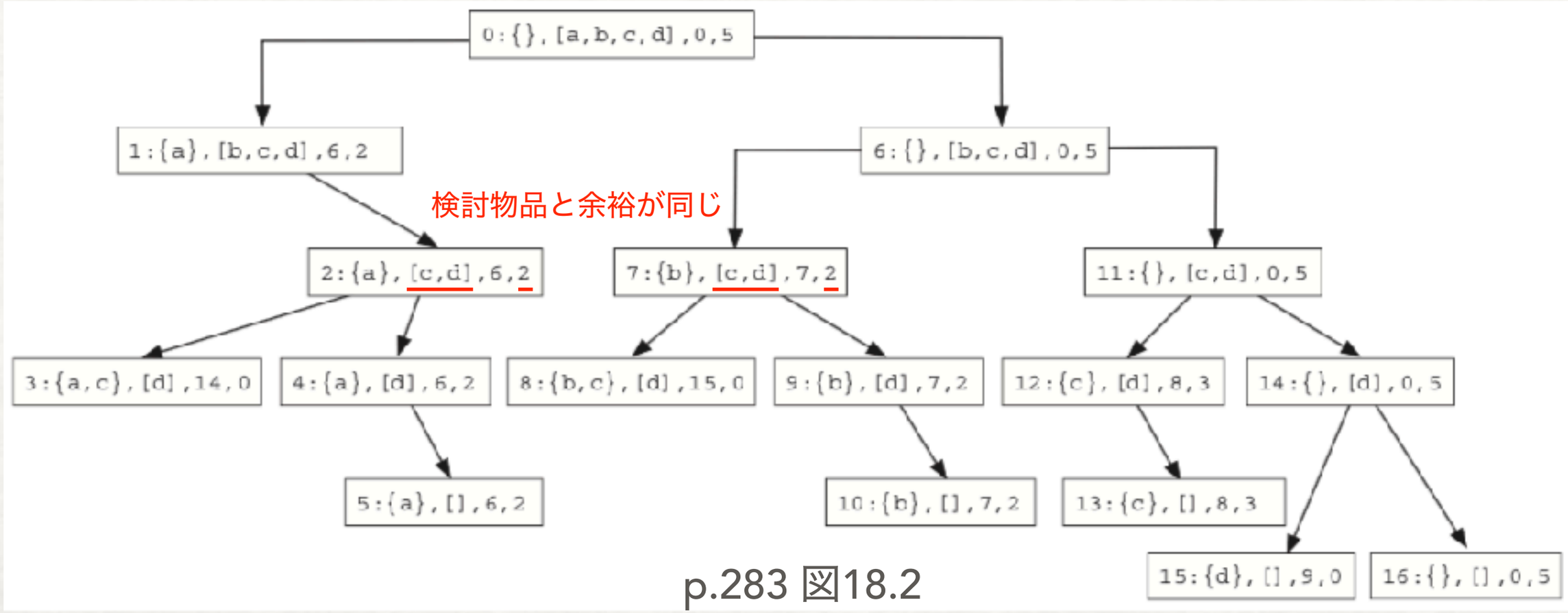
- 素直に実装したfib(120)は、関数の再帰呼び出しが1ナノ秒で、2.7億年くらい
- 対してfastFib(120)はおよそ1ミリ秒

```
def fastFib(n, memo = {}):
    """nを0以上の整数とする、memoは再帰呼び出しによってのみ使われる
    n番目のフィボナッチ数を返す"""
if n == 0 or n == 1:
    return 1
try:
    return memo[n] # 記録されたfib(n)を返す
except KeyError:
    result = fastFib(n-1, memo) + fastFib(n-2, memo)
    memo[n] = result # 計算結果を記録する
    return result
```

ナップサック問題を動的計画法で解く

Python勉強会@HACHINOHE

- ナップサックの容量 5
- a {価格:6, 重さ:3}、b {価格:7, 重さ:3}、c {価格:8, 重さ:2}、d {価格:9, 重さ:5}
- 記号の意味「ノード: {盗品}, [検討中], 盗品の価格, ナップサックの余裕」
- 左が検討中の先頭を盗む、右が検討中の先頭を除外する



p.283 図18.2

ナップサック問題の品物のクラス図

Python勉強会@HACHINOHE

- 前の章と同じ

品物
名前 価格 重さ
初期化(名前, 価格, 重さ) 名前を取得() 価格を取得() 重さを取得() 文字列化()

ナップサック問題の深さ優先木プログラム

Python勉強会@HACHINOHE

```
def maxVal(toConsider, avail):
    """toConsiderを検討中の品物のリスト、availをナップサックの余裕とする。
       それらをパラメータとする0/1ナップサック問題の解である
       総重量と品物のリストからなるタプルを返す"""
    if toConsider == [] or avail == 0:
        result = (0, ())
    elif toConsider[0].getWeight() > avail: # 検討中の先頭が余裕よりも重い
        # 右側の分岐のみを探索する
        result = maxVal(toConsider[1:], avail)
    else:
        nextItem = toConsider[0] # 検討中の先頭を盗むことにした
        # 左側の分岐を探索する
        withVal, withToTake = maxVal(toConsider[1:], avail - nextItem.getWeight())
        withVal += nextItem.getValue()
        # 右側の分岐を探索する
        withoutVal, withoutToTake = maxVal(toConsider[1:], avail)
        # いい方の分岐を選択
        if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))
        else:
            result = (withoutVal, withoutToTake)
    return result
```

ナップサック問題の動的計画法プログラム

Python勉強会@HACHINOHE

```
def fastMaxVal(toConsider, avail, memo = {}):
    “""toConsiderを検討中の品物のリスト、availをナップサックの余裕、
      memoは再帰呼び出しによってのみ使われるとする。
      それらをパラメータとする0/1ナップサック問題の解である
      総重量と品物のリストからなるタプルを返す"""
    if (len(toConsider), avail) in memo:
        result = memo[(len(toConsider), avail)]
    elif toConsider == [] or avail == 0:
        result = (0, ())
    elif toConsider[0].getWeight() > avail:
        # 右側の分岐のみを探索する
        result = fastMaxVal(toConsider[1:], avail, memo)
    else:
        nextItem = toConsider[0]
        # 左側の分岐を探索する
        withVal, withToTake = fastMaxVal(toConsider[1:], avail - nextItem.getWeight(), memo)
        withVal += nextItem.getValue()
        # 右側の分岐を探索する
        withoutVal, withoutToTake = fastMaxVal(toConsider[1:], avail, memo)
        # いい方の分岐を選択
        if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))
        else:
            result = (withoutVal, withoutToTake)
    memo[(len(toConsider), avail)] = result
    return result
```

ナップサック問題の計算時間の比較

Python勉強会@HACHINOHE