Algorithms (2020 Summer)
#6：動的計画法1

矢谷 浩司

動的計画法（dynamic programming）

（個人的には）花形アルゴリズムの1つ.

動的計画法の基本が身につくと，いろんな問題に取り組める（グラフ探索，パターンマッチング，文章間のdiff，など）.

言葉は難しそうだが，基本の考え方は簡単！（どう応用するかはむずいかも．．．）

動的計画法

Richard Bellmanさんによって考案．1954年にRAND研究所のテックノートとして発表．

THE THEORY OF DYNAMIC PROGRAMMING

Richard Bellman

§1. Introduction

Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time t is determined by a set of quantities which we call state parameters, or state variables. At certain times, which may be prescribed in advance, or which may be determined by the process itself, we are called upon to make decisions which will affect the state of the system. These decisions are equivalent to transformations of the state variables, the choice of a decision being identical with the choice of a transformation. The outcome of the preceding decisions is to be used to guide the choice of future ones, with the purpose of the whole process that of

https://www.rand.org/content/dam/rand/pubs/papers/2008/P550.pdf

なぜdynamic programmingという名前？

Dynamic：複数の段階に渡り，時間的に変化する問題.

Programming：「コーディング」という意味ではなく，
「最適な解を導出する方法」という意味.

“In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word “programming”. I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying I thought, lets kill two birds with one stone.”

-- Richard Bellman, Eye of the Hurricane: An Autobiography

THE THEORY OF DYNAMIC PROGRAMMING

Richard Bellman

§1. Introduction

Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time t is determined by a set of quantities which we call state parameters, or state variables. At certain times, which may be pre-

フィボナッチ数列

ある場所の要素の値はその2つ前の要素の和で定義される数列.

$$Fib(n) = Fib(n-1) + Fib(n-2)$$
$$Fib(1) = Fib(2) = 1$$

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

DPでは最も基本的な例として取り上げられる.

この数式通りに実装してみよう

実装は劇的に楽. 😄

```
def fib(n):
        if n<=2: return 1
        else:
                return fib(n-1) + fib(n-2)
```

実行してみると．．．

n	fib(n)計算にかかった おおよその時間 [msec]
30	284
31	
32	
33	
34	
35	
36	
37	
38	
39	
40	

実行してみると．．．

n	fib(n)計算にかかった おおよその時間 [msec]
30	284
31	461
32	786
33	
34	
35	
36	
37	
38	
39	
40	

実行してみると．．．

30から40に上げるだけで，
およそ計算時間100倍！

n	fib(n)計算にかかった おおよその時間 [msec]
30	284
31	461
32	786
33	1,279
34	2,044
35	3,236
36	5,238
37	8,237
38	13,227
39	22,027
40	35,057

実行してみると．．．

30から40に上げるだけで，
およそ計算時間100倍！

n	fib(n)計算にかかった おおよその時間 [msec]
30	284
31	461
32	786
33	1,279
34	2,044
35	3,236
36	5,238
37	8,237
38	13,227
39	22,027
40	35,057

計算量を確認

```
def fib(n):
    if n<=2: return 1
    else:
        return fib(n-1) + fib(n-2)
```

計算量を確認

```
def fib(n):
    if n<=2: return 1
    else:
        return fib(n-1) + fib(n-2)
```

n>1ならば，比較1回，
関数呼び出し2回，
足し算1回.

計算量を確認

```
def fib(n):
    if n<=2: return 1
    else:
        return fib(n-1) + fib(n-2)
```

n>1ならば，比較1回，
関数呼び出し2回，
足し算1回.

$T(n)$をn番目のフィボナッチ数にかかる計算時間とすると，

$$T(n) = T(n-1) + T(n-2) + O(1)$$

漸化式を解こう

$$T(n) = T(n-1) + T(n-2)$$

これの一般解は,

$$T(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

よって計算量は,

$$O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right) \approx O(1.6^n)$$

漸化式を解こう

$$T(n) = T(n-1) + T(n-2)$$

これの一般解は,

$$T(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

よって計算量は,

$$O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right) \approx O(1.6^n)$$

たしかに

n+1でだいたい1.6倍.

n	fib(n)計算にかかったおおよその時間 [msec]	n-1との比較[倍]
30	284	--
31	461	1.62
32	786	1.70
33	1,279	1.63
34	2,044	1.60
35	3,236	1.58
36	5,238	1.62
37	8,237	1.57
38	13,227	1.61
39	22,027	1.67
40	35,057	1.59

動的計画法

DPは以下の2つの条件を満たすようなアルゴリズムの総称.

- 小さい問題を解き，その結果を使ってより大きい問題を解く
- 小さい問題の計算結果を再利用する

<u>漸化式のような関係性</u>にどう着目するがポイントになる.

（累積和も似たような感じだが，小さい問題からより
大きい問題を解いているわけでない）

改良のポイント

非効率なところはどこか？

```
def fib(n):
        if n<=2: return 1
        else:
                return fib(n-1) + fib(n-2)
```

改良のポイント

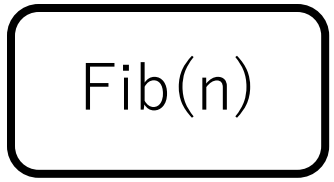
非効率なところはどこか？

```
def fib(n):
        if n<=2: return 1
        else:
                return fib(n-1) + fib(n-2)
```

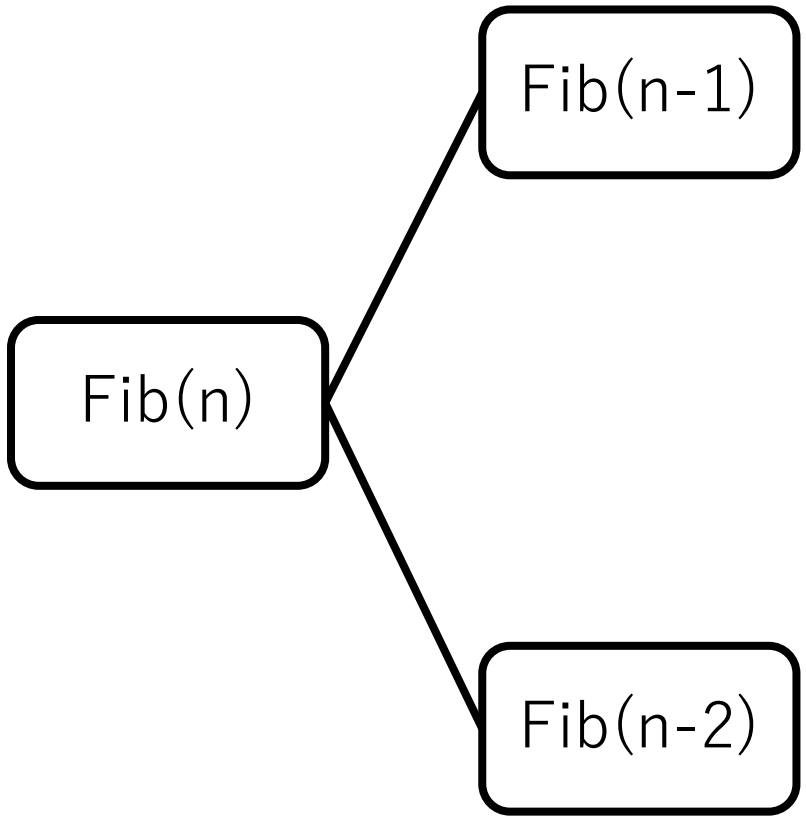
毎回呼び出す，毎回同じことをやっている系は多くの場合，非効率.

fib(n-2)はfib(n)とfib(n-1)の両方で必要になるが，お互いのやりとりがないので，同じ計算を繰り返す無駄がある.

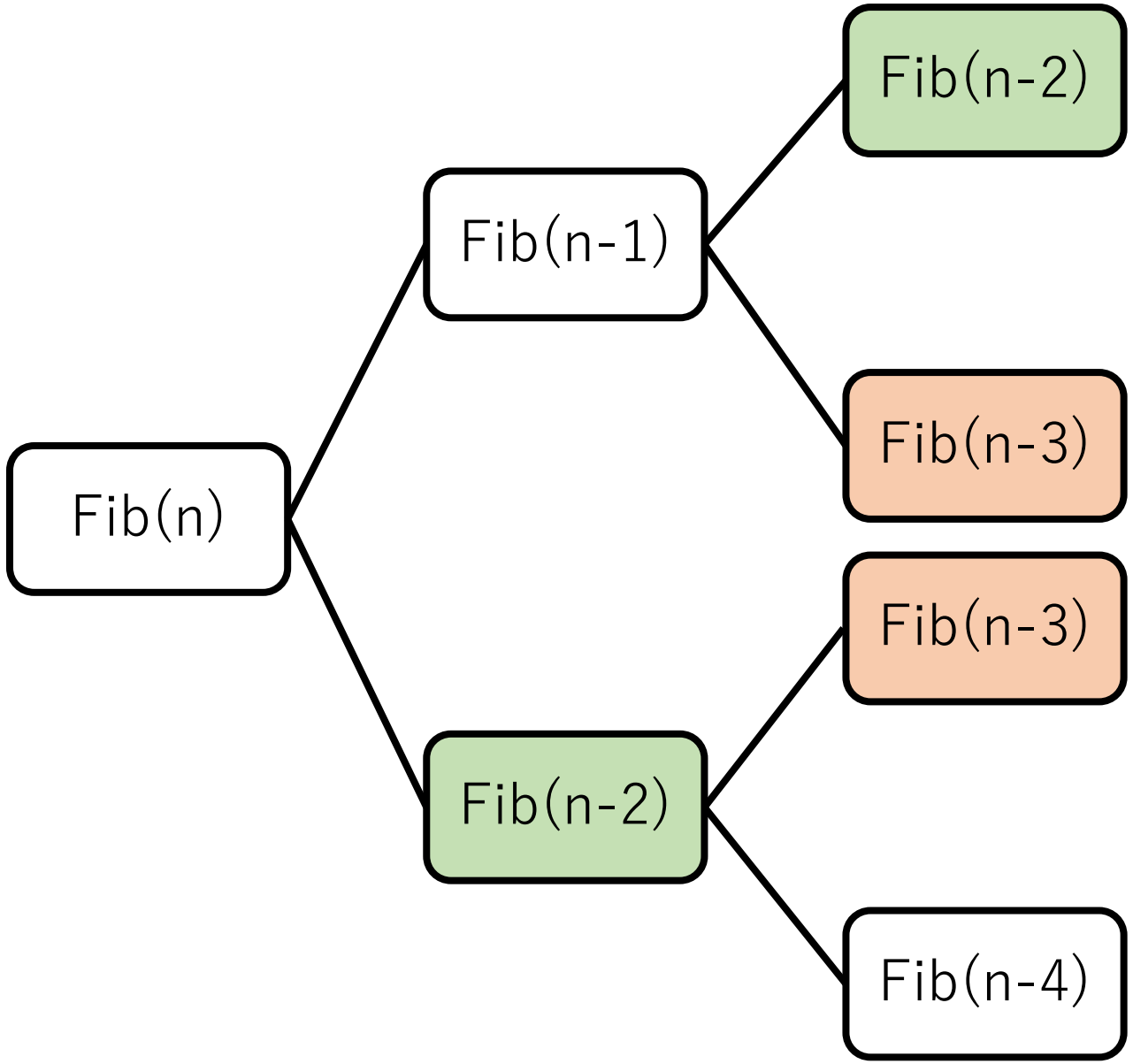
改良のポイント



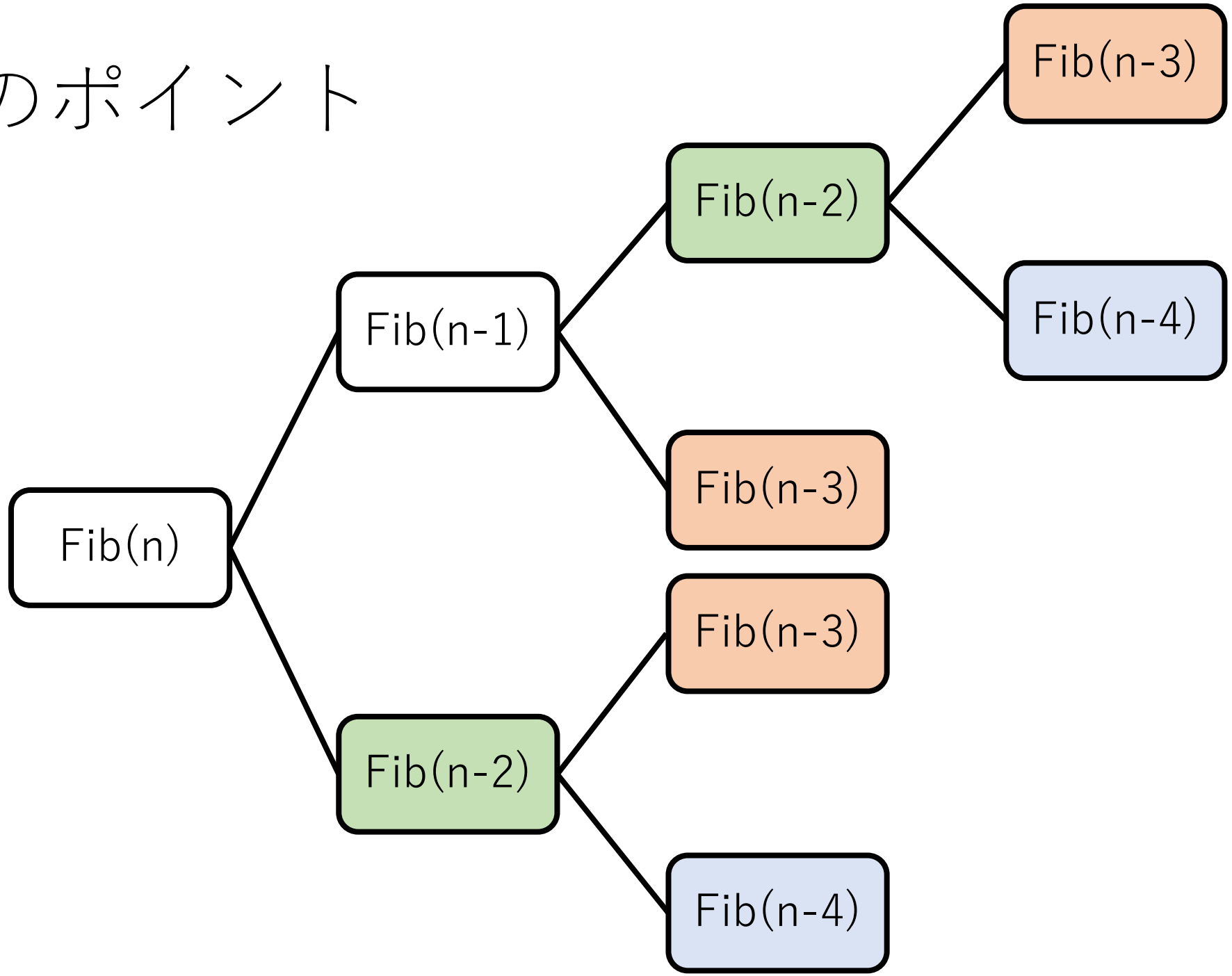
改良のポイント



改良のポイント



改良のポイント



改良のポイント

非効率なところはどこか？

```
def fib(n):
        if n<=2: return 1
        else:
                return fib(n-1) + fib(n-2)
```

一度は計算しないといけないが，計算したことあるものは再利用したい．

計算したら，記憶しておく！

改良の方針1

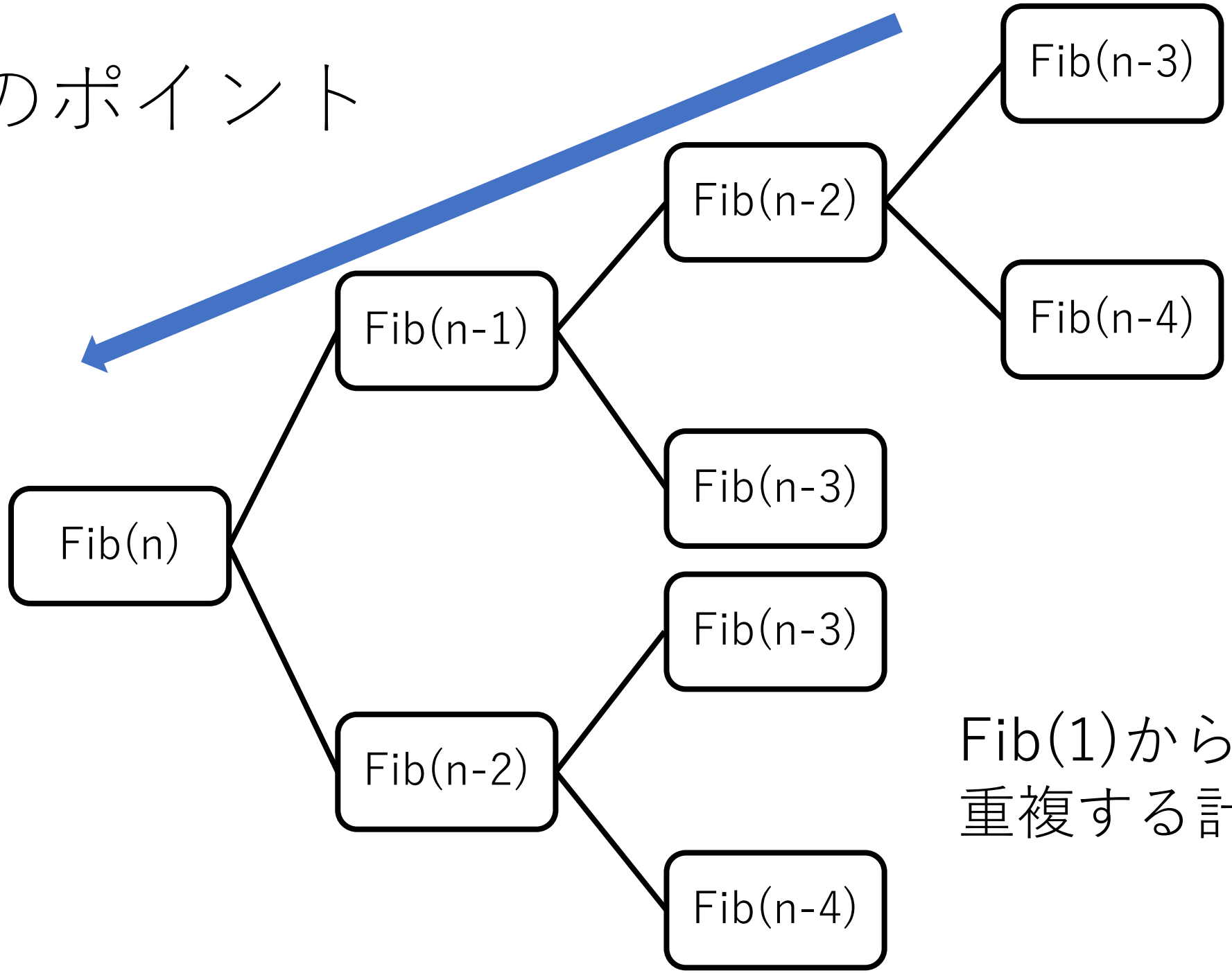
```
def fib_rev1(n):
        f = [0]*(n+1)      # 記憶用の配列
        def _fib(n):
                if n <= 2: return 1
                elif f[n] != 0: return f[n] # 記憶があればそれを使う
                else:  # なければ計算する
                        f[n] = _fib(n-1) + _fib(n-2)
                        return f[n]

        return _fib(n)
```

改良の方針1

この方法のポイントは，わかっている（計算したことがある）ものを保持しておき，再計算を減らすところ．

では，わかっているものから順に計算していってもよいのでは？

改良のポイント



Fib(1)から辿れば
重複する計算がない

フィボナッチ数列

$Fib(n)$を求めるためには，$Fib(n-1)$と$Fib(n-2)$がわかっていればよい．

フィボナッチ数列

$Fib(n)$を求めるためには，$Fib(n-1)$と$Fib(n-2)$がわかっていればよい．

$Fib(1), Fib(2)$は明らか（どちらも1）．

よって，$Fib(3)$ は，$Fib(2) + Fib(1)$で求まる．

さらに，$Fib(4)$ は，$Fib(3) + Fib(2)$で求まる．

フィボナッチ数列

$Fib(n)$を求めるためには，$Fib(n-1)$と$Fib(n-2)$がわかっていればよい．

$Fib(1), Fib(2)$は明らか（どちらも1）．
よって，$Fib(3)$ は，$Fib(2) + Fib(1)$で求まる．
さらに，$Fib(4)$ は，$Fib(3) + Fib(2)$で求まる．

これをどんどん辿っていけば，$Fib(n)$にたどり着く！

改良の方針2

```
# この方針の場合，配列である必要はない
def fib_rev2(n):
        if n<=2: return 1
        else:
                f = [0]*n
                f[0] = 0
                f[1] = 1
                for i in range(2, n):
                        f[i] = f[i-1] + f[i-2]
        return f[n-1]
```

n	fib(n) [msec]	fib_rev1(n) [msec]	fib_rev2(n) [msec]
30	284	0.0026	0.0004
31	461	0.0017	0.0004
32	786	0.0017	0.0005
33	1,279	0.0018	0.0004
34	2,044	0.0018	0.0004
35	3,236	0.0019	0.0004
36	5,238	0.0019	0.0004
37	8,237	0.0023	0.0004
38	13,227	0.0021	0.0005
39	22,027	0.0020	0.0004
40	35,057	0.0022	0.0004

改良の方針

メモ化再帰（改良1）

再帰をするが計算結果を記録しておき，次回以降はそれを利用して再計算を避ける．

漸化式方式（改良2）

わかっている値から計算をスタートし，漸化式の形で順に計算していくことで，再帰自体を避ける．
（狭義のDPとしてはこちらを意味する．）

改良版の計算量

メモ化再帰（改良1）

n番目のフィボナッチ数を計算するのは1回限りで,
この計算は定数回. あとは配列から呼び出されるのみ.

よって，$O(n)$. （ただし再帰呼び出しのオーバーヘッドあり）

漸化式方式（改良2）

1回のループは定数回の足し算と代入.

よって，$O(n)$.

最大値問題

「配列に与えられた整数n個のうち，任意の個数取り出して和を計算する．このときの和の最大値を求めよ．」

例）[4, 5, 1, 2, 3, -10]ならば，[4, 5, 1, 2, 3]を取り出して15.

（単に正の値を取り出して足し合わせる，でも解けますがDPの簡単な練習問題としても考えることができます．）

最大値問題のナイーブな解法

全組み合わせを計算.

各要素を入れるか入れないかは2通り．よって，全組み合わせ数は2^n.

よって，単純な方法では計算量は$O(2^n)$.

動的計画法（再掲）

DPは以下の2つの条件を満たすようなアルゴリズムの総称.

小さい問題を解き，その結果を使ってより大きい問題を解く
小さい問題の計算結果を再利用する

漸化式のような関係性にどう着目するがポイントになる.

（累積和も似たような感じだが，小さい問題からより
大きい問題を解いているわけでない）

漸化式的な関係性

$a[k]$：与えられた配列のk+1番目の要素.

S_k：1番目からk番目までの要素の最大の和．ただし，S_0は計算が全く始まっていない状態での最大の和とする.

このときに，S_{k+1}：1番目からk+1番目までの最大の和,がどうなるかを考えよう.

漸化式的な関係性

$a[k]$：与えられた配列のk+1番目の要素.
S_k：1番目からk番目までの要素の最大の和.

S_{k+1}が取りうる値の可能性は2つ.

S_kに$a[k]$を足したもの.

S_kに$a[k]$を足さなかったもの.

この2つのうちどちらか大きいほうが，今までの最大の和となる.

漸化式的な関係性

$a[k]$：与えられた配列のk+1番目の要素.
S_k：1番目からk番目までの要素の最大の和.

S_kとS_{k+1}の漸化式的関係性を考えると,

$$S_{k+1} = \max(S_k, S_k + a[k])$$

さらに, $S_0 = 0$（何も足し合わせるものがないので最大の和も当然0）.

最大和問題の解答コード例（漸化式方式）

```
def max_sum(a):
    s = [0]*(len(a)+1)
    for i in range(len(a)):
        s[i+1] = max(s[i], s[i]+a[i])

    return s[len(a)]
```

（sは配列である必要はないですが，ここでは漸化式的関係を明確にするために意図的に使っています．）

カエルとび問題

実行時間制限: 2 sec / メモリ制限: 1024 MB

配点 : 100 点

問題文

N 個の足場があります。 足場には $1, 2, \ldots, N$ と番号が振られています。 各 $i\,(1 \leq i \leq N)$ について、足場 i の高さは h_i です。

最初、足場 1 にカエルがいます。 カエルは次の行動を何回か繰り返し、足場 N まで辿り着こうとしています。

- 足場 i にいるとき、足場 $i+1$ または $i+2$ へジャンプする。 このとき、ジャンプ先の足場を j とすると、コスト $|h_i - h_j|$ を支払う。

カエルが足場 N に辿り着くまでに支払うコストの総和の最小値を求めてください。

制約

- 入力はすべて整数である。
- $2 \leq N \leq 10^5$
- $1 \leq h_i \leq 10^4$

https://atcoder.jp/contests/dp/tasks/dp_a

カエルとび問題

入力例 3 Copy

```
6
30 10 60 10 60 50
```

出力例 3 Copy

```
40
```

足場 $1 \rightarrow 3 \rightarrow 5 \rightarrow 6$ と移動すると、コストの総和は $|30 - 60| + |60 - 60| + |60 - 50| = 40$ となります。

https://atcoder.jp/contests/dp/tasks/dp_a

動的計画法（再掲）

DPは以下の2つの条件を満たすようなアルゴリズムの総称.

小さい問題を解き，その結果を使ってより大きい問題を解く
小さい問題の計算結果を再利用する

漸化式のような関係性にどう着目するがポイントになる.

（累積和も似たような感じだが，小さい問題からより
大きい問題を解いているわけでない）

漸化式的な関係性

$c(i,j)$： i番目の足場からj番目の足場に行くときのコスト.
S_i：i番目の足場にたどり着くまでのコストの総和の最小値.

i番目の足場に至るケースは2通り.

i-1番目の足場から1つジャンプして，　i番目に来た.

i-2番目の足場から2つジャンプして，　i番目に来た.

漸化式的な関係性

$c(i,j)$： i番目の足場からj番目の足場に行くときのコスト.
S_i：i番目の足場にたどり着くまでのコストの総和の最小値.

i-1番目の足場から1つジャンプ：

i-2番目の足場から2つジャンプ：

漸化式的な関係性

$c(i,j)$： i番目の足場からj番目の足場に行くときのコスト.
S_i：i番目の足場にたどり着くまでのコストの総和の最小値.

i-1番目の足場から1つジャンプ：$S_{i-1} + c(i-1, i)$

i-2番目の足場から2つジャンプ：

漸化式的な関係性

$c(i,j)$： i番目の足場からj番目の足場に行くときのコスト.
S_i：i番目の足場に行くまでのコストの総和の最小値.

i-1番目の足場から1つジャンプ：$S_{i-1}+c(i-1,i)$

i-2番目の足場から2つジャンプ：$S_{i-2}+c(i-2,i)$

このうちより小さい方がS_iになる.

漸化式的な関係性

よって,

$$S_i = \min(S_{i-1} + c(i-1, i), S_{i-2} + c(i-2, i))$$

という関係性をコードに落とせば良い！（ただし，S_1とS_2だけは個別対応が必要）.

いよいよDPらしい問題へ：

ナップサック問題

ナップサック問題

「n個の品物があり，各々その重さとその価値がw_i, v_iで表される．このとき重さの総和の制限Wを超えないように品物を選んだとき，価値の総和の最大値を求めよ．」

例）

[重さ，価値]: [11, 15], [2, 3], [3, 1], [4, 4], [1, 2], [5, 8]

W=15

ナップサック問題

「n個の品物があり，各々その重さとその価値がw_i, v_iで表される．このとき重さの総和の制限Wを超えないように品物を選んだとき，価値の総和の最大値を求めよ．」

例）

[重さ，価値]: [11, 15], [2, 3], [3, 1], [4, 4], [1, 2], [5, 8]

W=15

[11, 15], [2, 3], [1, 2]を選んで，20.

ナップサック問題の解法？

コスパ（v_i/w_i）の高い順に入れていって，Wを超える手前でストップする．

例）

[重さ，価値]: [11, 15], [2, 3], [3, 1], [4, 4], [1, 2], [5, 8]

W=15

[1, 2] -> [5, 8] -> [2, 3] -> [11, 15] -> [4, 4] -> [3, 1]

ナップサック問題の解法？

コスパ（v_i/w_i）の高い順に入れていって，Wを超える手前でストップする．

例）

[重さ，価値]: [11, 15], [2, 3], [3, 1], [4, 4], [1, 2], [5, 8]

W=15

[1, 2] -> [5, 8] -> [2, 3] -> [11, 15] -> [4, 4] -> [3, 1]

価値の総和は18．．．

ナップサック問題の近似的解法

「コスパ（v_i/w_i）の高い順に入れていって，Wを超える手前でストップする．」

近似アルゴリズムとして知られている．

全ての品物に対してコスパを計算し，ソートをした後，比較と足し算を順次行う．

ソートが一番重いが，$O(n \log n)$程度で実行可能．

DPに向けた指針

漸化式的な関係性を探そう.

knapsack(i, w)を総重量の上限がwである条件下で，i番目の品物まで考慮したときの価値の和の最大値を返すものとする.

i番目の品物（a[i]）を考慮する直前までの状態，すわなち，knapsack(i-1, w)との関係性を考えよう.

DPに向けた指針

i番目の品物（a[i]）に起こりうるケースは，以下の3つ．

a[i]を入れるとwを超える．

wは超えないが，a[i]を入れない方がよい．

wは超えず，a[i]を入れた方がよい．

これによって，knapsack(i, w)が求まる．

単なる再帰の擬似コード

```
w_limit= 15
weight = [11, 2, 3, 4, 1, 5]
value = [15, 3, 1, 4, 2, 8]

def knapsack(i, w):
        if  i<0: return 0
```

単なる再帰の擬似コード

```
def knapsack(i, w):
        if  i<0: return 0
        elif [a[i]を入れるとwを超える]:
                return knapsack(i-1, w)      # a[i]は入れられない
```

単なる再帰の擬似コード

```
def knapsack(i, w):
    if  i<0: return 0
    elif [a[i]を入れるとwを超える]:
        return knapsack(i-1, w)     # a[i]は入れられない
    else: #not_inは入れない，is_inは入れる場合
        not_in = knapsack(i-1, w)
        is_in = knapsack(i-1, w-[a[i]の重さ])+ a[i]の価値
        return max(not_in, is_in)
```

単なる再帰の擬似コード

このやり方では，フィボナッチ数列の時と同様に
同じ計算がなんども行われてしまう．

メモ化再帰をおこなってみよう！

メモ化再帰

メモとして2次元の配列を定義．note[i][w]

note[i][w]は，重さの上限がwであるとき，i番目の品物までを考慮した時点での価値の最大の和を記録する．

（とすると，配列の大きさは？）

メモ化再帰を使ったナップサック

```
def knapsack_rev1(i, w):
    if [note[i][w]があるなら]: return note[i][w]
    if i<0: return 0
```

メモ化再帰を使ったナップサック

```
def knapsack_rev1(i, w):
    if [note[i][w]があるなら]: return note[i][w]
    if i<0: return 0
    elif [[a[i]を入れるとwを超える]:
        note[i][w] = knapsack_rev1(i-1, w)
        return note[i][w]
```

メモ化再帰を使ったナップサック

```
def knapsack(i, w):
    …
    else: #not_inは入れない，is_inは入れる場合
        not_in = knapsack_rev1(i-1, w)
        is_in = knapsack_rev1(i-1, w-[a[i]の重さ])
                                + a[i]の価値
        note[i][w] = max(not_in, is_in)
        return note[i][w]
```

メモ化再帰を使ったナップサック

```
print(knapsack_rev1(5, 15))
-----
20
```

メモ化再帰を使ったナップサック

あるiとwに対して，knapsack_rev1(i, w)の計算は1回になる．

よって，$O(NW)$．Nは品物の総数．

ナイーブな解法だと$O(2^N)$なので，だいぶマシ．

メモ化再帰を使ったナップサック

```
def knapsack_rev1(i, w):
    if [note[i][w]があるなら]: return note[i][w]
    if i<0: return 0
    elif [a[i]の重さでwを超える]:
        [knapsack _rev1(i-1, w)をnote[i][w]に入れてreturn]
    else: #not_inは入れない，is_inは入れる場合
        not_in = knapsack _rev1(i-1, w)
        is_in = knapsack _rev1(i-1, w-[a[i]の重さ])+ a[i]の価値
        [2つの内，大きい方をnote[i][w]に入れてreturn]
```

再帰部分に注目

```
def knapsack_rev1(i, w):
    if [note[i][w]があるなら]: return note[i][w]
    if i<0: return 0
    elif [a[i]の重さでwを超える]:
        [knapsack _rev1(i-1, w)をnote[i][w]に入れてreturn]
    else: #not_inは入れない，is_inは入れる場合
        not_in = knapsack _rev1(i-1, w)
        is_in = knapsack _rev1(i-1, w-[a[i]の重さ])+ a[i]の価値
        [2つの内，大きい方をnote[i][w]に入れてreturn]
```

再帰部分に着目

knapsack(i-1, w)は，note[i-1][w]
knapsack(i-1, w-[a[i]の重さ])は，note[i-1][w-[a[i]の重さ]]
にそれぞれ対応する．

再帰部分に着目

knapsack(i-1, w)は，note[i-1][w]
knapsack(i-1, w-[a[i]の重さ])は，note[i-1][w-[a[i]の重さ]]
にそれぞれ対応する．

よって，これらの値を単に引っ張ってくれば良い！

再帰部分に着目

knapsack(i-1, w)は，note[i-1][w]
knapsack(i-1, w-[a[i]の重さ])は，note[i-1][w-[a[i]の重さ]]
にそれぞれ対応する．

よって，これらの値を単に引っ張ってくれば良い！

ただし，先にnote[i-1][w]の値を知っておかないといけない．
これをどんどん辿っていくと最終的には「初期状態」に
たどり着く．

初期状態は？

そもそも何も始まっていない状態を考える.

品物はなにもないので，どんな状況でも（総重量の制約によらず）価値の総和は0.

初期状態は？

そもそも何も始まっていない状態を考える.

品物はなにもないので，どんな状況でも（総重量の制約によらず）価値の総和は0.

この状況をnote[0][w]に入れておくことにする．つまり，メモ化再帰と比較して，noteの行の大きさが1つ増える．

初期状態は？

そもそも何も始まっていない状態を考える．

品物はなにもないので，どんな状況でも（総重量の制約によらず）価値の総和は0．

この状況をnote[0][w]に入れておくことにする．つまり，メモ化再帰と比較して，noteの行の大きさが1つ増える．

note[i+1][w] は，総重量の制約がwのときに，i番目の品物に対する判断をした後の，価値の和の最大が入る．

さらに

note[i][w]を計算するときに，どんなnote[i-1][w’]を使うのかを予測して必要なものだけ計算しておくのは，処理としては手間.

さらに

note[i][w]を計算するときに，どんなnote[i-1][w’]を使うのかを予測して必要なものだけ計算しておくのは，処理としては手間.

よって，何も考えずnote[i-1]の行を全部埋めておこう！

さらに

note[i][w]を計算するときに，どんなnote[i-1][w']を使うのかを予測して必要なものだけ計算しておくのは，処理としては手間.

よって，何も考えずnote[i-1]の行を全部埋めておこう！

したがって，重さの上限が0から最大の上限の場合まで予め全部計算してしまう.

knapsack()の漸化式方式の実装例

```
w_limit = 15
weight = [11, 2, 3, 4, 1, 5]
value = [15, 3, 1, 4, 2, 8]
note = [[-1 for _ in range(w_limit+1)]
            for _ in range(len(value)+1)]
```

knapsack()の漸化式方式の実装例

```
def knapsack_rev2():
        [note[0]の行を0で初期化]
```

knapsack()の漸化式方式の実装例

```
def knapsack_rev2():
        [note[0]の行を0で初期化]

        for [i: 0から品物の総数]:
                for [w: 0から総重量の上限]:
```

knapsack()の漸化式方式の実装例

```
def knapsack_rev2():
    …
    for [i: 0から品物の総数]:
        for [w: 0から総重量の上限]:
            if [品物iを入れると上限を超える]:
                [note[i+1][w]とnote[i][w]は同じになる]
```

knapsack()の漸化式方式の実装例

```
def knapsack_rev2():
    …
    for [i: 0から品物の総数]:
        for [w: 0から総重量の上限]:
            if [品物iを入れると上限を超える]:
                [note[i+1][w]とnote[i][w]は同じになる]
            else:
                [品物iを入れる場合，入れない場合を
                比較し，より大きい方をnote[i+1][w]に]
```

knapsack()の漸化式方式の実装例

```
def knapsack_rev2():
    …
    …
    return [返り値は何を指定すれば良い？]
```

漸化式方式のnoteをのぞいてみよう

[重さ，価値]: [11, 15], [2, 3], [3, 1], [4, 4], [1, 2], [5, 8]
W=15

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	15	15	15	15	15
2	0	0	3	3	3	3	3	3	3	3	3	15	15	18	18	18
3	0	0	3	3	4	4	4	4	4	4	4	15	15	18	18	18
4	0	0	3	3	4	4	7	7	7	8	8	15	15	18	18	19
5	0	2	3	5	5	6	7	9	9	9	10	15	17	18	20	20
6	0	2	3	5	5	8	10	11	13	13	14	15	17	18	20	20

漸化式方式のnoteをのぞいてみよう

1行目は初期状態が入っている.
（このスライド以降，この行は削除して説明．）

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	15	15	15	15	15
2	0	0	3	3	3	3	3	3	3	3	3	15	15	18	18	18
3	0	0	3	3	4	4	4	4	4	4	4	15	15	18	18	18
4	0	0	3	3	4	4	7	7	7	8	8	15	15	18	18	19
5	0	2	3	5	5	6	7	9	9	9	10	15	17	18	20	20
6	0	2	3	5	5	8	10	11	13	13	14	15	17	18	20	20

漸化式方式のnoteをのぞいてみよう

2行目は1つ目の品物（[重さ，価値]: [11, 15]）に対する判断を行ったあとの最適解が入っている．

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	0	0	0	0	0	0	0	0	0	15	15	15	15	15
2	0	0	3	3	3	3	3	3	3	3	3	15	15	18	18	18
3	0	0	3	3	4	4	4	4	4	4	4	15	15	18	18	18
4	0	0	3	3	4	4	7	7	7	8	8	15	15	18	18	19
5	0	2	3	5	5	6	7	9	9	9	10	15	17	18	20	20
6	0	2	3	5	5	8	10	11	13	13	14	15	17	18	20	20

漸化式方式のnoteをのぞいてみよう

[重さ，価値]: [11, 15]
重さの上限が10以下なら，1つ目の品物は入れられない．

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	0	0	0	0	0	0	0	0	0	15	15	15	15	15
2	0	0	3	3	3	3	3	3	3	3	3	15	15	18	18	18
3	0	0	3	3	4	4	4	4	4	4	4	15	15	18	18	18
4	0	0	3	3	4	4	7	7	7	8	8	15	15	18	18	19
5	0	2	3	5	5	6	7	9	9	9	10	15	17	18	20	20
6	0	2	3	5	5	8	10	11	13	13	14	15	17	18	20	20

漸化式方式のnoteをのぞいてみよう

[重さ，価値]: [11, 15]
重さの上限が11以上なら，1つ目の品物は入れるのが最適.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	0	0	0	0	0	0	0	0	0	15	15	15	15	15
2	0	0	3	3	3	3	3	3	3	3	3	15	15	18	18	18
3	0	0	3	3	4	4	4	4	4	4	4	15	15	18	18	18
4	0	0	3	3	4	4	7	7	7	8	8	15	15	18	18	19
5	0	2	3	5	5	6	7	9	9	9	10	15	17	18	20	20
6	0	2	3	5	5	8	10	11	13	13	14	15	17	18	20	20

漸化式方式のnoteをのぞいてみよう

2つ目の品物についても同じ.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	0	0	0	0	0	0	0	0	0	15	15	15	15	15
2	0	0	3	3	3	3	3	3	3	3	3	15	15	18	18	18
3	0	0	3	3	4	4	4	4	4	4	4	15	15	18	18	18
4	0	0	3	3	4	4	7	7	7	8	8	15	15	18	18	19
5	0	2	3	5	5	6	7	9	9	9	10	15	17	18	20	20
6	0	2	3	5	5	8	10	11	13	13	14	15	17	18	20	20

漸化式方式のnoteをのぞいてみよう

[重さ，価値]: [11, 15], [2, 3]
重さの上限が1以下なら何も入れられない.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	0	0	0	0	0	0	0	0	0	15	15	15	15	15
2	0	0	3	3	3	3	3	3	3	3	3	15	15	18	18	18
3	0	0	3	3	4	4	4	4	4	4	4	15	15	18	18	18
4	0	0	3	3	4	4	7	7	7	8	8	15	15	18	18	19
5	0	2	3	5	5	6	7	9	9	9	10	15	17	18	20	20
6	0	2	3	5	5	8	10	11	13	13	14	15	17	18	20	20

漸化式方式のnoteをのぞいてみよう

[重さ，価値]: [11, 15], [2, 3]
重さの上限が2〜10なら，2つ目の品物のみ入れる．

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	0	0	0	0	0	0	0	0	0	15	15	15	15	15
2	0	0	3	3	3	3	3	3	3	3	3	15	15	18	18	18
3	0	0	3	3	4	4	4	4	4	4	4	15	15	18	18	18
4	0	0	3	3	4	4	7	7	7	8	8	15	15	18	18	19
5	0	2	3	5	5	6	7	9	9	9	10	15	17	18	20	20
6	0	2	3	5	5	8	10	11	13	13	14	15	17	18	20	20

漸化式方式のnoteをのぞいてみよう

[重さ，価値]: [11, 15], [2, 3]
重さの上限が11か12なら，1つ目の品物のみ入れる．

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	0	0	0	0	0	0	0	0	0	15	15	15	15	15
2	0	0	3	3	3	3	3	3	3	3	3	15	15	18	18	18
3	0	0	3	3	4	4	4	4	4	4	4	15	15	18	18	18
4	0	0	3	3	4	4	7	7	7	8	8	15	15	18	18	19
5	0	2	3	5	5	6	7	9	9	9	10	15	17	18	20	20
6	0	2	3	5	5	8	10	11	13	13	14	15	17	18	20	20

漸化式方式のnoteをのぞいてみよう

[重さ，価値]: [11, 15], [2, 3]
重さの上限が13以上なら，両方入れる．

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	0	0	0	0	0	0	0	0	0	15	15	15	15	15
2	0	0	3	3	3	3	3	3	3	3	3	15	15	18	18	18
3	0	0	3	3	4	4	4	4	4	4	4	15	15	18	18	18
4	0	0	3	3	4	4	7	7	7	8	8	15	15	18	18	19
5	0	2	3	5	5	6	7	9	9	9	10	15	17	18	20	20
6	0	2	3	5	5	8	10	11	13	13	14	15	17	18	20	20

漸化式方式のnoteをのぞいてみよう

これを繰り返すと，求めたいものは6つ目の品物まで考え，
総重量の制約が15の場合．-> 茶色のセル

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	0	0	0	0	0	0	0	0	0	15	15	15	15	15
2	0	0	3	3	3	3	3	3	3	3	3	15	15	18	18	18
3	0	0	3	3	4	4	4	4	4	4	4	15	15	18	18	18
4	0	0	3	3	4	4	7	7	7	8	8	15	15	18	18	19
5	0	2	3	5	5	6	7	9	9	9	10	15	17	18	20	20
6	0	2	3	5	5	8	10	11	13	13	14	15	17	18	20	20

DPの根本：最適性の原理

「最適な計画となるためには，初期状態・条件に関係なく，残りの決定が最初の決定から生じた状態に対して最適な計画とならなくてはいけない．」

Principle of Optimality: An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decisions.

https://www.rand.org/content/dam/rand/pubs/papers/2008/P550.pdf

DPの根本：最適性の原理

「次の状態での最適解」
＝「今に至るまで最適解」＋「この時点での最適な選択」

これを順次繰り返すことによって，最終的に求めたい状態での最適解にたどり着く．

メモ化再帰のnoteをのぞいてみよう

[重さ，価値]: [11, 15], [2, 3], [3, 1], [4, 4], [1, 2], [5, 8]
W=15

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
0	0	0	0	0	0	0	0	0	0	0	0	15	15	15	15	15
1	--	--	3	3	--	3	3	3	3	3	3	15	15	--	18	18
2	--	--	--	--	--	4	4	--	--	4	4	15	15	--	--	18
3	--	--	--	--	--	--	--	--	--	8	8	--	--	--	18	19
4	--	--	--	--	--	--	--	--	--	--	10	--	--	--	--	20
5	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	20

漸化式方式のnoteをのぞいてみよう

[重さ，価値]: [11, 15], [2, 3], [3, 1], [4, 4], [1, 2], [5, 8]
W=15

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
1	0	0	0	0	0	0	0	0	0	0	0	15	15	15	15	15
2	0	0	3	3	3	3	3	3	3	3	3	15	15	18	18	18
3	0	0	3	3	4	4	4	4	4	4	4	15	15	18	18	18
4	0	0	3	3	4	4	7	7	7	8	8	15	15	18	18	19
5	0	2	3	5	5	6	7	9	9	9	10	15	17	18	20	20
6	0	2	3	5	5	8	10	11	13	13	14	15	17	18	20	20

メモ化再帰のnoteをのぞいてみよう

再帰による呼び出しは必ずしも全部のケースでは行われないので，この表の一部は計算されないまま終わる．

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
0	0	0	0	0	0	0	0	0	0	0	0	15	15	15	15	15
1	--	--	3	3	--	3	3	3	3	3	3	15	15	--	18	18
2	--	--	--	--	--	4	4	--	--	4	4	15	15	--	--	18
3	--	--	--	--	--	--	--	--	--	8	8	--	--	--	18	19
4	--	--	--	--	--	--	--	--	--	--	10	--	--	--	--	20
5	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	20

漸化式方式の場合のナップサック

2重ループになっているだけ.

よって, $O(NW)$. Nは品物の総数.

こちらも，だいぶマシ.

まとめ

DPの紹介

2つの方法：メモ化再帰，漸化式方式

フィボナッチ数，最大和問題，カエルとび問題，
ナップサック問題

改良の方針（再掲）

メモ化再帰（改良1）

再帰をするが計算結果を記録しておき，次回以降はそれを利用して再計算を避ける．

漸化式方式（改良2）

わかっている値から計算をスタートし，漸化式の形で順に計算していくことで，再帰自体を避ける．
（狭義のDPとしてはこちらを意味する．）

メモ化再帰

メリット：

わかりやすい，再帰で実装できていればすぐに効率化可能.

起こりうる全てのケースを計算する必要がない.

デメリット：

再帰分のオーバーヘッドがつきまとう.

再帰が深くなる場合はスタックオーバーフローを起こすことも.

漸化式方式

メリット：

再帰がなく，ループだけで記述可能.

計算量の見積もりが非常にわかりやすい.

応用の幅がより大きい（と思う）.

デメリット：

どのように設計するか，少しコツがいる.

実際問題としては,

どちらかの方式でないと解けない，時間が大幅にかかりすぎる，ということは（多くの場合）ないと思います.

ネット上でもどちらの流派を好むかで色々議論があるようです.

自分なりにポリシーを決めて書くことができれば良いと思いますが，どちらの方式のコードを見ても理解できるようになっていることがベスト.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	0	0	0	0	0	0	0	0	0	15	15	15	15	15
2	0	0	3	3	3	3	3	3	3	3	3	15	15	18	18	18
3	0	0	3	3	4	4	4	4	4	4	4	15	15	18	18	18
4	0	0	3	3	4	4	7	7	7	8	8	15	15	18	18	19
5	0	2	3	5	5	6	7	9	9	9	10	15	17	18	20	20
6	0	2	3	5	5	8	10	11	13	13	14	15	17	18	20	20

やっていることは一緒！

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	15	15	15	15	15
1	--	--	3	3	--	3	3	3	3	3	3	15	15	--	18	18
2	--	--	--	--	--	4	4	--	--	4	4	15	15	--	--	18
3	--	--	--	--	--	--	--	--	--	8	8	--	--	--	18	19
4	--	--	--	--	--	--	--	--	--	--	10	--	--	--	--	20
5	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	20

来週は？

漸化式方式をもう少し詳しく見ていきましょう.

矢谷式のDPの考え方もお伝えしたいと思います！

コードチャレンジ：基本課題#6-a [1.5点]

授業中に紹介した「カエルとび問題」を解くコードを書いてください.

メモ化再帰，漸化式方式でもどちらでも構いません.

コードチャレンジ：基本課題#6-b [1.5点]

ナップサック問題を漸化式方式で解くコードを，授業中に紹介した擬似コードに従って書いてください.

こちらは漸化式方式で書いてください.

コードチャレンジ：Extra課題#6 [3点]

DPを使った問題.