Algorithms (2023 Summer)
#7：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，先の実装の計算量は,

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

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，先の実装の計算量は,

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

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

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）

改良のポイント

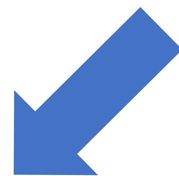
非効率なところはどこか？

```
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)
```



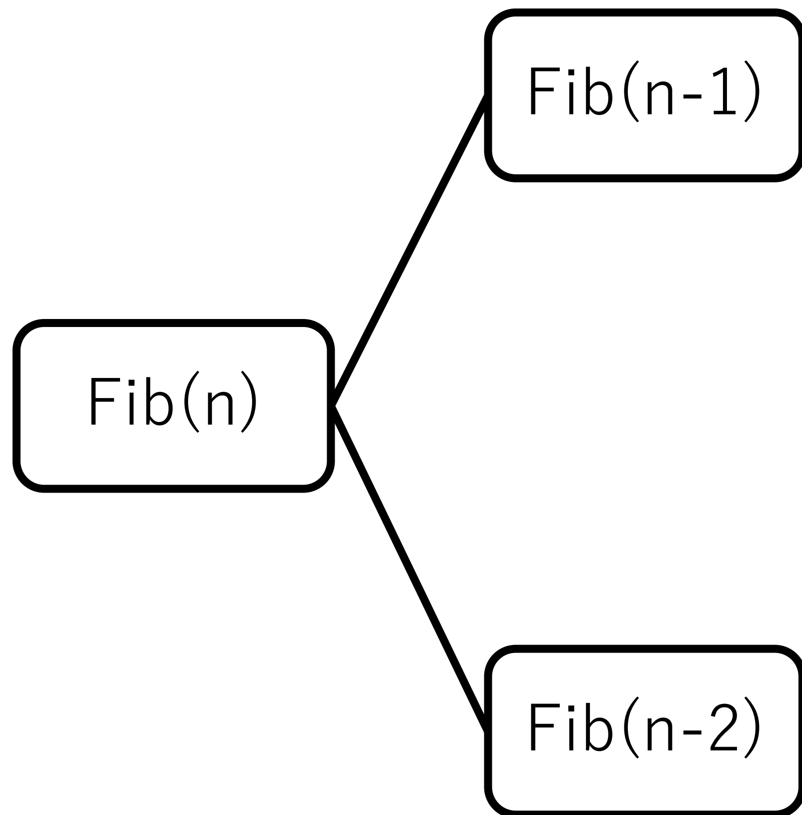
毎回呼び出す，毎回同じことをやっている系は多くの場合，非効率．

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．

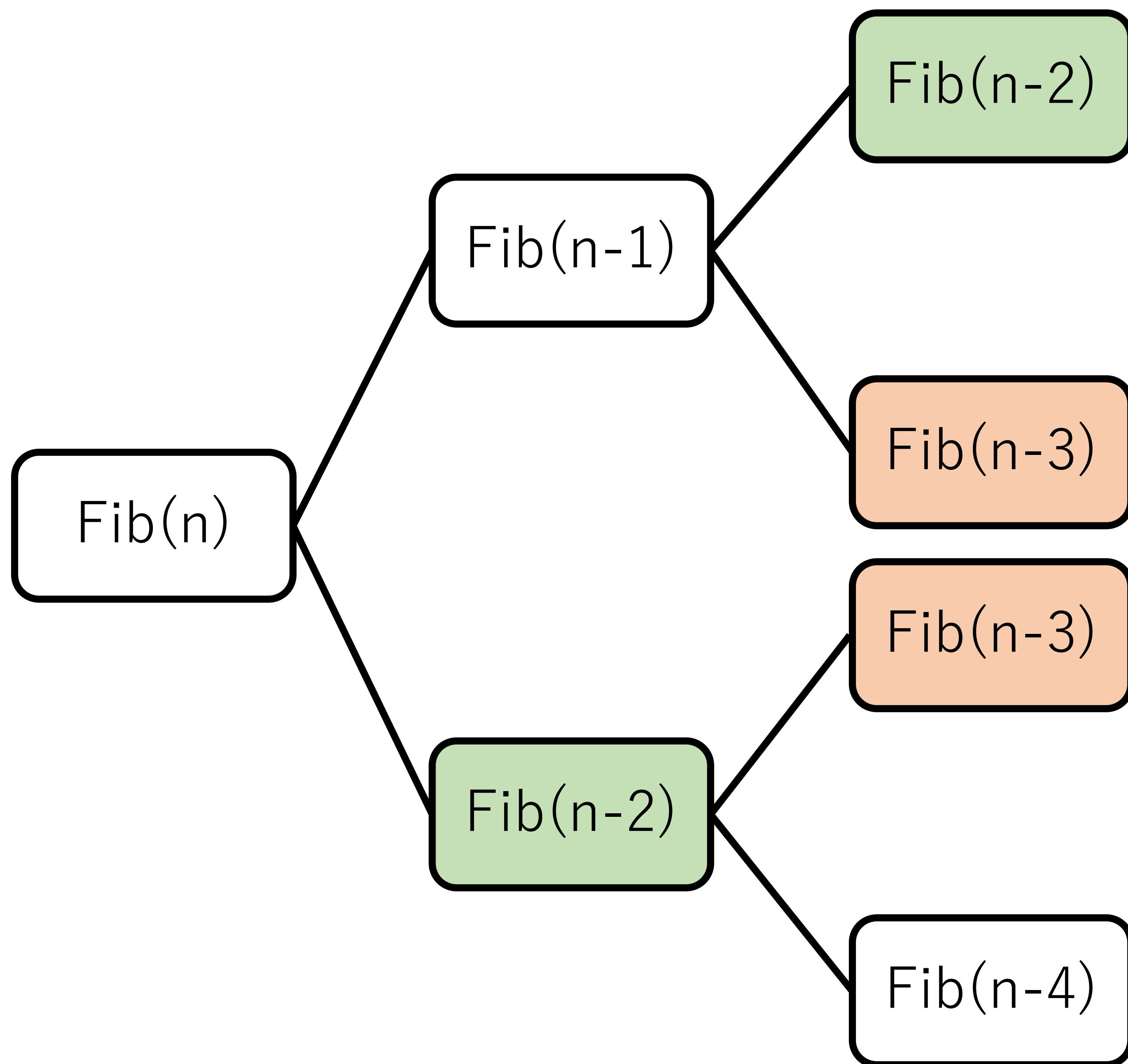
改良のポイント

Fib(n)

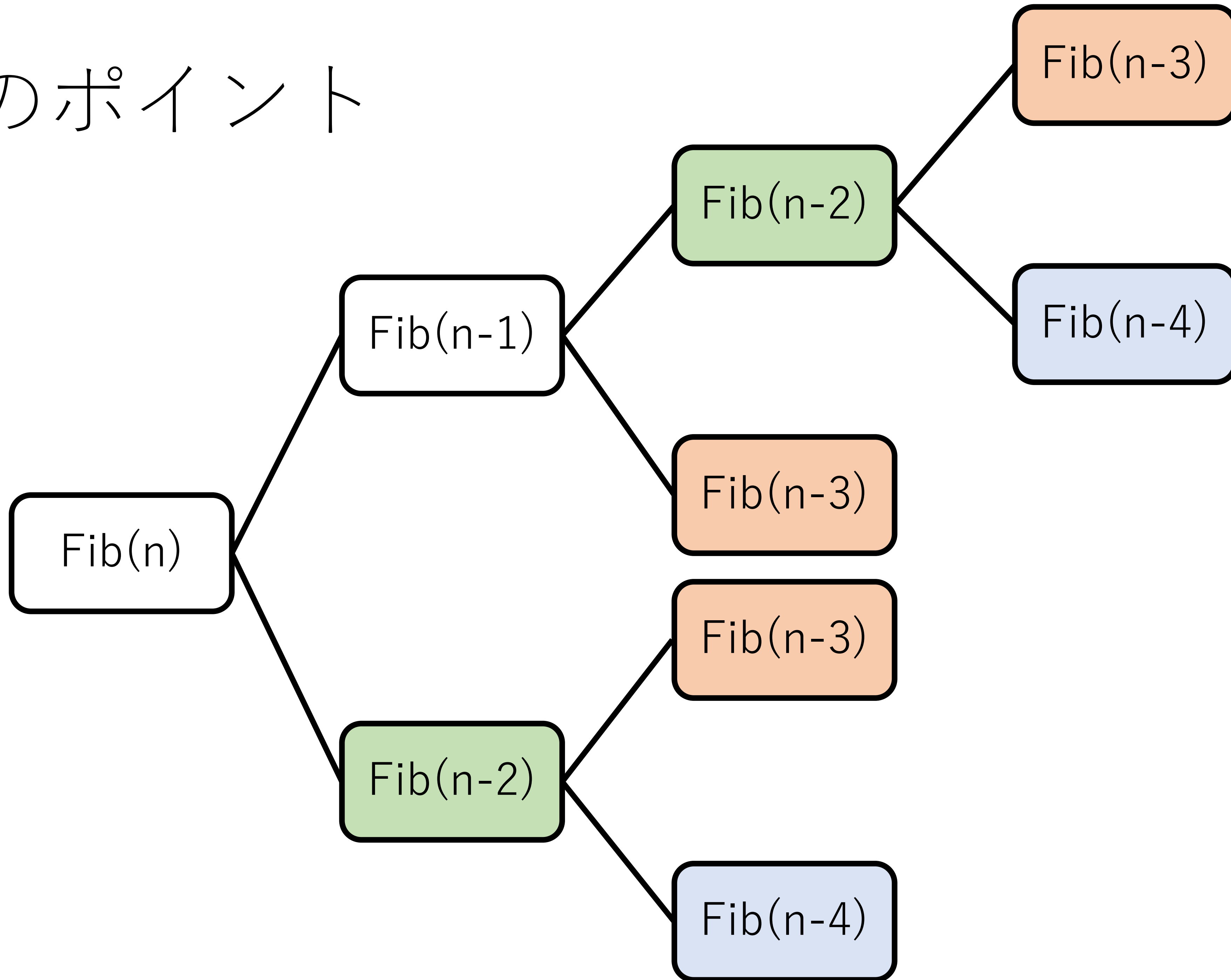
改良のポイント



改良のポイント



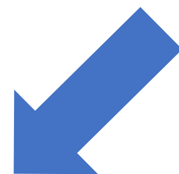
改良のポイント



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

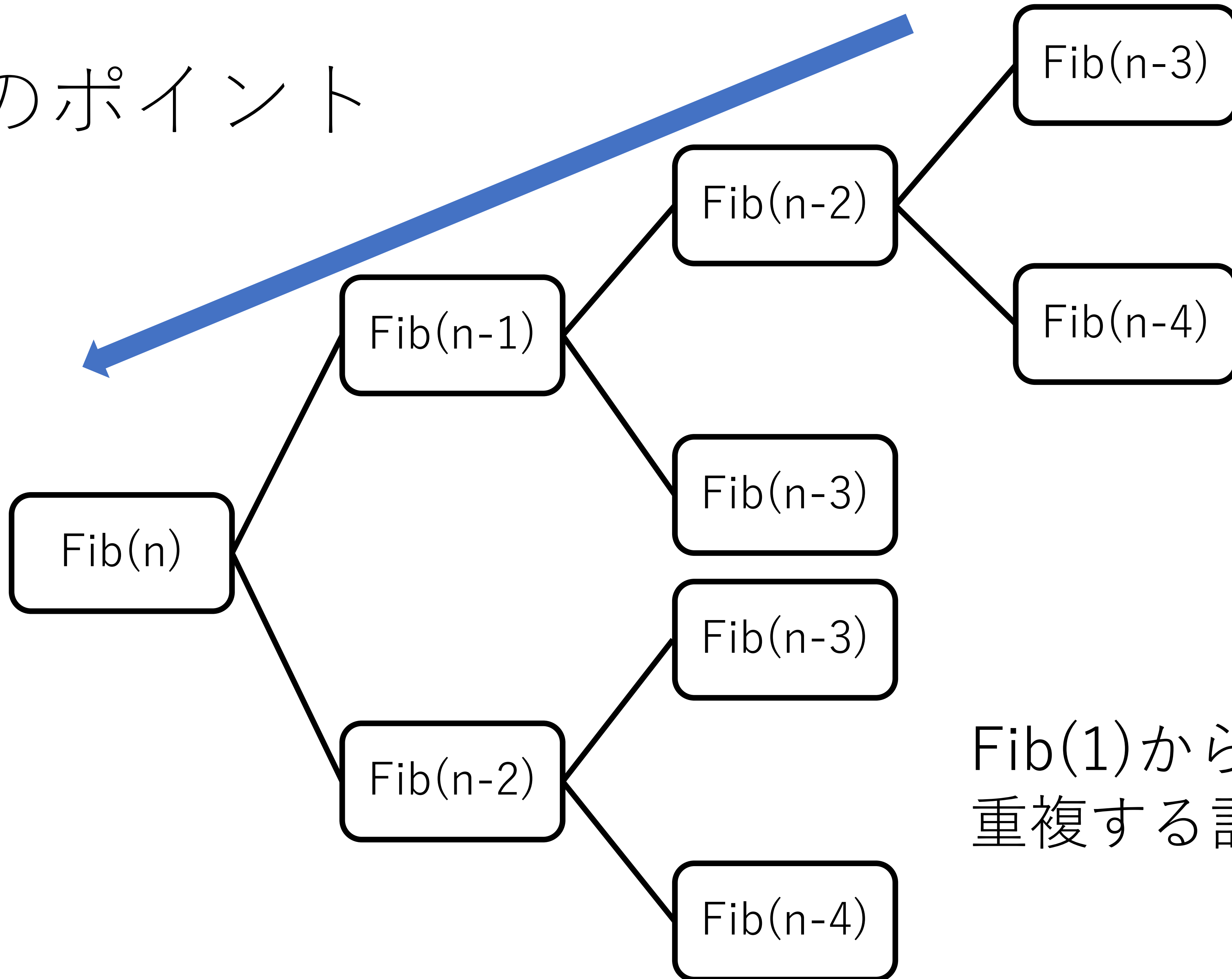
```
def fib_rev1(n):
    f = [0]*(n+1)      # 記憶用の配列
    def _fib(n):
        if n <= 2: return 1
        elif f[n] != 0: return f[n] # 記憶があればそれを使う
        else:  # なければ計算する
            f[n] = _fib(n-1) + _fib(n-2)
            return f[n]

    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

$Fib(n-1)$と$Fib(n-2)$がわかっていれば，　$Fib(n)$を求めることができる．

フィボナッチ数列

$Fib(n-1)$と$Fib(n-2)$がわかっていれば，　$Fib(n)$を求めることができる．

$Fib(1), Fib(2)$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

$Fib(n-1)$と$Fib(n-2)$がわかっていれば，　$Fib(n)$を求めることができる．

$Fib(1), Fib(2)$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+1)
        f[0] = 0; f[1] = 1; f[2] = 1
        for i in range(3, n+1):
            f[i] = f[i-1] + f[i-2]
    return f[n]
```

n	fib(n) [msec]	fib_rev1(n) [msec]	fib_rev2(n) [msec]
30	284	0.016	0.004
31	461	0.015	0.004
32	786	0.015	0.005
33	1,279	0.015	0.004
34	2,044	0.015	0.004
35	3,236	0.016	0.004
36	5,238	0.016	0.004
37	8,237	0.016	0.004
38	13,227	0.017	0.005
39	22,027	0.018	0.004
40	35,057	0.018	0.004

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，
ここではDPの簡単な練習問題として考えてみましょう．🙂

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

S_k：与えられた配列の1番目（indexでは0）からk番目（indexではk-1）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

S_{k+1}が取りうる値の可能性は2つ.

S_kにk+1番目の要素を足したもの.

S_kにk+1番目の要素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

S_kとS_{k+1}の漸化式的関係性を考えると，

$$S_{k+1} = \max(S_k, S_k + [k+1\text{番目の要素}])$$

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．）

最大和問題の漸化式方式での計算量

単なる1重のforループ！→ $O(n)$.

```
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)]
```

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

Copy

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

出力例 3 Copy

Copy

```
40
```

足場 $1 \to 3 \to 5 \to 6$ 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：

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.

S_1は最初の位置なので，最小コストは0.

S_2は1番目の足場から飛んでくるしかないので,
最小コストは$c(0,1)$.

DPの実装方針（再掲）

メモ化再帰

再帰をするが計算結果を記録する.

計算結果があるものはそれを利用して再計算を避ける.

漸化式方式

漸化式の形で計算を表現して，再帰を避ける.

DPの実装方針

メモ化再帰は，再帰で実装できれば比較的すぐに
実現できる．

漸化式方式は再帰呼び出しがない，計算量を簡単に
見積もれるなどのメリットがあり，実装上有利な
ことがある．

漸化式方式をよりダイレクトに実装できないか？

DPテーブル

DPを実装したときにできるテーブル．

上の例で言えば，漸化式方式で出てきたリスト f や s．

これをいかに設計し，解釈するかがDP成功の鍵！

矢谷式DPの考え方😆

#1 DPテーブルを設計する.

#2 DPテーブルを初期化する.

#3 DPテーブル上のあるセルに対して，1ステップの操作で他のどのセルから遷移できるかを調べる.

#4 #3でわかったことをコードに落とし込む.

（DPの全部の問題がうまく解けるわけではありません．あしからず．．．）

コイン問題

「m種類の額面のコインc[0]，c[1]，…，　c[m-1]円が与えられたとき，n円を支払う最小枚数を求めよ．
各コインは何枚でも使って良い．」

例）1，8，13円の3種類のコインで25円を支払う．

答え：4枚（8，8，8，1）

コイン問題

「m種類の額面のコインc[0]，c[1]，…，　c[m-1]円が与えられたとき，n円を支払う最小枚数を求めよ．
各コインは何枚でも使って良い．」

例）1，8，13円の3種類のコインで25円を支払う．

大きい額のコインから順に払うのでは，うまく求まらない場合がある．

25円の場合，（13，8，1，1，1，1）で6枚となる．

矢谷式DPの考え方：#1 DPテーブルの設計

DPテーブルの設計ヒューリスティックス
（まずはこれを考えよう！）

DP**テーブルのセル：求めたいもの**

テーブルの行と列：セルの説明変数の取りうる「より小さな状態」を全部並べたもの

$[\text{セル}] = f(x, y)$となるx, yが行と列の候補.

より小さな値や先頭からの部分集合（prefix）**など.**

矢谷式DPの考え方：#1 DPテーブルの設計

DP**テーブルのセル：求めたいもの**

この場合，最小枚数

矢谷式DPの考え方：#1 DPテーブルの設計

テーブルの行と列：セルの説明変数の取りうる「より小さな状態」を全部並べたもの

［最小枚数］= f([支払う金額(1~n円)])

今回は支払う金額1~n円のみなので，1次元のテーブルを作る．

矢谷式DPの考え方：#1 DPテーブルの設計

これを表にすると，以下の通りになる．

	1	2	3	4	5	6	7	8

矢谷式DPの考え方：#1 DPテーブルの設計

これを表にすると，以下の通りになる．

	1	2	3	4	5	6	7	8

1円払う時の
最小枚数

矢谷式DPの考え方：#1 DPテーブルの設計

これを表にすると，以下の通りになる．

	1	2	3	4	5	6	7	8

2円払う時の
最小枚数

矢谷式DPの考え方：#2 DPテーブルの初期化

「初期状態」をDPテーブルに追加する．

例えば，探索が始まる前状態など．これらの状態では，
計算をしなくても答えがわかっている．

矢谷式DPの考え方：#2 DPテーブルの初期化

「初期状態」をDPテーブルに追加する．

例えば，探索が始まる前状態など．これらの状態では，
計算をしなくても答えがわかっている．

今回の場合は，支払いをしていない，つまり0円支払う時
がある．この時の最小枚数は明らかに0.

矢谷式DPの考え方：#2 DPテーブルの初期化

初期状態を追加する．

	0	1	2	3	4	5	6	7	8
	0								

矢谷式DPの考え方：#3 操作のマッピング

DPテーブルのセルが全部埋まれば，答えが求まる！

ただし，いきなり答えにつながるセルは求まらないので，初期状態のセルなどを取っ掛かりにして，順番に埋めていくことになる．

1ステップ分で行える操作が，DPテーブル上においてどういう処理に当たるか，を考える．

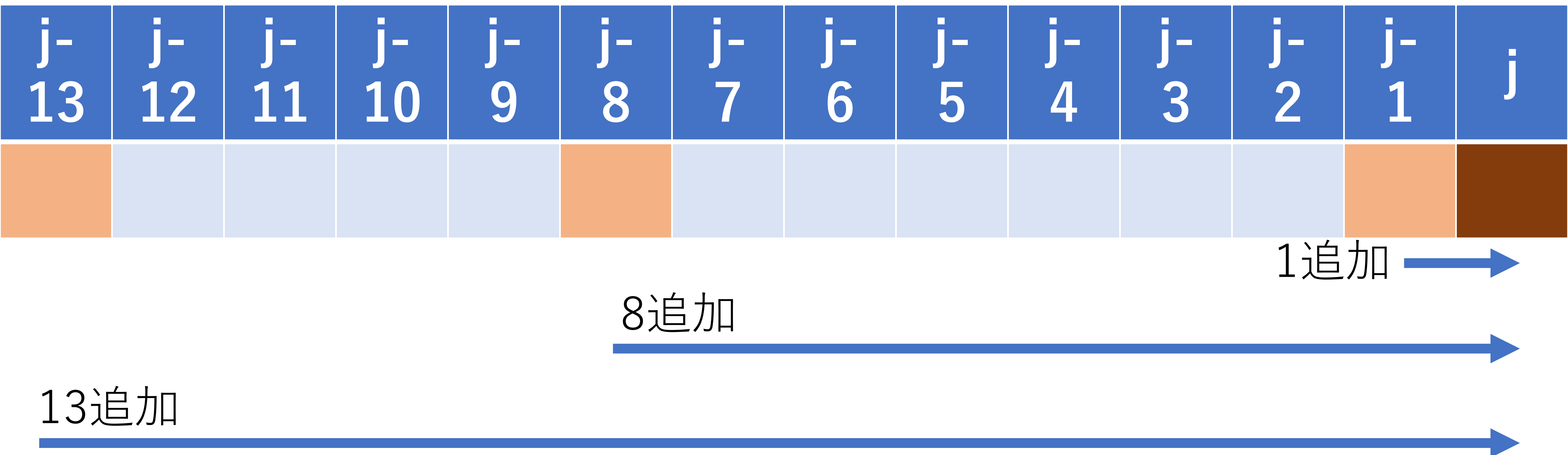
矢谷式DPの考え方：#3 操作のマッピング

jのセルに1ステップで到達してくるケースを考える．
つまり，1枚コインを足すとj円になるケース．

j-13	j-12	j-11	j-10	j-9	j-8	j-7	j-6	j-5	j-4	j-3	j-2	j-1	j

矢谷式DPの考え方：#3 操作のマッピング

jのセルに到達してくるケースを考えると，3パターンある．
（実際にはm種類の硬貨があるので，mパターン）



矢谷式DPの考え方：#4 コード化

この遷移を式で表せばよい.

1，8，13を1枚追加してセルjに遷移してくる．この内,
最小のものだけ，記録しておけば良い.

すなわち，（dp[j-1], dp[j-8], dp[j-13]の最小値）+ 1 がdp[j]
の値となる．（ただし，indexが負にならないように注意．）

これをm種類のコインがある一般的な場合で書く.

矢谷式DPの考え方：#4 コード化

```
# amount: 支払う額，coins: 硬貨の種類
# 例）coins = [1, 8, 13]

# dpテーブルの準備．とりあえず全部0にしておく．
dp = [0 for i in range(amount+1)]
```

矢谷式DPの考え方：#4 コード化

```
…
for i in range(1, amount+1):
    # (i - coins[j])円の中で最も枚数が少ないものを取り出す
    tmp_min = 10**6     # 十分に大きい数を設定
    for j in range(len(coins)):
        if (i - coins[j] > -1) and (tmp_min > dp[i - coins[j]]):
            tmp_min = dp[i - coins[j]]
```

矢谷式DPの考え方：#4 コード化

```
…
for i in range(1, amount+1):
        tmp_min = 10**6
        for j in range(len(coins)):
                if (i - coins[j] > -1) and (tmp_min > dp[i - coins[j]]):
                        tmp_min = dp[i - coins[j]]
        dp[i] = tmp_min + 1          # 1枚加えてi円の枚数にする
```

矢谷式DPの考え方：#4 コード化

```
…
for i in range(1, amount+1):
        tmp_min = 10**6
        for j in range(len(coins)):
                if (i - coins[j] > -1) and (tmp_min > dp[i - coins[j]]):
                        tmp_min = dp[i - coins[j]]
        dp[i] = tmp_min + 1

return dp[amount]    # 求めたいものはdpテーブルの端にある
```

実行例

amount = 25

coins = [1, 8, 13]

結果：4

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	2	3	4	5	6	7	1	2	3	4	5	1

14	15	16	17	18	19	20	21	22	23	24	25
2	3	2	3	4	5	6	2	3	4	3	4

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

「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．」

各品物に対して，入れるか入れないかの2択なので，「0-1ナップサック問題」とも言われる．

$$\max \sum_i v_i x_i \quad \text{ただし，} \sum_i w_i x_i \leq W, x_i \in \{0,1\}$$

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]

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．（貪欲法）

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.

矢谷式DPの考え方：#1 DPテーブルの設計

DPテーブルの設計ヒューリスティックス
（まずはこれを考えよう！）

DP**テーブルのセル：求めたいもの**

テーブルの行と列：セルの説明変数の取りうる「より小さな状態」を全部並べたもの

$[\text{セル}] = f(x, y)$となるx, yが行と列の候補.

より小さな値や先頭からの部分集合（prefix）**など.**

矢谷式DPの考え方：#1 DPテーブルの設計

DP**テーブルのセル：求めたいもの**

この場合，価値の総和

テーブルの行と列：セルの説明変数の取りうる「より小さな状態」を全部並べたもの

求める答えで設定されているアイテムの数，重さよりも少ない/小さい状態

矢谷式DPの考え方：#1 DPテーブルの設計

テーブルの行と列：セルの説明変数の取りうる「より小さな状態」を全部並べたもの

アイテムで言えば，「0個考えた状態」，「1番目のアイテムを考えた状態」，「2番目のアイテムを考えた状態」．．．

重さの制限で言えば，「重さの上限が0の状態」，「重さの上限が1の状態」，「重さの上限が2の状態」．．．

矢谷式DPの考え方：#1 DPテーブルの設計

列は重さの上限を表す.

右端は今与えられている重さの上限（W=15）.

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

矢谷式DPの考え方：#1 DPテーブルの設計

行は各アイテムを考慮することを表す.
（入れているかどうかを直接表すわけではない．）

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

矢谷式DPの考え方：#1 DPテーブルの設計

「0」の行は，どのアイテムも考えていないことを表す．

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

矢谷式DPの考え方：#1 DPテーブルの設計

「1」の行は，1つ目のアイテム（重さ：11，価値：15）を入れるかどうかを考えたことを表す．

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

矢谷式DPの考え方：#1 DPテーブルの設計

セル(i, j)は，i番目のアイテムまで考えたときに，重さの上限がjであることを満たす状況を表す．

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

3つ目のアイテムまで考えたときに，重さの上限5を満たす状況.

矢谷式DPの考え方：#2 DPテーブルの初期化

「初期状態」をDPテーブルに追加する．

今回はアイテムが全く入っていない場合が明示的にわかる．

そもそも何もまだ考えていないので，ナップサックは確実に空のはず．

矢谷式DPの考え方：＃２DPテーブルの初期化

「0」の行のセルは全部0.

	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																
2																
3																
4																
5																
6																

矢谷式DPの考え方：#3 操作のマッピング

1**ステップ分で行える操作が，**DP**テーブル上において**
どういう処理に当たるか，を考える．

今回の問題における1ステップ

あるアイテムを入れるか，入れないかを考えること．

これを各セルについて考える．

矢谷式DPの考え方：#3 操作のマッピング

茶色のセルについて考えてみよう．

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

矢谷式DPの考え方：#3 操作のマッピング

このセルは5つ目のアイテム（重さ：1，価値：2）まで考えたときに，重さの上限7を満たしている状況を意味している．

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

矢谷式DPの考え方：#3 操作のマッピング

このセル（状況）に1ステップで至るケースを考えてみよう．

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

矢谷式DPの考え方

考えられるのは2つ．5番目のアイテムを入れても重さの上限7を満たすか，入れずにいて重さの上限7を満たすか．

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

入れる場合，斜めの矢印
入れない場合，下向きの矢印

矢谷式DPの考え方

5番目のアイテムを入れる場合の総価値：dp[4][7-1]+value[5]

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

入れる場合，斜めの矢印

矢谷式DPの考え方

5番目のアイテムを入れない場合の総価値：dp[4][7]

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

入れない場合，下向きの矢印

矢谷式DPの考え方

この2つのうち，より総価値が高い方だけ記録すれば良い．

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

入れる場合，斜めの矢印
入れない場合，下向きの矢印

矢谷式DPの考え方：#4 コード化

この遷移を一般化して，表せばよい．

i番目のアイテムを入れて重さの上限jを満たすときの総価値：dp[i-1][j-weight[i]]+value[i]

i番目のアイテムを入れずに重さの上限jを満たすときの総価値：dp[i-1][j]

よって，dp[i][j] = max(dp[i-1][j], dp[i-1][j-weight[i]]+value[i])

（ただし，indexが負にならないように注意．）

矢谷式DPの考え方：#4 コード化

```
W = 15; N = 6
weight = [11, 2, 3, 4, 1, 5]
value = [15, 3, 1, 4, 2, 8]
dp = [[-1 for _ in range(W+1)]
               for _ in range(N+1)]

# 「0」の行のセルは全部0
for i in range(W+1): dp[0][i] = 0
```

矢谷式DPの考え方：#4 コード化

```
for i in range(N):
    for j in range(W+1):
        # indexが負にならないように注意.
        if weight[i] <= j:
            dp[i+1][j] = max(dp[i][j],
                             dp[i][j-weight[i]]+value[i])
        else: dp[i+1][j]=dp[i][j]

return dp[N][W]
```

DPテーブルをのぞいてみよう

[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	3	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テーブルをのぞいてみよう

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	3	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テーブルをのぞいてみよう

「1」の行は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	3	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テーブルをのぞいてみよう

[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	3	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テーブルをのぞいてみよう

[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	3	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テーブルをのぞいてみよう

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	3	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テーブルをのぞいてみよう

[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	3	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テーブルをのぞいてみよう

[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	3	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テーブルをのぞいてみよう

[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	3	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テーブルをのぞいてみよう

[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	3	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テーブルをのぞいてみよう

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	3	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

矢谷式DPの考え方：#1 DPテーブルの設計

テーブルの行と列：セルの説明変数の取りうる「より小さな状態」を全部並べたもの

アイテムで言えば，「0個考えた状態」，
「1番目のアイテムを考えた状態」，
「2番目のアイテムを考えた状態」．．．

総重さで言えば，「総重さが0の状態」，
「総重さが1の状態」，「総重さが2の状態」．．．

もしこう考えたとすると？

矢谷式DPの考え方：#1 DPテーブルの設計

テーブルの形自体は同じ．ただし，今度は列が「総重さがピッタリjの時」を表す．

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

矢谷式DPの考え方：#2 DPテーブルの初期化

「初期状態」をDPテーブルに追加する．

今回はアイテムが全く入っていない場合が明示的にわかる．

そして，このときは総重さは必ず0．

矢谷式DPの考え方：＃２ DPテーブルの初期化

よって，（0, 0）のセルは0．それ以外は値を持たない．
（実装上は-float('inf')とかにする．）

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

矢谷式DPの考え方

あとは先程と一緒！

実行結果

列が「総重さがピッタリj」であることを表す場合.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1	0	-	-	-	-	-	-	-	-	-	-	15	-	-	-	-
2	0	-	3	-	-	-	-	-	-	-	-	15	-	18	-	-
3	0	-	3	1	-	4	-	-	-	-	-	15	-	18	16	-
4	0	-	3	1	4	4	7	5	-	8	-	15	-	18	16	19
5	0	2	3	5	4	6	7	9	7	8	10	15	17	18	16	19
6	0	2	3	5	4	8	10	11	13	12	14	15	17	18	20	19

実行結果

この場合は一番最後の列を見て，その最大のセルを取ることになる（どのセルも重さの上限Wを満たす）ので，20.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1	0	-	-	-	-	-	-	-	-	-	-	15	-	-	-	-
2	0	-	3	-	-	-	-	-	-	-	-	15	-	18	-	-
3	0	-	3	1	-	4	-	-	-	-	-	15	-	18	16	-
4	0	-	3	1	4	4	7	5	-	8	-	15	-	18	16	19
5	0	2	3	5	4	6	7	9	7	8	10	15	17	18	16	19
6	0	2	3	5	4	8	10	11	13	12	14	15	17	18	20	19

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.

ナイーブな解法だと$O(2^N)$なので，だいぶマシ.

部分和問題

「n個の整数a[0]，a[1]，…，a[n-1]が与えられたとき，そのいくつかを組み合わせて総和がSにできるかどうかを判定せよ．」

矢谷式DPの考え方：#1 DPテーブルの設計

DP**テーブルのセル：求めたいもの**

和をSにすることができるかどうか（Boolean）

矢谷式DPの考え方：#1 DPテーブルの設計

テーブルの行と列：セルの説明変数の取りうる「より小さな状態」を全部並べたもの

与えられた整数に関しては，「0個考えた状態」，「1番目の整数を考えた状態」，「2番目の整数を考えた状態」．．．

総和に関しては，「総和が0の状態」，「総和が1の状態」，「総和が2の状態」．．．

矢谷式DPの考え方：部分和問題の場合

#1 DPテーブルの設計

行が与えられた整数，列は1~S，セルはSになるかどうかのBoolean

		1	2	3	4	5	…
a[0]							
a[1]							
a[2]							
a[3]							
…							

矢谷式DPの考え方：部分和問題の場合

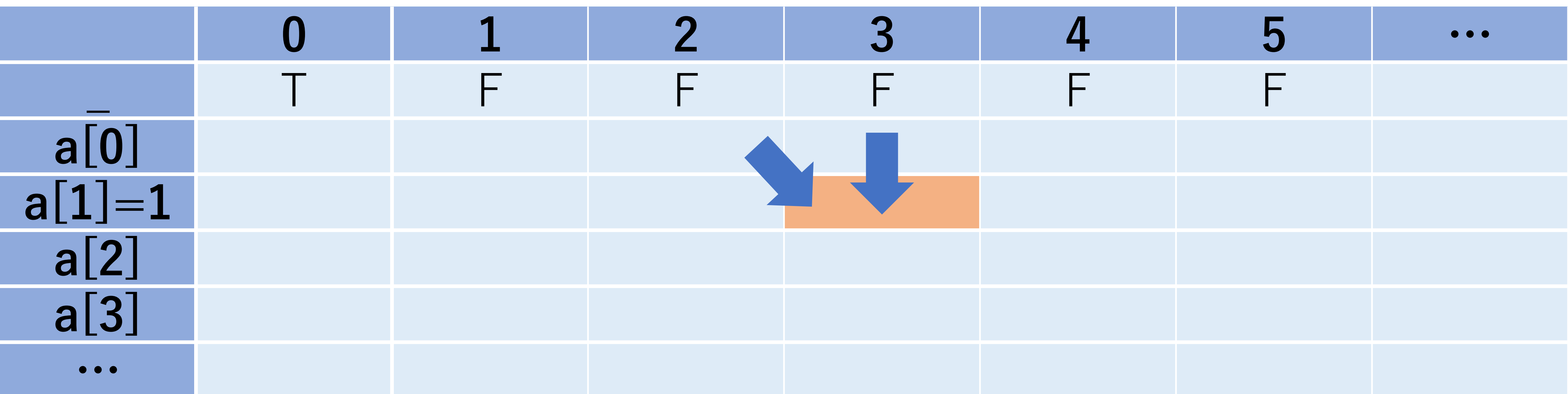
#2 DPテーブルの初期化
数字がなにもない状態なら取りうる部分和は0のみ.

	0	1	2	3	4	5	…
_	T	F	F	F	F	F	
a[0]							
a[1]							
a[2]							
a[3]							
…							

矢谷式DPの考え方：部分和問題の場合

#3 操作のマッピング

仮にa[1]=1とすると，dp[2][3]に移ってくるのは，dp[1][3]（a[1]を入れない）かdp[1][2]（a[1]を入れる）．

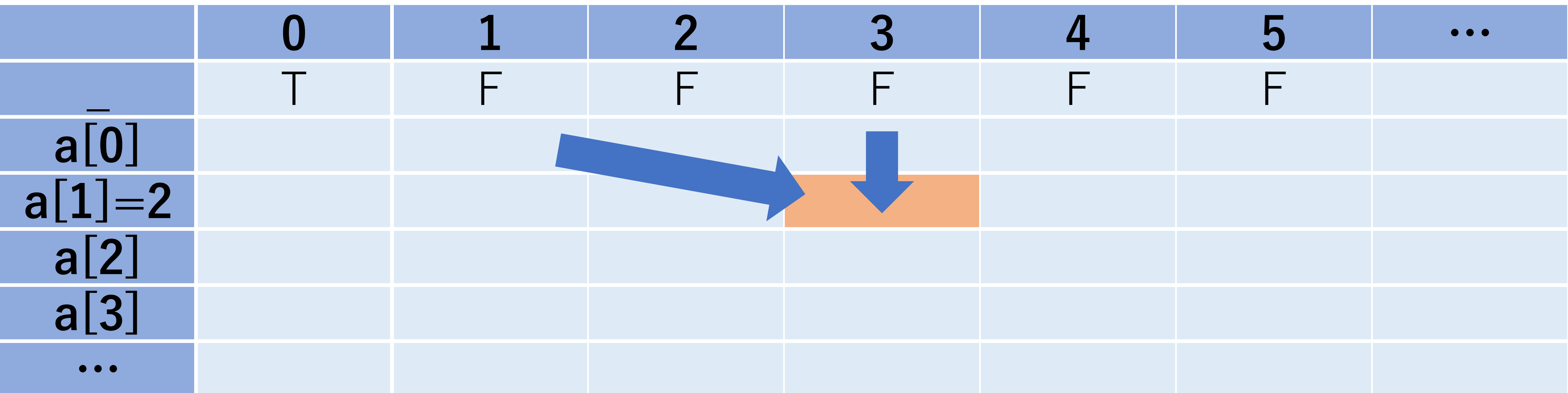


	0	1	2	3	4	5	…
_	T	F	F	F	F	F	
a[0]							
a[1]=1							
a[2]							
a[3]							
…							

矢谷式DPの考え方：部分和問題の場合

#3 操作のマッピング

もしa[1]=2とすると，dp[2][3]に移ってくるのは，dp[1][3]（a[1]を入れない）かdp[1][1]（a[1]を入れる）．



	0	1	2	3	4	5	…
_	T	F	F	F	F	F	
a[0]							
a[1]=2							
a[2]							
a[3]							
…							

矢谷式DPの考え方：部分和問題の場合

#3 操作のマッピング

よって，dp[i][j]に関係するセルは，dp[i-1][j]（a[i]を入れない）かdp[i-1][j-a[i]]（a[i]を入れる），となる．

	0	1	2	3	4	5	…
_	T	F	F	F	F	F	
a[0]							
a[1]							
a[2]							
a[3]							
…							

矢谷式DPの考え方：部分和問題の場合

#4 コード化

今までのことをどうやってコードに落とし込めばよいか,
ぜひ考えてみてください. 😊

部分和問題に似たものも同様にできる！

「n個の整数a[0], a[1], …, a[n-1]が与えられ，そのいくつかを組み合わせて総和がSにできるのは何通りか．」

「n個の整数a[0], a[1], …, a[n-1]が与えられ，そのいくつかを組み合わせて総和がSにできる時，選ぶ整数の最小個数を求めよ．」

セルに何を保持したらいいかを考えてみると，自ずとできるはずです．😊

どうしてこんなことができるの？

DPの根本：最適性原理

「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：最適性原理

「次の状態での最適解」
＝「今に至るまでの最適解」＋「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．

なので，1ステップ分だけ考えていけば良い！

これが成立しない場合はDPは使えない．

まとめ

DPの紹介

2つの方法：メモ化再帰，漸化式方式

矢谷式DPの考え方

DPテーブルを意識しよう！

DPの基本問題

フィボナッチ数，最大和問題，コイン問題，ナップサック問題

矢谷式DPの考え方😆

#1 DPテーブルを設計する.

#2 DPテーブルを初期化する.

#3 DPテーブル上のあるセルに対して，1ステップの操作で他のどのセルから遷移できるかを調べる.

#4 #3でわかったことをコードに落とし込む.

（DPの全部の問題がうまく解けるわけではありません．あしからず．．．）

コードチャレンジ：基本課題#7-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.

メモ化再帰の場合は，再帰・スタックサイズの制限の設定に気をつけてください．授業のページも参考にしてコードを書いてください.

コードチャレンジ：基本課題#7-b [1.5点]

授業中に紹介した「部分和問題」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#7 [3点]

DPを使った問題.