Algorithms (2020 Summer)
#7：動的計画法2

矢谷 浩司

前回のQA

「基本課題の1つ目で，メモ化再帰だとエラーが出ることがある．」

→再帰回数の上限を超えてしまうためであったようです．失礼いたしました．漸化式方式でやっていただければ問題なく解けたかと思います．

前回のQA

「メモ化再帰では解けるが漸化式型では解けない、あるいはその逆みたいなケースってありますか？」

→スタックオーバーフローする場合を除いて,
基本的にはないと思います.

前回のQA

「先ほどの計算速度比較表（フィボナッチ数列の実行例）から考えるとメモ化再帰と漸化式方式では、基本的に漸化式方式の方が定数倍早いと考えて大丈夫ですか？」

→必ず漸化式方式の方が定数倍速い，とは限りません．漸化式方式は常にすべてのセルを埋めることになるので，計算過程によってはメモ化再帰より遅くなることもありえます．ただ関数呼び出しはそれなりに時間がかかるので，それを削減できるメリットはあります．

前回のQA

「i-1まででは入れていたけど、iの方がコスパいいから、それまで入れていたものを出すというのはこのコードのどこで実現されているのですか？」

→（重さ，価値）=[1, 1], [1, 2], [1, 3]で重さの和の上限が1のナップサック問題を考えましょう．
noteは右のようになります．

	0	1
_	0	0
a[0]	0	1
a[1]	0	2
a[2]	0	3

前回のQA

「i-1まででは入れていたけど、iの方がコスパいいから、それまで入れていたものを出すというのはこのコードのどこで実現されているのですか？」

→このとき，オレンジの2という値は，1つ前の状態から決まります．すなわち，a[0]を入れない（黄色）かa[0]が入っている（黄緑）という2つの状態から，さらにa[1]を入れるか，という判断をすることになります．

	0	1
_	0	0
a[0]	0	1
a[1]	0	2
a[2]		

前回のQA

「i-1まででは入れていたけど、iの方がコスパいいから、それまで入れていたものを出すというのはこのコードのどこで実現されているのですか？」

→今回の場合，[黄色]+2 か [黄緑]を比較して，より大きい値がオレンジのセルに入ることになります.
したがって，ここは2という値になり，
それはa[0]入れずにa[1]を入れたほうがよいということに対応します.

	0	1
_	0	0
a[0]	0	1
a[1]	0	2
a[2]		

前回のQA

「i-1まででは入れていたけど、iの方がコスパいいから、それまで入れていたものを出すというのはこのコードのどこで実現されているのですか？」

→DPは最適性の原理を実装しているものになります．まず1つ前までの状態で最適なものがわかっているという状態になっています．ここで，現時点での最適な判断を下すとき，1つ前までの状態のあるセルの情報を使って，現時点に対応するセルを更新するという作業を行います．そして，それを最後の状態まで行っていく，ということ実装しています．

前回のQA

「なぜ二次元配列にする必要があるのでしょうか？wは問題文に与えられていて変化しないのではと思ったのですが」

→「W以下で取りうる全ての状態」を列挙するためにw（小文字のw）を使っています．この全ての取りうる状態を考慮するのがDPのミソであり，今日はこれを深く見ていきたいと思います．

今日の目標

DPのいろんな問題を解いてみよう！

習うより慣れろ，ということで．😆

あとは，矢谷なりのDPの考え方を共有したいと思います．

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）

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.

先週のスライドで言えば，漸化式方式で出てきたnote.

<u>これをいかに設計し，解釈するかがDP成功の鍵！</u>

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押し込む.

（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）

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

[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

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．**

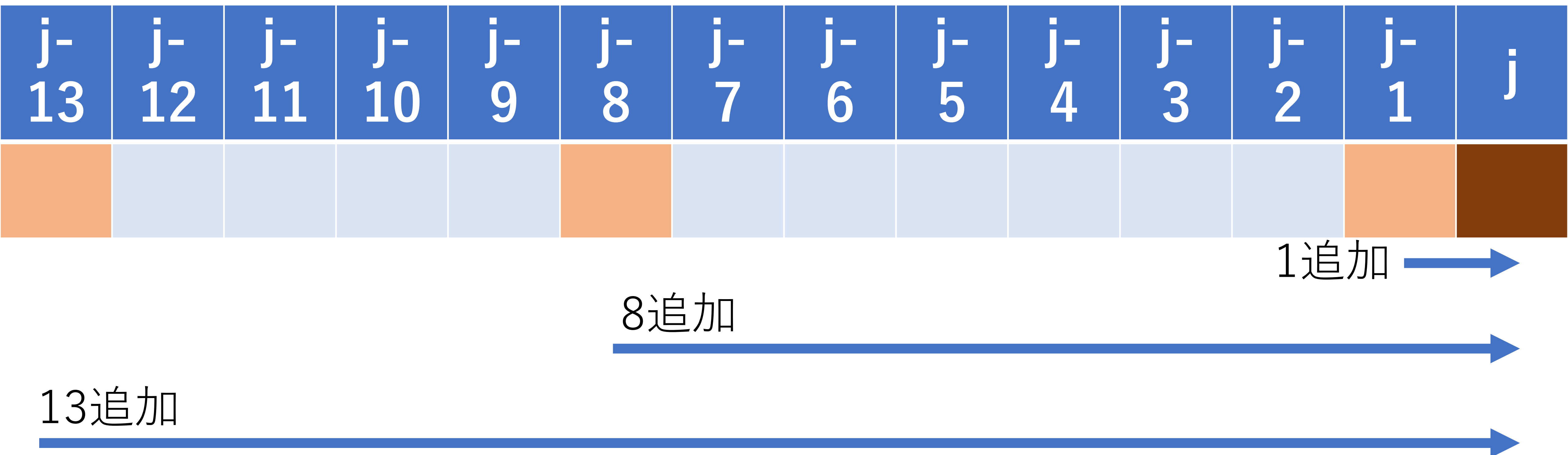
矢谷式DPの考え方：#3 操作のマッピング

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] = min(dp[j-1], dp[j-8], dp[j-13]) + 1
（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 = float('inf’)
    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 = float('inf’)
        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	14	15	16	17	18	19	20	21	22	23	24	25
0	1	2	3	4	5	6	7	1	2	3	4	5	1	2	3	2	3	4	5	6	2	3	4	3	4

2次元になるテーブルでもやってみよう！

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）

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

$$[\text{和をSにできる}] = f([\text{n個の整数のprefix}], S)$$

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

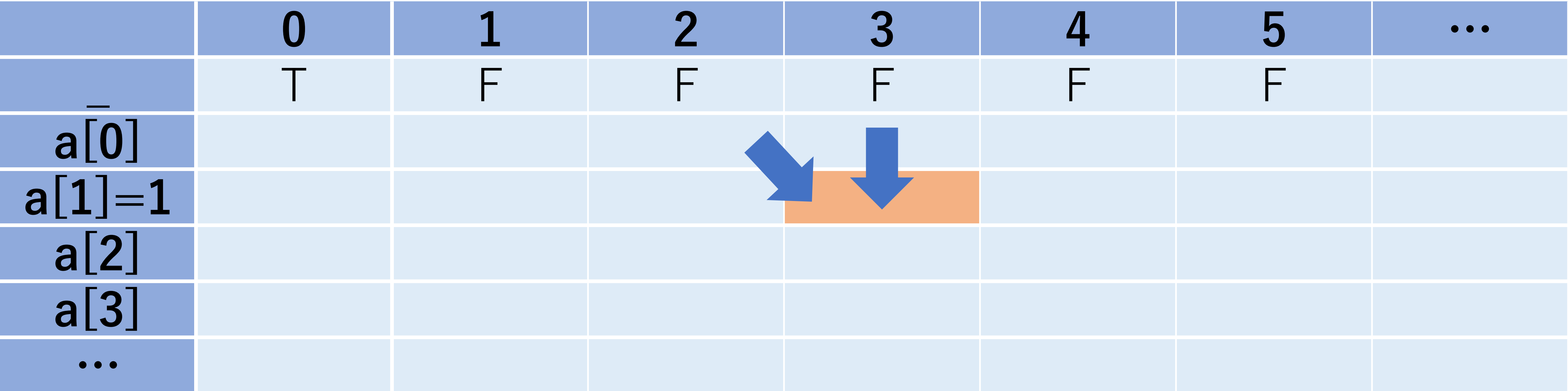
#2 DPテーブルの初期化
数字がなにもない状態なら取りうる部分和は0のみ.

	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）．

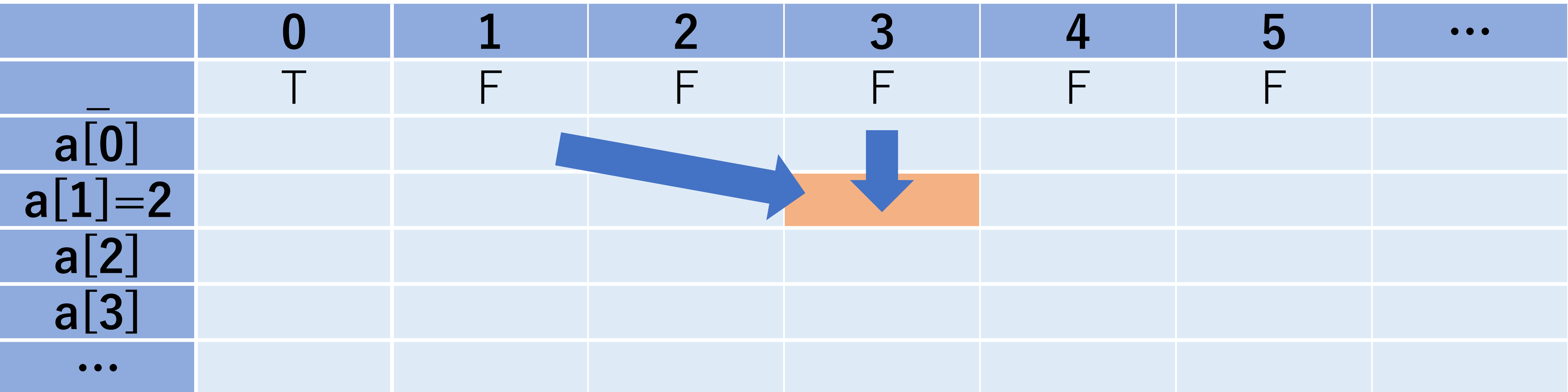


	0	1	2	3	4	5	…
_	T	F	F	F	F	F	
a[0]							
a[1]=1							
a[2]							
a[3]							
…							

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）．



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

2通りのパターンがある．

a[i]を入れる：dp[i][j] = dp[i-1][j-a[i]]

a[i]を入れない：dp[i][j] = dp[i-1][j]

この内，どちらか一方でもTrueならdp[i][j]もTrue．
そうでなければ，False．

（ただし，j-a[i]は0以上．そうでない場合，自動的に
dp[i][j] = dp[i-1][j]）

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．」

レーベンシュタイン距離

2つの文字列の「差」を表す距離.

ある文字列からもう1つの文字列に変換するために
必要な編集（追加，削除，置換）の最小回数を表す.

レーベンシュタイン距離

2つの文字列の「差」を表す距離.

例）static -> dynamic

最小回数の編集例：

static ->（置換）-> dtatic ->（置換）-> dyatic –>（追加）-> dynatic ->（置換）-> dynamic

よって，距離は4.

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押し込む.

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**

今回は最小の編集回数.

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

$$[\text{最小編集回数}] = f([\text{文字列1のprefix}], [\text{文字列2のprefix}])$$

矢谷式DPの考え方：#1 DPテーブルの設計

abcとadcdの距離を求める例を考える.

レーベンシュタイン距離の説明変数は2つの文字列.
例の場合を考えれば，abcとadcd.

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．

	a	d	c	d
a				
b				
c				

矢谷式DPの考え方：#1 DPテーブルの設計

この表のセルは，その行までの部分文字列と，その列までの部分文字列の距離を表す．

	a	d	c	d
a	aとa の距離			
b				
c				

矢谷式DPの考え方：#1 DPテーブルの設計

この表のセルは，その行までの部分文字列と，その列までの部分文字列の距離を表す．

	a	d	c	d
a		aとad の距離		
b				
c				

矢谷式DPの考え方：#1 DPテーブルの設計

この表のセルは，その行までの部分文字列と，その列までの部分文字列の距離を表す．

	a	d	c	d
a				
b	abとaの距離			
c				

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.

レーベンシュタイン距離でいえば，空文字列との比較.

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．最初の行と列は，空文字との距離を表す．

	_	a	d	c	d
_					
a					
b					
c					

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．最初の行と列は，空文字との距離を表す．

	_	a	d	c	d
_					
a	aと_				
b					
c					

矢谷式DPの考え方：#2 DPテーブルの初期化

「初期状態」のセルはすぐに求まる．ある文字列と空文字との距離は，ある文字列の長さに同じ．

	_	a	d	c	d
_	0	1	2	3	4
a	1				
b	2				
c	3				

矢谷式DPの考え方：#3 操作のマッピング

オレンジの行は，空文字からadcdの部分文字列への変更に必要な追加回数を表す．

	_	a	d	c	d
_	0	1	2	3	4
a	1				
b	2				
c	3				

矢谷式DPの考え方：#3 操作のマッピング

オレンジの列は，abcの部分文字列から空文字への変更に必要な削除回数を表す．

	_	a	d	c	d
_	0	1	2	3	4
a	1				
b	2				
c	3				

矢谷式DPの考え方：#3 操作のマッピング

右に1セル行くは「追加」，下に1セル行くは「削除」に対応することがわかる！

	_	a	d	c	d
_	0	1	2	3	4
a	1				
b	2				
c	3				

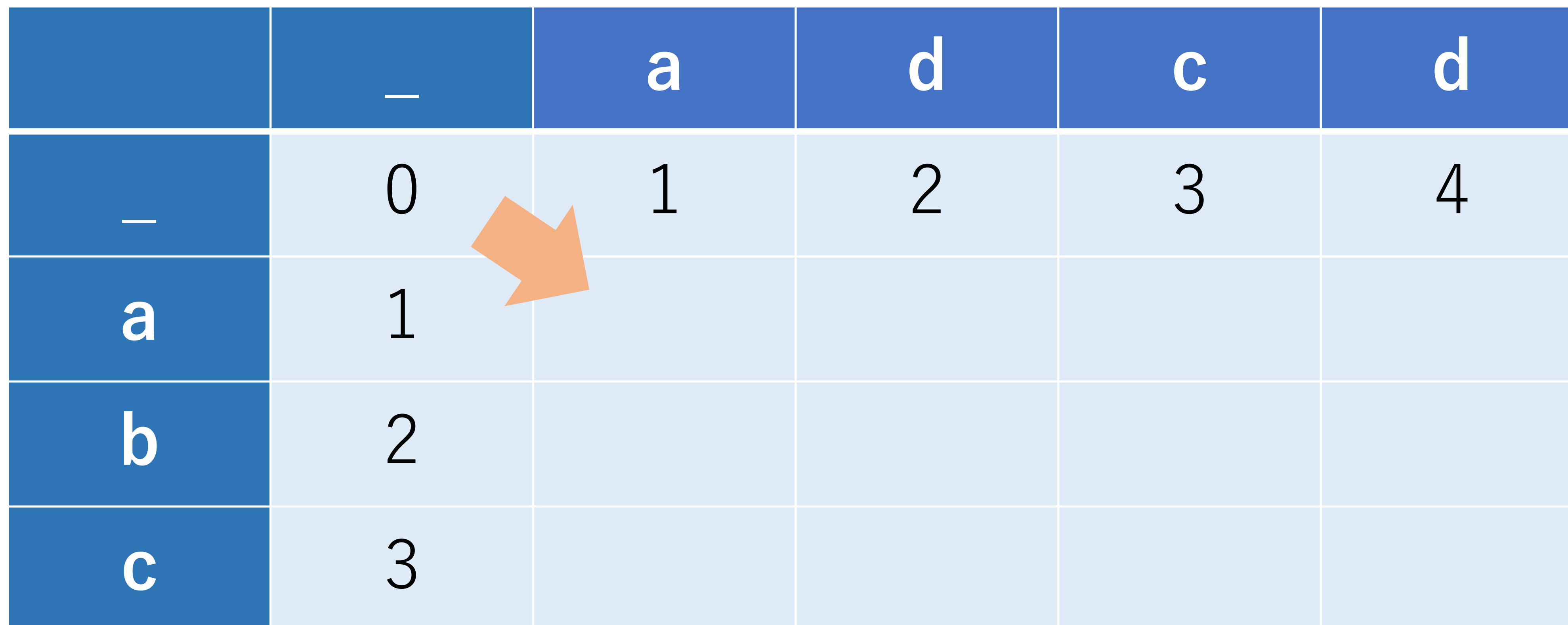
矢谷式DPの考え方：#3 操作のマッピング

では「置換」と「何もしない」は？

	_	a	d	c	d
_	0	1	2	3	4
a	1				
b	2				
c	3				

矢谷式DPの考え方：#3 操作のマッピング

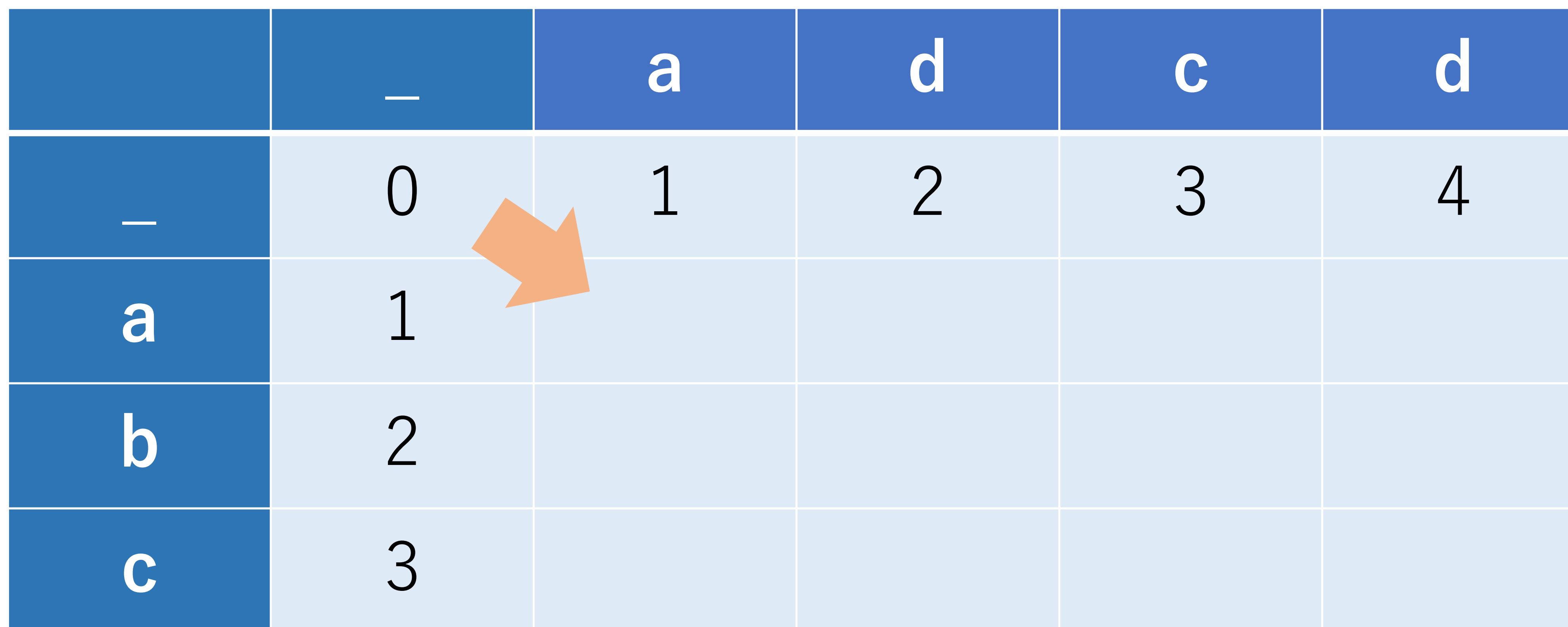
斜めに1セル移動することは，どちらの文字列も1つずつ進めることに対応する．



	_	a	d	c	d
_	0	1	2	3	4
a	1				
b	2				
c	3				

矢谷式DPの考え方：#3 操作のマッピング

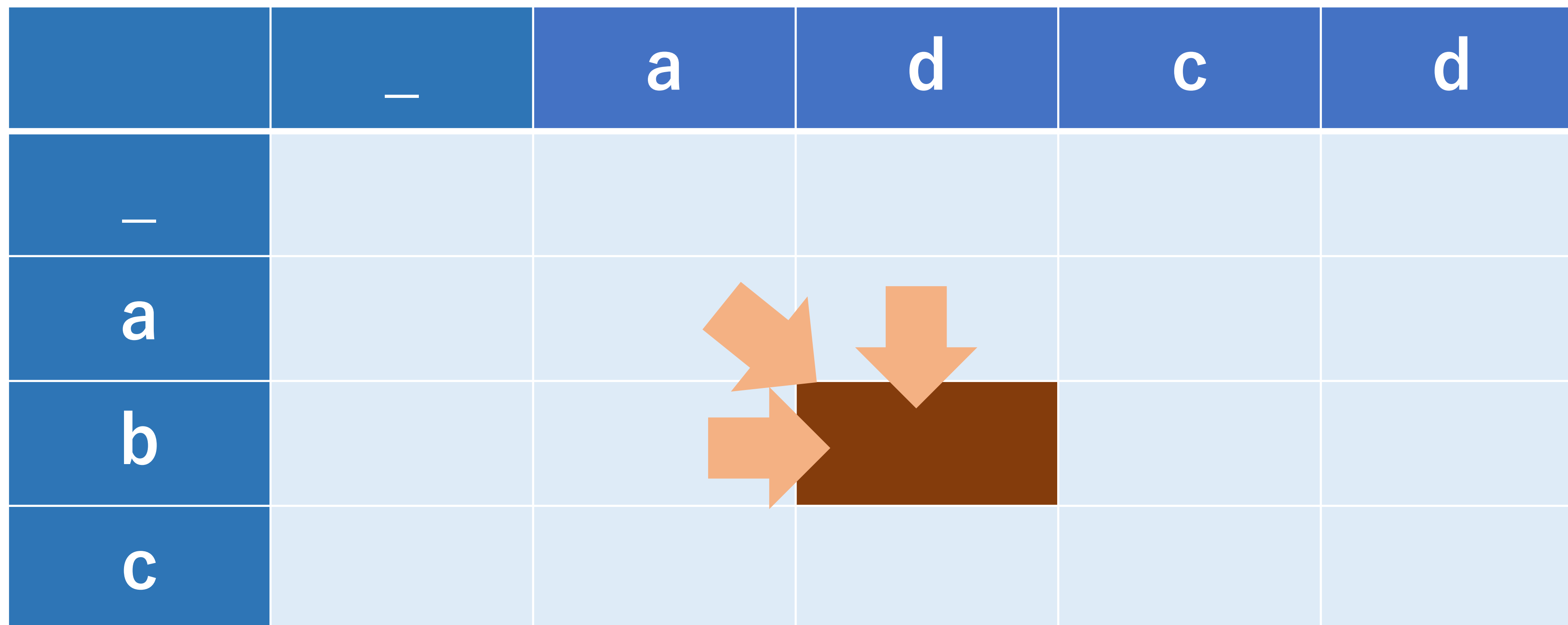
このときの行と列の文字が違う -> 置換
行と列の文字が同じ -> 何もしない（一致している）



	_	a	d	c	d
_	0	1	2	3	4
a	1				
b	2				
c	3				

矢谷式DPの考え方：#3 操作のマッピング

以上をまとめると，ある1つのセルに遷移する道としては3つのセルが絡んでいることになる．



矢谷式DPの考え方：#4 コード化

ここまでくればもう少し！この遷移を式で表せばよい．

追加，削除，置換のときは編集回数1回とカウントする．
つまり，dpに1を足すことになる．

追加：dp[i][j] = dp[i-1][j] + 1
削除：dp[i][j] = dp[i][j-1] + 1
置換：dp[i][j] = dp[i-1][j-1] + 1
何もしない：dp[i][j] = dp[i-1][j-1]

矢谷式DPの考え方：#4 コード化

今考えたいのは，最小の編集回数なので，この4つのうち
最小になるものだけ保持すれば良い．

つまり，行と列の文字が同じ場合には，

dp[i][j] = min(dp[i-1][j] + 1, dp[i][j-1] + 1, dp[i-1][j-1])

行と列の文字が違う場合には，

dp[i][j] = min(dp[i-1][j] + 1, dp[i][j-1] + 1, dp[i-1][j-1] + 1)

DPテーブルの更新例

オレンジのセルの場合は？

	_	a	d	c	d
_	0	1	2	3	4
a	1	?			
b	2				
c	3				

DPテーブルの更新例

関係するセルは薄オレンジのもの.
さらに，今回は文字が一致しているケース.

	_	a	d	c	d
_	0	1	2	3	4
a	1	?			
b	2				
c	3				

DPテーブルの更新例

よって，min(dp[1][0]+1, dp[0][1]+1, dp[0][0]) -> 0

	_	a	d	c	d
_	0	1	2	3	4
a	1	**0**			
b	2				
c	3				

DPテーブルの更新例

では，このオレンジのセルの場合は？

	_	a	d	c	d
_	0	1	2	3	4
a	1	0	?		
b	2				
c	3				

DPテーブルの更新例

関係するセルは薄オレンジのもの.
ただし，今回は文字が一致して<u>いない</u>ケース.

	_	a	d	c	d
_	0	1	2	3	4
a	1	0	?		
b	2				
c	3				

DPテーブルの更新例

よって， min(dp[1][1]+1, dp[0][2]+1, <u>dp[0][1]+1</u>) -> 1

	_	a	d	c	d
_	0	1	2	3	4
a	1	0	**1**		
b	2				
c	3				

DPテーブルの更新例

では，ここは？

	_	a	d	c	d
_	0	1	2	3	4
a	1	0	1		
b	2				
c	3				

DPテーブルの更新例

順番に埋めていくと，一番右下が求めたいものになる．

	_	a	d	c	d
_	0	1	2	3	4
a	1	0	1	2	3
b	2	1	1	2	3
c	3	2	2	1	**2**

実行例

levenshtein('similar', 'similarity') -> 3

levenshtein('similar', 'difference’) -> 9

文字の見た目上の類似度を計算するときなどにも使える.

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押し込む．

（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．．．）

貰うDPと配るDP

正式なアルゴリズム用語ではないですが，競技プログラミング界隈などは使われている表現.

貰うDP

「ある状態」を「1ステップ前の状態」から計算する

配るDP

「ある状態」から「1ステップ後の状態」を計算する

貰うDPと配るDP

矢谷式で説明したのは全て「貰うDP」です.

ただし，#3を以下のように書き換えれば，配るDPに変更することが出来ます.

#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<u>他のどのセル**へ**遷移できる</u>かを調べる.

矢谷式DPの考え方

#3 操作をDPテーブル上にマッピング（貰うDP）

品物5（重さ1，価値2）を入れるか入れないか

	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：ナップサックで配るDP

#3’ 操作をDPテーブル上にマッピング

品物5（重さ1，価値2）を入れるか入れないか

	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：ナップサックで配るDP

#3’ 操作をDPテーブル上にマッピング

2つのセルnote[j+1][w]とnote[j+1][k+weight[j]]が影響.

	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

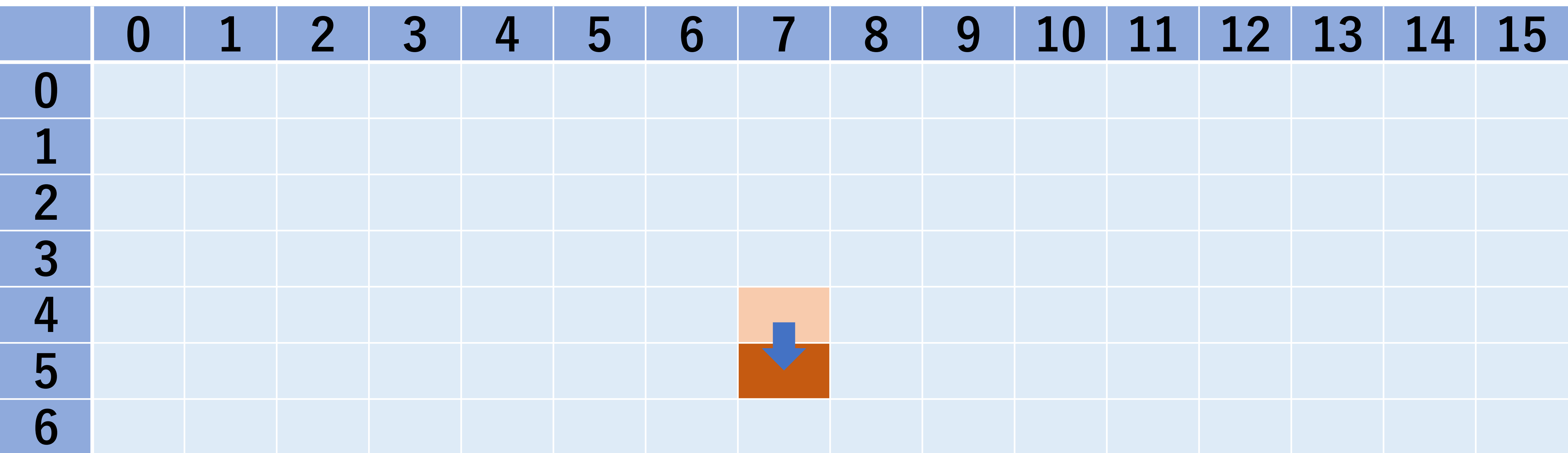
矢谷式DPの考え方：ナップサックで配るDP

#4 コード化

note[i+1][w]，note[i-1][w+weight[i]]の2つをnote[i][w]を使って更新する．更新前のものと比較して大きい方のみ残す．

矢谷式DPの考え方：ナップサックで配るDP

#4 note[i+1][w]の更新.
（入れない，入れられない場合を考慮することに相当．）



矢谷式DPの考え方：ナップサックで配るDP

#4 note[i+1][w]の更新.

すでにnote[i+1][w]にある値（他のケースですでにこの
セルが更新されている場合がある）とnote[i][w]（入れない,
入れられないケース）とを比較.

note[i+1][w]=max(note[i+1][w], note[i][w])

矢谷式DPの考え方：ナップサックで配るDP

#4 note[i+1][w+weight[i]]の更新.
（入れる場合を考慮することに相当．）

	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：ナップサックで配るDP

```
#4 note[i+1][w+weight[i]]の更新.

# すでにnote[i+1][w+weight[i]]にある値とnote[i][w]+value[i]
# （入れるケース）とを比較.

if w+weight[i] <= w_limit:
        note[i+1][w+weight[i]]=max(note[i+1][w+weight[i]],
                                   note[i][w]+value[i])
```

矢谷式DPの考え方：ナップサックで配るDP

```
def knapsack_distribute():
    [DPテーブルの初期化などは貰うDPと同じ]

    [i: 0からNまで,　w: 0からWまで]:
        note[i+1][w]=max(note[i+1][w], note[i][w])

        if w+weight[i] <= w_limit:
            note[i+1][w+weight[i]]=
            max(note[i+1][w+weight[i]], note[i][w]+value[i])
```

前回の実装例（ナップサックで貰うDP）

```
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]=note[i][w]
            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]
```

貰うDPと配るDPの違い

貰うDP

1回のループで更新するかどうかをチェックする
DPテーブルのセルは1つ.

配るDP

1回のループで更新するかどうかをチェックする
DPテーブルのセルは複数.

貰うDPと配るDPの違い

多くの問題ではどちらも大きく変わらない（はず）．

人によっては配るDPのほうが考えやすいかも．

1ステップ進むほうが，1ステップ立ち戻るより
やりやすいかも．

ただし，配る先が多い場合には注意．

貰うDP vs. 配るDP

問題文

N 人の子供たちがいます。 子供たちには $1, 2, \ldots, N$ と番号が振られています。

子供たちは K 個の飴を分け合うことにしました。 このとき、各 $i\,(1 \leq i \leq N)$ について、子供 i が受け取る飴の個数は 0 以上 a_i 以下でなければなりません。 また、飴が余ってはいけません。

子供たちが飴を分け合う方法は何通りでしょうか？ $10^9 + 7$ で割った余りを求めてください。 ただし、2 通りの方法が異なるとは、ある子供が存在し、その子供が受け取る飴の個数が異なることを言います。

制約

- 入力はすべて整数である。
- $1 \leq N \leq 100$
- $0 \leq K \leq 10^5$
- $0 \leq a_i \leq K$

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

配るDPで考えると,

DPテーブル：dp[i][j]

格納される値：i番目の子供まででアメj個を配る場合の数

i：子供

j：アメの個数

初期化

dp[i][0]=1：アメが0なら子供の数によらず配り方は1通り

dp[0][j]=0：子供に配る前は0通り

dp[i][j]=0で初期化.

配るDPで考えると,

dp[i]からdp[i+1]列への遷移で影響するセルは,

dp[i+1][j]：i+1の子供に0個配る（下限）
dp[i+1][j+1]：i+1の子供に1個配る
dp[i+1][j+2]：i+1の子供に2個配る
…
dp[i+1][j+a[i+1]]：i+1の子供にa[i+1]個配る（上限）

配るDPで考えると，

いま場合の数を考えているので，dp[i][j]にある値を以下の全てのセルに足してあげる．

dp[i+1][j]：i+1の子供に0個配る（下限）
dp[i+1][j+1]：i+1の子供に1個配る
dp[i+1][j+2]：i+1の子供に2個配る
…
dp[i+1][j+a[i+1]]：i+1の子供にa[i+1]個配る（上限）

配るDPで考えると,

```
[dpテーブル初期化]
for i in range(len(a)): # 子供のループ
    for j in range(drops+1): # 飴のループ
        for k in range(a[i]+1): # 各子供に配れる数のループ
            if j+k < drops+1:
                if (j + k) == 0: dp[i+1][0] = 1
                else:
                    dp[i+1][j+k] += dp[i][j]
```

配るDPで考えると,

ループ3重なのでdropsやa[i]の値が大きくなるとかなり重い.

```
[dpテーブル初期化]
for i in range(len(a)):
    for j in range(drops+1):
        for k in range(a[i]+1):
            if j+k < drops+1:
                if (j + k) == 0: dp[i+1][0] = 1
                else:
                    dp[i+1][j+k] += dp[i][j]
```

貰うDPで考えると，

dp[i][j]の値になるのは，dp[i-1][j-a[i]]からdp[i-1][j]までの和.

→子供iに0からa[i]個のアメを配り，配布総数がjになる場合の総和.

貰うDPで考えると，

dp[i][j]の値になるのは，dp[i-1][j-a[i]]からdp[i-1][j]までの和.

dp[i][j+1]の値になるのは，dp[i-1][j+1-a[i]]からdp[i-1][j+1]までの和.

dp[i][j+2]の値になるのは，dp[i-1][j+2-a[i]]からdp[i-1][j+2]までの和.

…

貰うDPで考えると,

dp[i][j]の値になるのは，dp[i-1][j-a[i]]からdp[i-1][j]までの和.

dp[i][j+1]の値になるのは，dp[i-1][j+1-a[i]]からdp[i-1][j+1]までの和.

dp[i][j+2]の値になるのは，dp[i-1][j+2-a[i]]からdp[i-1][j+2]までの和.

…

貰うDPで考えると，

dp[i][j]の値になるのは，dp[i-1][j-a[i]]からdp[i-1][j]までの和.

dp[i][j+1]の値になるのは，dp[i-1][j+1-a[i]]からdp[i-1][j+1]までの和.

dp[i][j+2]の値になるのは，dp[i-1][j+2-a[i]]からdp[i-1][j+2]までの和.

累積和の時間ですー. 😉

計算量の比較

子供の総数がN，飴の総数がK，さらに各子供が受け取ることのできる飴の数の最大がK.

計算量の比較

子供の総数がN，飴の総数がK，さらに各子供が受け取ることのできる飴の数の最大がK.

配るDPの場合，ループが3つあるので，$O(NK^2)$. 😭

計算量の比較

子供の総数がN，飴の総数がK，さらに各子供が受け取ることのできる飴の数の最大がK.

配るDPの場合，ループが3つあるので，$O(NK^2)$. 😭

貰うDPの場合，各子供に対して最初に累積和を準備しておくことが必要で，以降は定数回の処理．したがって，$O(N(K+K)) \to O(NK)$. 😊

パフォーマンスの比較

drops = 10000　　　　# アメの総数
a = [10000, 10000, 10000, 10000]　　# 子供の配列
の場合,

実行時間（一例）
　　配るDP：183 [sec]
　　貰うDP：0.05 [sec]

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押し込む.

（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．．．）

矢谷式DPの考え方

#1 DP**テーブルを設計する.**

まずはヒューリスティックスにしたがって.

#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.**

具体的な小さめの例から考えて一般化する.

TLEしちゃう時は？

配るから貰うへの変換を考える.

その他，計算量を減らそうなところを見る.

DPテーブルの大きさを確認する.

行や列があまりにも長すぎる場合には，格納している値と入れ替えてみる.

ナップサック問題 その2

「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．」
（問題文自体は前回紹介したものと全く同じ．）

w_i やWがものすごく大きい（例えば，10^9）場合は？
（価値の総和が取りうる値はWよりもかなり小さいと仮定）

$O(nW)$なので，Wが大きいと計算が大変．😭

ナップサック問題 その2

dp[i][sum_v] = min_w

i：品物

sum_v：価値の総和

min_w：品物iまで入れたときに価値の総和がsum_vになる時の最小の重さの総和

値がとても大きくなる可能性があるものをセルにするという設計をする．この場合，Vを取りうる価値の最大値とすると，計算量は$O(nV)$なので速くなる．

ナップサック問題 その2

初期化に気をつける．一番最初にセルに入っておくべき値はなにか？（最小を求める，がヒント．）

明らかに最小の重さの総和がわかるケースはどこにある？

求める解はDPテーブルのどこにある？

まとめ

矢谷式DPの解き方

スタンダードなものはこれでやってみる．

DPの高速化が必要な場合

DPには色んなバリエーションがあるので，ぜひ調べてみてください．

コードチャレンジ：基本課題#7-a [1.5点]

ナップサック問題その2を解くコードを漸化式方式で書いてください.

コードチャレンジ：基本課題#7-b [1.5点]

貰うDP vs. 配るDPで検討した問題において，配るDPから貰うDPへと変更し，3重ループを2重ループにして，高速化を実現してください．（漸化式方式で実装してください．）

10^9+7の剰余で答えを求めてください．

配るDPも実装して，パフォーマンス比較をローカル環境でしてみてください．

コードチャレンジ：Extra課題#7 [3点]

DPを使う問題．どのように使うとよいか，いろいろ考えてみてください．