Algorithms (2022 Summer)
#8：動的計画法2

矢谷 浩司

今日の目標

前回学んだことのちょっとした復習＆追加の説明.

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. 😆

動的計画法（前回からの再掲）

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（前回からの再掲）

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.

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．．．）

ナップサック問題（前回からの再掲）

「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．」

前回は矢谷式DPの考え方に従い，漸化式方式で実装.

ナップサック問題のメモ化再帰での実装

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.

ナップサック問題のメモ化再帰での実装

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．

ナップサック問題のメモ化再帰での実装

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．

iは0から$n-1$まで，wは0からWまでの値を取りうる．

漸化式方式のときと違い，初期状態の行は不要．

メモ化再帰を使った場合の実装例

```
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(weight))]
```

メモ化再帰を使った場合の実装例

```
# 再帰で呼び出す関数
# cur_i：その時点で考慮したアイテムのインデックス
# 後ろからいくので，最初はcur_i = 5
# cur_w：その時点での重さの上限
# 最初はcur_w = 15
def knapsack_rec(cur_i, cur_w):
```

メモ化再帰を使った場合の実装例

```
def knapsack_rec(cur_i, cur_w):
    # すべての品物を考慮したあとは価値が増える
    # ことはない
    if cur_i < 0: return 0

    # メモがあれば，すぐさまそれを使う
    if note[cur_i][cur_w] > -1: return note[cur_i][cur_w]
```

メモ化再帰を使った場合の実装例

```
def knapsack_rec(cur_i, cur_w):
    ...

    # 今考えているアイテムが現在の入れられる残りの
    # 重さを超えている場合は，入れずに1つ前に.
    if w < weight[cur_i]:
        note[cur_i][w] = knapsack_rec(i-1, cur_w)
        return note[cur_i][cur_w]
```

メモ化再帰を使った場合の実装例

```
def knapsack_rec(cur_i, cur_w):
        …
        else: #not_inは入れない，is_inは入れる場合
                not_in = knapsack_rec(i-1, cur_w)
                is_in = knapsack_rec(i-1, cur_w-weight[i])+ value[i]
                # より大きい方をメモに残す
                note[i][cur_w] = max(not_in, is_in)
                return note[i][cur_w]
```

実行例

```
knapsack_rec(5, 15)


----
20
```

メモ化再帰を使ったナップサック

あるcur_iとcur_wに対して，knapsack_rec(cur_i, cur_w)の計算は1回になる．

よって，$O(NW)$．Nは品物の総数．
　　漸化式方式と同じ．

ナイーブな解法だと$O(2^N)$なので，だいぶマシ．

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.

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dp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	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

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.

私個人としては,

漸化式方式をおすすめ！

ループで書けちゃう美しさが良い.
　　あとから見たときにコードの挙動がわかりやすい.

再帰にまつわる細かな制約を気にしなくて良い.
　　再帰上限回数やスタックオーバーフローなど.

	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

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

もっといろんな問題を見てみよう！😁

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：

sタtatic ->（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落とし込む.

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

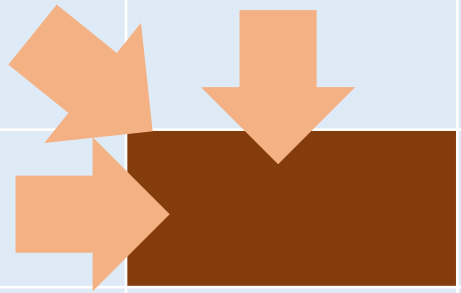
このときの行と列の文字が違う -> 置換
行と列の文字が同じ -> 何もしない（一致している）

	_	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．

	_	a	d	c	d
_					
a					
b					
c					



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いないケース．

	_	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, dp[0][1]+1) -> 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.

動的伸縮法（Dynamic Time Warping, DTW）

2つの時系列データの類似度を測る時に利用.

波形の類似性を定量的に評価したいなど.

この2つのデータは長さが異なっていても良い.

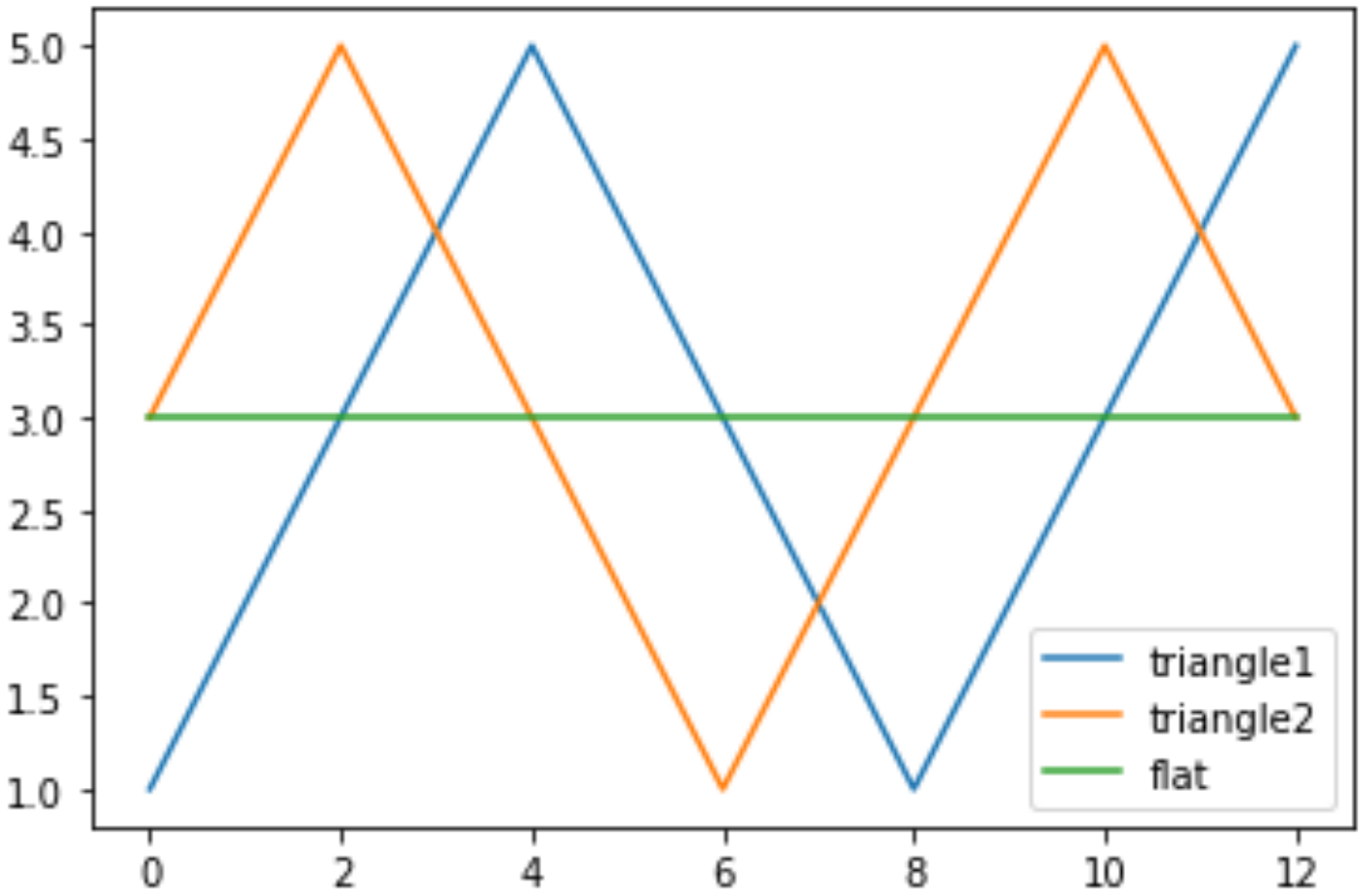
弾性マッチング（elastic matching）とも呼ばれる.

DPマッチングと呼ばれることもあるが日本語だけ？

音声認識などのナイーブな方法としても知られる.

時系列データ間の距離

以下のような3つの時系列データ（triangle1，triangle2，flat）を考える．



時系列データの距離

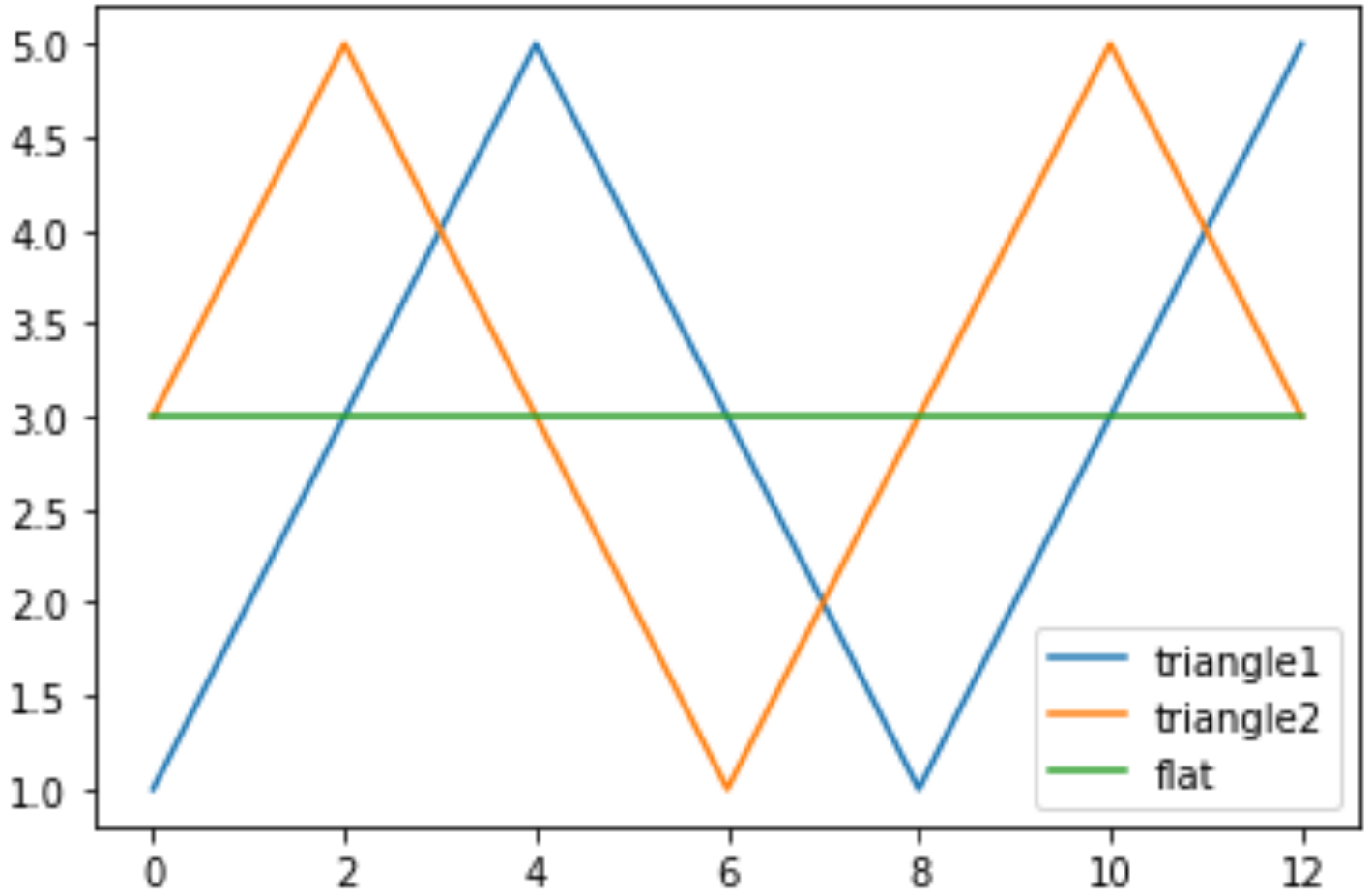
3つの時系列データは離散化しており，各々，

tri1 = [1, 2, 3, 4, 5, 4, 3, 2, 1, 2, 3, 4, 5]
tri2 = [3, 4, 5, 4, 3, 2, 1, 2, 3, 4, 5, 4, 3]
flat = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]

となっているとする．

時系列データ間の距離

感覚的には，triangle1はflatよりはtriangle2に近いと言いたい．



時系列データ間の距離

平均二乗誤差で2つの時系列データ間の距離を計算してみる.

これは時系列データのi番目の要素同士の距離を
合算し，平均を取ったもの.

```
def mse(x,y):
    L = min(len(x), len(y)), tmp = 0
    for i in range(L):
        tmp += (x[i] - y[i]) ** 2
    return tmp/L
```

時系列データ間の距離

平均二乗誤差での計算結果.

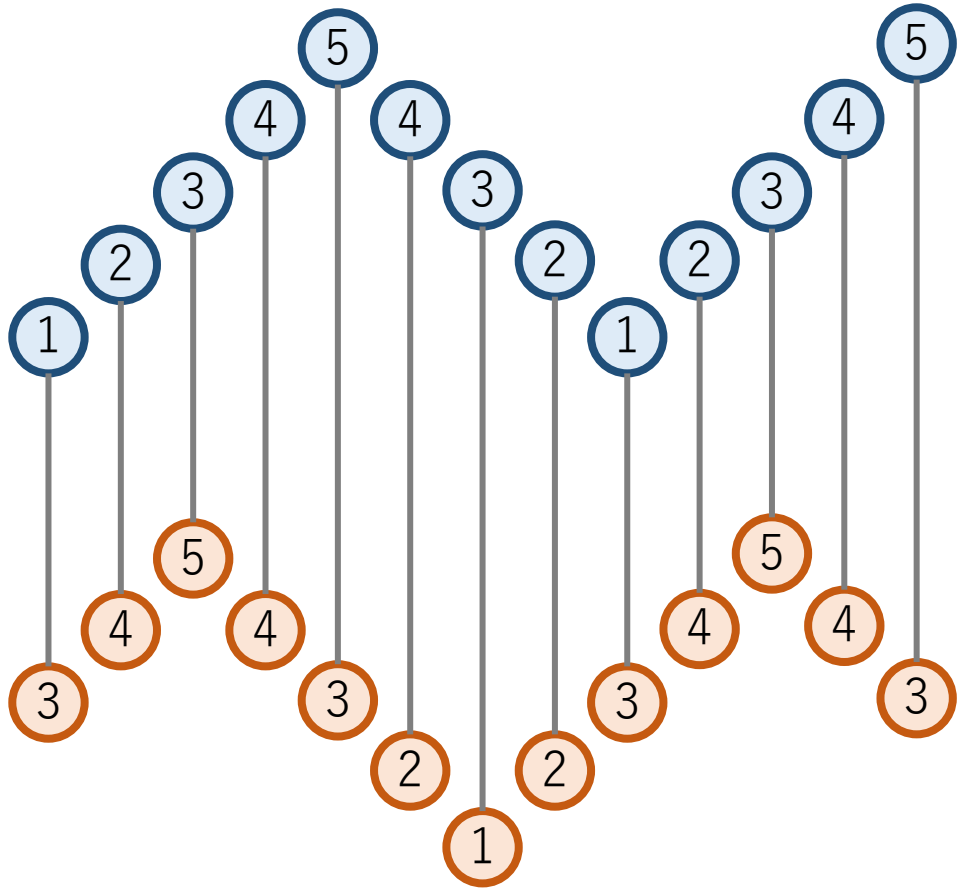
mse(tri1, tri2) -> 3.08

mse(tri1, flat) -> 1.69

平均二乗誤差を使うと，triangle1はtriangle2よりもflatの方が近いという結果に．．．😭

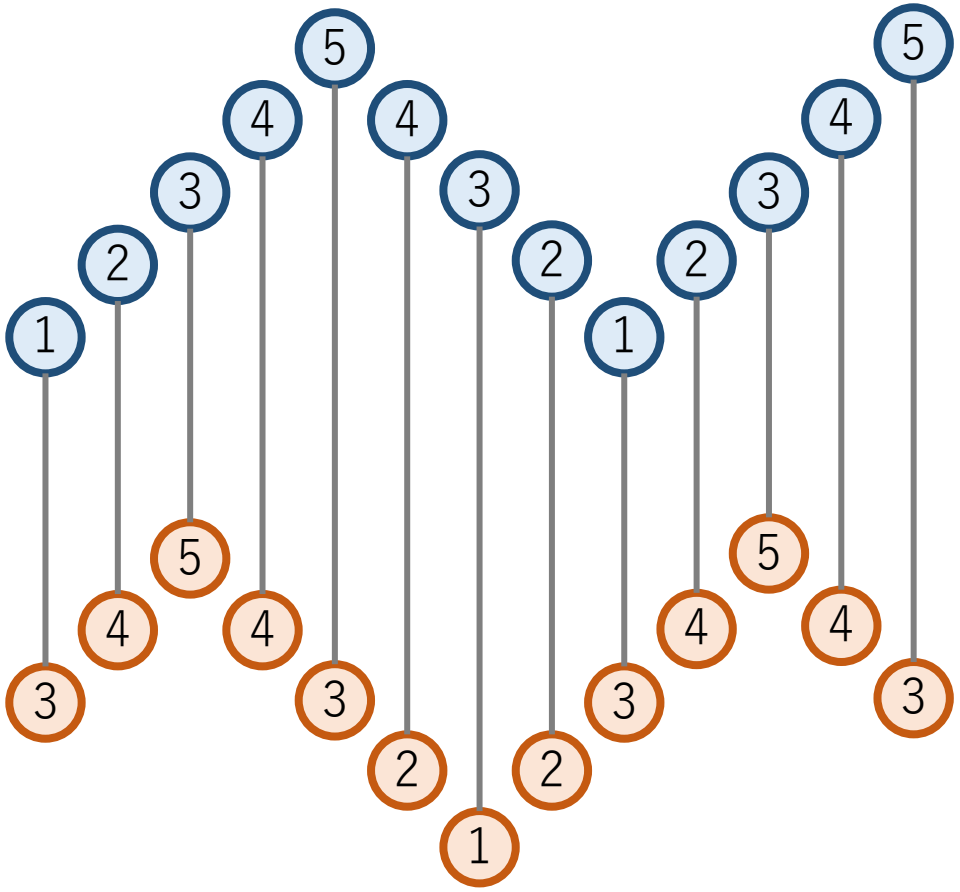
時系列データ間の距離

平均二乗誤差では，時系列データ間の要素に関して，以下の図に示すようなペア組1通りしか考えない．



時系列データ間の距離

もっと，よいペア組はないだろうか？



動的時間伸縮法

2つの時系列データの要素同士の全てのペア組を考え，
距離の総和が最も小さいものを選ぶ．

2つのデータは配列a，bとして表されている．

動的時間伸縮法

ペア組は1対多でもよい.

1つの時系列データのある要素は，もう1つの時系列データの複数の要素とペア組になってもよい.

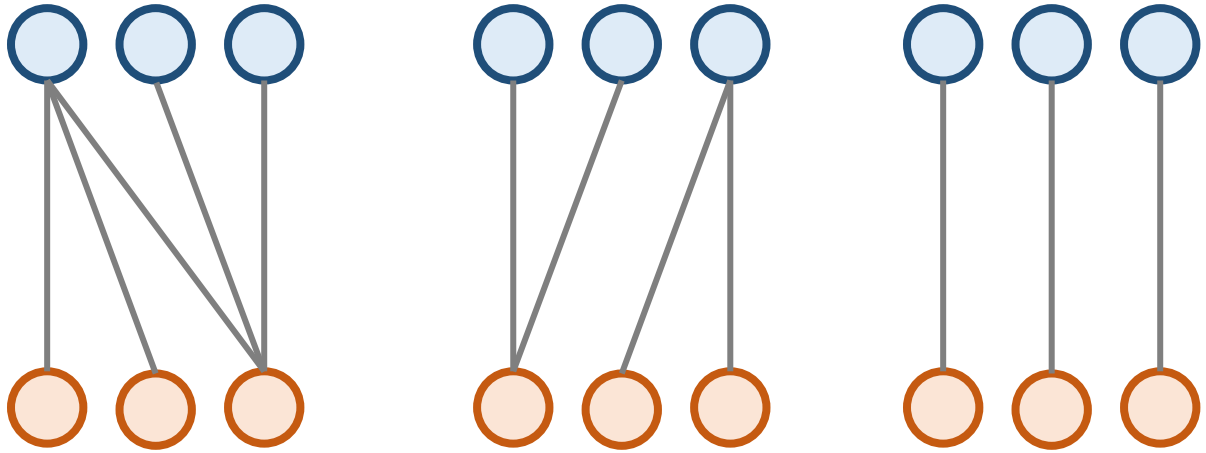
ただし，全てのi，jに対して，a[i]とb[j]がペア組されている場合，以下が成立する.

a[i+1]以降の要素は，b[j]かそれ以降の要素に，
b[j+1]以降の要素は，a[i]かそれ以降の要素にしかペア組されない.

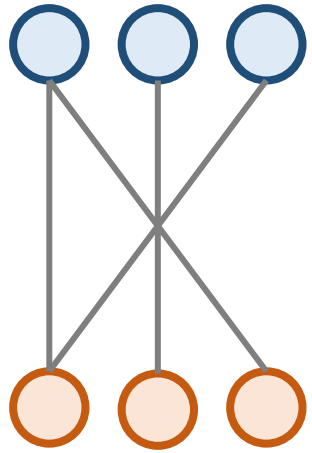
→つまり，「交差」するようなことはない.

動的時間伸縮法

「さかのぼってつながる」というようなペア組は起きない.



ありえるペア組の例



ありえないペア組の例

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.

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

今回は距離の総和のうち最小のもの.

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

[距離の総和の最小] =
$f([\text{時系列データ1のprefix}], [\text{時系列データ2のprefix}])$

以下では，2つの時系列データをa，b（配列）とする.

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．

	b[0]	b[1]	b[2]	b[3]	b[4]
a[0]					
a[1]					
a[2]					
a[3]					
a[4]					

矢谷式DPの考え方：#1 DPテーブルの設計

以下のオレンジのセルには，a[1]とb[1]までのペア組をした上で，距離の総和の最小値が入る．

	b[0]	b[1]	b[2]	b[3]	b[4]
a[0]					
a[1]					
a[2]					
a[3]					
a[4]					

矢谷式DPの考え方：#1 DPテーブルの設計

仮にtri1，tri2の最初の5つの値を入れたとすると，
こうなる．

	b[0]=3	b[1]=4	b[2]=5	b[3]=4	b[4]=3
a[0]=1					
a[1]=2					
a[2]=3					
a[3]=4					
a[4]=5					

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.

今回は，少なくとも片方のデータの要素が1つしかない場合が該当する.

→要素が1つずつしかないので，ペア組の仕方も自動的に1通りしかない.

矢谷式DPの考え方：#2 DPテーブルの初期化

以下のオレンジのセルには，a[0]とb[0]までのペア組をした上で，距離の総和の最小値が入る．

	b[0]=3	b[1]=4	b[2]=5	b[3]=4	b[4]=3
a[0]=1					
a[1]=2					
a[2]=3					
a[3]=4					
a[4]=5					

矢谷式DPの考え方：#2 DPテーブルの初期化

これは1（a[0]）と3（b[0]）の距離しかなので，
4（(1-3) ** 2）となる．

	b[0]=3	b[1]=4	b[2]=5	b[3]=4	b[4]=3
a[0]=1	4				
a[1]=2					
a[2]=3					
a[3]=4					
a[4]=5					

矢谷式DPの考え方：#2 DPテーブルの初期化

続いて，1行目を考えると，a[0]とbとの距離を考えることになる．

	b[0]=3	b[1]=4	b[2]=5	b[3]=4	b[4]=3
a[0]=1	4				
a[1]=2					
a[2]=3					
a[3]=4					
a[4]=5					

矢谷式DPの考え方：#2 DPテーブルの初期化

これもaの要素を全てb[0]にペア組するしかないので,
自動的に埋められる.

	b[0]=3	b[1]=4	b[2]=5	b[3]=4	b[4]=3
a[0]=1	4	13	29	38	42
a[1]=2					
a[2]=3					
a[3]=4					
a[4]=5					

矢谷式DPの考え方：#2 DPテーブルの初期化

1列目も同様に考えられる（aとb[0]との距離）．

	b[0]=3	b[1]=4	b[2]=5	b[3]=4	b[4]=3
a[0]=1	4	13	29	38	42
a[1]=2	5				
a[2]=3	5				
a[3]=4	6				
a[4]=5	10				

矢谷式DPの考え方：#3 操作のマッピング

1ステップでの操作の後，a[i]とb[j]まで考えた状況に至るためにはどのような状況があるかを考える．

a[i]とb[j]まで考えた状況は，DPテーブルのセル（i, j）に該当する．

この場合，1ステップはペア組1つを行うことに該当．

つまり，ペア組を1つ行ったあと，a[i]とb[j]までの全ての要素のペア組ができている状態になる．

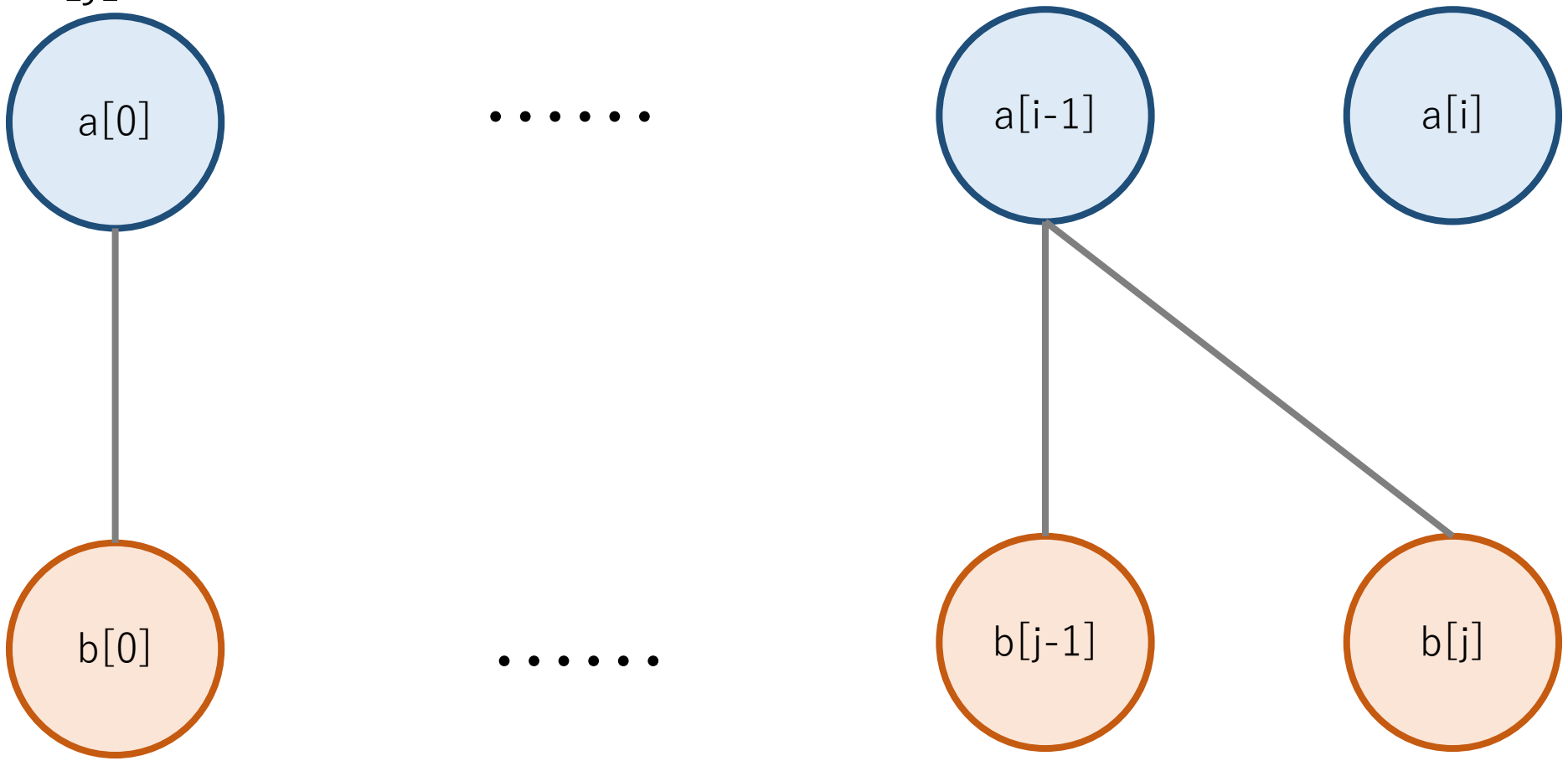
矢谷式DPの考え方：#3 操作のマッピング

この状況に1ステップで至るのは，以下の3つのケース.

1. a[i-1]とb[j]まではすでにペア組が終わっていて，
   a[i]とb[j]をペアとする.
2. a[i]とb[j-1]まではすでにペア組が終わっていて，
   a[i]とb[j]をペアとする.
3. a[i-1]とb[j-1]まではすでにペア組が終わっていて，
   a[i]とb[j]をペアとする.

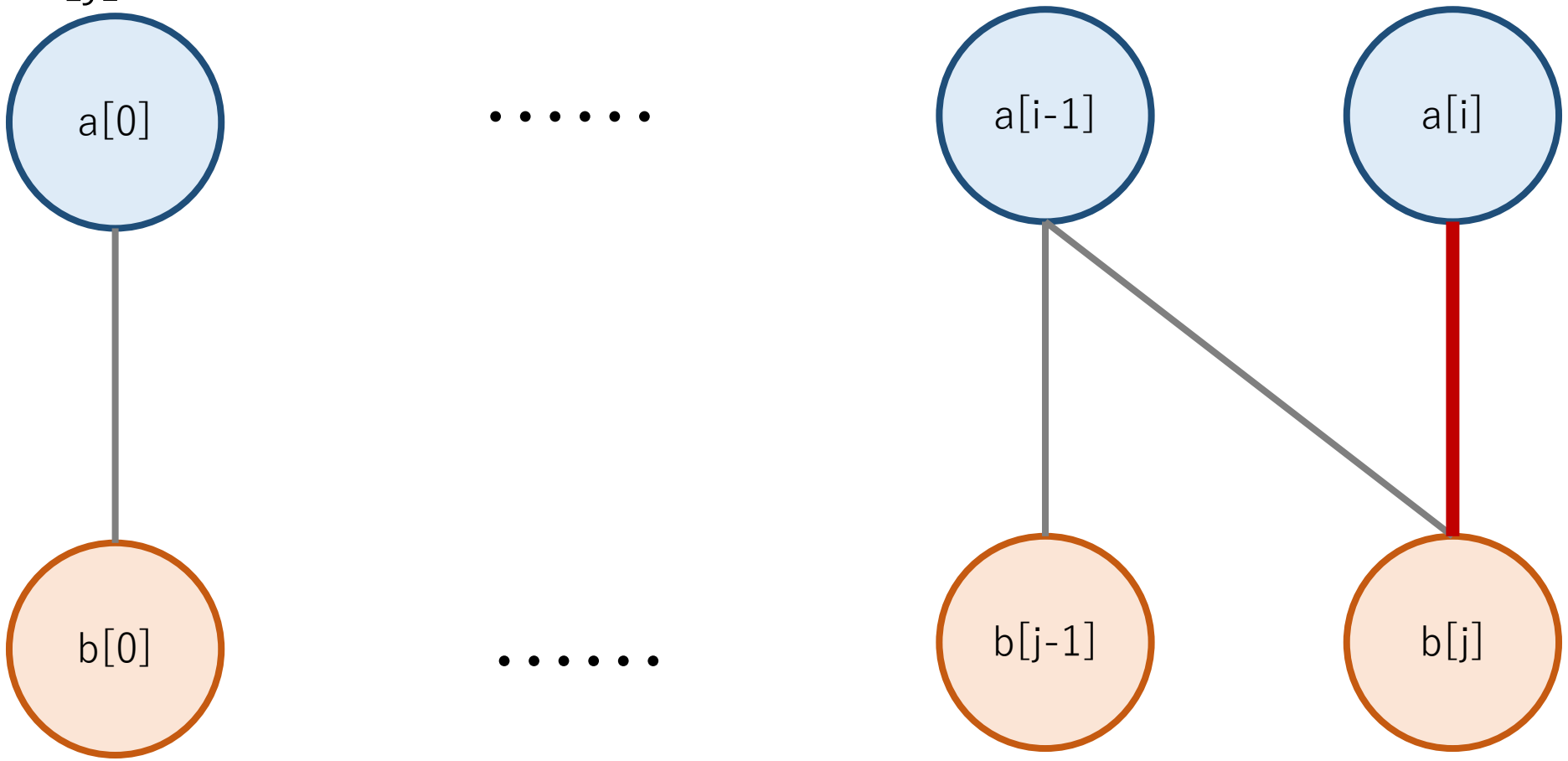
矢谷式DPの考え方：#3 操作のマッピング

1. a[i-1]とb[j]まではすでにペア組が終わっていて，
a[i]とb[j]をペアとする



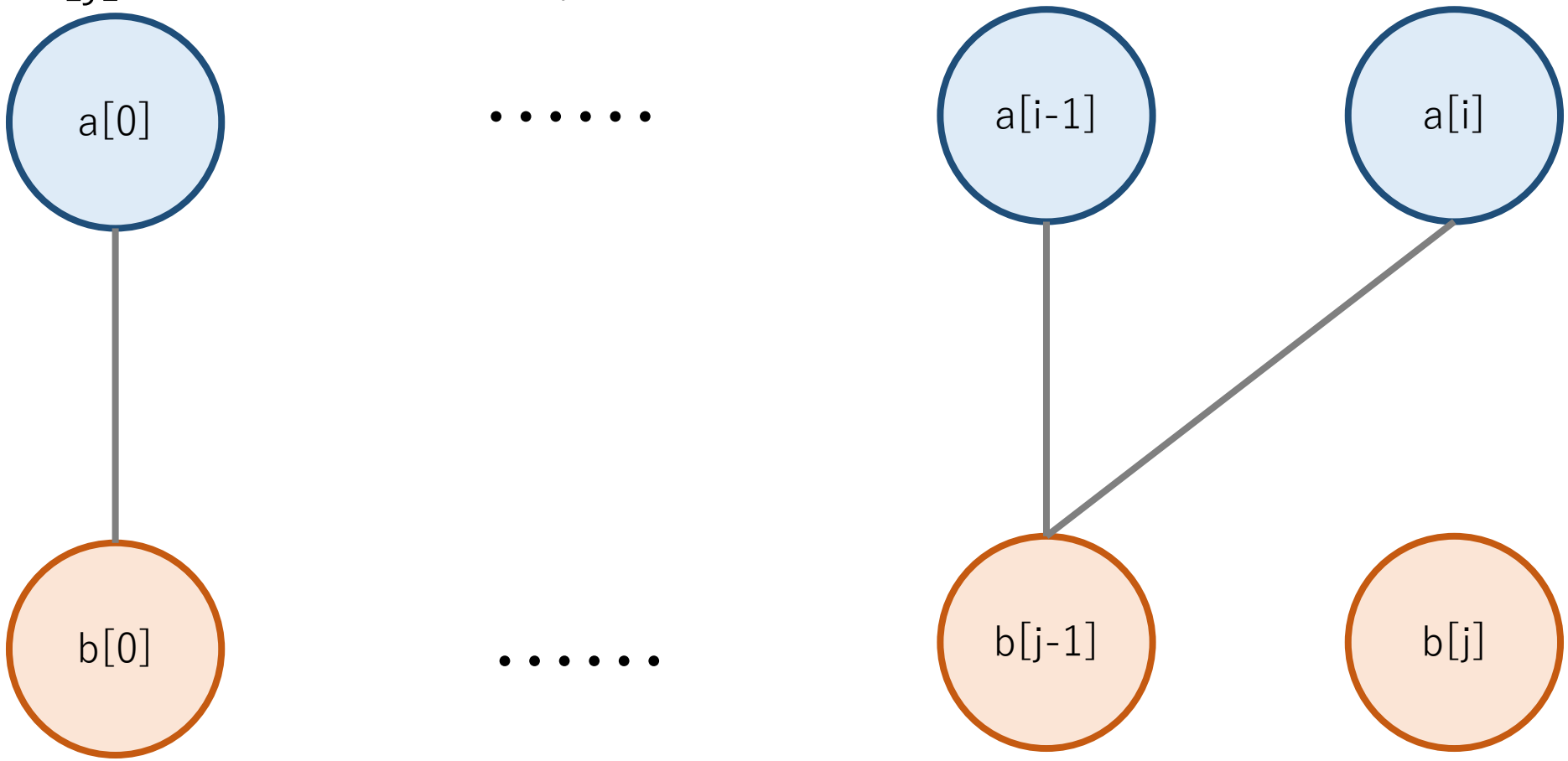
矢谷式DPの考え方：#3 操作のマッピング

1. a[i-1]とb[j]まではすでにペア組が終わっていて，
a[i]とb[j]をペアとする



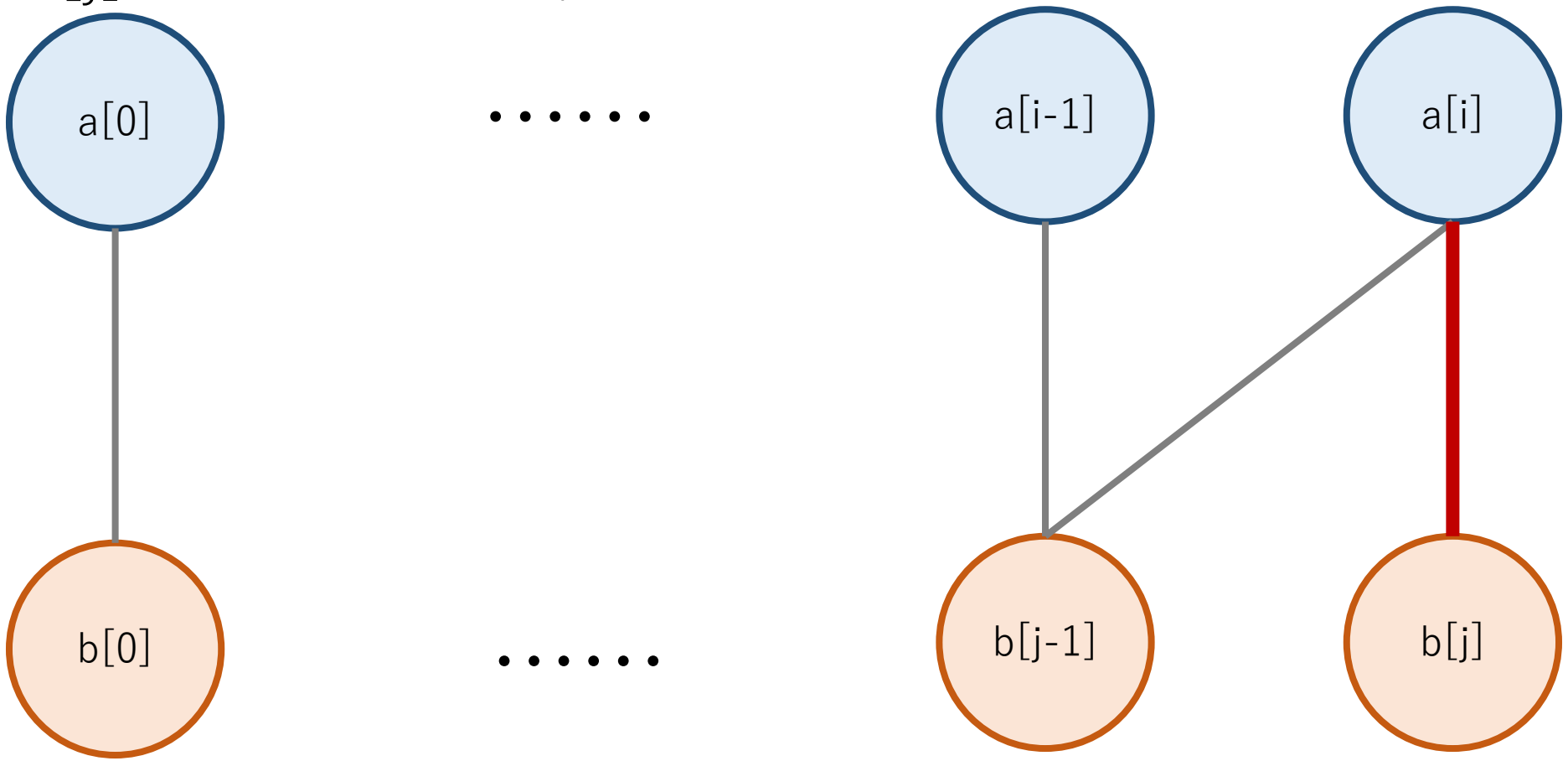
矢谷式DPの考え方：#3 操作のマッピング

2. a[i]とb[j-1]まではすでにペア組が終わっていて，
a[i]とb[j]をペアとする．



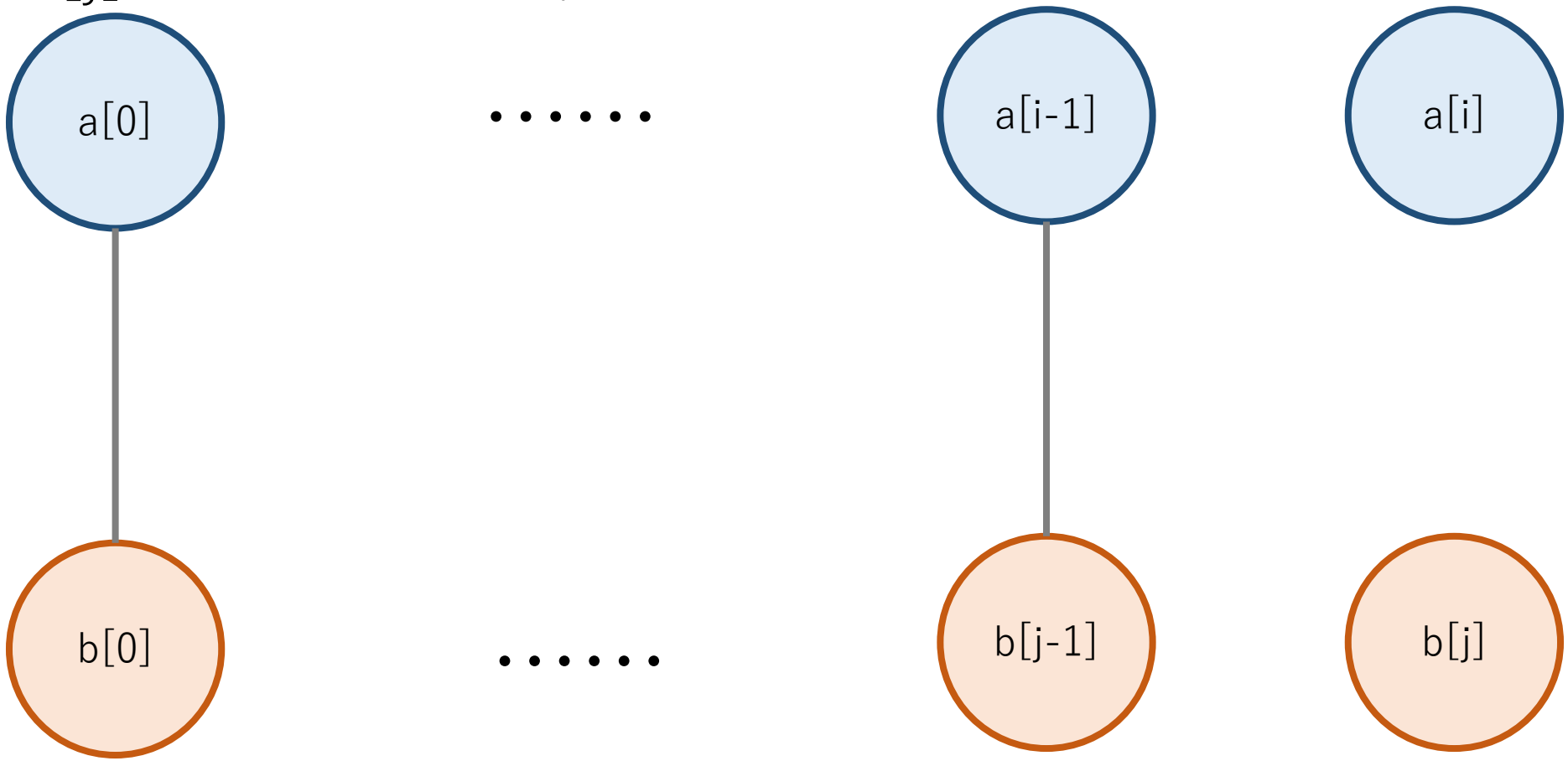
矢谷式DPの考え方：#3 操作のマッピング

2. a[i]とb[j-1]まではすでにペア組が終わっていて，
a[i]とb[j]をペアとする．



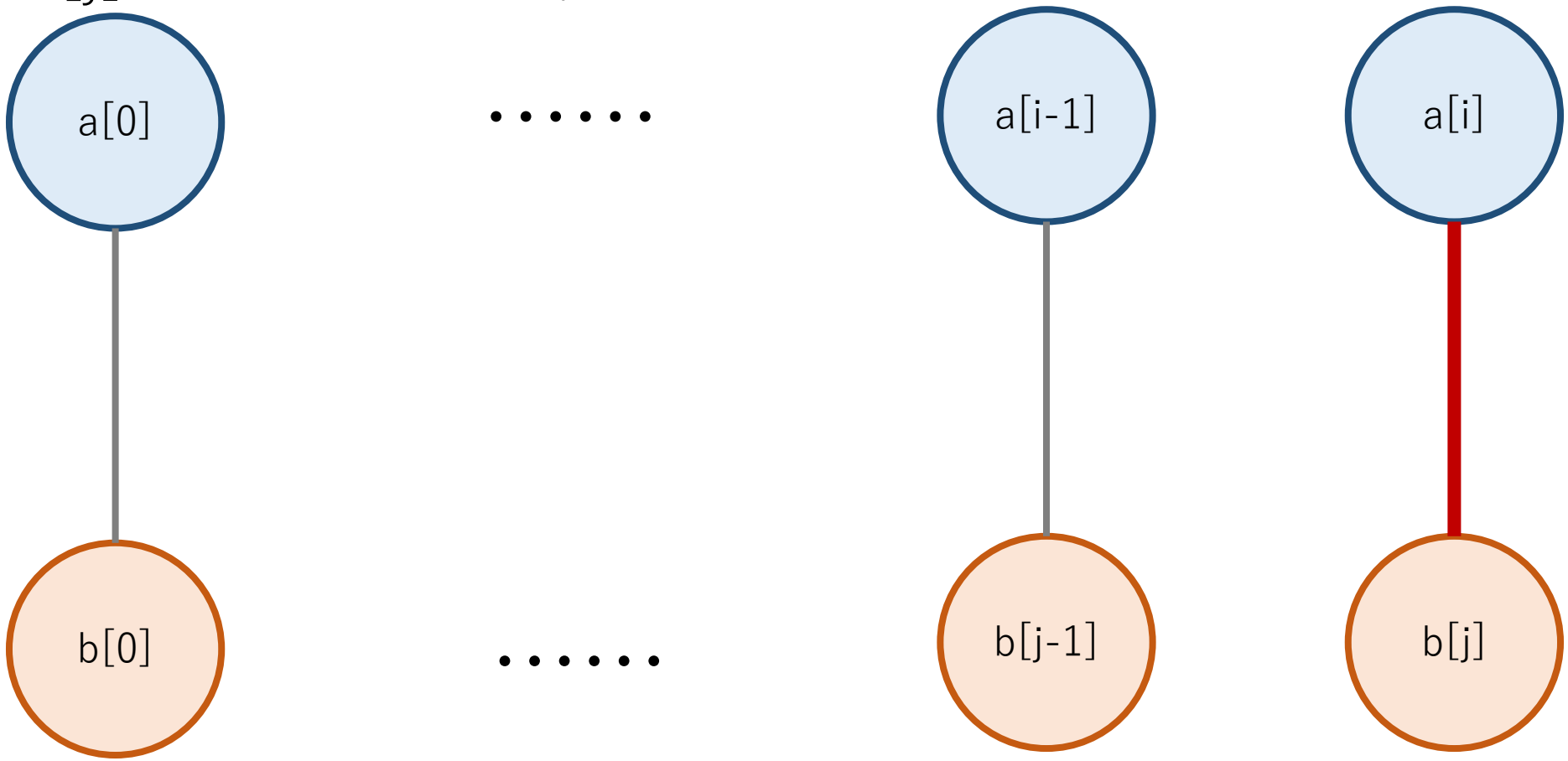
矢谷式DPの考え方：#3 操作のマッピング

3. a[i-1]とb[j-1]まではすでにペア組が終わっていて，
a[i]とb[j]をペアとする．



矢谷式DPの考え方：#3 操作のマッピング

3. a[i-1]とb[j-1]まではすでにペア組が終わっていて，
a[i]とb[j]をペアとする．



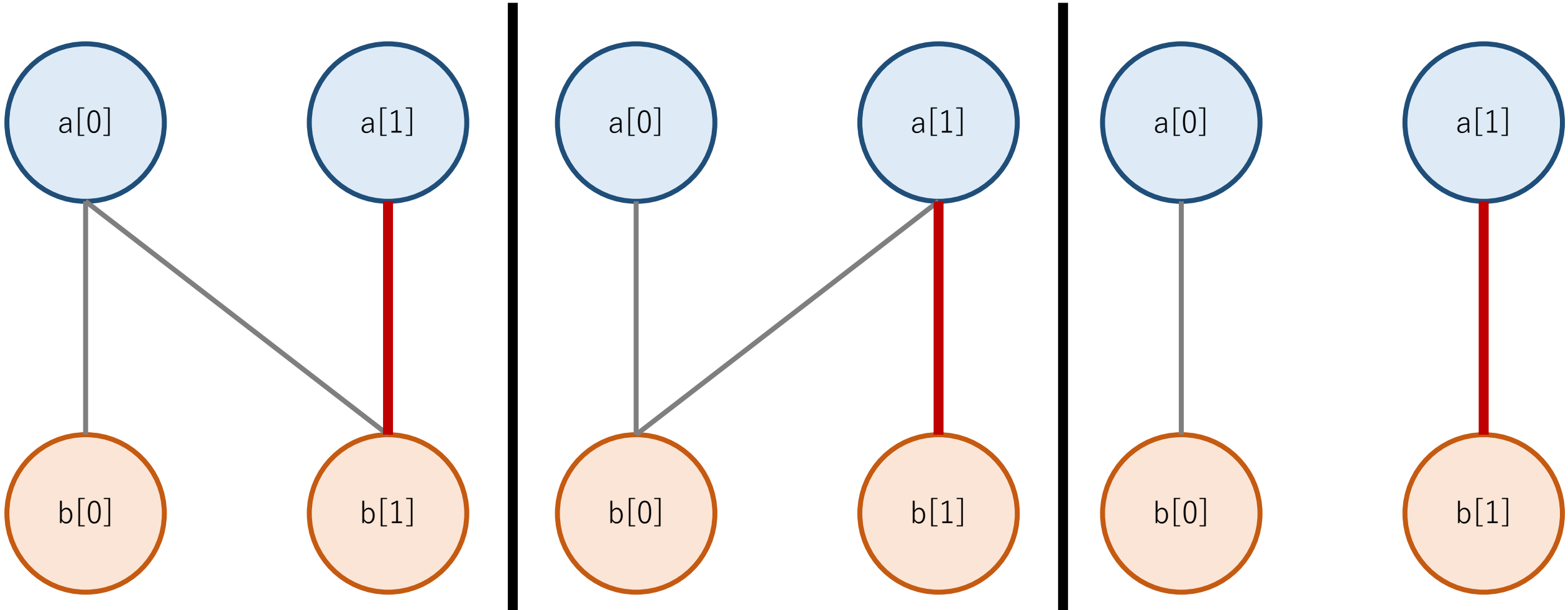
矢谷式DPの考え方：#3 操作のマッピング

具体的な例として，オレンジのセルを考えよう．ここは a[1]とb[1]までのペア組をした状況を表している．

	b[0]=3	b[1]=4	b[2]=5	b[3]=4	b[4]=3
a[0]=1	4	13	29	38	42
a[1]=2	5				
a[2]=3	5				
a[3]=4	6				
a[4]=5	10				

矢谷式DPの考え方：#3 操作のマッピング

この場合，以下の3通りがある．



矢谷式DPの考え方：#3 操作のマッピング

これは，隣接する3つのセルから遷移できる．

	b[0]=3	b[1]=4	b[2]=5	b[3]=4	b[4]=3
a[0]=1	4	13	29	38	42
a[1]=2	5				
a[2]=3	5				
a[3]=4	6				
a[4]=5	10				

矢谷式DPの考え方：#3 操作のマッピング

これらのセルの値にa[1]とb[1]の距離（4）を加えたもので一番小さいものを取る.

	b[0]=3	b[1]=4	b[2]=5	b[3]=4	b[4]=3
a[0]=1	4	13	29	38	42
a[1]=2	5				
a[2]=3	5				
a[3]=4	6				
a[4]=5	10				

矢谷式DPの考え方：#3 操作のマッピング

今回の場合は，セル（0, 0）の値を採用して，4 + 4 = 8.

	b[0]=3	b[1]=4	b[2]=5	b[3]=4	b[4]=3
a[0]=1	4	13	29	38	42
a[1]=2	5	**8**			
a[2]=3	5				
a[3]=4	6				
a[4]=5	10				

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.

dp[i][j] = [dp[i-1][j], dp[i][j-1], dp[i-1][j-1]のうち最小]
+ [a[i]とb[j]の距離]

DTWコード例

```
def dist(x, y): return (x-y) ** 2　# 距離は二乗で定義

def dtw(a, b):
        N = len(a)
        M = len(b)

        # DPテーブル
        dp = [[0 for j in range(M)] for i in range(N)]
```

DTWコード例

```
def dtw(a, b):
    …
    # DPテーブルの初期化
    dp[0][0] = dist(a[0], b[0])

    for i in range(1, N):
        dp[i][0] = dp[i-1][0] + dist(a[i], b[0])
    for j in range(1, M):
        dp[0][j] = dp[0][j-1] + dist(a[0], b[j])
```

DTWコード例

```
def dtw(a, b):
    …
    # DPテーブルを埋めていく
    for i in range(1, N):
        for j in range(1, M):
            dp[i][j] = min(min(dp[i-1][j], dp[i][j-1]),
                           dp[i-1][j-1]) + dist(a[i], b[j])

    print(dp[N-1][M-1])
```

DTWコード実行例

```
tri1 = [1, 2, 3, 4, 5, 4, 3, 2, 1, 2, 3, 4, 5]
tri2 = [3, 4, 5, 4, 3, 2, 1, 2, 3, 4, 5, 4, 3]
flat = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
dtw(tri1, tri2)
dtw(tri1, flat)


-----
10    # tri1とtri2のDTW
22    # tri1とflatのDTW
```

tri1とtri2で実行したときのDPテーブル

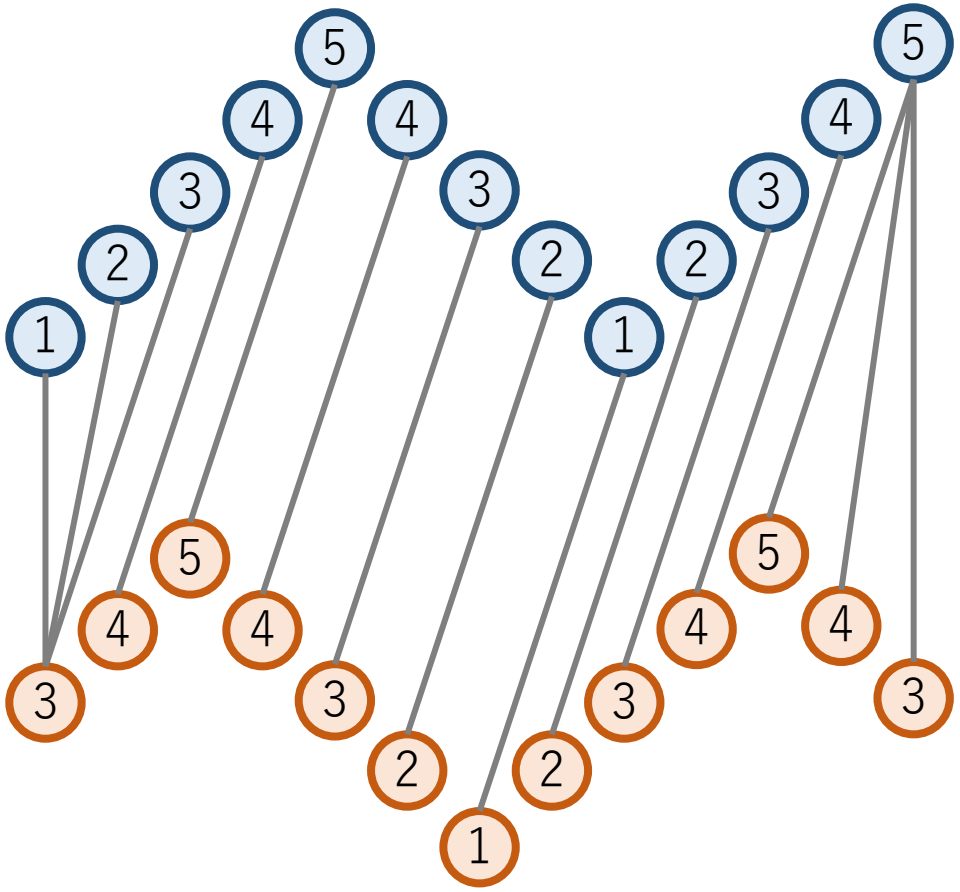
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	4	13	29	38	42	43	43	44	48	57	73	82	86
1	5	8	17	21	22	22	23	23	24	28	37	41	42
2	5	6	10	11	11	12	16	17	17	18	22	23	23
3	6	5	6	6	7	11	20	20	18	17	18	18	19
4	10	6	5	6	10	16	27	29	22	18	17	18	22
5	11	6	6	5	6	10	19	23	23	18	18	17	18
6	11	7	10	6	5	6	10	11	11	12	16	17	17
7	12	11	16	10	6	5	6	6	7	11	20	20	18
8	16	20	27	19	10	6	5	6	10	16	27	29	22
9	17	20	29	23	11	6	6	5	6	10	19	23	23
10	17	18	22	23	11	7	10	6	5	6	10	11	11
11	18	17	18	18	12	11	16	10	6	5	6	6	7
12	22	18	17	18	16	20	27	19	10	6	5	6	10

(0, 0) から最小のところをたどると,

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	4	13	29	38	42	43	43	44	48	57	73	82	86
1	5	8	17	21	22	22	23	23	24	28	37	41	42
2	5	6	10	11	11	12	16	17	17	18	22	23	23
3	6	5	6	6	7	11	20	20	18	17	18	18	19
4	10	6	5	6	10	16	27	29	22	18	17	18	22
5	11	6	6	5	6	10	19	23	23	18	18	17	18
6	11	7	10	6	5	6	10	11	11	12	16	17	17
7	12	11	16	10	6	5	6	6	7	11	20	20	18
8	16	20	27	19	10	6	5	6	10	16	27	29	22
9	17	20	29	23	11	6	6	5	6	10	19	23	23
10	17	18	22	23	11	7	10	6	5	6	10	11	11
11	18	17	18	18	12	11	16	10	6	5	6	6	7
12	22	18	17	18	16	20	27	19	10	6	5	6	10

DTWで求まったペア組

最小値のところを追っていくとペア組を見つけることができる.



DTWの計算量

二重ループで，$O(nm)$.

n, mは2つの時系列データそれぞれの長さ.

FastDTWという改良版では，$O(\max(n, m))$になる.

Stan Salvador, and Philip Chan. “FastDTW: Toward accurate dynamic time warping in linear time and space.” Intelligent Data Analysis 11.5 (2007): 561-580.

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．．．）

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

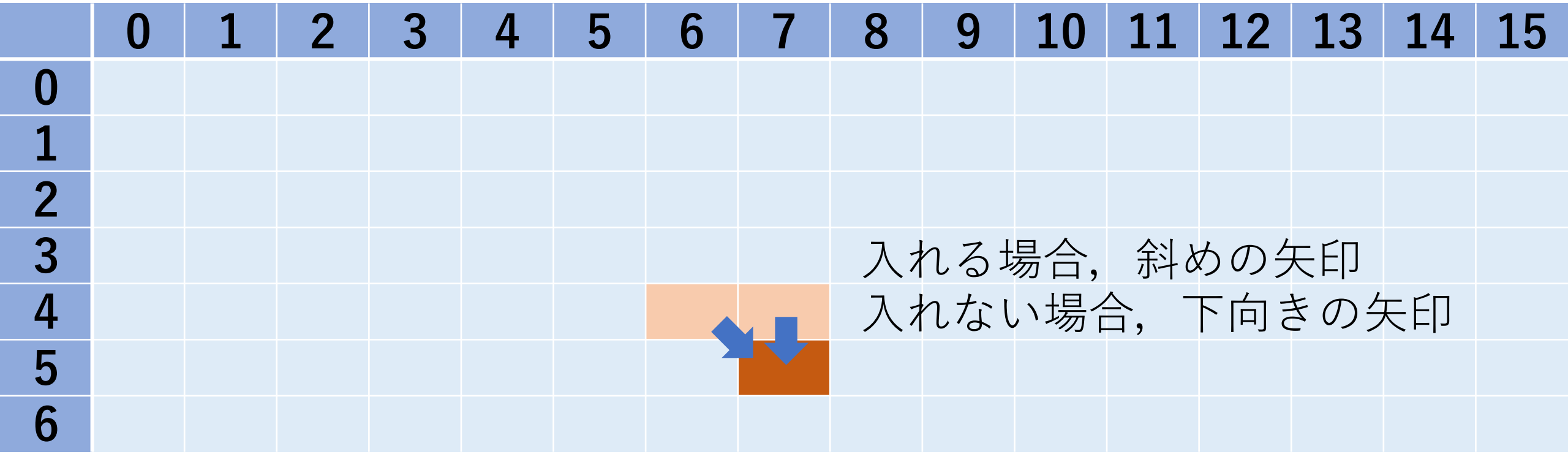
矢谷式で説明したのは全て「貰うDP」です.

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
で他のどのセル**へ**遷移できるかを調べる.

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



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．）

	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

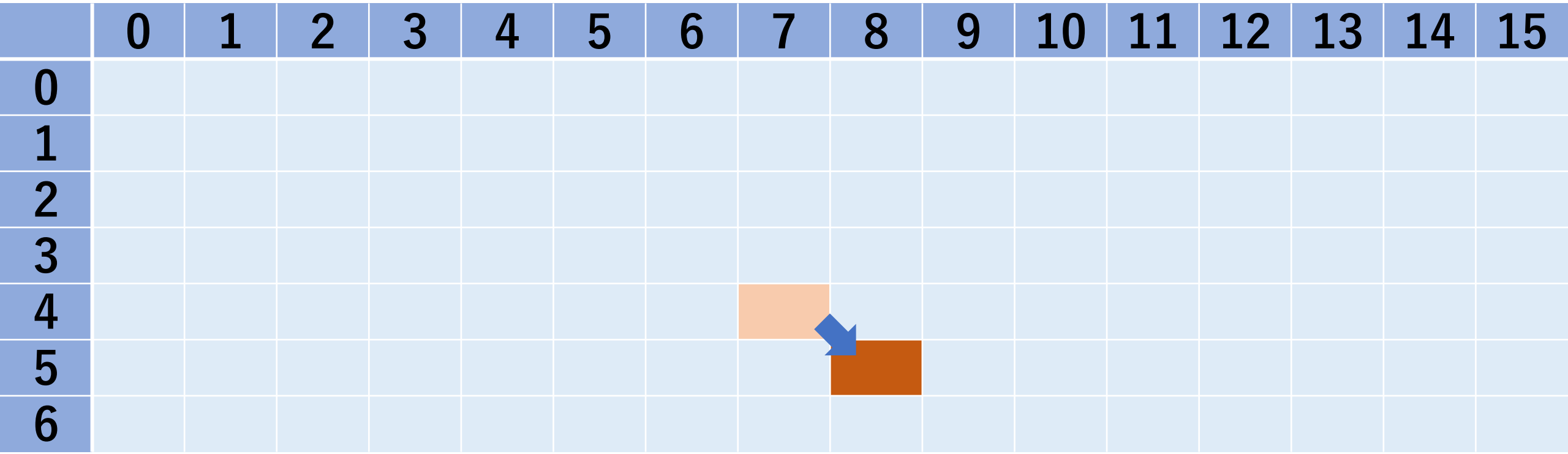
#4 note[i+1][w]の更新.

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．）



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

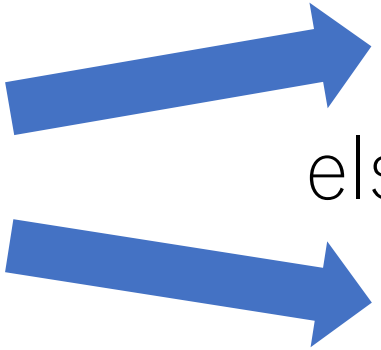
```
def knapsack_distribute():
    [DPテーブルの初期化などは貰うDPと同じ]

    [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])
```

2つのセルを更新

前回の実装例（ナップサックで貰うDP）

```
def knapsack():
    ...
    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:
                note[i+1][w]=[品物iを入れる場合,
                入れない場合の大きい方を]
```

1つのセルのみ更新



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ありうる.

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.

しゃくとり法の時間ですー. 😉

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．．．）

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.

ペンと紙でテーブルを書き，自分がDPになったつもりでテーブルを埋めてみるのもよい.

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.

セグメント木やBITを使うことで高速化できることも.

先程の飴の問題ではBITを使うこともできます.

ある区間の最大値や最小値を出してくることが必要な場合は，セグメント木が有効になり得る.

TLEしちゃう時は？

- DPテーブルの大きさを確認する.
  - 行や列があまりにも長くなる場合には，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

メモ化再帰と漸化式方式のDPテーブル
　　どちらもやっていることは同じ！

レーベンシュタイン距離，DTW

配るDPと貰うDP

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#8-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#8-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#8 [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．