Algorithms (2024 Summer)
#6：文字列照合

矢谷 浩司
東京大学工学部電子情報工学科

文字列照合

あるテキスト（文字列）において，所望の文字列が
現れる場所を探し出す.

「“BABABCBABABABDA”から，“ABABD”の場所を探す．」

文字列検索，文字列探索なども.

力任せ法（brute force）

ごく単純な方法．頭から順番にマッチしているかどうか
を1文字ずつ確認．

マッチしなかったら，1つ右に移動し，また最初から
マッチングを確認．

全一致しているか，最後までいってしまった場合は終了．

このスライドでの説明

「X文字目」という表現をすると，自然言語での「X文字目」か，プログラミングでの「X文字目（自然言語ではX+1文字目）」か，途中でわからなくなってしまうので，text[i]，pattern[i]といった表現で話を進めます．

text

B	**A**	**B**	**A**	**B**	**C**	**B**	**A**	**B**	**A**	**B**	**D**	**B**
A	B	A	B	D								

pattern

力任せ法の例

B	**A**	**B**	**A**	**B**	**C**	**B**	**A**	**B**	**A**	**B**	**D**	**B**
A	B	A	B	D								

text[0]とpattern[0]で照合開始.

力任せ法の例

B	**A**	**B**	**A**	**B**	**C**	**B**	**A**	**B**	**A**	**B**	**D**	**B**
A	B	A	B	D								
X												

ここは一致しないので，次の文字へと移動する．

力任せ法の例

B	A	B	A	B	C	B	A	B	A	B	D	B
	A	B	A	B	D							
	○											

text[1]とpattern[0]で照合．ここは一致する．

力任せ法の例

B	A	B	A	B	C	B	A	B	A	B	D	B
	A	B	A	B	D							
	O	O										

text[2]とpattern[1]で照合．ここも一致．

力任せ法の例

B	A	B	A	B	C	B	A	B	A	B	D	B
	A	B	A	B	D							
	O	O	O									

text[3]とpattern[2]で照合． ここも一致．

力任せ法の例

B	A	B	A	B	C	B	A	B	A	B	D	B
	A	B	A	B	D							
	O	O	O	O								

text[4]とpattern[3]で照合． ここも一致．

力任せ法の例

B	**A**	**B**	**A**	**B**	**C**	**B**	**A**	**B**	**A**	**B**	**D**	**B**
	A	B	A	B	D							
	O	**O**	**O**	**O**	**X**							

text[5]とpattern[4]で照合．ここは一致しない．

力任せ法の例

B	A	B	A	B	C	B	A	B	A	B	D	B
		A	B	A	B	D						

text[1]からの照合は失敗に終わったので，text[2]に戻り，
pattern[0]から照合を再開する．

力任せ法の例

B	A	B	A	B	C	B	A	B	A	B	D	B
							A	B	A	B	D	

見つかったらその場所を返す（例えば先頭のindexの7など）．

力任せ法の実装例

```
def brute_force(text, pattern):
    t_len = len(text)
    p_len = len(pattern)

    # カーソル位置を保持する変数
    t_i = 0
    p_i = 0
```

力任せ法の実装例

```
def brute_force(text, pattern):
    …
    while t_i < t_len and p_i < p_len:
        # 一致している場合は両方のカーソルを進める
        if text[t_i] == pattern[p_i]:
            t_i += 1
            p_i += 1
```

力任せ法の実装例

```
def brute_force(text, pattern):
    …
    while t_i < t_len and p_i < p_len:
        …
        else: # 一致しなかったら後戻り
            t_i = t_i - p_i + 1
            p_i = 0
```

力任せ法の実装例

```
def brute_force(text, pattern):
    …
    while t_i < t_len and p_i < p_len:
        …


    if p_i == p_len:        # 見つかった場合
        return t_i - p_i
    return -1           # 見つからなかった場合
```

力任せ法の計算量

照合対象の文字列の長さがn，照合パターンの長さがlならば，最悪の場合$O(nl)$.

（最悪の場合はどんな場合？）

力任せ法の計算量

とはいえ，実際にはそれほど悪くないことも多い．

照合が失敗する場合，パターンの初めの数文字であることが多く，使われている文字の種類が多くなればパターンの初めの方で失敗する可能性はさらに高くなる．

照合が失敗する位置の期待値

出現しうる文字の種類がK種類（$K > 1$）であるとし，
さらに文字の出現は互いに独立であるとする（前後の
文字の種類に依存しない）．

pattern[0]で初めて照合失敗する確率： $\frac{K-1}{K}$

pattern[1]で初めて照合失敗する確率： $\frac{1}{K} \times \frac{K-1}{K}$

…

pattern[l-1]で初めて照合失敗する確率： $\left(\frac{1}{K}\right)^{l-1} \times \frac{K-1}{K}$

照合が失敗する位置の期待値

よって，照合が失敗する位置の期待値は，

$$\frac{K-1}{K}\sum_{i=1}^{l} i\left(\frac{1}{K}\right)^{i-1}$$

これは，

$$\frac{K}{K-1}\left(1-(l+1)\left(\frac{1}{K}\right)^{l}+l\left(\frac{1}{K}\right)^{l+1}\right)$$

であり，Kが大きい時は第1項以外が十分小さくなって，
$\frac{K}{K-1}$に近似でき，これはほぼ1（pattern[0]に相当）となる．

力任せ法の計算量

照合が失敗する場合，パターンの初めの数文字であることが多く，使われている文字の種類が多くなればパターンの初めの方で失敗する可能性はさらに高くなる．

よって，それほど照合対象文字列のカーソル（t_i）が大きく後戻りすることがそれほど起きない．

処理が単純なので，比較的高速に動く．

力任せ法の問題点

B	A	B	A	B	C	B	A	B	A	B	D	B
	A	B	A	B	D							
	O	O	O	O	X							

text[1]からの照合は失敗に終わったので，**text[2]に戻り**，pattern[0]から照合を再開する．

力任せ法の問題点

B	**A**	**B**	**A**	**B**	**C**	**B**	**A**	**B**	**A**	**B**	**D**	**B**
	A	B	A	B	D							
	O	**O**	**O**	**O**	**X**							

この時点でわかっていること

pattern[2]（B）はマッチしない．

pattern[3]，pattern[4]（AB）はマッチする．

KMP法

Knuth-Morris-Pratt法.

照合が失敗した時点の状況（patternのどこまで照合したか）に応じて，次の照合位置を変更.

KMP法

B	A	B	A	B	C	B	A	B	A	B	D	B
	A	B	A	B	D							
	O	O	O	O	X							

KMP法

B	A	B	A	B	C	B	A	B	A	B	D	B

照合が失敗したCからスタート．（照合対象の方の開始位置は更新しない．）

KMP法

B	**A**	**B**	**A**	**B**	**C**	**B**	**A**	**B**	**A**	**B**	**D**	**B**
			A	B	A	B	D					

ただし，pattern[2]**から照合を始める．**

最初のABは照合することはわかっているのでスキップできるため．

KMP法

B	**A**	**B**	**A**	**B**	**C**	**B**	**A**	**B**	**A**	**B**	**D**	**B**
			A	B	A	B	D					
					X							

今回の場合は，それでも照合失敗．

KMP法

B	**A**	**B**	**A**	**B**	**C**	**B**	**A**	**B**	**A**	**B**	**D**	**B**
					A	B	A	B	D			

更にパターンの開始位置を動かす．今回はpattern[1]のBから始めても照合しないのは明らか（その前のAがそもそも照合しない）ので，pattern[0]まで戻る．

KMP法

B	A	B	A	B	C	B	A	B	A	B	D	B
					A	B	A	B	D			
					X							

ただし，今回はこれもだめ．．．

KMP法

B	A	B	A	B	C	B	A	B	A	B	D	B
						A	B	A	B	D		

よって，これ以上この位置で考えられる候補はなくなったので，次の位置に進む．

KMP法の照合再開場所の表

照合パターンの中に重複な並びが存在する場合には,
照合パターンのどこから再スタートするかが変わる.

ただし，これは固定した情報であるため，毎回計算
していると非効率.

予め表を作っておき，照合中はそれを参照する.

KMP法の事前準備

もしpattern[0]で照合失敗なら,

C	**C**	**C**	**B**	**A**	**B**	**C**	**A**	**C**	**C**	**C**	**C**	**C**
			A	B	A	B	D					
			X									

KMP法の事前準備

もしpattern[0]で照合失敗なら，照合対象のカーソルを1つ右に移動して次の照合を行う．

C	**C**	**C**	**B**	**A**	**B**	**C**	**A**	**C**	**C**	**C**	**C**	**C**
			A	B	A	B	D					
			X									

C	**C**	**C**	**B**	**A**	**B**	**C**	**A**	**C**	**C**	**C**	**C**	**C**
				A	B	A	B	D				

KMP法の事前準備

もしpattern[1]で照合失敗なら,

C	**C**	**C**	**A**	**A**	**B**	**C**	**A**	**C**	**C**	**C**	**C**	**C**
			A	B	A	B	D					
			O	**X**								

KMP法の事前準備

もしpattern[1]で照合失敗なら，次の照合はpattern[0]から始めることが出来る．照合対象のカーソル位置はそのまま．

C	**C**	**C**	**A**	**A**	**B**	**C**	**A**	**C**	**C**	**C**	**C**	**C**
			A	B	A	B	D					
			O	**X**								

C	**C**	**C**	**A**	**A**	**B**	**C**	**A**	**C**	**C**	**C**	**C**	**C**
				A	B	A	B	D				

KMP法の事前準備

もしpattern[2]で照合失敗なら,

C	**C**	**C**	**A**	**B**	**B**	**C**	**A**	**C**	**C**	**C**	**C**	**C**
			A	B	A	B	D					
			O	**O**	**X**							

KMP法の事前準備

もしpattern[2]で照合失敗なら，次の照合はpattern[0]から始めることが出来る．照合対象のカーソル位置はそのまま．

C	**C**	**C**	**A**	**B**	**B**	**C**	**A**	**C**	**C**	**C**	**C**	**C**
			A	B	A	B	D					
			O	**O**	**X**							

C	**C**	**C**	**A**	**B**	**B**	**C**	**A**	**C**	**C**	**C**	**C**	**C**
					A	B	A	B	D			

KMP法の事前準備

もしpattern[3]で照合失敗なら,

C	**C**	**C**	**A**	**B**	**A**	**C**	**A**	**C**	**C**	**C**	**C**	**C**
			A	B	A	B	D					
			O	O	O	X						

KMP法の事前準備

もしpattern[3]で照合失敗なら，次の照合はpattern[1]から始めることが出来る．照合対象のカーソル位置はそのまま．

C	**C**	**C**	**A**	**B**	**A**	**C**	**A**	**C**	**C**	**C**	**C**	**C**
			A	B	A	B	D					
			O	**O**	**O**	**X**						

C	**C**	**C**	**A**	**B**	**A**	**C**	**A**	**C**	**C**	**C**	**C**	**C**
					A	B	A	B	D			
					O							

KMP法の事前準備

もしpattern[4]で照合失敗なら,

C	**C**	**C**	**A**	**B**	**A**	**B**	**A**	**C**	**C**	**C**	**C**	**C**
			A	B	A	B	D					
			O	**O**	**O**	**O**	**X**					

KMP法の事前準備

もしpattern[4]で照合失敗なら，次の照合はpattern[2]から
始めることが出来る．照合対象のカーソル位置はそのまま．

C	**C**	**C**	**A**	**B**	**A**	**B**	**A**	**C**	**C**	**C**	**C**	**C**
			A	B	A	B	D					
			O	O	O	O	X					

C	**C**	**C**	**A**	**B**	**A**	**B**	**A**	**C**	**C**	**C**	**C**	**C**
					A	B	A	B	D			
					O	O						

KMP法の事前準備

このような飛ばす位置はどうやって決めることができる？

照合パターン同士を照合開始位置をずらしながら，照合パターンの内に部分一致するような場所がないかを確認することで求めることができる．

これをテーブルにまとめたものをスキップテーブルと呼ぶ．

KMP法の事前準備：スキップテーブル

以下のように仮想的に照合パターン同士を位置をずらしながら照合していく．

A	A	B	A	A	A	B	A	C	
	A	A	B	A	A	A	B	A	C

このとき，2つの可能性がある．

文字がマッチする場合

文字がマッチしない場合

KMP法の事前準備：スキップテーブル

文字がマッチする場合

直近の最長部分一致の長さを記録し，そのまま照合を次の文字へ進める.

文字がマッチしない場合

マッチが失敗した位置から，パターンの中で可能な最長の部分一致の場所に移動する.

（つまりKMPそのもの）

KMP法の事前準備：スキップテーブル

pattern[i]の文字で照合が失敗した場合，skip[i-1]の値を見て次の照合におけるパターンの開始位置を決める．

A	A	B	A	A	A	B	A	C	A

skip	0	1	2	3	4	5	6	7	8
要素の値									

KMP法の事前準備：スキップテーブル

よって，スキップテーブルの大きさは[照合パターンの長さ] − 1でよい．（今回の場合，スキップテーブルの大きさは9となる．）

A	A	B	A	A	A	B	A	C	A

skip	0	1	2	3	4	5	6	7	8
要素の値									

KMP法の事前準備：スキップテーブル

テーブルの一番最初の値は常に0. つまりskip[0]の値は0.

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**

skip	0	1	2	3	4	5	6	7	8
要素の値	0								

KMP法の事前準備：スキップテーブル

pattern[1]で照合失敗のときは，skip[1-1] = 0 から，
pattern[0]に戻って照合を開始することになる．

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**

skip	0	1	2	3	4	5	6	7	8
要素の値	0								

KMP法の事前準備：スキップテーブル

pattern[1]で照合失敗の場合は，pattern[0]からスタートした場合に上手くいくかどうかをチェックしないといけないため.

B	B	A	B	A	B	B	A	A	A
B	A	B	A						
	B	A	B	A					

skip	0	1	2	3	4	5	6	7	8
要素の値	0								

KMP法の事前準備：スキップテーブル

これ以降は照合パターン同士を照合開始位置をずらしながら，照合パターンの内に部分一致するような場所がないかを確認する．

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**	
	A	A	B	A	A	A	B	A	C	A

skip	0	1	2	3	4	5	6	7	8
要素の値	0								

KMP法の事前準備：スキップテーブル

pattern[2]に対応する値を考えるため，青pattern[1]と緑pattern[0]を照合する．今回は一致する．

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**	
	A	A	B	A	A	A	B	A	C	A
	O									

skip	0	1	2	3	4	5	6	7	8
要素の値	0								

KMP法の事前準備：スキップテーブル

一致した場合は直近の最長部分一致の長さを記録するので，skip[1]には1が入る．

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**	
	A	A	B	A	A	A	B	A	C	A
	O									

skip	0	1	2	3	4	5	6	7	8
要素の値	0	**1**							

KMP法の事前準備：スキップテーブル

このことは，もしpattern[2]で照合失敗した場合は，pattern[1]から照合を再度行うことで一致する可能性があることを意味している．

A	**A**	**A**	**A**	**B**	**B**	**B**	**A**	**B**	**A**
A	A	B	B						
	A	A	B	B					

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1							

KMP法の事前準備：スキップテーブル

青pattern[2]と緑pattern[1] は一致しない.

A	A	B	A	A	A	B	A	C	A	
	A	A	B	A	A	A	B	A	C	A
	O	X								

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1							

KMP法の事前準備：スキップテーブル

この時は，今までにわかっているスキップテーブルの値を使い，どこまでずらせるかを確認する．

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**	
	A	A	B	A	A	A	B	A	C	A
	O	**X**								

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1							

KMP法の事前準備：スキップテーブル

現在の緑patternにおける照合の位置は1なので，
skip[1-1] = skip[0]を見ると0.

A	A	B	A	A	A	B	A	C	A	
	A	A	B	A	A	A	B	A	C	A
	O	X								

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1							

KMP法の事前準備：スキップテーブル

つまり先頭まで戻すことを指している.

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**		
		A	A	B	A	A	A	B	A	C	A

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1							

KMP法の事前準備：スキップテーブル

さらに今回はずらした後も一致しない.

A	A	B	A	A	A	B	A	C	A		
		A	A	B	A	A	A	B	A	C	A
		X									

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1							

KMP法の事前準備：スキップテーブル

よってskip[2]には0を入れ，さらに照合パターンを1つ右にずらす．

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**						
			A	A	B	A	A	A	B	A	C	A			

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1	**0**						

KMP法の事前準備：スキップテーブル

青pattern[3]と緑pattern[0]，青pattern[4]と緑pattern[1]はそれぞれ一致するので，その時点における直近の部分一致の最長の値を入れる.

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**			
			A	A	B	A	A	A	B	A	C	A
			o	**o**								

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1	0						

KMP法の事前準備：スキップテーブル

よって，skip[3]は1，skip[4]は2.

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**			
			A	A	B	A	A	A	B	A	C	A
			O	**O**								

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1	0	**1**	**2**				

KMP法の事前準備：スキップテーブル

青pattern[5]と緑pattern[2]は一致しない．

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**						
			A	A	B	A	A	A	B	A	C	A			
			O	**O**	**X**										

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1	0	1	2				

KMP法の事前準備：スキップテーブル

現在の緑patternにおける照合の位置は2なので,
skip[2-1] = skip[1]を見ると1.

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**			
			A	A	B	A	A	A	B	A	C	A
			O	**O**	**X**							

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1	0	1	2				

KMP法の事前準備：スキップテーブル

よって，青pattern[5]の位置に緑pattern[1]が来るように照合パターンをずらして照合を再開すれば良い．この時，青pattern[4]と緑pattern[0]は調べなくても一致している．

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**				
				A	A	B	A	A	A	B	A	C	
				O									

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1	0	1	2				

KMP法の事前準備：スキップテーブル

さらに今回の場合は，青pattern[5]と緑pattern[1]も一致するので，その時点における直近の部分一致の最長の値である2をskip[5]に入れる．

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**				
				A	A	B	A	A	A	B	A	C
				O	**O**								

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1	0	1	2	**2**			

KMP法の事前準備：スキップテーブル

スキップテーブルで照合開始位置を求めてその位置で一致がある場合には，スキップテーブルで求めた値に1を加えれば良い．

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**					
				A	A	B	A	A	A	B	A	C		
				O	**O**									

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1	0	1	2	**2**			

KMP法の事前準備：スキップテーブル

この場合の2は，skip[1]の1にさらに1を加えることで求めることができる．

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**				
				A	A	B	A	A	A	B	A	C
				O	**O**							

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1	0	1	2	**2**			

KMP法の事前準備：スキップテーブル

青pattern[6]と緑pattern[2]，青pattern[7]と緑pattern[3]もそれぞれ一致するので，その時点における直近の部分一致の最長の値skip[6]，skip[7]に入れる．

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**			
				A	A	B	A	A	A	B	A	C
				O	**O**	**O**	**O**					

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1	0	1	2	2	**3**	**4**	

KMP法の事前準備：スキップテーブル

青pattern[8]と緑pattern[4]は一致しない．skip[4-1] = skip[3]を見ると1となっている．

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**				
				A	A	B	A	A	A	B	A	C
				o	o	o	o						

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1	0	1	2	2	3	4	

KMP法の事前準備：スキップテーブル

つまり，青pattern[8]に緑pattern[1]を合わせる，と言っているが，そこに合わせても一致しない．

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**			
							A	A	B	A	A	A
								X				

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1	0	1	2	2	3	4	

KMP法の事前準備：スキップテーブル

緑pattern[1]でも一致しないので，さらにskip[1-1] = skip[0]を見る．

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**						
							A	A	B	A	A	A			
								X							

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1	0	1	2	2	3	4	

KMP法の事前準備：スキップテーブル

緑pattern[1]でも一致しないので，さらにskip[1-1] = skip[0]を見るが，今回はそれでも一致しない．

A	**A**	**B**	**A**	**A**	**A**	**B**	**A**	**C**	**A**					
								A	A	B	A	A		
								X						

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1	0	1	2	2	3	4	

KMP法の事前準備：スキップテーブル

よって，skip[8]は0にして，ここでスキップテーブルの構築が終了する．

A	A	B	A	A	A	B	A	C	A			
								A	A	B	A	A
								X				

skip	0	1	2	3	4	5	6	7	8
要素の値	0	1	0	1	2	2	3	4	**0**

KMP法の事前準備：スキップテーブル

文字がマッチする場合

直近の最長部分一致の長さを記録し，そのまま照合を次の文字へ進める.

文字がマッチしない場合

マッチが失敗した位置から，パターンの中で可能な最長の部分一致の場所に移動する.

→文字が一致するか先頭に行くまで，今までに構築したスキップテーブルを繰り返し参照する.

スキップテーブルを構築する方法の実装

```
def create_table(pattern):
        table = [0] * (len(pattern) - 1)
        j = 0          # 緑patternのカーソル位置
```

スキップテーブルを構築する方法の実装

```
def create_table(pattern):
    …
    for i in range(1, len(pattern)-1):
        # 文字が一致している時
        if pattern[i] == pattern[j]:
            j += 1
            table[i] = j
```

スキップテーブルを構築する方法の実装

```
def create_table(pattern):
    …
    for i in range(1, len(pattern)-1):
        …
        else: # 文字が一致しない時
            while [文字が一致するか先頭に行くまで]:
                [スキップテーブルを使ってjを更新]
```

スキップテーブルを構築する方法の実装

```
def create_table(pattern):
    …
    for i in range(1, len(pattern)-1):
        …
        else:
            …
        # jの値を更新した後，一致していれば+1する．
        if pattern[i] == pattern[j]: j += 1
        table[i] = j
    return table
```

スキップテーブルを構築する方法の実装

上記コード例は実は冗長な書き方になっていて，以下のように書いても同じように動きます.

```
for i in range(1, len(pattern)-1):
    while [文字が一致するか先頭に行くまで]:
        [スキップテーブルを使ってjを更新]

    if pattern[i] == pattern[j]: j += 1
    table[i] = j
```

KMP法の実装例

```
def kmp(text, pattern):
        # スキップテーブルを作る
        skip = createTable(pattern)

        t_len = len(text)
        p_len = len(pattern)
        t_i = p_i = 0     # テキストとパターンのカーソル位置
```

KMP法の実装

```
def kmp(text, pattern):
    …
    # 照合を行うループ
    while t_i < t_len and p_i < p_len:
        # 一致している場合は両方のカーソルを進める
        if text[t_i] == pattern[p_i]:
            t_i += 1
            p_i += 1
```

KMP法の実装例

```
def kmp(text, pattern):
    …
    while t_i < t_len and p_i < p_len:
        …
        # pattern[0]で失敗した場合は，textのカーソルを
        # 1つすすめるだけ.
        elif p_i == 0:
            t_i += 1
```

KMP法の実装例

```
def kmp(text, pattern):
    …
    while t_i < t_len and p_i < p_len:
        …
        else:
            # pattern[0]以外で失敗した場合
            [スキップテーブルを使ってp_iを更新]
```

KMP法の実装例

```
def kmp(text, pattern):
    …
    while t_i < t_len and p_i < p_len:
        …


    if p_i == p_len:        # 見つかった場合
        return t_i - p_i
    return -1           # 見つからなかった場合
```

KMP法の計算量

t_iは文字がマッチした時とpattern[0]でマッチしない時に進み，**後戻りすることはない**.

照合対象の文字列の長さがn，照合パターンの長さがl

スキップテーブルの作成： $O(l)$.
照合：最悪でも$O(n)$.　（最悪の場合はどんな場合？）

よって，$O(n+l)$.

ただ実際には．．．

力任せ法の最悪ケースはかなり意地悪なケースなので,
実際問題そんなには起きない.

特殊な文字列ではない場合，照合しないときは最初の数文字でそれがわかるので，毎回の照合に$O(l)$かかることは，めったに起こらない.

KMP法のほうが処理が少し複雑なので，遅くなることも.

ただ実際には．．．

KMP法は理論的によくできているアルゴリズムだが，
性能向上はあまり望めないことも．

KMP法の教科書的な良いところは以下の2つ．
- 照合対象文字列のカーソルが後戻りしない．
- スキップテーブルを機械的に作り出せる．

このためアルゴリズムの教科書ではよく紹介されている
手法になります． 😊

力任せ法とKMP法の考察

先ほどの2つのアルゴリズムは，頭から比較を行なっていくという共通点がある．

意地悪なケースでなければ，照合の多くは最初の数文字で失敗する．

このために大概の場合で数文字分しか進めない．

BM法

Boyer-Moore法.

照合パターンの前からではなく，後ろから照合する.

照合パターンに出てくる文字かどうか，出てくる場合どこに出てくるか，に着目する.

実用上は結構速く処理ができる.

BM法の例

B	**A**	**B**	**A**	**C**	**C**	**B**	**A**	**B**	**A**	**B**	**D**	**B**
A	B	A	B	D								

照合パターンの最後（D）からマッチング開始.

BM法の例

B	**A**	**B**	**A**	**C**	**C**	**B**	**A**	**B**	**A**	**B**	**D**	**B**
A	B	A	B	D								

Cはパターンにない文字．なので，このtext[4]のCが含まれるような照合開始位置は全て却下すべき．

BM法の例

B	**A**	**B**	**A**	**C**	**C**	**B**	**A**	**B**	**A**	**B**	**D**	**B**
					A	B	A	B	D			

よって，一気にジャンプ！

BM法の例

B	**A**	**B**	**A**	**C**	**C**	**B**	**A**	**B**	**A**	**B**	**D**	**B**
					A	B	A	B	D			
									X			

DはAとは不一致.

BM法の例

B	**A**	**B**	**A**	**C**	**C**	**B**	**A**	**B**	**A**	**B**	**D**	**B**
					A	B	A	B	D			
									X			

DはAとは不一致．しかし先ほどの場合と違って，Aは照合パターンに存在する文字．後ろからたどって最初にマッチするのは，照合パターンpatter[2]のA．

BM法の例

B	**A**	**B**	**A**	**C**	**C**	**B**	**A**	**B**	**A**	**B**	**D**	**B**
							A	B	A	B	D	

よって，2文字分照合開始位置を進めて，再度末尾のDから照合開始．（今回はここで完全に一致．）

BM法の事前準備

KMP法と同じく，スキップテーブルを作成する.

照合パターンの長さをlとして,

- パターンに含まれていない文字に対しては，移動量l.
- パターンに含まれている文字に対して,
  - 末尾にしかその文字が現れない場合は，移動量l.
  - 末尾に最も近い出現位置がi $(0 \leq i < l - 1)$ならば,
移動量$l - i - 1$.
    - 末尾に加えてそれ以外の場所でも出現する場合は
こちらを採用.

BM法の事前準備

照合パターンがABABDの場合のスキップテーブル.

A	B	C	D	E
2	1	5	5	5

先頭から見ていって，Dが一番最後に現れるのは末尾なので，移動量は5（＝パターンの長さ）.

照合が失敗した場合，上記の値分だけ照合開始位置を進めて照合を再開.

確かに良さそう

B	**A**	**B**	**A**	**C**	**C**	**B**	**A**	**B**	**A**	**B**	**D**	**B**
A	B	A	B	D								
					A	B	A	B	D			
							A	B	A	B	D	

A	B	C	D	E
2	1	5	5	5

でも，スキップテーブルだけだと．．．

・	・	・	・	**D**	**B**	**A**	**B**	**D**	・	・	・	・
				A	C	A	B	D				

A	B	C	D	E
2	1	3	5	5

（照合パターンが変更され，スキップテーブルの値が変更されていることに注意）

パターンの途中でマッチしない場合

・	・	・	・	**D**	**B**	**A**	**B**	**D**	・	・	・	・
				A	C	A	B	D				

A	B	C	D	E
2	1	3	5	5

パターンの途中でマッチしない場合

・	・	・	・	**D**	**B**	**A**	**B**	**D**	・	・	・	・
				A	C	A	B	D				

A	B	C	D	E
2	1	3	5	5

後戻りしている？

・	・	・	・	**D**	**B**	**A**	**B**	**D**	・	・	・	・
		A	C	A	B	D						

濃い青色のセルは進んでいるが，マッチングの先頭（”A”）の位置は後戻りしてしまっている．

A	B	C	D	E
2	1	3	5	5

パターンの途中で不一致になるときの注意点

照合パターン（スキップテーブルの値）によっては,
照合パターンの最初の位置が後戻りすることがある.

力任せ法のように毎回後戻りするわけではないので,
効率が極端に悪くなるわけではないが，この場合を
ちゃんと考慮しないとやばいことが起きそうな．．．

パターンの途中で不一致になるときの注意点

・	・	・	・	**B**	**A**	**D**	**D**	**D**	**B**	・	・	・
				A	C	A	D	B				

Dでマッチしなかったので，右に1つ移動．

A	B	C	D	E
2	5	3	1	5

（照合パターンが変更され，スキップテーブルの値が変更されていることに注意）

パターンの途中で不一致になるときの注意点

・	・	・	・	**B**	**A**	**D**	**D**	**D**	**B**	・	・	・
					A	C	A	D	B			

移動したら，照合再開.

A	B	C	D	E
2	5	3	1	5

パターンの途中で不一致になるときの注意点

・	・	・	・	B	A	D	D	D	B	・	・	・
					A	C	A	D	B			
							X	O	O			

pattern[2]で一致しない．Dでマッチしなかったので，
今のカーソルの位置から1つ右に進む．

A	B	C	D	E
2	5	3	1	5

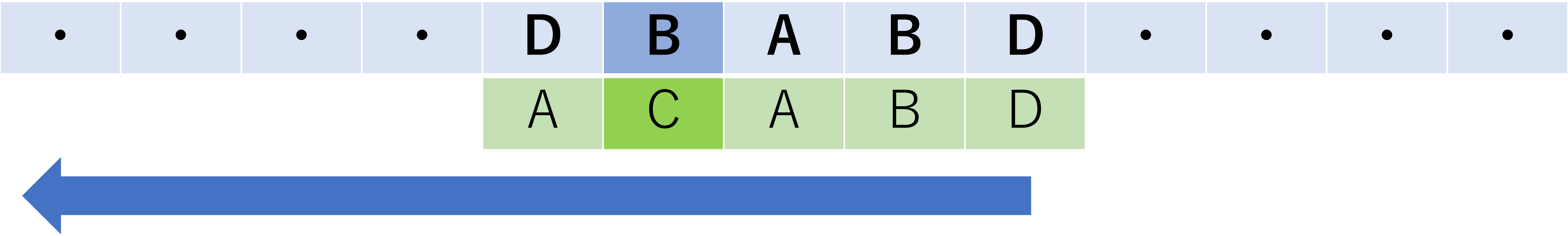
パターンの途中で不一致になるときの注意点

・	・	・	・	**B**	**A**	**D**	**D**	**D**	**B**	・	・	・
				A	C	A	D	B				

あれ？😨

A	B	C	D	E
2	5	3	1	5

後戻りしないために



照合をしている最後の文字から左側にはマッチするもの
はないので，戻る意味はない．

よって，強制的に1つ右にずらしたい．

後戻りしないために

・	・	・	・	**D**	**B**	**A**	**B**	**D**	・	・	・	・
				A	C	A	B	D				

不一致になるまでに照合した文字数は4.

もし，照合対象のカーソルをこの数進めると，

後戻りしないために

・	・	・	・	**D**	**B**	**A**	**B**	**D**	・	・	・	・
					A	C	A	B	D			

1つ前に進んだことになる！

そして，この位置から照合を再開する．

後戻り防止策

照合パターンの一番右の文字で照合失敗の場合は,
スキップテーブルの値に従えば良い.

照合パターンの途中の文字で照合失敗の場合は,
スキップテーブルの値と不一致になるまでに照合した
文字数を比較し，より大きい方の値を採用する.

この防止策を含めてBM法となっている.

BM法の実装例

```
def bm_search(text, pattern):
        t_len = len(text)
        p_len = len(pattern)
        # アルファベット小文字のみの想定
        # 照合パターンの長さで初期化
        skip = [p_len]*26
```

BM法の実装例

```
def bm_search(text, pattern):
    …
    # スキップテーブルの作成・照合パターン最後の文字
    # は飛ばす．先頭から見ていって一番最後に現れるの
    # は末尾になる場合，対応するskipの値はp_lenになる．
    for i in range(p_len - 1):
        skip[ord(pattern[i])-97] = p_len - i - 1
```

BM法の実装例

```
def bm_search(text, pattern):
    …
    # 照合対象側のカーソルを予め進める
    t_i = p_len – 1

    # 照合対象の文字列全てをチェック
    while t_i < t_len:
        # パターンのカーソルを一番最後にする
        p_i = p_len - 1
```

BM法の実装例

```
def bm_search(text, pattern):
        …
        t_i = p_len – 1
        while t_i < t_len:
                p_i = p_len - 1
                [パターンの後ろから一致を確認するループ]:
                        [1番先頭まで行っていればreturn]
                        [それ以外なら2つのカーソルを1つ後ろに進める]
                [マッチしなかったら，t_iを前に進める]
        return -1     # 見つからなかった場合
```

BM法の実装例

```
def bm_search(text, pattern):
    …
    t_i = p_len - 1
    while t_i < t_len:
        p_i = p_len - 1
        while text[t_i] == pattern[p_i]:
            if p_i == 0: return t_i
            t_i -= 1; p_i -= 1
        t_i += max(skip[ord(text[t_i])-97], p_len - p_i)
    return -1
```

BM法の計算量：スキップテーブルの構築

出現しうる文字の種類の総数をKとすると，$O(K+l)$.

$O(K)$：配列の初期化

（上記実装例では配列を一括で初期化している．）

$O(l)$：その後のforループ部分に対応.

BM法の計算量：照合

最良の場合，1文字目でマッチせず，かつlだけスキップできることがずっと続く．

よって，$O(n/l)$.

ただし，最悪のケースは$O(nl)$. （それはどんな場合？）

通常$n \gg K, l$なので，$O(n/l)$の場合でも照合がスキップテーブル構築よりも支配的．

BM法の計算量

現実的には，$O(n/l)$で動くことが期待できる．

パターン文字列において各文字の出現確率が均等なら，
スキップテーブルの値は平均的には $l/2$になる．

もしパターン文字列中に出現する文字が偏っている
場合，パターン文字中に出てこない文字に対して
スキップの大きさはlになり，それが大部分になる．

よって，どちらの場合でも1回のジャンプで
$O(l)$程度動けることが期待できる．

BM法の計算量

現実的には，$O(n/l)$で動くことが期待できる．

毎回の照合で失敗するのは，後ろから数えて数
文字目で多くの場合起きると考えられ，その度に
$O(l)$ジャンプできることが期待できる．

よって，lが小さくなければ，それなりに速く動くことが
通常の場合は期待できる．

BM法の計算量

出現しうる文字の種類が少なかったり，文字の出現の仕方に偏りがあると不利になり得るが，そのような意地悪なケースでなければ，高速に動くアルゴリズムとして認知されている．

BMH法

Boyer-Moore-Horspool.

Nigel HorspoolさんによるBM法の改良.

不一致が出た時，その場所での文字ではなく，**照合している部分の末尾の文字で**スキップする大きさを決定する.

スキップテーブルの作り方はBM法に同じ.

BMH方の考え方

・	・	・	・	**B**	**A**	**D**	**D**	**D**	**B**	・	・	・
					A	C	A	D	B			
							X	**O**	**O**			

pattern[2]で照合失敗.

A	B	C	D	E
2	5	3	1	5

BMH方の考え方

・	・	・	・	**B**	**A**	**D**	**D**	**D**	**B**	・	・	・
					A	C	A	D	B			
							X	**O**	**O**			

照合失敗した場所の文字ではなく，一番末尾の文字で移動量を決定.

A	B	C	D	E
2	5	3	1	5

BMH方の考え方

・	・	・	・	**B**	**A**	**D**	**D**	**D**	**B**	・	・	・
					A	C	A	D	B			
										A	C	A

その文字ではなく，一番末尾の文字で移動量を決定.

A	B	C	D	E
2	5	3	1	5

BM法とBMH法の違い

BM法：不一致発生時，**照合中文字列の不一致発生場所にある文字**に対して，次にマッチするところまで進む.

A	**B**	**A**	**C**	**B**	**B**	**D**	**A**	**B**	**A**	**C**	**C**	**A**
		D	B	A	B	D						
			D	B	A	B	D					

BMH法：不一致発生時，**照合中文字列の末尾の文字**に対して，次にマッチするところまで進む.

A	**B**	**A**	**C**	**B**	**B**	**D**	**A**	**B**	**A**	**C**	**C**	**A**
		D	B	A	B	D						
						D	B	A	B	D		

BMH法の実装例

```
def bmh_search(text, pattern):
    （スキップテーブル部分はBM法に同じ）
    t_i = p_len - 1
    while t_i < t_len:
        p_i = p_len - 1
        while text[t_i - (p_len - 1 - p_i)] == pattern[p_i]:
            if p_i == 0: return t_i - p_len + 1
            p_i -= 1
        t_i += skip[ord(text[t_i])-97]
    return -1
```

照合を行っている
末尾の文字に応じて
スキップ

BMH法の計算量

計算量としてはBM法に同じ.

BM法よりもより多くスキップできることが多く,
BM法よりはやや効率的なことが多い.

ラビン・カープ（Rabin-Karp）法

部分文字列をハッシュに変換しておき，文字列の照合をハッシュ値の一致の問題へと変換.

ハッシュ値が一致したものに対して文字列の一致を確認.

ハッシュにより，比較の計算量を$O(l)$から$O(1)$にできる.

この方法ではローリングハッシュというものを使う.

ローリングハッシュ

互いに素な定数a, h $(1 < a < h)$を準備して，以下のような式で，検索パターン文字列$p_0, p_1 \dots, p_{l-1}$に対してハッシュ値を計算する．

$$H(P) = \left(a^{l-1} p_0 + \cdots + \ a^0 p_{l-1}\right) \bmod \ h$$

これはa進法に変換していることと同じ．与えられた文字列の種類以上の素数を使うことになる．ただし大きな値になるので，剰余にしている．

ローリングハッシュ

次に検索対象文字列Sの部分文字列に対しても同様にハッシュ値を計算する．text[0]からtext[l-1]までの部分文字列のハッシュは,

$$H(S,0,l-1) = \left(a^{l-1}s_0 + \cdots + a^0 s_{l-1}\right) \bmod h$$

この$H(S,0,l-1)$と$H(P)$とが一緒かどうかをチェック．

ローリングハッシュ

ハッシュ値が違っていたら，次にtext[1]の文字からtext[l]番目までの部分文字列のハッシュを計算する．すなわち，

$$H(S, 1, l) = \left(a^{l-1} s_1 + \cdots + \ a^0 s_l\right) \bmod h$$

を計算し，照合を行う．以降，これを繰り返してハッシュ値がマッチすることを見つける．

（ラビン・カープ法ではさらに，文字列の一致を確認する．）

ローリングハッシュ

ただし，このハッシュの計算をまともに毎回やってしまうと，最悪$O(n)$回発生する．

ハッシュ1回の計算が$O(l)$なので，全体として$O(nl)$となり，力任せ法と変わらない．．．

剰余の計算などがあるので，力任せ法よりも定数倍遅くなりえる．

ローリングハッシュ

$a^{l-1}s_0 + \cdots + \ a^0 s_{l-1} = H(S, 0, l-1) + Ah$と表される．

Aは何かしらの整数値であり，ハッシュを計算する時にはhの剰余とするので，無視できる．

$H(S, 1, l)$の計算をする際に，この$a^{l-1}s_0 + \cdots + \ a^0 s_{l-1}$をうまく再利用できないだろうか？

ローリングハッシュ

$$H(S,1,l) = a^{l-1}s_1 + \cdots + a^1 s_{l-1} + \ a^0 s_l$$

ローリングハッシュ

$$a^{l-1}s_1 + \cdots + a^1 s_{l-1} + \ a^0 s_l$$

$$= a\left(a^{l-2}s_1 + \cdots + \ a^0 s_{l-1}\right) + \ a^0 s_l$$

ローリングハッシュ

$$a^{l-1}s_1 + \cdots + a^1 s_{l-1} + \ a^0 s_l$$

$$= a\left(a^{l-2}s_1 + \cdots + \ a^0 s_{l-1}\right) + \ a^0 s_l$$

$$= a\left(a^{l-1}s_0 + a^{l-2}s_1 + \cdots + \ a^0 s_{l-1}\right) - a^l s_0 + \ a^0 s_l$$

ローリングハッシュ

$$a^{l-1}s_1 + \cdots + a^1 s_{l-1} + \ a^0 s_l$$

$$= a\left(a^{l-2}s_1 + \cdots + \ a^0 s_{l-1}\right) + \ a^0 s_l$$

$$= a\left(a^{l-1}s_0 + a^{l-2}s_1 + \cdots + \ a^0 s_{l-1}\right) - a^l s_0 + \ a^0 s_l$$

$$= a(H(S,0,l-1) + Ah) - a^l s_0 + \ a^0 s_l$$

ローリングハッシュ

よって,

$$H(S,1,l) = \left(aH(S,0,l-1) - a^l s_0 + a^0 s_l\right) \bmod h$$

を計算すればよく，これは$O(1)$で計算可能.

しゃくとり法的な話. 😊
（累積和のように実装することも可能です.）

ローリングハッシュによる照合の実装例

```
def RollingHashMatch(text, pattern):
        base = 31                   # 基数
        h = 998244353               # 除数

        t _len = len(text)
        p_len = len(pattern)
        t_hash = 0          # 照合対象側のハッシュ
        p_hash = 0          # 照合パターン側のハッシュ
```

ローリングハッシュによる照合の実装例

```
def RollingHashMatch(text, pattern):
        …
        for i in range(p_len):
```

i番目まで：$\left(a^{i-1}p_1 + \cdots + \ a^1 p_{i-1} + a^0 p_i\right) \bmod h$

i-1番目まで：$\left(a^{i-2}p_1 + \cdots + \ a^0 p_{i-1}\right) \bmod h$

[上の関係性を使ってt_hashを更新]

[同様にp_hashも計算]

[値が大きくなるので毎回剰余を計算し，それを渡していく]

ローリングハッシュによる照合の実装例

```
def RollingHashMatch(text, pattern):
        …
        # 一番先頭でマッチしていればそこで終わり
        if t_hash == p_hash:
                return 0
```

ローリングハッシュによる照合の実装例

```
def RollingHashMatch(text, pattern):
    …
    [textの残りの部分文字列に対してのループ]:
        # a^lを予めどこかで計算しておくと良い.
        # ただし，値がとても大きくなる可能性に注意.
        [t_hashを更新（しゃくとり法的考え方）]
        if t_hash == p_hash:
            [マッチした場所のindexを返す]
    return -1
```

ローリングハッシュによる照合の計算量

照合パターンのハッシュ化： $O(l)$
ハッシュを用いての照合：

一番最初だけは$O(l)$，あとは$O(1)$.
最悪の場合は$n-l$回の比較が必要.
よって，$O(n)$.

以上より$O(n+l)$．ただしハッシュの計算コストが高い場合があり，他の手法より劣ることも．一方で，文字の出現パターンに左右されにくい.

基数と除数の選び方

出来る限りハッシュの衝突を避けたい.

除数は大きい素数であるほうが良いとされる.

基数は出現しうる文字の総種類数より大きいものを選ぶ.

それでも衝突は起きるので，ハッシュを複数用意する,
基数を乱択化するなどの方法が用いられることもある.

意図的に適当に選んだときにどのくらい衝突してしまうか,
ぜひ試してみてください.

パフォーマンス比較例

照合対象文字列：’aa'*(10**6)+'a’*100+'b’
“aaaaaaaaaaaaaaaaaaaaaaaaaaaaa…aaaaaaaaaaab”
パターン文字列：'a’*100+’b’　“[a100個]b”

力任せ法： 33,635 msec
KMP：680 msec
BM：826 msec
BMH：759 msec
ローリングハッシュ：818 msec

パフォーマンス比較例

照合対象文字列：’ab'*(10**6)+'a’*100+'b’
“abababababababababababababababa…aaaaaaaaaaab”
パターン文字列：'a’*100+’b’　“[a100個]b”

力任せ法： 529 msec
KMP：381 msec
BM：30 msec
BMH：25 msec
ローリングハッシュ：856 msec

パフォーマンス比較例

照合対象文字列：’ac'*(10**6)+'a’*100+'b’
“acacacacacacacacacacacacacaca…aaaaaaaaaaab”
パターン文字列：'a’*100+’b’　“[a100個]b”

力任せ法： 485 msec
KMP：505 msec
BM：22 msec
BMH：20 msec
ローリングハッシュ：854 msec

パフォーマンス比較例

照合対象文字列：aとbのみのランダムな文字列200,000文字
パターン文字列：末尾の100文字

（10回行なった平均）
力任せ法：72.1 msec
KMP：49.3 msec
BM：99.1 msec
BMH：72.8 msec
ローリングハッシュ：94.5 msec

パフォーマンス比較例

照合対象文字列：aからzのランダムな文字列200,000文字
パターン文字列：末尾の100文字

（10回行なった平均）
力任せ法：37.3 msec
KMP：33.8 msec
BM：3.49 msec
BMH：3.29 msec
ローリングハッシュ：94.3 msec

まとめ

力任せ法
KMP法
BM法
BMH法
ローリングハッシュ

これ以外にも色々ありますので，ぜひ調べてみてください！

コードチャレンジ：基本課題#6-a [1.5点]

KMP法を自分で実装してください.

（余裕があれば自分のローカル環境で力任せ法の場合と比較をしてみてください．）

コードチャレンジ：基本課題#6-b [1.5点]

ローリングハッシュによる照合を自分で実装してください.

ord関数の使用は問題ありません.

（余裕があれば自分のローカル環境で力任せ法の場合と比較をしてみてください．）

コードチャレンジ：Extra課題#6 [3点]

今日習った文字列探索アルゴリズムを活用する問題.