データ構造とアルゴリズム (第13回)

文字列のアルゴリズム

文字列のアルゴリズム

- 7.1 文字列照合問題とは
- 7.2 力まかせ法
- 7.3 ラビン-カープ法
- 7.4 クヌース-モリス-プラッツ法
- 7.5 ボイヤー-ムーア法 (概要)

この章の学習目標

- 文字列照合問題とは何か，応用例を用いて説明できる
- 力まかせ法とは何か，また，その時間計算量を説明できる
- ハッシュを利用した文字列照合アルゴリズムのアイデアと
その時間計算量を説明できる
- 線形時間の文字列照合アルゴリズムのアイデアと
その時間計算量を説明できる
- 文字列照合アルゴリズムのプログラムを書ける

Part1:文字列照合問題，
BF法，Rabin-Karp法

文字列照合問題(パターンマッチング)

- 利用分野の例
  - テキストエディタ，検索エンジン，遺伝子解析…
- **文字列照合問題**
  - 入力:
    テキスト（長さ **n** の文字列）
    パターン（長さ **m** の文字列）
  - 出力:
    テキスト中にパターンが現れるかどうか判定
    テキスト中でパターンの位置を求める
  - 多くの場合， **n>>m**

テキスト: T[0]T[1]...T[n-1]
パターン: P[0]P[1]...P[m-1]

文字列照合アルゴリズム

T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]	T[11]	T[12]	T[13]
こ	の	中	か	ら	パ	タ	ー	ン	を	見	つ	け	る
					パ	タ	ー	ン					
					P[0]	P[1]	P[2]	P[3]					

n=14, m=4
→ 答え:i = 5

アルゴリズム	計算時間
力まかせ(Brute force)法	$O(nm)$
Rabin-Karp法	平均$O(n)$ 最悪$O(nm)$
Knuth-Morris-Pratt法	最悪$O(n+m)$
Boyer-Moore法	最悪$O(n+m)$

Brute Force 法

i ↓

T	a	a	a	b	b	a	a	a	b	a	b	a	b	a	a	b	a	b	a	a	a	b	b	a	a	a	a
P	a	a	a	b	a	b																					

一致せず！

Brute Froce 法

i ↓

T	a	a	a	b	b	a	a	a	b	a	b	a	b	a	a	b	a	b	a	a	a	b	b	a	a	a	a
P		a	a	a	b	a	b																				

一致せず！

これを繰り返せばよい
1回の比較：最悪m
比較開始位置：最悪n-(m-1)

Brute Force法アルゴリズム

```
// テキスト T[0]T[1] … T[n-1]
// パターン  P[0]P[1] … P[m-1]
  for ( i = 0;  i <= n-m; i++) {
    for (j = 0; j <m; j++)
        if (T[i+j] != P[j]) break;
    if (j==m) return i;
 }
 return -1;
```

ここはフルには回らない

最悪O(nm)
(実用的にはそこまで
悪くならないことも多い)

最悪を実現する場合

T=aaaaa…aaaaab
P=aaaaab

Rabin-Karp法

- Brute Force + ハッシュ法
  - ハッシュ関数を用意する
    - 長さmの文字列を引数に取って，整数値を返す
    - （mは入力パターンのパターン長）
  - ハッシュ値は偽陽性(False-positive性)を持つ
    - 2つの同じ入力文字列が与えられると，必ず同じ値
    - 異なる２つの文字列を与えても（たまに）同じ値(衝突)
      - どれくらいの頻度で起きるかはハッシュ値の値域による
      - （基本的には，値域が広いほうが衝突が起きにくい）

Rabin-Karp法

ハッシュ値

6 5 3 0 6 5 3 6 …

T | a | a | a | b | b | a | a | a | b | a | b | a | b | a | a | b | a | b | a | a | a | b | b | a | a | a | a |

P | a | a | a | b | a | b |

h(a a a b a b) = 5

h(a a a b b a) = 6

h(a a b b a a) = 5

h(a b b a a a) = 3

ハッシュ値が異なるときは
確実に異なるパターン

⇩

ハッシュ値が等しいとき
だけBrute-force法と
同様にチェックする
（衝突があるのでチェックは必要）

Rabin-Karp法アルゴリズム

```
// テキスト T[0]T[1] … T[n-1]
// パターン   P[0]P[1] … P[m-1]
//hp（パターンのハッシュ値）と
//ht（テキストのハッシュ値）の初期値(=h(T[0]...T[m-1])

for ( i = 0;  i <= n-m; i++) {
  if (hp==ht) {   // ハッシュ値が等しい時だけチェック
    for (j = 0; j <m; j++)
      if (T[i+j] != P[j]) break;
      if (j==m) return i;
    }
  }
  // htの更新(ht =h([T[i+1]...T[i+m]))
}
return -1;
```

動画でT[0]... T[m-1] -> T[1] ... T[m]と訂正してますが，嘘です．
このままで正しいです．すいませんm(_ _)m

ハッシュ関数

- ハッシュ関数 h
  - 長さ m の文字列 → ハッシュ値（整数値）
  - $h(a_1 a_2 \cdots a_m) = (a_1 d^{m-1} + a_2 d^{m-2} + \cdots + a_m) \bmod q$

d：文字種数
q：素数

各文字は0～$d-1$の数字で符号化されているとする
(e.g., アルファベット={a,b}ならa=1, b=0のように)

d 進数として数値化

$$h_i^T = h(T[i]T[i+1]\cdots T[i+m-1])$$
$$= (T[i]d^{m-1} + T[i+1]d^{m-2} + \cdots + T[i+m-1]) \bmod q$$

$$h_{i+1}^T = h\ (T[i+1]T[i+2] \ldots T[i+m])$$
$$= (T[i+1]d^{m-1} + T[i+2]d^{m-2} + \cdots + T[i+m]) \bmod q$$
$$= ((h_i^T - T[i]d^{m-1})d + T[i+m]) \bmod q$$

- h_{i+1}^T は h_i^T から定数時間で計算可能

Rabin-Karp法アルゴリズム(完全版)

```
// テキスト T[0]T[1] … T[n-1]
// パターン   P[0]P[1] … P[m-1]
 dm=1; hp=0; ht=0;
 for(i=0;i<m-1;i++) dm=(d*dm) % q; // dm-1を事前に計算
 for(i=0;i<m;i++) {
         hp=(hp*d+P[i]) % q; ht=(ht*d+T[i]) % q;
 }
  for ( i = 0;  i <= n-m; i++) {
   if (hp==ht) {   // ハッシュ値が等しい時だけチェック
    for (j = 0; j <m; j++)
         if (T[i+j] != P[j]) break;
    if (j==m) return i;
   }
  ht=((ht-((T[i]*dm) % q) + q)*d+T[i+m]) % q;
 }
 return -1;
```

Part2：Knuth-Morris-Pratt法, Boyer-Moore法

Rabin-Karp法

- Rabin-Karp法はハッシュ値の衝突が毎回生じるとパフォーマンスが低下
  - 平均時間　O(n+m)
  - 最悪時間　O(nm)

最悪時間 O(n+m) のアルゴリズムは？

テキストに対してパターンをひとつずつずらすのが，効率を悪くしている．
　　じゃあ，どうすればいいか？

Knuth-Morris-Pratt法

Knuth-Morris-Pratt法＝照合位置スキップ

T　a a a b b a a a b a b a b a a b a b a a a b b a a a a

P　a a a b a b a b　（5文字目不一致）→**4文字目まで一致**

この情報を使えないか？

Knuth-Morris-Pratt法

Knuth-Morris-Pratt法＝照合位置スキップ

T　a a a b b a a a b a b a b a a b a b a a a b b a a a a

P　a a a b a b a b　（5文字目不一致）→**4文字目まで一致**

a a a b a b a b

a a a b a b a b

a a a b a b a b

この情報を使えないか？

ずらしても元と等しくならないからスキップ(スキップできる事実はPの内容だけから決まる！)

a a a b a b a b　4位置シフト可能

Knuth-Morris-Pratt法

- 実際にはP側の添字位置を「巻き戻す」

i

T　a a a b b b a a b a b a b a a b a b a a a b b a a a a

j

P　a a b b a b a b

Knuth-Morris-Pratt法

- 実際にはP側の添字位置を「巻き戻す」

i

T　a a a b b b a a b a b a b a a b a b a a a b b a a a a

j

P　a a b b a b a b

Knuth-Morris-Pratt法

- 実際にはP側の添字位置を「巻き戻す」

i

T　a a a b b b a a b a b a b a a b a b a a a b b a a a a

不一致！

j

P　a a b b a b a b

Knuth-Morris-Pratt法

- 実際にはP側の添字位置を「巻き戻す」

i

T　a a a b b b a a b a b a b a a b a b a a a b b a a a a

不一致！

j を後ろに戻して続きを実行
(相対的にPの位置を前に出す)

j

P　a a b b a b a b

問題：どこに戻す？

next[k] (-1≦k≦m-1): j=kで不一致が起きたときの巻戻し位置

これを前もって計算しておけばよい

Knuth-Morris-Pratt法

- 実際にはP側の添字位置を「巻き戻す」

i

T　a a a b b b a a b a b a b a a b a b a a a b b a a a a

j

P　a a b b a b a b

Knuth-Morris-Pratt法

- 実際にはP側の添字位置を「巻き戻す」

i

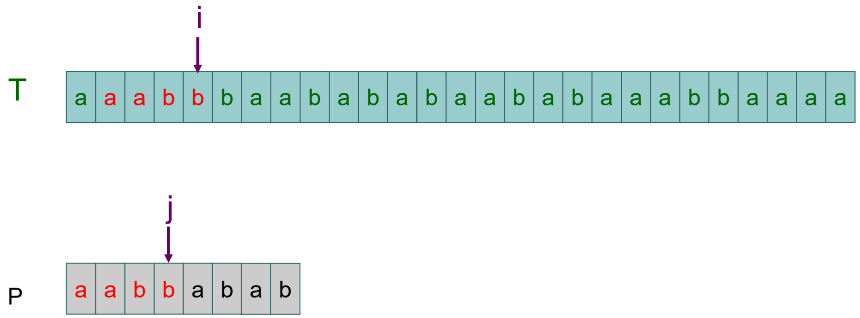
T　a a a b b b a a b a b a b a a b a b a a a b b a a a a

j

P　a a b b a b a b

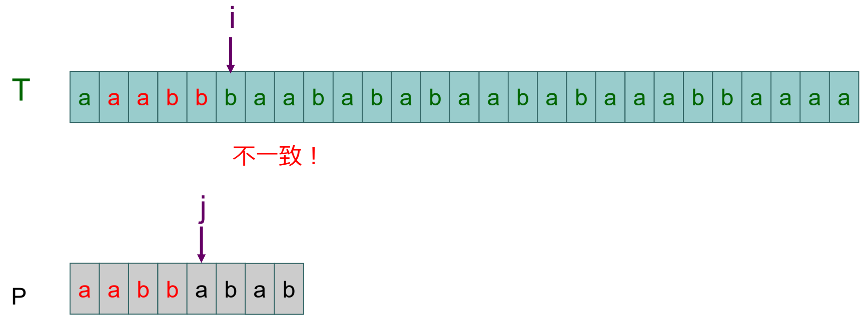
Knuth-Morris-Pratt法

- 実際にはP側の添字位置を「巻き戻す」



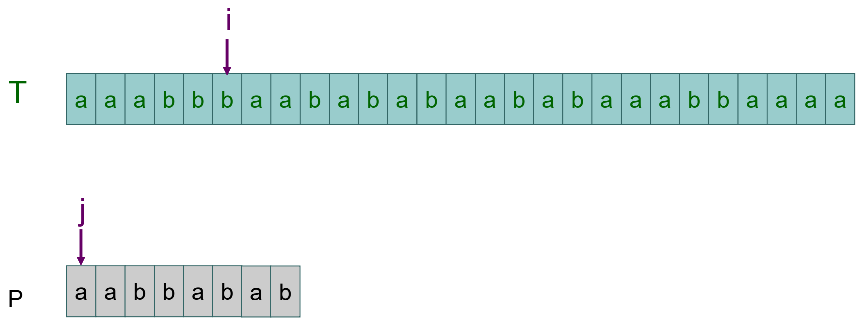
Knuth-Morris-Pratt法

- 実際にはP側の添字位置を「巻き戻す」



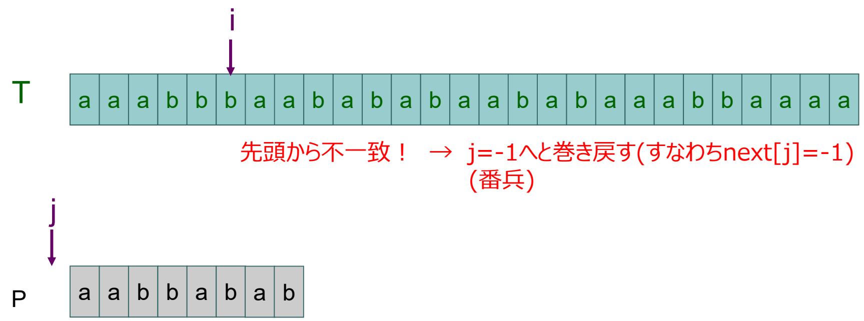
Knuth-Morris-Pratt法

- 実際にはP側の添字位置を「巻き戻す」



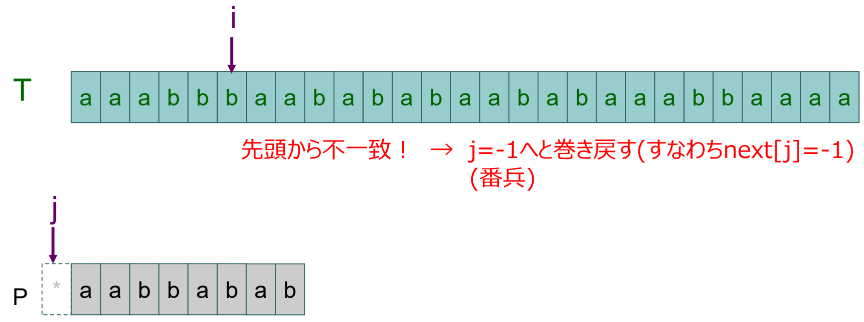
Knuth-Morris-Pratt法

- 実際にはP側の添字位置を「巻き戻す」



Knuth-Morris-Pratt法

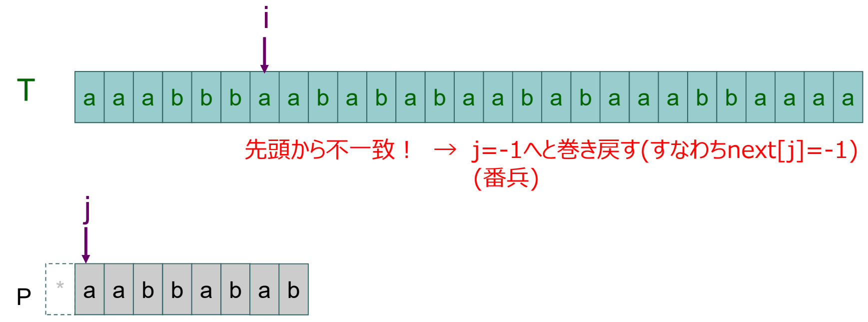
- 実際にはP側の添字位置を「巻き戻す」



P[-1]には何にでもマッチする文字が
入っているとみなす→先頭から不一致でもiが進む

Knuth-Morris-Pratt法

- 実際にはP側の添字位置を「巻き戻す」



P[-1]には何にでもマッチする文字が
入っているとみなす→先頭から不一致でもiが進む

Knuth-Morris-Pratt法

```
// テキスト T[0]T[1] … T[n-1]
// パターン   P[0]P[1] … P[m-1]
// 再開位置：next[0]=-1, next[1]～next[m-1]
  j=0;
 for ( i = 0;  i < n; i++) {
    while ((j>=0) && (T[i]!=P[j]))
        j = next[j];
    if (j == m-1) return i-m+1;
    j++;
 }
 return -1;
```

i: テキストの何文字目か
j: パターンの何文字目か

KMP法の計算時間

- KMP法は悪くとも以下の状況では終了する
  - iがnに到達したとき
  - パターンを右までシフトしきったとき
    (実際には，それに伴ってiの値も大きくなっている)
- While文の中のループ実行1回で，パターンは少なくとも右に1個ずれる
  - パターンが左にずれることはない
    →Whileループ内部は高々n回実行

nextの構成を除いて$O(n)$時間

nextの計算法

- 巻き戻し位置はどこか?
  - パターンのj文字目で不一致が発生

$$P[0]\ \boxed{\ldots P[k] \ldots \quad P[j-1]}\ \ldots \quad P[m-1]$$

位置をkずらす

$$\boxed{P[0] \ldots P[j-k-1]} \quad \ldots \quad P[m-1]$$

ここが一致する必要がある

  - $P[k]P[k+1]\ldots P[j-1] = P[0]P[1]\ldots P[j-k-1]$ なる最小の kに対して，next[j]=$j-k$となる

nextの計算法

- パタンとパタンを文字列照合する
  - 例：next[2]を計算してみる

	0	1	2	3	4	5
P	a	a	a	b	a	b
P’	a	a	a	b	a	b

右にずらす

nextの計算法

- パタンとパタンを文字列照合する
  - 例：next[2]を計算してみる

	0	1	2	3	4	5	
P	a	a	a	b	a	b	
		=					
P		a	a	a	b	a	b

next[2]=1

nextの計算法

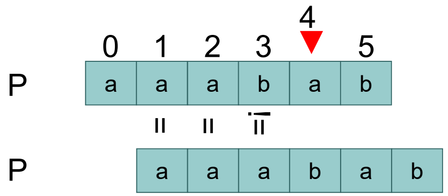
- パタンとパタンを文字列照合する
  - 例：next[4]を計算してみる

	0	1	2	3	4	5
P	a	a	a	b	a	b
P	a	a	a	b	a	b

右へずらす

nextの計算法

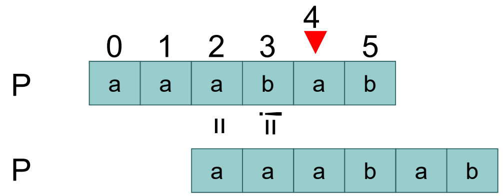
- パタンとパタンを文字列照合する
  - 例：next[4]を計算してみる



マッチしないのでもう一度

nextの計算法

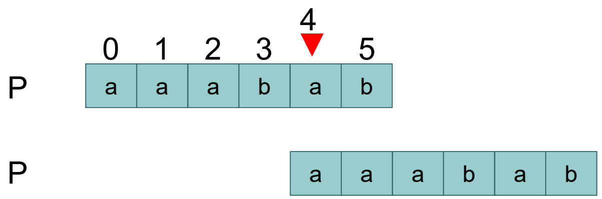
- パタンとパタンを文字列照合する
  - 例：next[4]を計算してみる



マッチしないのでもう一度
で，これを繰り返して…

nextの計算法

- パタンとパタンを文字列照合する
  - 例：next[4]を計算してみる



マッチしないのでもう一度
で，これを繰り返して…

next[4]=0

nextの計算

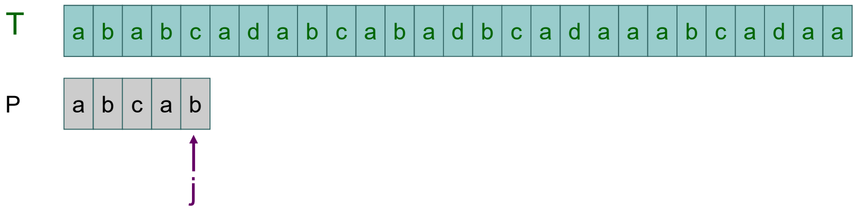
- 力まかせでやると$O(m^2)$，
- 工夫してやると$O(m)$：nextの計算自体にKMPを使う

```
// パターン  P[0]P[1] … P[m-1]
// 再開位置：next[i] の計算
j=-1;
for (i=0; i<m; i++) {
    next[i]=j;
    while ((j>=0) && (P[i] !=P[j]))
        j=next[j];
    j++;
};
```

  - 詳細は省略するが，これでnextを正しく計算できる
    - 実際にはiについての帰納法で証明する

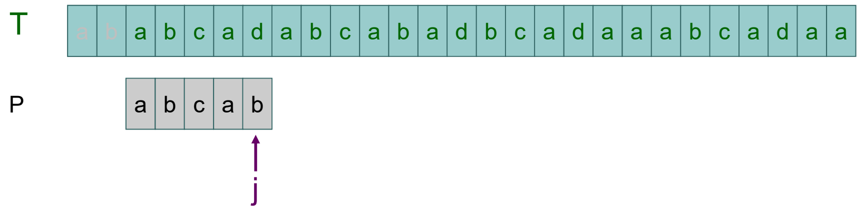
Boyer-Moore法(素朴版)

- 基本アイデア
  - テキストを前から，パタンを後ろから照合する



Boyer-Moore法(素朴版)

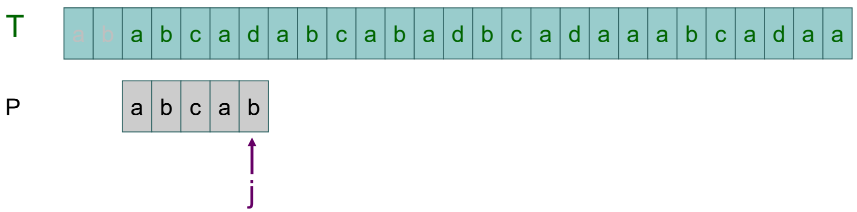
- 基本アイデア
  - テキストを前から，パタンを後ろから照合する



  - T[j]=cで不一致なので，右からチェックして最初にcが出てくるところまでPをシフトできる

Boyer-Moore法(素朴版)

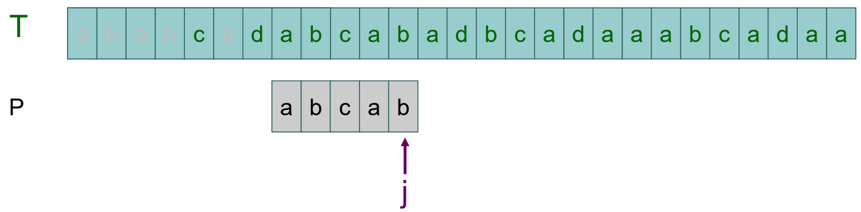
- 基本アイデア
  - テキストを前から，パタンを後ろから照合する



  - T[j]=bで不一致なので，右からチェックして最初にdが出てくるところまでPをシフトできる
→Pはdを含まないので端までシフトできる

Boyer-Moore法(素朴版)

- 基本アイデア
  - テキストを前から，パタンを後ろから照合する



  - T[j]=bで不一致なので，右からチェックして最初にdが出てくるところまでPをシフトできる
→Pはdを含まないので端までシフトできる

Boyer-Moore法(素朴版)

- 基本アイデア
  - テキストを前から，パタンを後ろから照合する

i ↓

T	a	b	a	b	c	a	d	a	b	c	a	b	a	d	b	c	a	d	a	a	a	b	c	a	d	a	a

P	a	b	c	a	b

↑ j

  - T[j]=cで不一致なので，右からチェックして最初にcが出てくるところまでPをシフトできる

Boyer-Mooreのご利益

- Boyer-Mooreはテキストを全部見ないで済むことがある
  - うまくいくと$O(n/m)$時間で終わる（し，実用的にはそうなることも多い）
  - 素朴版では最悪時は$O(mn)$時間
  - KMPのように可能シフト量を事前計算する方法と組み合わせると，最悪時を$O(n+m)$にできる(詳細は略)

参考：KMP法でのnextの計算(例)

i ↓

	0	1	2	3	4	5
P	a	a	a	b	a	b

P	a	a	a	b	a	b

↑ j

```
// パターン  P[0]P[1] … P[m-1]
// 再開位置：next[i] の計算
j=-1;
for (i=0; i<m; i++) {
    next[i]=j;
    while ((j>=0) && (P[i] !=P[j]))
        j=next[j];
    j++;
};
```

参考：KMP法でのnextの計算(例)

i ↓

	0	1	2	3	4	5
P	a	a	a	b	a	b

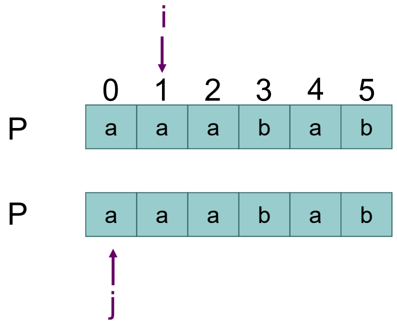
P	a	a	a	b	a	b

↑ j

```
// パターン  P[0]P[1] … P[m-1]
// 再開位置：next[i] の計算
j=-1;
for (i=0; i<m; i++) {
    next[i]=j;
    while ((j>=0) && (P[i] !=P[j]))
        j=next[j];
    j++;
};
```

next[0] = -1

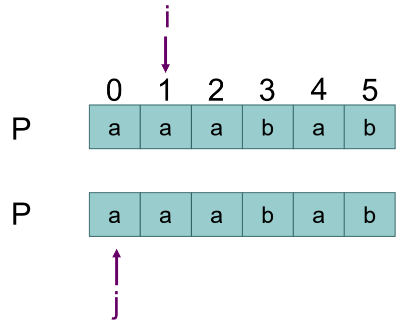
参考：KMP法でのnextの計算(例)




```
// パターン　P[0]P[1] … P[m-1]
// 再開位置：next[i] の計算
j=-1;
for (i=0; i<m; i++) {
    next[i]=j;
    while ((j>=0) && (P[i] !=P[j]))
        j=next[j];
    j++;
};
```

next[0] = -1

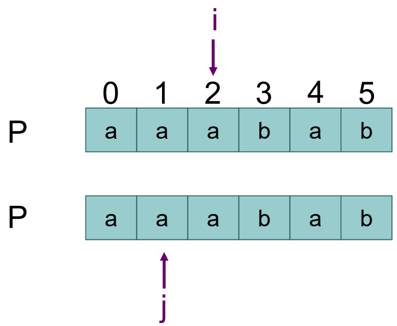
参考：KMP法でのnextの計算(例)




```
// パターン　P[0]P[1] … P[m-1]
// 再開位置：next[i] の計算
j=-1;
for (i=0; i<m; i++) {
    next[i]=j;
    while ((j>=0) && (P[i] !=P[j]))
        j=next[j];
    j++;
};
```

next[0] = -1, next[1] = 0,

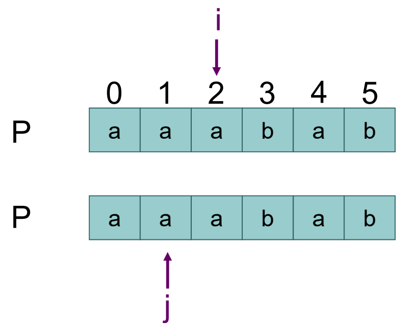
参考：KMP法でのnextの計算(例)




```
// パターン　P[0]P[1] … P[m-1]
// 再開位置：next[i] の計算
j=-1;
for (i=0; i<m; i++) {
    next[i]=j;
    while ((j>=0) && (P[i] !=P[j]))
        j=next[j];
    j++;
};
```

next[0] = -1, next[1] = 0

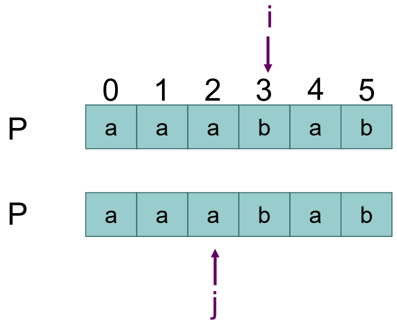
参考：KMP法でのnextの計算(例)




```
// パターン　P[0]P[1] … P[m-1]
// 再開位置：next[i] の計算
j=-1;
for (i=0; i<m; i++) {
    next[i]=j;
    while ((j>=0) && (P[i] !=P[j]))
        j=next[j];
    j++;
};
```

next[0] = -1, next[1] = 0, next[2] = 1,

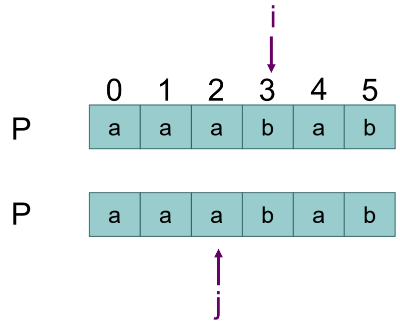
参考：KMP法でのnextの計算(例)




```
// パターン   P[0]P[1] … P[m-1]
// 再開位置：next[i] の計算
j=-1;
for (i=0; i<m; i++) {
    next[i]=j;
    while ((j>=0) && (P[i] !=P[j]))
        j=next[j];
    j++;
};
```

next[0] = -1, next[1] = 0, next[2] = 1

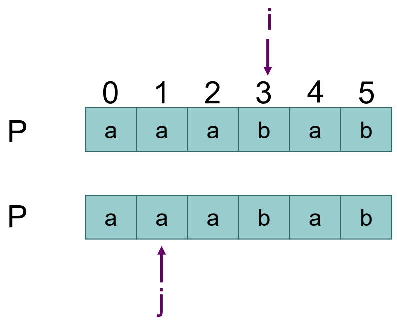
参考：KMP法でのnextの計算(例)




```
// パターン   P[0]P[1] … P[m-1]
// 再開位置：next[i] の計算
j=-1;
for (i=0; i<m; i++) {
    next[i]=j;
    while ((j>=0) && (P[i] !=P[j]))
        j=next[j];
    j++;
};
```

next[0] = -1, next[1] = 0, next[2] = 1, next[3]=2

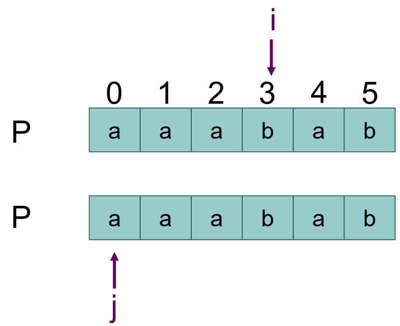
参考：KMP法でのnextの計算(例)




```
// パターン   P[0]P[1] … P[m-1]
// 再開位置：next[i] の計算
j=-1;
for (i=0; i<m; i++) {
    next[i]=j;
    while ((j>=0) && (P[i] !=P[j]))
        j=next[j];
    j++;
};
```

next[0] = -1, next[1] = 0, next[2] = 1, next[3]=2

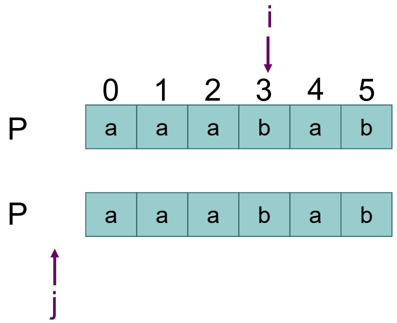
参考：KMP法でのnextの計算(例)




```
// パターン   P[0]P[1] … P[m-1]
// 再開位置：next[i] の計算
j=-1;
for (i=0; i<m; i++) {
    next[i]=j;
    while ((j>=0) && (P[i] !=P[j]))
        j=next[j];
    j++;
};
```

next[0] = -1, next[1] = 0, next[2] = 1, next[3]=2

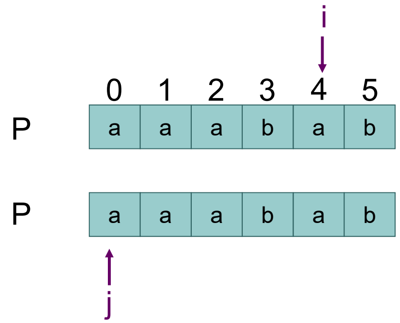
参考：KMP法でのnextの計算(例)




```
// パターン  P[0]P[1] … P[m-1]
// 再開位置：next[i] の計算
j=-1;
for (i=0; i<m; i++) {
    next[i]=j;
    while ((j>=0) && (P[i] !=P[j]))
        j=next[j];
    j++;
};
```

next[0] = -1, next[1] = 0, next[2] = 1, next[3]=2

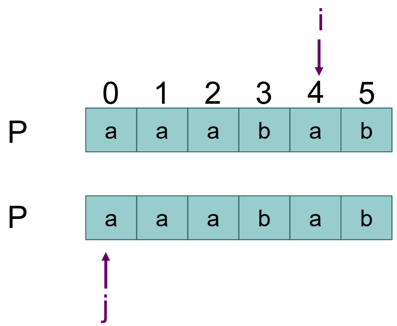
参考：KMP法でのnextの計算(例)




```
// パターン  P[0]P[1] … P[m-1]
// 再開位置：next[i] の計算
j=-1;
for (i=0; i<m; i++) {
    next[i]=j;
    while ((j>=0) && (P[i] !=P[j]))
        j=next[j];
    j++;
};
```

next[0] = -1, next[1] = 0, next[2] = 1, next[3]=2

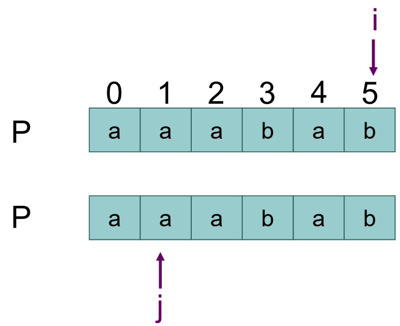
参考：KMP法でのnextの計算(例)




```
// パターン  P[0]P[1] … P[m-1]
// 再開位置：next[i] の計算
j=-1;
for (i=0; i<m; i++) {
    next[i]=j;
    while ((j>=0) && (P[i] !=P[j]))
        j=next[j];
    j++;
};
```

next[0] = -1, next[1] = 0, next[2] = 1, next[3]=2, next[4]=0,

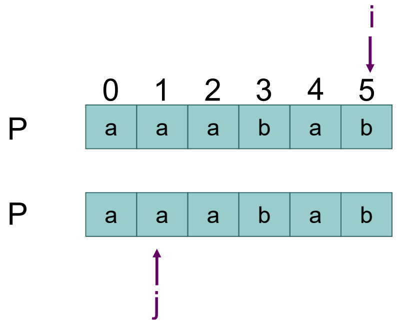
参考：KMP法でのnextの計算(例)




```
// パターン  P[0]P[1] … P[m-1]
// 再開位置：next[i] の計算
j=-1;
for (i=0; i<m; i++) {
    next[i]=j;
    while ((j>=0) && (P[i] !=P[j]))
        j=next[j];
    j++;
};
```

next[0] = -1, next[1] = 0, next[2] = 1, next[3]=2, next[4]=0

参考：KMP法でのnextの計算(例)




```
// パターン   P[0]P[1] … P[m-1]
// 再開位置：next[i] の計算
j=-1;
for (i=0; i<m; i++) {
    next[i]=j;
    while ((j>=0) && (P[i] !=P[j]))
        j=next[j];
    j++;
};
```

next[0] = -1, next[1] = 0, next[2] = 1, next[3]=2, next[4]=0,
next[5] = 1