Algorithms (2020 Summer)
#4：文字列照合

矢谷 浩司

前回の質問

「基本課題の1つ目で，タイムアウトしてしまう．」

→「ハッシュ値を計算して該当する場所を見たときに，そこが空いているか否か」で場合分けします．

もし空いていれば，他の場所にも存在しないと考えられるので，その時点で探索を止めることができます．

もう1つの場合は，オープンアドレス法で別の場所に格納されていることがあるので，線形探索する必要があります．

前回の質問

「基本課題の1つ目で，タイムアウトしてしまう．」

→「ハッシュ値を計算して該当する場所を見たときに，そこが空いているか否か」で場合分けします．

ただし削除の操作が加わると，話は少し厄介です．削除をしたというフラグを別に保存する，削除した箇所に別の場所に保存されている値を移動させてくるなどの実装が必要になってきます．

前回の質問

「二分探索木だと木を構成する際にそれぞれ要素ごとに$O(\log(n))$かかり、合計で$O(n\log(n))$の時間がかかるような気がしたのですがそんなことはないのですか？」

→よい線いっています！もう少し厳密にいくなら，二分探索木の要素の数がiのときには，要素の挿入に$O(\log(i))$回の比較が必要ですので，木全部を構成するためには，

$$\sum_{i}^{n} \log(i) \to O(\log(n!))$$

となり，これは$n\log(n) = \log(n^n)$よりは小さいです．

前回の質問

「AVL木で回転するのが$O(1)$でできるのは分かったのですが、ノードを追加する時に各ノードのバランスファクターを変えていくのは$O(1)$で出来ますか？それとも普通に$O(\log(n))$ですか？」

→授業中は$O(1)$で出来ると答えましたが，間違っておりました．失礼いたしました．結論からすると，$O(\log(n))$で合っています．

前回の質問

「AVL木で回転するのが$O(1)$でできるのは分かったのですが、ノードを追加する時に各ノードのバランスファクターを変えていくのは$O(1)$で出来ますか？それとも普通に$O(\log(n))$ですか？」

→挿入した要素ははとりあえず葉ノードになります．その後，その葉ノードに到達するまでの根ノードからの経路上において，バランスが取れているかを確認する必要があります．そのために，追加で新しくできた葉ノードから順にバランスファクターを更新します．

前回の質問

「AVL木で回転するのが$O(1)$でできるのは分かったのですが、ノードを追加する時に各ノードのバランスファクターを変えていくのは$O(1)$で出来ますか？それとも普通に$O(\log(n))$ですか？」

→最悪の場合は根ノードまで辿る必要がありますが,
根ノードに到達する前までにどこかで回転をしたり（すなわちその部分木の高さを-1する），バランスファクターが0のノードにぶつかれば（挿入したことでその部分木の左右バランスが完全に取れた），
バランスのチェックを終了することが出来ます.

前回の質問

「AVL木で回転するのが$O(1)$でできるのは分かったのですが、ノードを追加する時に各ノードのバランスファクターを変えていくのは$O(1)$で出来ますか？それとも普通に$O(\log(n))$ですか？」

→したがって，平均的には木の高さの半分でチェックを終えられると考えられるので，$O(\log(n))$となります．ただし，ノード全部のバランスファクターを更新する必要はないので，$O(\log(n))$で収まります．

授業アンケート1回目実施中！

ぜひ皆さんの声をお聞かせください．slackにてURLを流しております．無記名のアンケートです．

始まって1ヶ月経った所ですが，みなさんの感じるところをお教えください．

次回は6月上旬くらいを予定しております．

UTAS

163名もの登録があります！ありがとうございます！

UTASでの登録がありながら，講義の登録フォームからは登録をされていない方は2名いらっしゃいます．（なので，trackでの演習課題を受け取っていない．）

1名はシステム創成の方，もう1名は新領域の方です．

私かも，と思う方は至急TAさんまでご連絡をください．

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．

「“BABABCBABABDA”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．

力任せ法の例

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

1文字目からマッチング開始.

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

1文字目がダメなので，照合対象の2文字目に移動.

力任せ法の例

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

1文字目がダメなので，照合対象の2文字目に移動.

力任せ法の例

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

A	B	A	B	D

0

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			

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

力任せ法の例

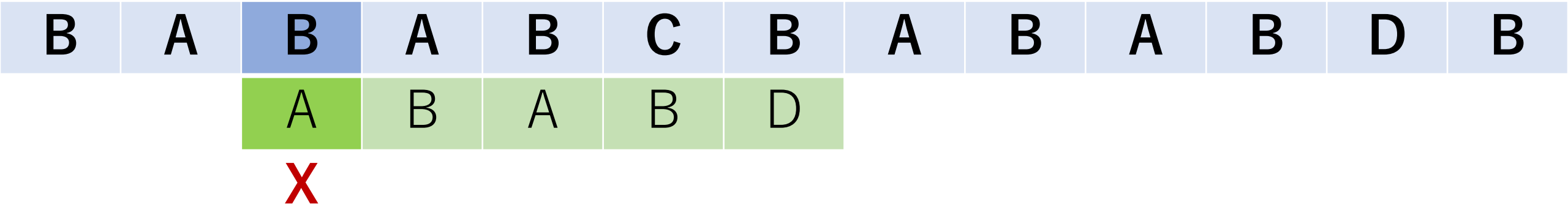
B	A	B	A	B	C	B	A	B	A	B	D	B

	A	B	A	B	D
	O	O	O	O	

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							

力任せ法の例



照合対象の2文字目からの照合がダメだったので，照合対象の3文字目に移動．以降，マッチしなかったら，照合対象の文字列における照合開始位置を1つ進めて，照合を行う．

力任せ法の例

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

見つかった場合にはその場所を返す．（例えば，先頭のindexである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:
        if text[t_i] == pattern[p_i]: # 一致している場合
            t_i += 1
            p_i += 1
        else: # 一致しなかったら後戻り
            t_i = t_i - p_i + 1
            p_i = 0
```

[見つかったことの判定は？返り値はどうなる？]

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．

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							

照合対象の2文字目からのマッチングがダメだったので,
照合対象の3文字目に移動.

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

照合対象の3文字目（B）はマッチしない.

照合対象の4，5文字目（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（何文字目まで照合したか）に応じて，次の照合位置を変更.

KMP法

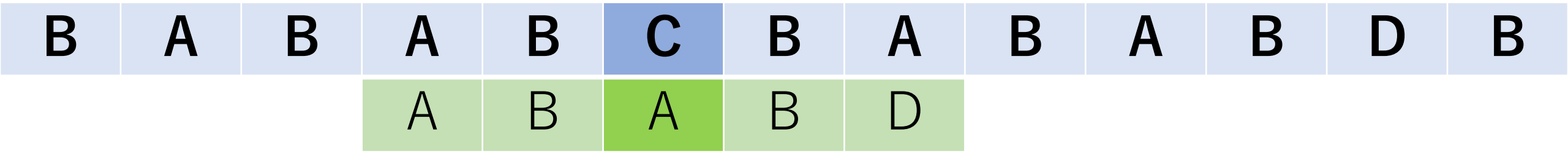
B	A	B	A	B	C	B	A	B	A	B	D	B
	A	B	A	B	D							
	O	O	O	O	X							

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



ただし，照合パターンの3文字目のA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

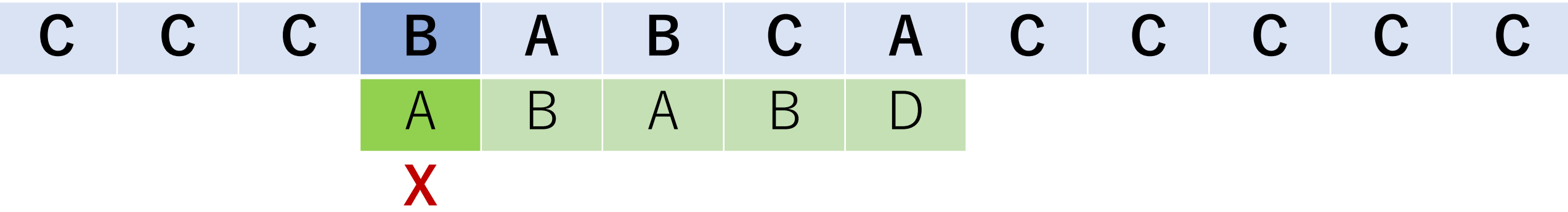
照合パターンの中に重複な並びが存在する場合には，
照合パターンのどこから再スタートするかが変わる．

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．

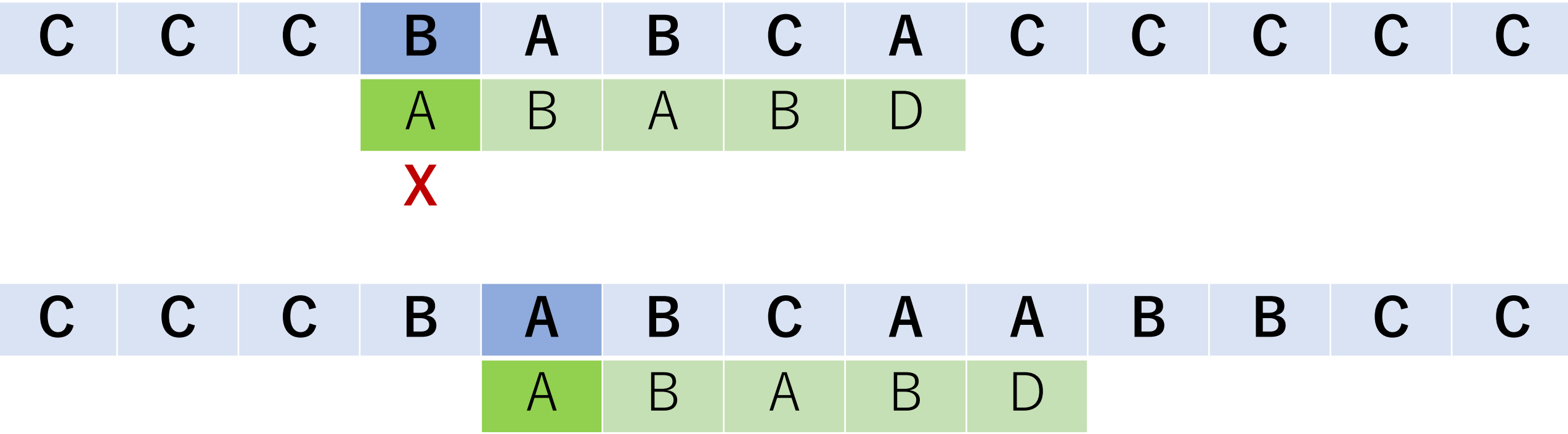
KMP法の事前準備

もし1文字目で照合失敗なら，



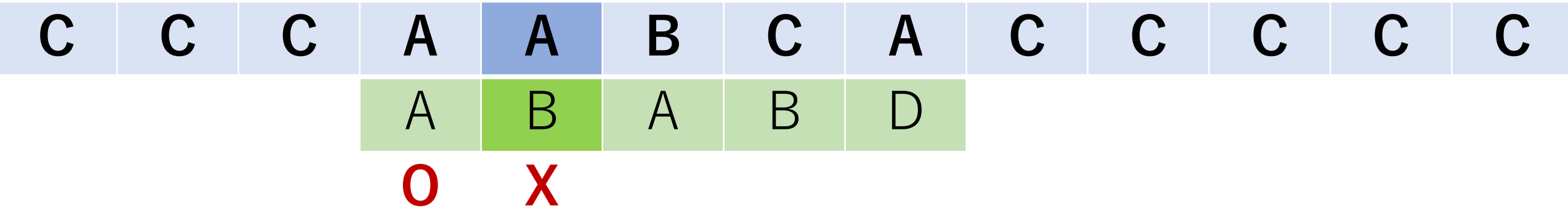
KMP法の事前準備

もし1文字目で照合失敗なら，照合対象の方を1つ移動して次の照合を行う．



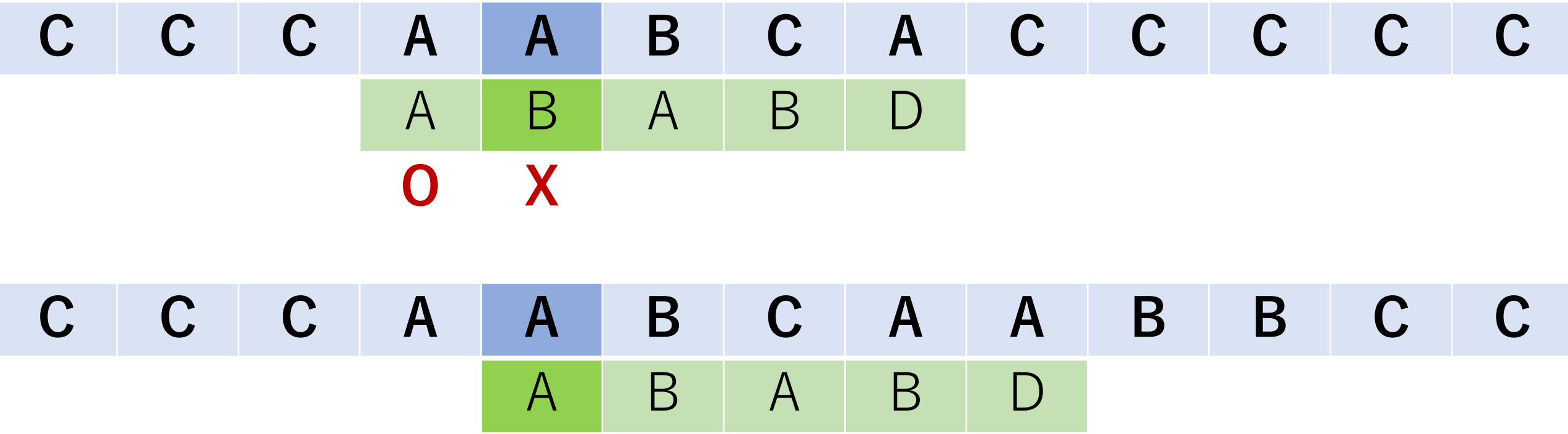
KMP法の事前準備

もし2文字目で照合失敗なら，



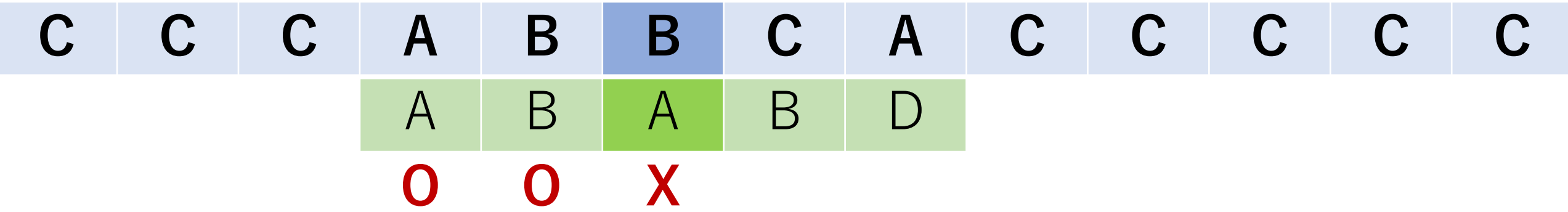
KMP法の事前準備

もし2文字目で照合失敗なら，次の照合は1文字目のAから始めることが出来る．



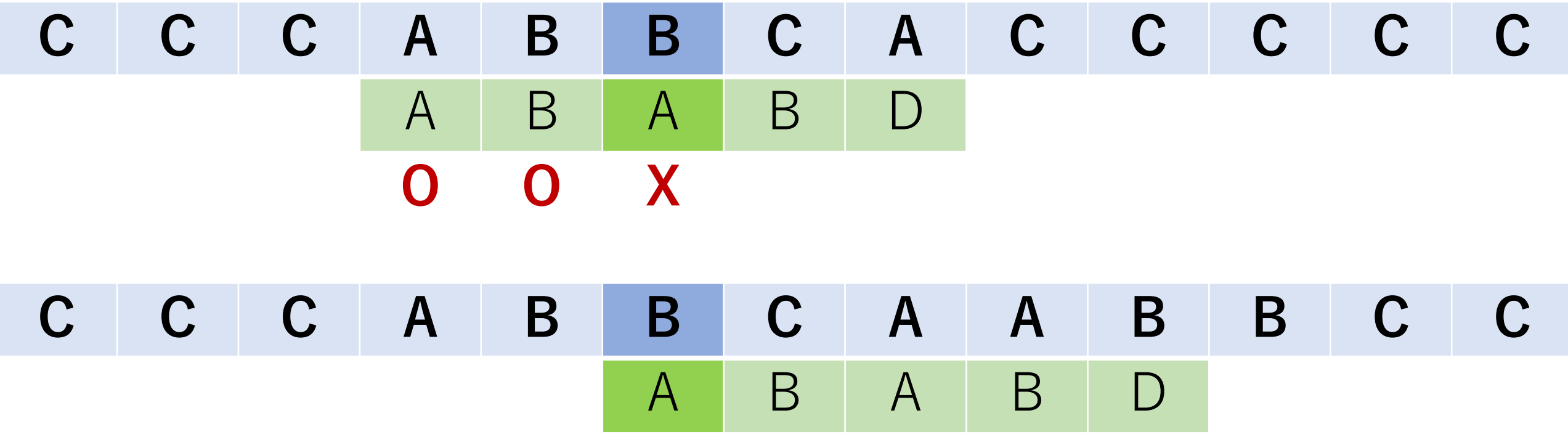
KMP法の事前準備

もし3文字目で照合失敗なら，



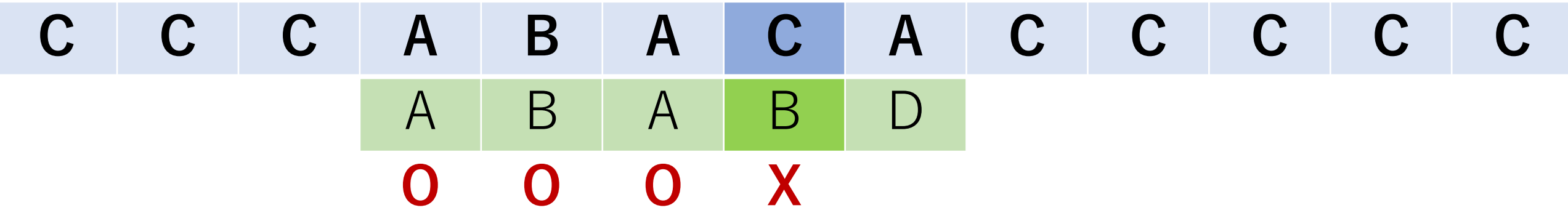
KMP法の事前準備

もし3文字目で照合失敗なら，次の照合は1文字目のAから始めることが出来る．



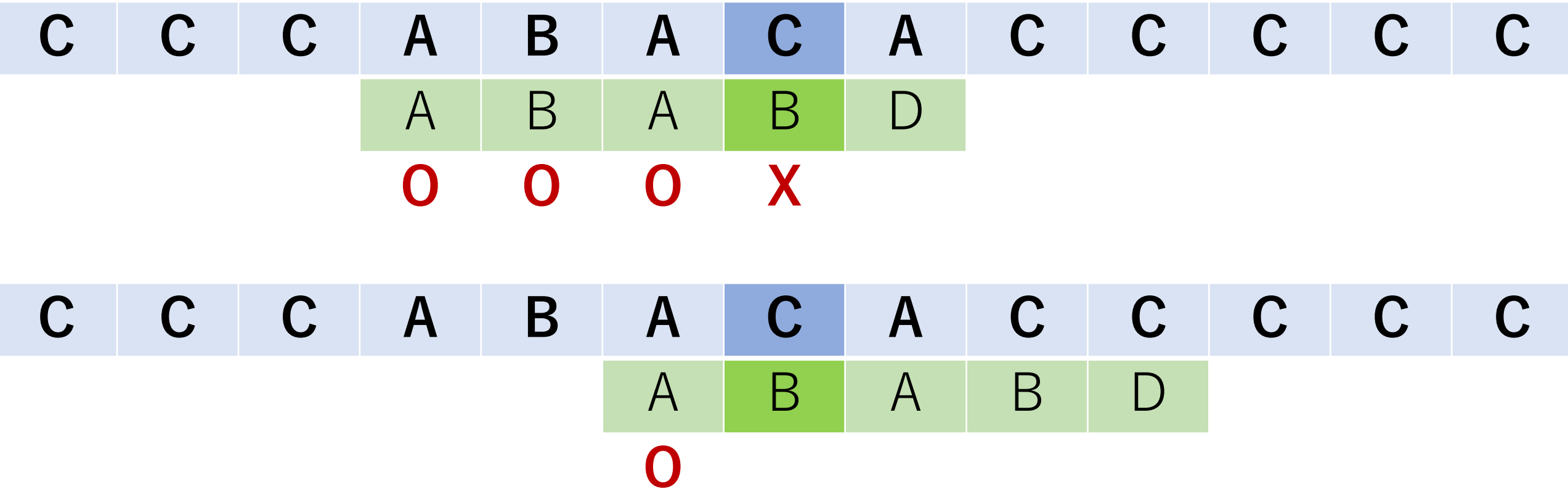
KMP法の事前準備

もし4文字目で照合失敗なら，



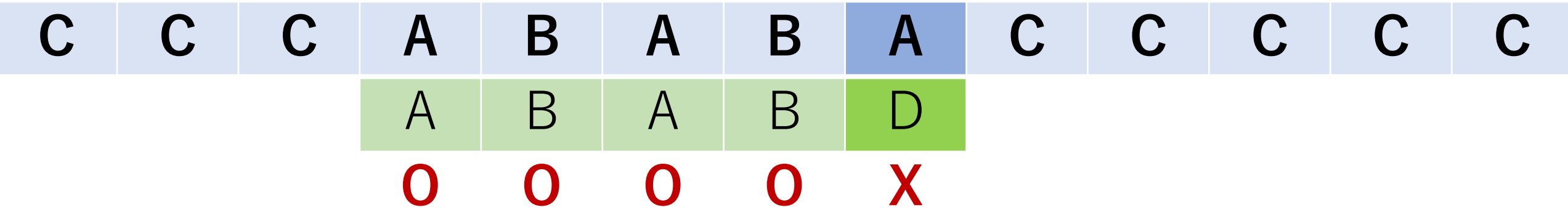
KMP法の事前準備

もし4文字目で照合失敗なら，次の照合は2文字目のBから始めることが出来る．



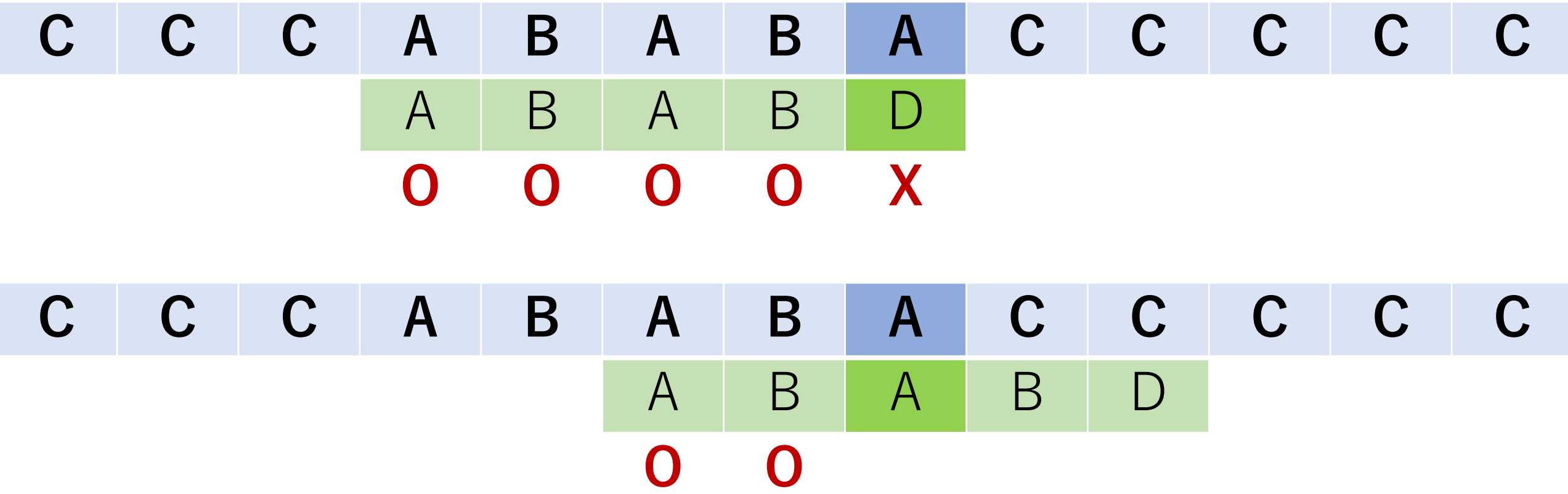
KMP法の事前準備

もし5文字目で照合失敗なら，



KMP法の事前準備

もし5文字目で照合失敗なら，次の照合は3文字目のAから始めることが出来る．



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

照合パターンのn番目（index：n-1）の文字で照合が失敗した場合，n-2番目のテーブルの値を見て，次の照合におけるパターンの開始位置を決める.

0	1	2	3	4
0				

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0．これは，2文字目（index：1）で照合失敗のときは，パターンの1文字目（index：0）に戻って照合を開始する，ことを意味している．

0	1	2	3	4
0				

KMP法の事前準備：スキップテーブル

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.

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

0	1	2	3	4
0				

KMP法の事前準備：スキップテーブル

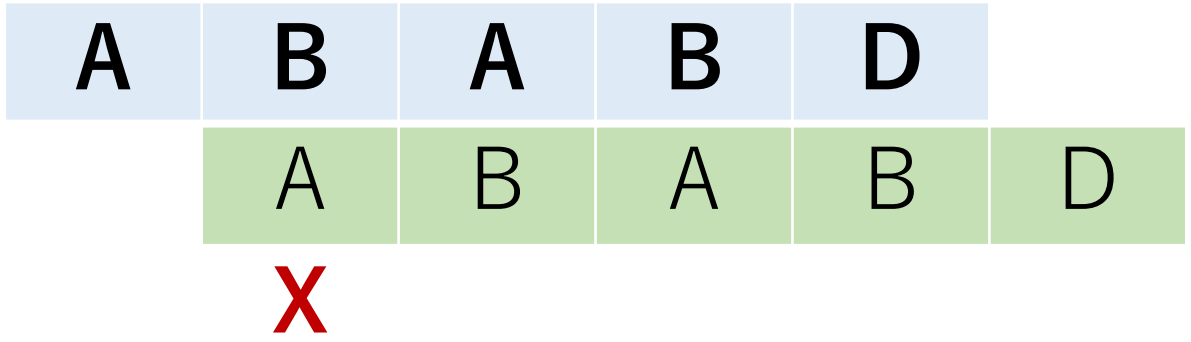
2文字目からの照合では最初から一致しないので，再開位置の値として0とする．

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

0	1	2	3	4
0	0			

KMP法の事前準備：スキップテーブル

つまり，3文字目（index：2）で照合失敗のときは，
パターンの1文字目（index：0）に戻って照合を開始する．



0	1	2	3	4
0	0			

KMP法の事前準備：スキップテーブル

3文字目からの照合ではA，Bが照合する．

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

0	1	2	3	4
0	0			

KMP法の事前準備：スキップテーブル

照合した個数をスキップテーブルに順に入れていく.

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

0	1	2	3	4
0	0	1	2	

KMP法の事前準備：スキップテーブル

4文字目（index：3）で照合失敗のときは，パターンの
2文字目（index：1）に戻って照合を開始する．

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

0	1	2	3	4
0	0	1	2	

KMP法の事前準備：スキップテーブル

5文字目（index：4）で照合失敗のときは，パターンの
3文字目（index：2）に戻って照合を開始する．

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

0	1	2	3	4
0	0	1	2	

KMP法の事前準備：スキップテーブル

最後の文字まで同様の処理を繰り返す．（テーブルの最後の値はこの実装では使用しない．照合する場所全部見つけてこようとするならこの値を使う．）

A	B	A	B	D

A	B	A	B	D

X

0	1	2	3	4
0	0	1	2	0

KMP法の実装例

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

        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法の実装例

```
def kmp(text, pattern):
    …
    # 照合を行うループ
    while t_i < t_len and p_i < p_len:
```

KMP法の実装

```
def kmp(text, pattern):
        …
        while t_i < t_len and p_i < p_len:
                if text[t_i] == pattern[p_i]:      # 一致している場合
                        t_i += 1      # お互いのカーソルを1つ進める
                        p_i += 1
```

KMP法の実装例

```
def kmp(text, pattern):
        …
        while t_i < t_len and p_i < p_len:
                if text[t_i] == pattern[p_i]:     # 一致している場合
                        t_i += 1      # お互いのカーソルを1つ進める
                        p_i += 1
                elif p_i == 0:        #照合パターン1文字目でマッチ失敗
                        t_i += 1
```

KMP法の実装例

```
def kmp(text, pattern):
        …
        while t_i < t_len and p_i < p_len:
                if text[t_i] == pattern[p_i]:     # 一致している場合
                        t_i += 1      # お互いのカーソルを1つ進める
                        p_i += 1
                elif p_i == 0:        #照合パターン1文字目でマッチ失敗
                        t_i += 1
                else:
                        [スキップテーブルを使ってジャンプ]
```

KMP法の実装

```
def kmp(text, pattern):
    ...
```

[どういう条件なら文字列が見つかったことになる？]
[返り値はどうなる？]

KMP法の計算量

力任せ法と違って必ず前に1つは進む．t_iが後戻りしない，というのが大きな特徴.

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．．．

$n \gg l$ なので，力任せ法 $O(nl)$ とKMP法に大きな差は出ない．

力任せ法の最悪ケースはかなり意地悪なケースなので，
実際問題ほぼ起きない．

KMP法のほうが処理が少し複雑なので，遅くなることも．

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．．．

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．

照合の多くは1〜数文字目ですぐ失敗する．

このために，大概の場合で1〜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								

X

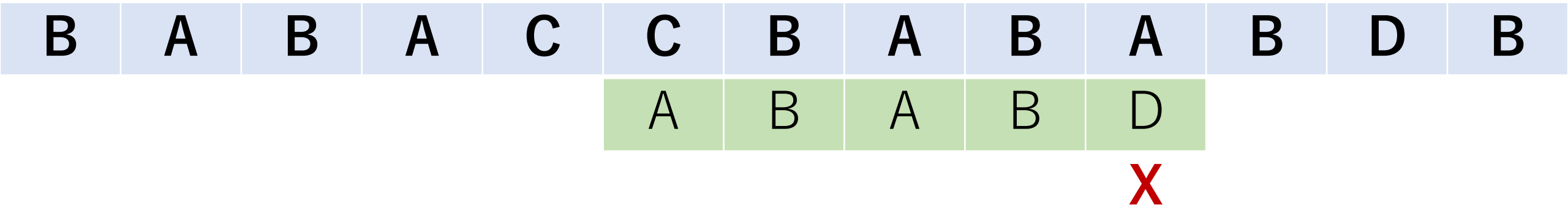
Cはパターンにない文字．なので，この5文字目のCが含まれるような照合開始位置は全て却下すべき．

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



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3文字目の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照合．）

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，最後に出現する位置が$i\ (0 \leq i \leq l-1)$ならば，移動量$l-i-1$.

BM法の事前準備

先ほどの例のABABDの場合.

A	B	C	D	E
2	1	5	0	5

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.

BM法の実装例

```
def bm_search(text, pattern):
        t_len = len(text)
        p_len = len(pattern)
        skip = [p_len]*26          #アルファベット小文字のみの想定

        # ord()：文字をアスキーコードにする関数
        # 例）ord('a') -> 97
        [aからzまでのスキップテーブルを作る]
```

BM法の実装例

```
def bm_search(text, pattern):
    …
    t_i = p_len – 1     # 照合対象側のカーソルを予め進める.
    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，skipをみてt_iを前に進める]
    return -1     # 見つからなかった場合
```

BM法の計算量

頻繁に似たようパターンが現れない場合，毎回の照合で長さlだけすっ飛ばせる．

多くの場合$O(n/l)$．ただし，最悪のケースは$O(nl)$．
（最悪のケースはどんな場合？）

工夫をすることで最悪のケースでも$O(n+l)$にできる．

ビット列のような同じ文字が頻繁に出現するケースでは不利．

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<u>ローリングハッシュ</u>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$p_1, p_2 \ldots, p_l$に対してハッシュ値を計算する．

$$H(P) = \left(a^{l-1}p_1 + \cdots + \ a^0 p_l\right) \bmod h$$

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．1番目の文字からl番目までの部分文字列のハッシュは,

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

この$H(S,1,l)$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2番目の文字から$l+1$番目までの部分文字列のハッシュを計算する．すなわち，

$$H(S, 2, l+1) = \left(a^{l-1}s_2 + \cdots + a^0 s_{l+1}\right) \bmod h$$

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をまともに毎回やってしまうと，結局$O(l)$の計算量になってしまい，全体では$O(n)$回計算が発生するので，力任せ法と変わらない．．．

ローリングハッシュ

$a^{l-1}s_1 + \cdots + a^0 s_l = H(S, 1, l) + Ah$と表される.

$H(S, 2, l+1)$の計算では$a^{l-1}s_1 + \cdots + a^0 s_l$を再利用できるところがある.

ローリングハッシュ

$$a^{l-1}s_2 + \cdots + a^1 s_l + \ a^0 s_{l+1}$$

ローリングハッシュ

$$a^{l-1}s_2 + \cdots + a^1 s_l + \ a^0 s_{l+1}$$

$$= a\left(a^{l-2}s_2 + \cdots + \ a^0 s_l\right) + \ a^0 s_{l+1}$$

ローリングハッシュ

$$a^{l-1}s_2 + \cdots + a^1 s_l + a^0 s_{l+1}$$

$$= a\left(a^{l-2}s_2 + \cdots + a^0 s_l\right) + a^0 s_{l+1}$$

$$= a\left(a^{l-1}s_1 + a^{l-2}s_2 + \cdots + a^0 s_l\right) - a^l s_1 + a^0 s_{l+1}$$

ローリングハッシュ

$$a^{l-1}s_2 + \cdots + a^1 s_l + a^0 s_{l+1}$$

$$= a\left(a^{l-2}s_2 + \cdots + a^0 s_l\right) + a^0 s_{l+1}$$

$$= a\left(a^{l-1}s_1 + a^{l-2}s_2 + \cdots + a^0 s_l\right) - a^l s_1 + a^0 s_{l+1}$$

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

ローリングハッシュ

よって,

$$H(S,2,l+1) = \left(aH(S,1,l) - a^{l}s_1 + a^{0}s_{l+1}\right) \bmod h$$

を計算すればよく，これは$O(1)$で計算可能.

累積和的な話. 😊

ローリングハッシュによる照合の実装例

```
def RollingHashMatch(text, pattern):
    base = 1007    # 基数
    h = 10**9 + 7  # 除数

    t _len = len(text)
    p_len = len(pattern)
    t_hash = 0
    p_hash = 0
```

ローリングハッシュによる照合の実装例

```
def RollingHashMatch(text, pattern):
    …
    # 文字列先頭からのハッシュを計算
    for i in range(p_len):

        # このループを使って，下のものを計算したい.
        # H(P) = (a^{l-1}p_1 + … + a^0 p_l) mod h
```

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(t_len):
        # i番目まで：(a^{i-1}p_1 + … + a^0 p_{i-1} + a^0 p_i) mod h
        # i-1番目まで：(a^{i-2}p_1 + … + a^0 p_{i-1}) mod 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.
        [t_hashを更新（累積和的考え方）]
        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，$O(n+l)$．ただし各回の照合が$O(1)$なので,
複数パターンの検索に有効.

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.

除数は大きいほうが良い.

基数は素数であるほうが良いとされる.

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’
“aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa‥aaaaaaaaaab”
パターン文字列：'a’*100+’b’
“[a100個]b”

力任せ法： 33,635 msec
KMP：771 msec
BM：909 msec
ローリングハッシュ：871 msec

パフォーマンス比較例

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

力任せ法：519 msec
KMP：468 msec
BM：1,281 msec
ローリングハッシュ：837 msec

パフォーマンス比較例

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

力任せ法：508 msec
KMP：461 msec
BM：21 msec
ローリングハッシュ：861 msec

今までの文字列照合アルゴリズム

今までに紹介した方法は，照合の度に頭からマッチングをしていく．前回の話でいえば，処理で問題を解決する方法.

では，データ構造で解決できない？
照合に便利な「索引」を予め作っておいて，
実際の照合時にはそれを利用する.

Suffix Array

与えられた文字列の接尾辞とその開始位置を全てリストアップし，ソートしたもの．

suffix（接尾辞）：文字列の後ろi文字分を取ったもの．

suffixの接尾辞：suffix, uffix, ffix, fix, ix, x

Suffix Array

例：abracadabra

#1：全suffixをリストアップ.

開始位置	suffix
0	abracadabra
1	bracadabra
2	racadabra
3	acadabra
4	cadabra
5	adabra
6	dabra
7	abra
8	bra
9	ra
10	a

Suffix Array

例：abracadabra

#1：全suffixをリストアップ.

#2：辞書順にソート.

開始位置	suffix
10	a
7	abra
0	abracadabra
3	acadabra
5	adabra
8	bra
1	bracadabra
4	cadabra
6	dabra
9	ra
2	racadabra

Suffix Array

このテーブルを使うことで,
「照合パターンをprefixに
持つようなsuffixを探し出す」
という問題に帰着できる.

例えば“ra”を見つけたければ,
各suffixの最初の2文字を見れば
良い.

開始位置	suffix
10	a
7	abra
0	abracadabra
3	acadabra
5	adabra
8	bra
1	bracadabra
4	cadabra
6	dabra
9	ra
2	racadabra

Suffix Array

さらに，suffixは辞書順にソートされているので，二分探索を使って効率的に探索範囲を絞ることが可能.

開始位置	suffix
10	a
7	abra
0	abracadabra
3	acadabra
5	adabra
8	bra
1	bracadabra
4	cadabra
6	dabra
9	ra
2	racadabra

Suffix Array

例）“abra”を見つけたい.

二分探索をして，一致を調べる範囲を絞っていく.

開始位置	suffix
10	a
7	abra
0	abracadabra
3	acadabra
5	adabra
8	bra
1	bracadabra
4	cadabra
6	dabra
9	ra
2	racadabra

Suffix Array

例）“abra”を見つけたい.

braはabraよりも辞書順では
後ろになる.

開始位置	suffix
10	a
7	abra
0	abracadabra
3	acadabra
5	adabra
8	bra
1	bracadabra
4	cadabra
6	dabra
9	ra
2	racadabra

Suffix Array

例）“abra”を見つけたい.

よって，bra以後のものは
探索範囲から除外.

開始位置	suffix
10	a
7	abra
0	abracadabra
3	acadabra
5	adabra
8	bra
1	bracadabra
4	cadabra
6	dabra
9	ra
2	racadabra

Suffix Array

例）“abra”を見つけたい.

二分探索を続ける.

開始位置	suffix
10	a
7	abra
0	abracadabra
3	acadabra
5	adabra
8	bra
1	bracadabra
4	cadabra
6	dabra
9	ra
2	racadabra

Suffix Array

例）“abra”を見つけたい.

この場合も同じく，　acadabra
以降の範囲を除外.

開始位置	suffix
10	a
7	abra
0	abracadabra
3	acadabra
5	adabra
8	bra
1	bracadabra
4	cadabra
6	dabra
9	ra
2	racadabra

Suffix Array

例）“abra”を見つけたい.

開始位置	suffix
10	a
7	abra
0	abracadabra
3	acadabra
5	adabra
8	bra
1	bracadabra
4	cadabra
6	dabra
9	ra
2	racadabra

Suffix Array

例）“abra”を見つけたい.

abraが先頭一致するが,
abracadabraは辞書順では
abraより後ろになるので,
探索を続ける.

abracadabraは探索範囲から
は除外しない.

開始位置	suffix
10	a
7	abra
0	abracadabra
3	acadabra
5	adabra
8	bra
1	bracadabra
4	cadabra
6	dabra
9	ra
2	racadabra

Suffix Array

例）“abra”を見つけたい.

abraに完全にマッチ．ここで，
Suffix arrayではこれより上は，
abraよりも辞書順で前の単語に
なるので，調べる必要がない.

開始位置	suffix
10	a
7	abra
0	abracadabra
3	acadabra
5	adabra
8	bra
1	bracadabra
4	cadabra
6	dabra
9	ra
2	racadabra

Suffix Array

例）“abra”を見つけたい.

abraに完全にマッチ．ここで，
Suffix arrayではこれより上は，
abraよりも辞書順で前の単語に
なるので，調べる必要がない.

開始位置	suffix
10	a
7	abra
0	abracadabra
3	acadabra
5	adabra
8	bra
1	bracadabra
4	cadabra
6	dabra
9	ra
2	racadabra

Suffix Array

例）“abra”を見つけたい.

あとは，この位置から絞り込んだ範囲の中で順に照合をし，先頭一致が確認できた開始位置を返す.

開始位置	suffix
10	a
7	abra
0	abracadabra
3	acadabra
5	adabra
8	bra
1	bracadabra
4	cadabra
6	dabra
9	ra
2	racadabra

Suffix Arrayの計算量（構築）

Suffix Arrayのソートが最も時間がかかるパート．

長さnの文字列のsuffixは$O(n)$なので，比較には$O(n)$がかかる．さらに，クイックソート分の$O(n \log n)$かかるので，合わせて$O(n^2 \log n)$.

ただし，$O(n (\log n)^2)$や$O(n)$にできる方法なども存在する．（例，SA-IS ← 文字列の巡回シフトのソートを$O(n)$ で実現できるもの）．

Suffix Arrayの計算量（照合）

二分探索をし，各段階でパターンの長さl分照合を行うので，全体として$O(l \log n)$.

こちらもさらに高速化できて，$O(l + \log n)$ などにできる.（例，最長共通接頭辞）.

出現位置をすべて返すことが可能.

まとめ

力任せ法，KMP法，BM法

ローリングハッシュ

Suffix Array

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#4-a [1.5点]

KMP法を実装してください.

（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#4-b [1.5点]

ローリングハッシュによる照合を実装してください.

（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：Extra課題#4 [3点]

今日習ったアルゴリズムなどを活用して，文字列探索をする問題.