

Algorithms (2021 Summer)

#5 : 文字列照合

矢谷 浩司

前回出た質問

「先週のエクストラ課題で明らかに時間計算量が間に合わない解法が通ってしまったのですが、これはテストケースの網羅性をあえて緩めているのでしょうか」

→TAの役山さんより

基本的にはそのような解法は落とそうと作っています。制約を緩めに作っている関係で完全とはいかないのでご了承ください。

前回出た質問

「while i < len(s): を for i in range(len(s)): に変えることで比較の回数ないし計算量は減らせますか？」

→C言語のようにforループの実装でも比較がある場合は、変更しても比較の回数は減りません。

例) for (int i = 0; i < n; i++)

Pythonの場合は、forループで書くと比較をしない（range型のオブジェクトから順に要素を取得して処理をする）ので、比較の回数は減りますが、計算量としては $O(n)$ に変わりありません。

前回出た質問

「今回のスライドの95ページなどで、AVL木における回転をしたのち、2のノードのbalance factorが0になると思ったのですが、どうして1なのでしょう。」

→ご指摘ありがとうございます。こちらの表記ミスです。修正したものをこのあとすぐ、あげておきました。

前回出た質問

「2分木では同じ値はどのように扱うのですか？」

→同じ値の場合にどちらの子に移動させるかを固定してあれば問題ありません.

前回出た質問

「（花束の問題の解法の説明に関して）不等式の両辺を足した変形で得られた K は必要条件だと思うのですが、十分性の確認は必要ありませんか？」

→TAの役山さんより.

除算を切り捨てとして扱うと必要十分になります.

前回出た質問

「Treapから目的の値を探索するにはどうするのですか？
また、木の回転で、「要素の順序を崩さずに」とはどういう意味ですか？」

→TAの鈴木さんより.

探索は通常の二分木と同じ方法でOKです(根ノードから順に、現在見ているノードと目的の値の大小から左側の子か右側の子を選んで進む).

「要素の順序を崩さずに」というのは、[左の子]
< [親] < [右の子] の大小関係を崩さずに、という
意味です.

前回出た質問

「(ハッシュの衝突は実際のところどのくらい起こるのでしょうか)」

→TAの役山さんより.

hashの取り方に依存します(例えばスライドの mod9の例だと1/9の確率で衝突します).

前回出た質問

「Extra課題の大体の難易度を毎問書いておいていただくことは可能ですか。参考にしたいです。」

→一概に難易度を決めるのはなかなか難しいのですが、slackでの反応を見る限りアンケートを試みるのは良さそうなので、zoom上で流して見たいと思います。ぜひご回答ください。

前回出た質問

「基本課題4_bのテストケース15-20で全て10002番目のデータでエラーが出るのですが、何か考えられうる原因はあるでしょうか。」

→これはtrack側のバグであることがわかりました。すでにシステム側での対応が進んでおり、こちらもどのようなときにバグが発生しうるかを確認していますので、今後は発生しないと思います。ご迷惑をおかけしました。

前回出た質問

「基本課題4_bのテストケース15-20で全て10002番目のデータでエラーが出るのですが、何か考えられうる原因はあるでしょうか。」

→なお、Giveryさんからは「この問題によるジャッジへの影響（正答が誤答として見なされる、またその逆）はありません。」という回答も頂戴しています。

前回出た質問

「基本課題4_bのテストケース15-20で全て10002番目のデータでエラーが出るのですが、何か考えられうる原因はあるでしょうか。」

→エラーの場所をより詳しく見るために、以下のコマンドを利用することも可能ですので、ご参考までに.

```
python3 main.py < ./test/in/basic/[入力するファイル名] |  
(cat ./test/out/basic/[想定出力結果のファイル名] | diff  
/dev/fd/3 -) 3<&0
```

前回出た質問

「A問題の大きいケースで必ずタイムアウトします。小さいケースは全て通っているのもので間違ったコードではないと思うのですが、どこか引っかかりそうなポイントはあるでしょうか？」

→以下の2つがポイントかと思います。

- ・ハッシュ初期値でハッシュテーブルを見た時に値が見つからない場合は、そこで探索を終える。
- ・ハッシュテーブル上を線形探索をする場合、どこかで空のセルにぶつかればそこで探索を終了する。

これらがないと不必要な探索を行ってしまいます。

前回出た成果

「dequeを履修していたおかげで今日のAtCoderのD問題が解けました。皆さんも是非復習がてら解いてみてください。
https://atcoder.jp/contests/zone2021/tasks/zone2021_d」

→よかったですね。 😊 Great job !

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。

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，文字列探索などとも．

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

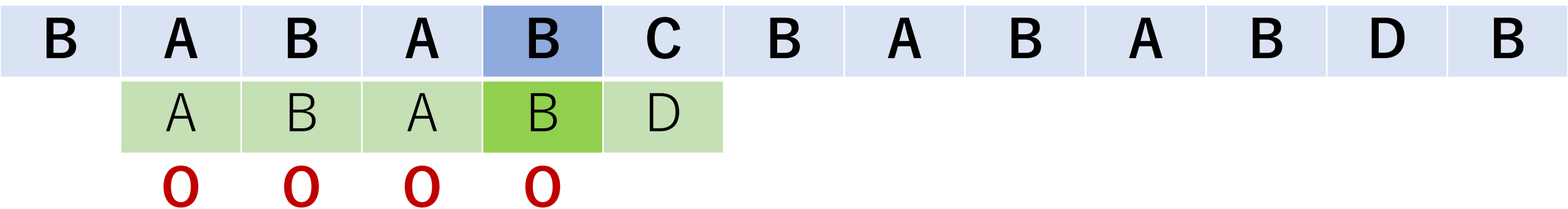
1文字目がダメなので、照合対象の2文字目に移動.

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.

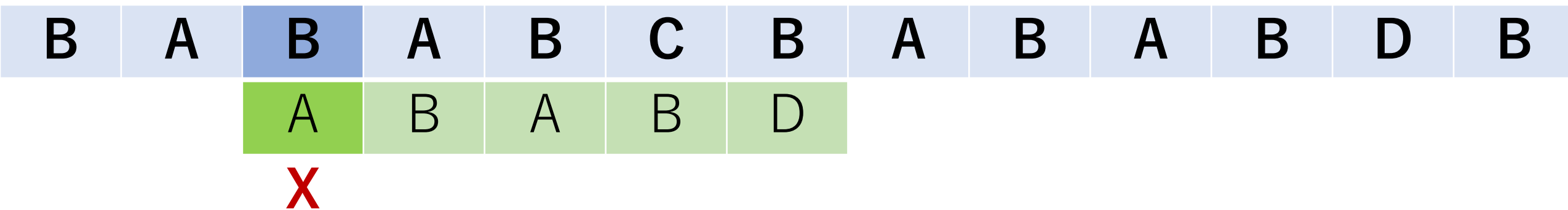
力任せ法の例



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	0	0	0	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:
```

```
        # 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 $O(nl)$.

(最悪の場合にはどんな場合？)

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							
	0	0	0	0	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							
	0	0	0	0	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

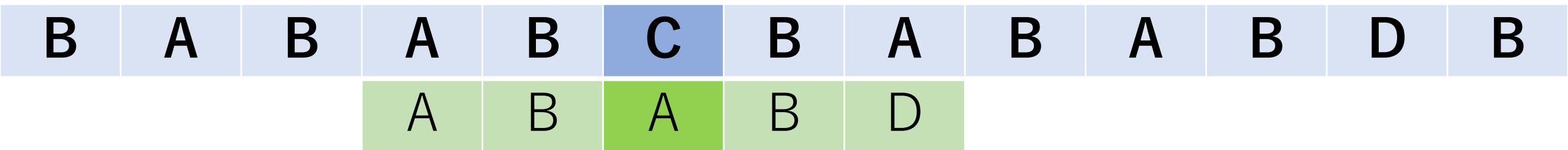
B	A	B	A	B	C	B	A	B	A	B	D	B
	A	B	A	B	D							
	0	0	0	0	X							

KMP法

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

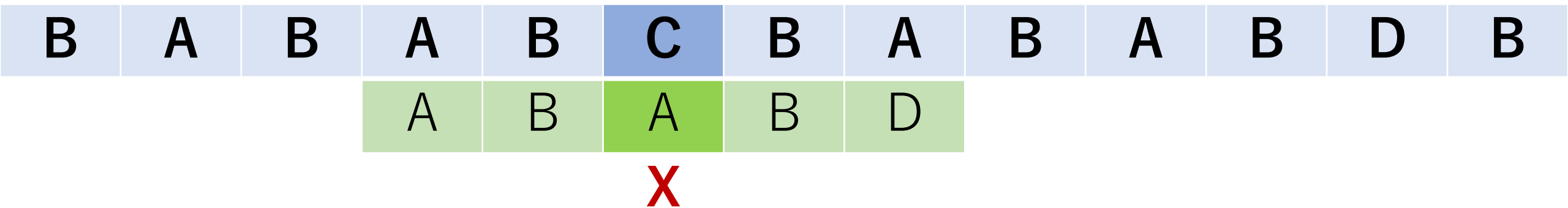
照合が失敗したCからスタート。（照合対象の方の開始位置は更新しない。）

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



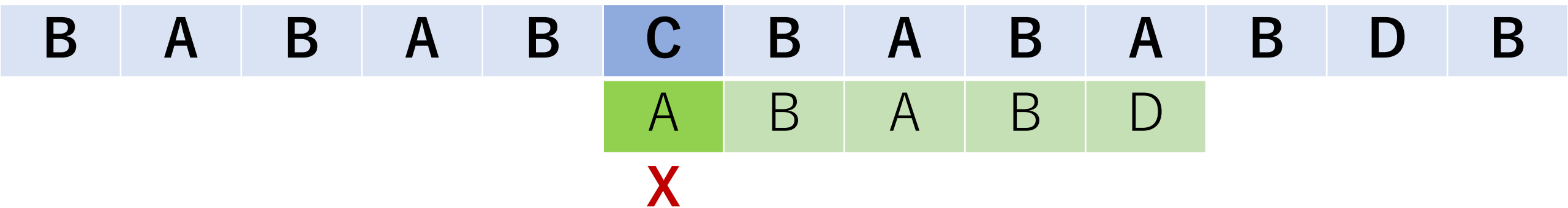
今回の場合は、それでも照合失敗.

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2文字目の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、1文字目まで移動。

KMP法



ただし，今回はこれもだめ．．．

KMP法



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

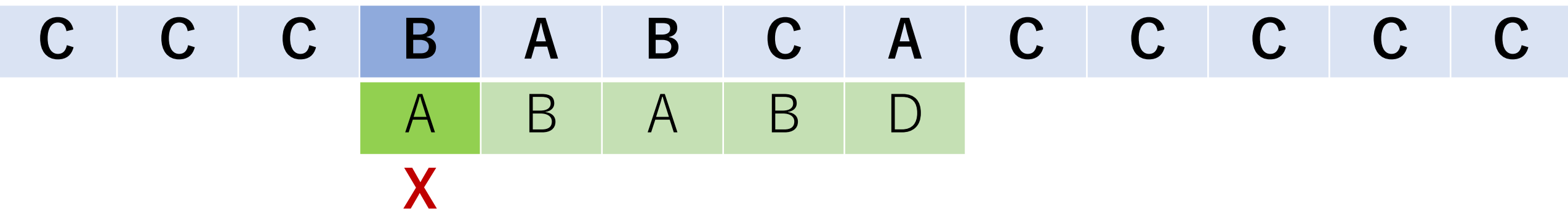
照合パターンの中に重複な並びが存在する場合には、照合パターンのどこから再スタートするかが変わる。

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。

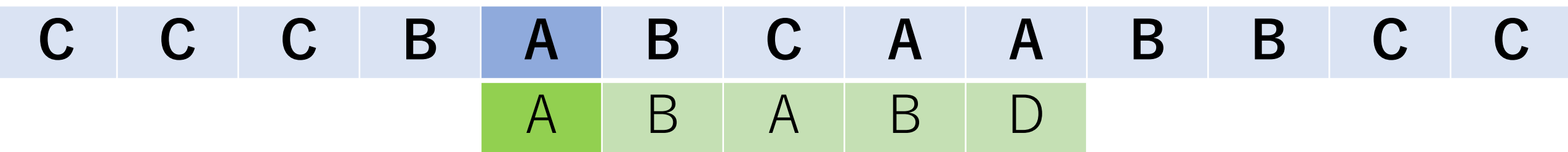
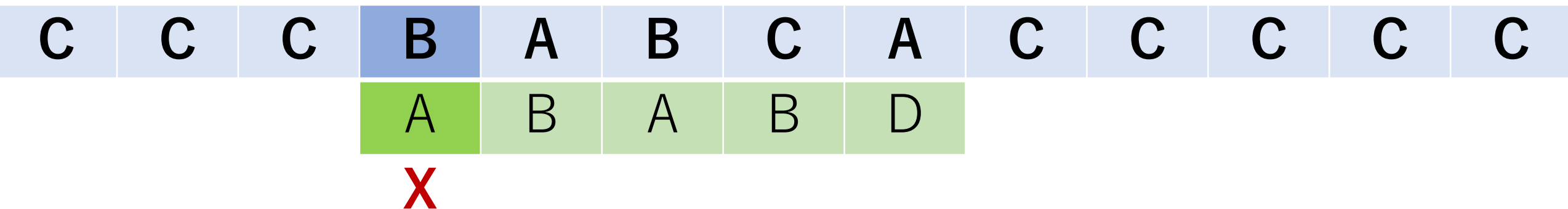
KMP法の事前準備

もし1文字目で照合失敗なら、



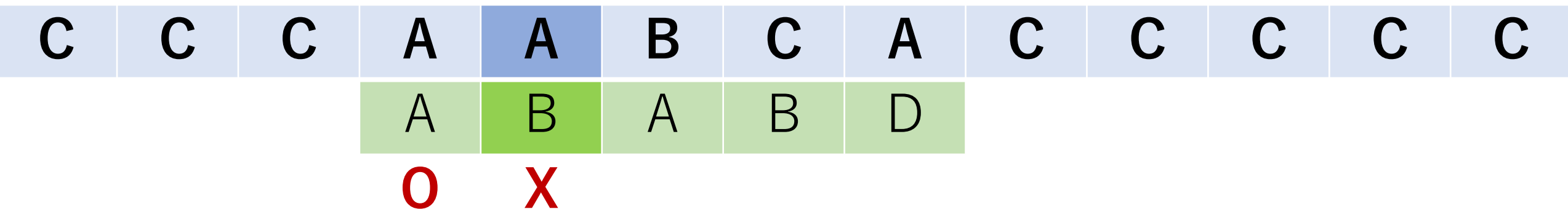
KMP法の事前準備

もし1文字目で照合失敗なら，照合対象のカーソルを1つ右に移動して次の照合を行う。



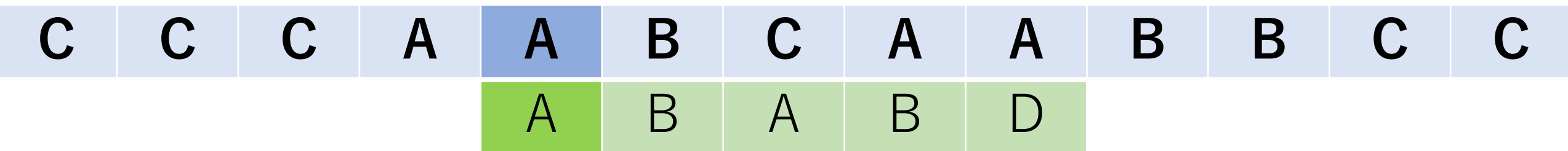
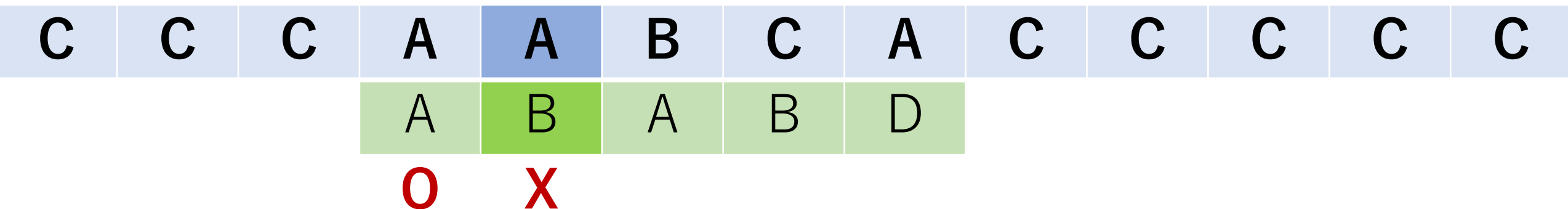
KMP法の事前準備

もし2文字目で照合失敗なら、



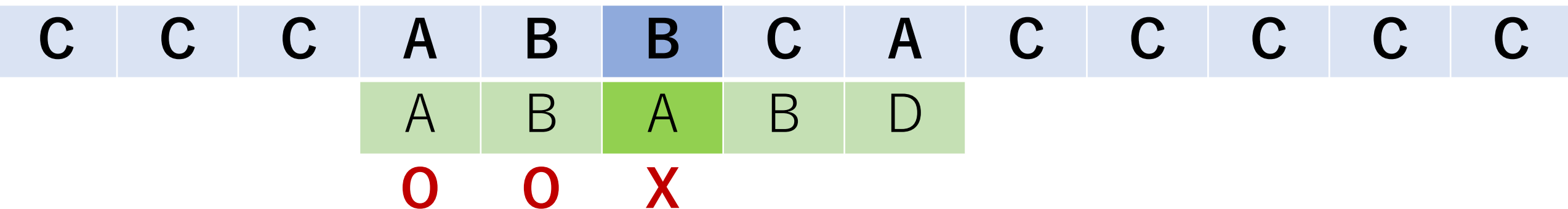
KMP法の事前準備

もし2文字目で照合失敗なら，次の照合は1文字目のAから始めることができる．照合対象のカーソル位置はそのまま．



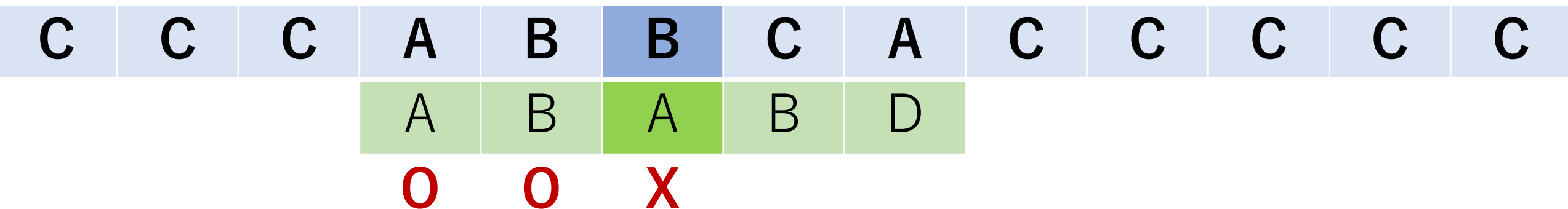
KMP法の事前準備

もし3文字目で照合失敗なら、



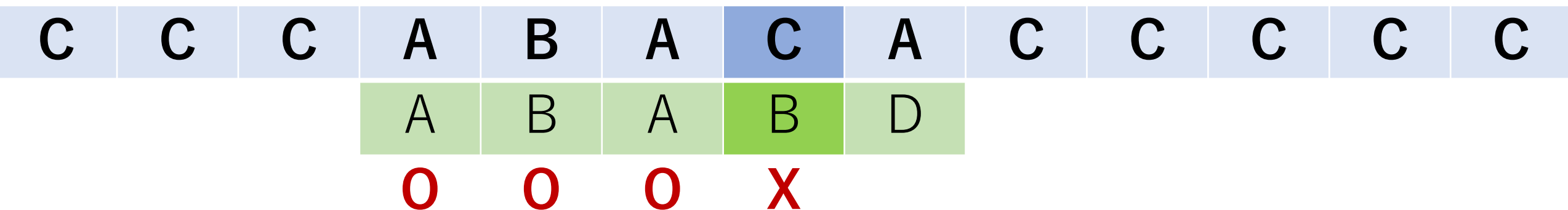
KMP法の事前準備

もし3文字目で照合失敗なら，次の照合は1文字目のAから始めることができる．照合対象のカーソル位置はそのまま．



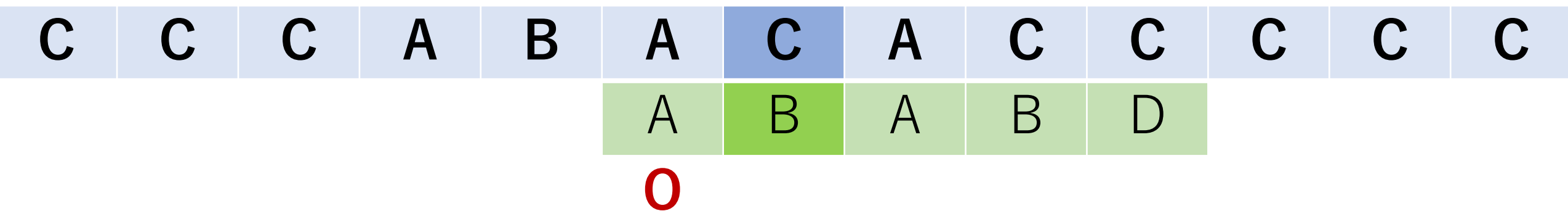
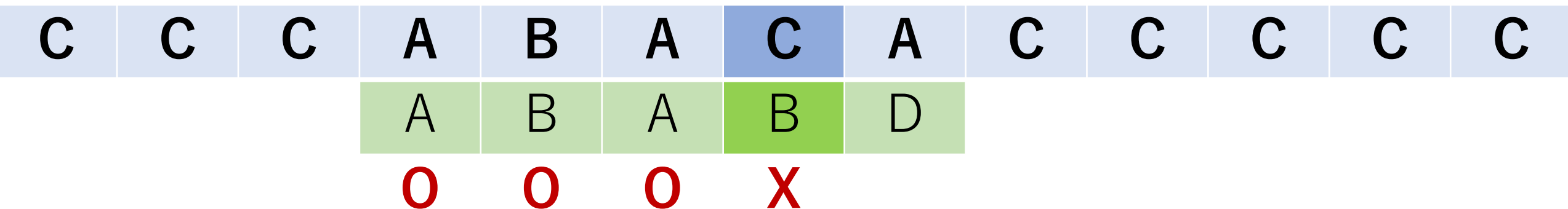
KMP法の事前準備

もし4文字目で照合失敗なら、



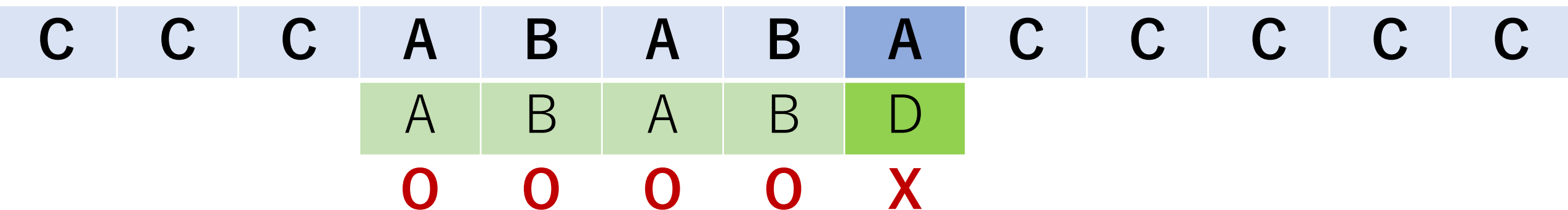
KMP法の事前準備

もし4文字目で照合失敗なら，次の照合は2文字目のBから始めることができる．照合対象のカーソル位置はそのまま．



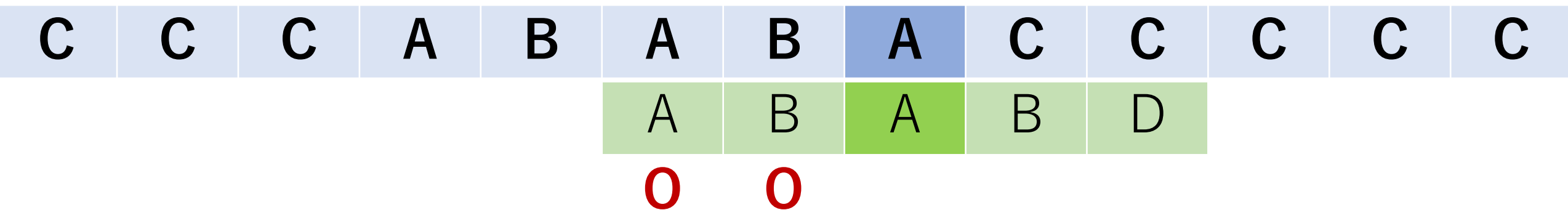
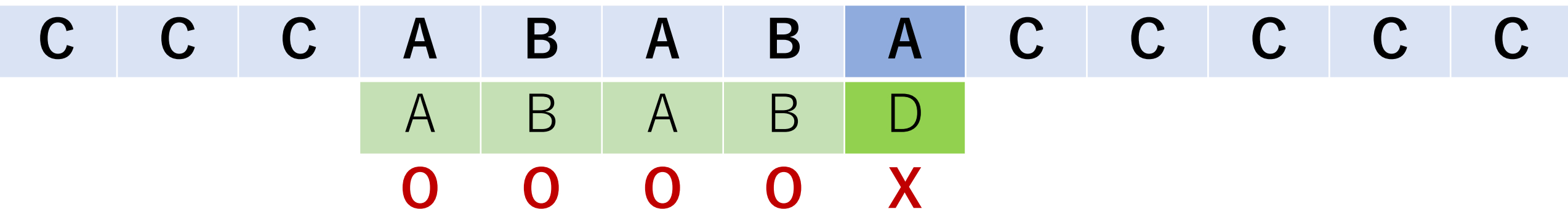
KMP法の事前準備

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から始めることができる．照合対象のカーソル位置はそのまま．



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*i*番目（`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。（ただし、以下の実装では、テーブルの0番目は使わない。）

skip	0	1	2	3	4	5
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[1]の値は0.

skip	0	1	2	3	4	5
要素の値	-	0				

KMP法の事前準備：スキップテーブル

2文字目（`pattern[1]`）で照合失敗のときは，`pattern[skip[1]]` → `pattern[0]`，つまり1文字目に戻って照合を開始する，ということの意味している。

skip	0	1	2	3	4	5
要素の値	-	0				

KMP法の事前準備：スキップテーブル

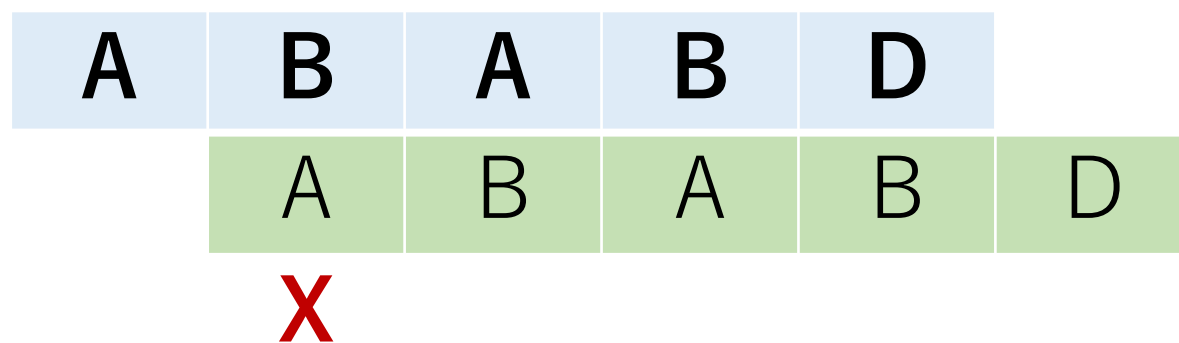
これ以降は，照合パターン同士を照合開始位置をずらしながら，照合パターンの内に部分一致するような場所がないかを確認する。

A	B	A	B	D	
	A	B	A	B	D

skip	0	1	2	3	4	5
要素の値	-	0				

KMP法の事前準備：スキップテーブル

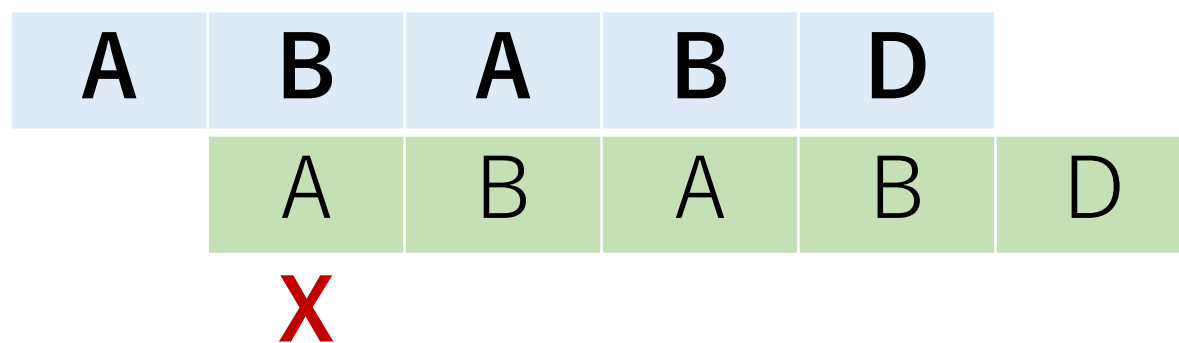
パターン文字同士の照合の最初は，今回の例では一致しないので，再開位置の値として0. これに伴い，skip[2]を0にする.



skip	0	1	2	3	4	5
要素の値	-	0				

KMP法の事前準備：スキップテーブル

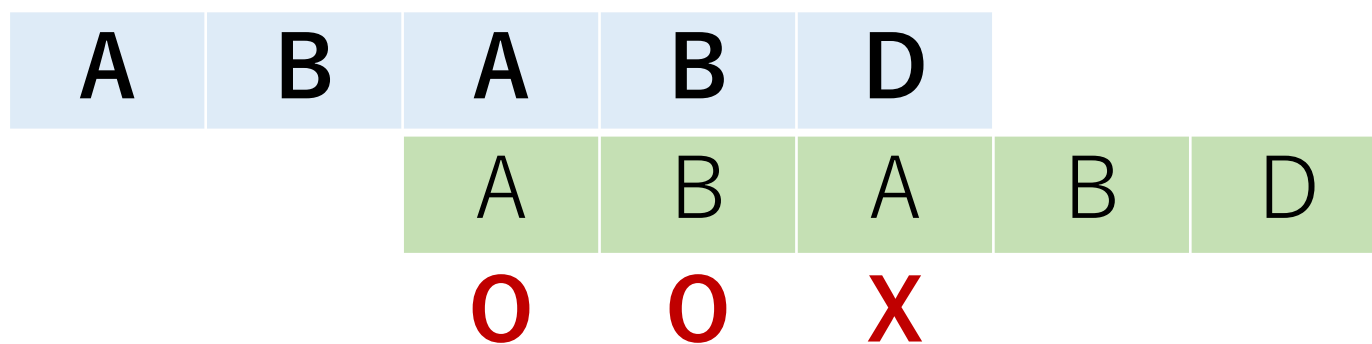
つまり，3文字目（`pattern[2]`）で照合失敗のときは，`skip[2]`の値に従い，パターンの1文字目（`pattern[0]`）に戻って照合を開始する。



skip	0	1	2	3	4	5
要素の値	-	0	0			

KMP法の事前準備：スキップテーブル

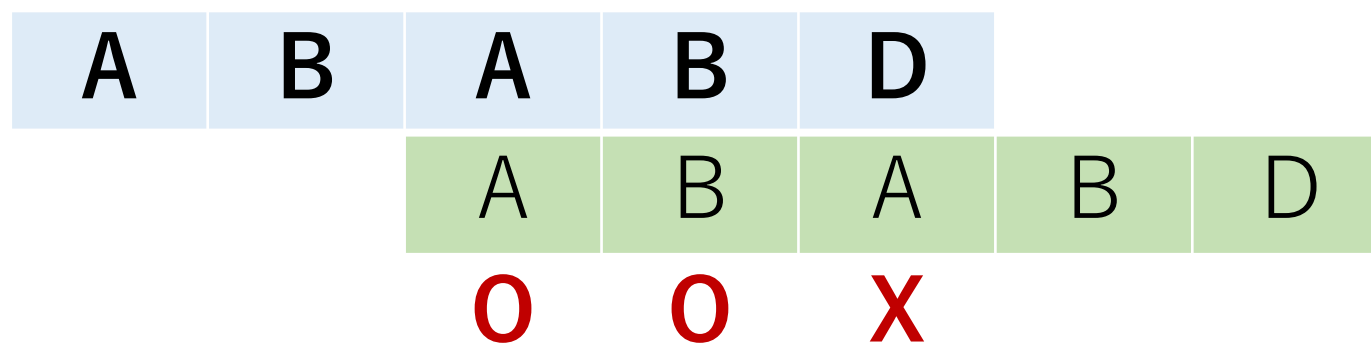
パターン文字同士の次の照合ではA, Bが照合する。



skip	0	1	2	3	4	5
要素の値	-	0	0			

KMP法の事前準備：スキップテーブル

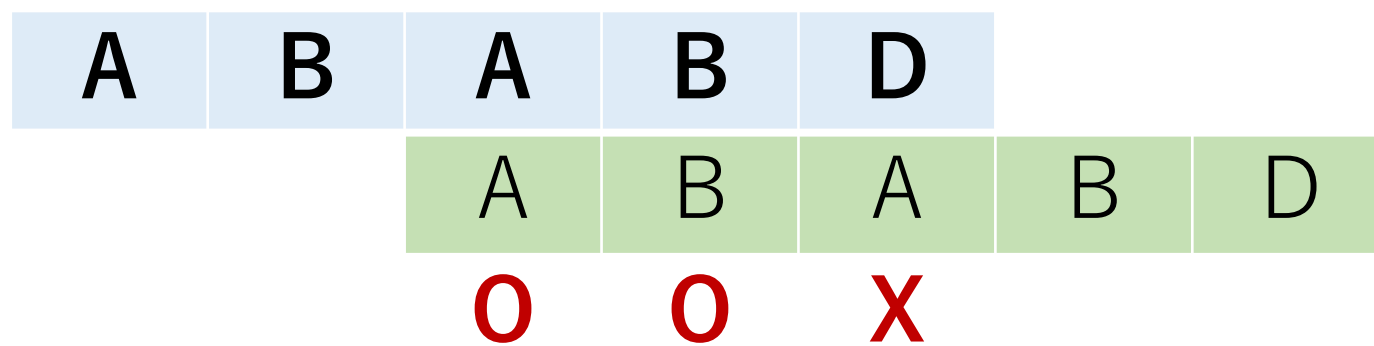
照合した個数をスキップテーブルに順に入れていく。
skip[3], skip[4]をそれぞれ1, 2にする。



skip	0	1	2	3	4	5
要素の値	-	0	0			

KMP法の事前準備：スキップテーブル

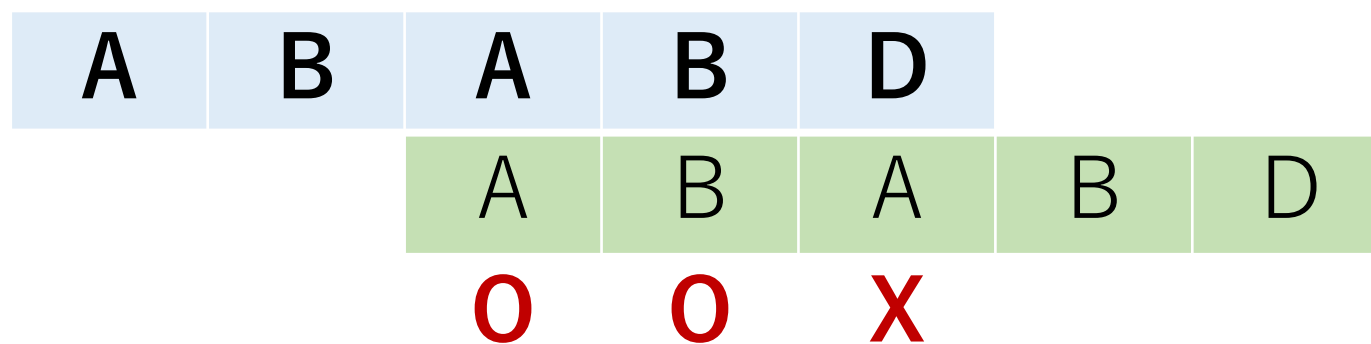
4文字目（pattern[3]）で照合失敗のときは，skip[3]の値に従い，パターンの2文字目（pattern[1]）に戻って照合を開始する。



skip	0	1	2	3	4	5
要素の値	-	0	0	1		

KMP法の事前準備：スキップテーブル

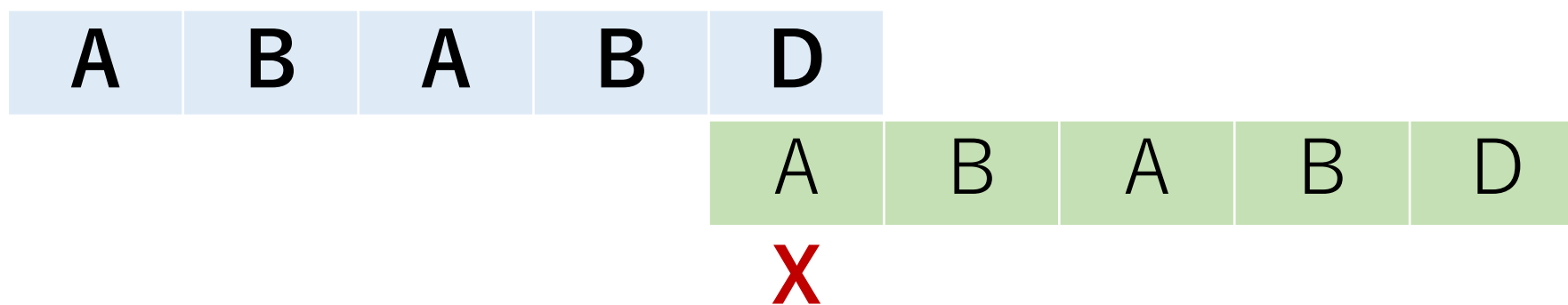
5文字目（pattern[4]）で照合失敗のときは，skip[4]の値に従い，パターンの3文字目（pattern[2]）に戻って照合を開始する。



skip	0	1	2	3	4	5
要素の値	-	0	0	1	2	

KMP法の事前準備：スキップテーブル

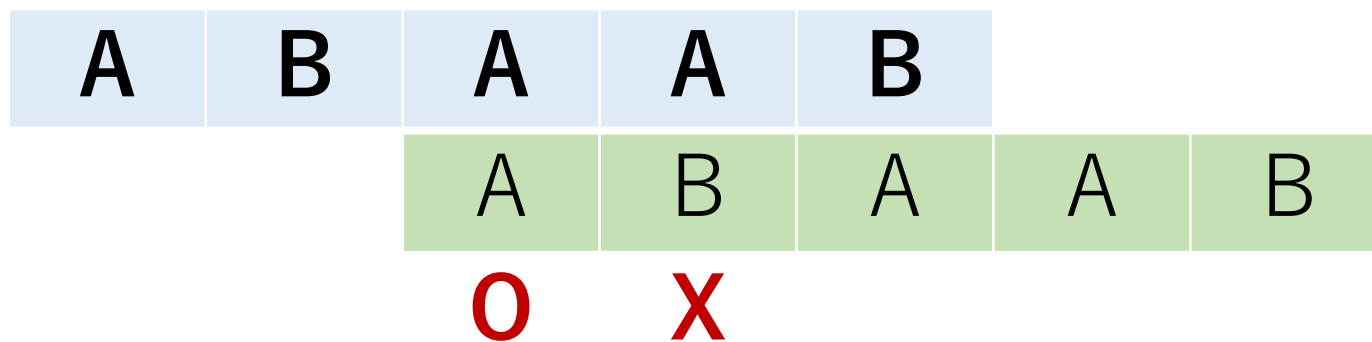
最後の文字まで同様の処理を繰り返す。（テーブルの最後の値はこの実装では使用しない。）



skip	0	1	2	3	4	5
要素の値	-	0	0	1	2	(0)

KMP法の事前準備：スキップテーブル

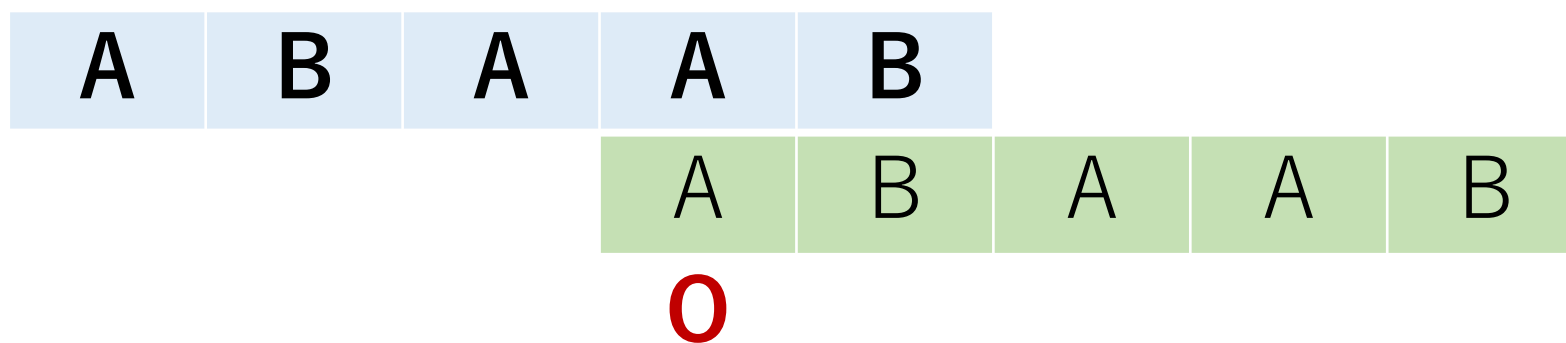
マッチしなくなったら，その場所を次に再開地点にし，
パターンの先頭からチェックする必要があることに注意。



skip	0	1	2	3	4	5
要素の値	-	0	0	1		

KMP法の事前準備：スキップテーブル

マッチしなくなったら，その場所を次に再開地点にし，
パターンの先頭からチェックする必要があることに注意。



skip	0	1	2	3	4	5
要素の値	-	0	0	1	1	(2)

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:  
        ...  
        # 照合パターン1文字目で失敗した場合は,  
        # テキスト側のカーソルを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:
```

```
        [スキップテーブルを使ってp_iを更新]
```


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

t_i は文字がマッチした時と1文字目から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。

ただし力任せ法と違って、**後戻りしない**，というところがしっかり実装されている所が良い。

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1～数文字目で失敗する。

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								

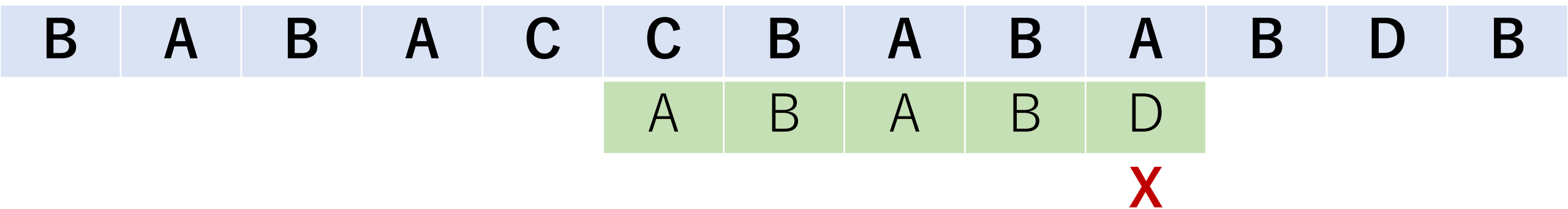
Cはパターンにない文字. なので, この5文字目のCが含まれるような照合開始位置は全て却下すべき.

BM法の例

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

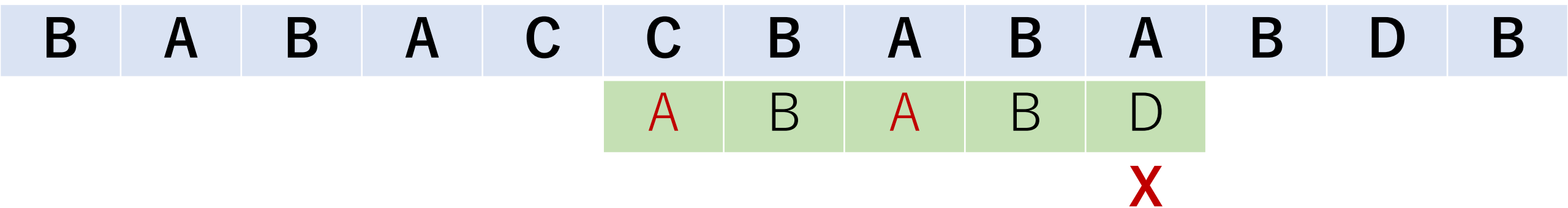
よって、一気にジャンプ！

BM法の例



DはAとは不一致.

BM法の例



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，先頭から順に見ていき，最後に出現する位置が i ($0 \leq i < l - 1$)ならば，移動量 $l - i - 1$ 。
 - ただし，最後に出現する位置が末尾になる場合は移動量 l 。

BM法の事前準備

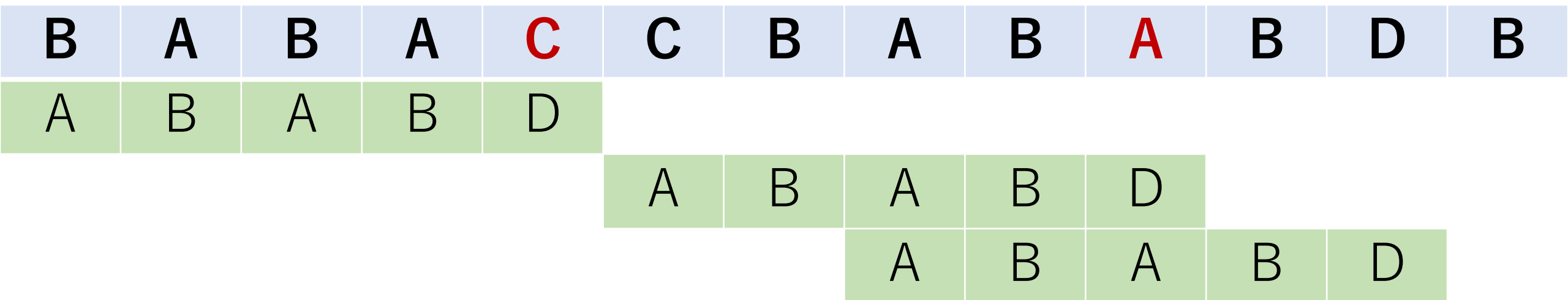
照合パターンがABABDの場合のスキップテーブル.

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



A	B	C	D	E
2	1	5	5	5

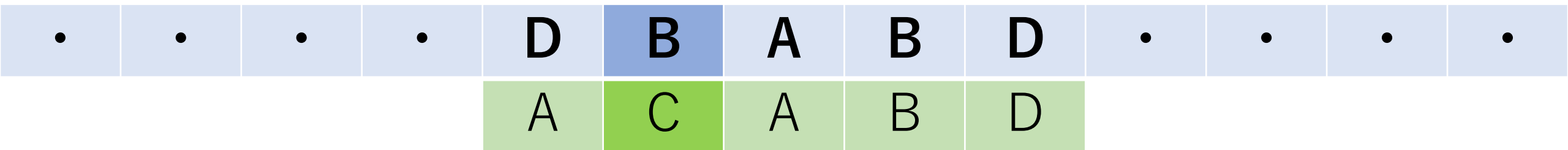
でも、スキップテーブルだけだと. . .

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

A	B	C	D	E
2	1	3	5	5

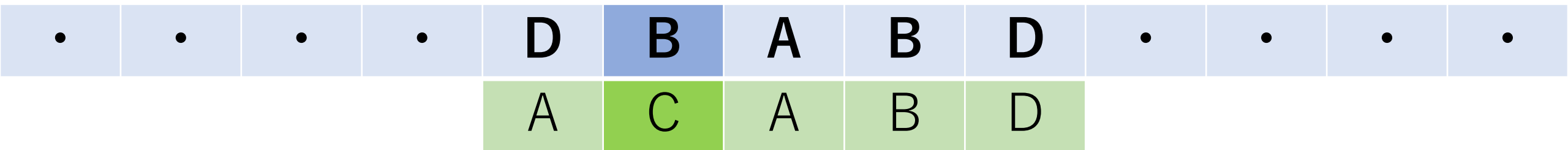
(照合パターンが変更され、スキップテーブルの値が変更されていることに注意)

パターンの途中でマッチしない場合



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



A	B	C	D	E
2	1	3	5	5

後戻りしている？



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的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だな...

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ため、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)

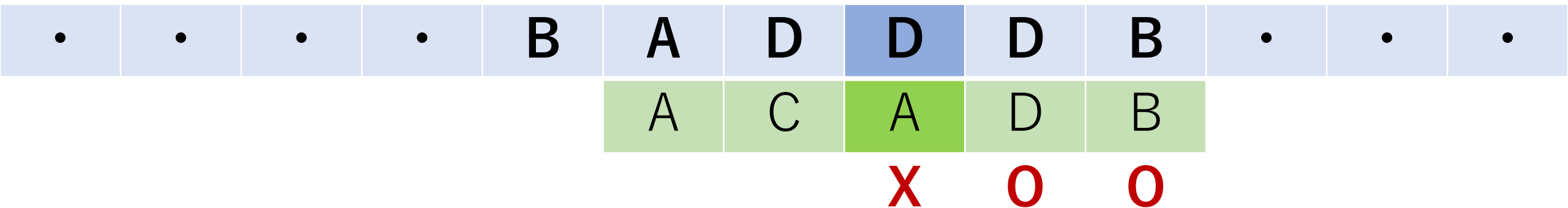
パターンの途中で不一致になるときの注意点

•	•	•	•	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



3つ目の文字で照合失敗。Dでマッチしなかったため、**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

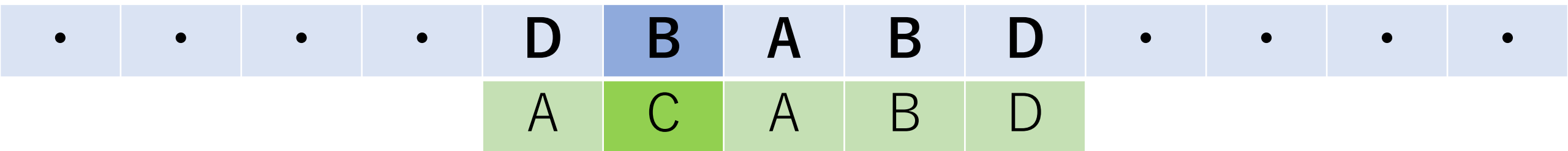
パターンの途中で不一致になるときの注意点

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

あれ？ 😬

A	B	C	D	E
2	5	3	1	5

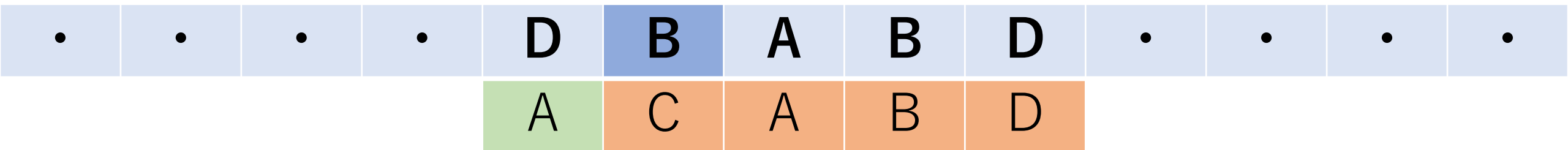
後戻りしないために



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。

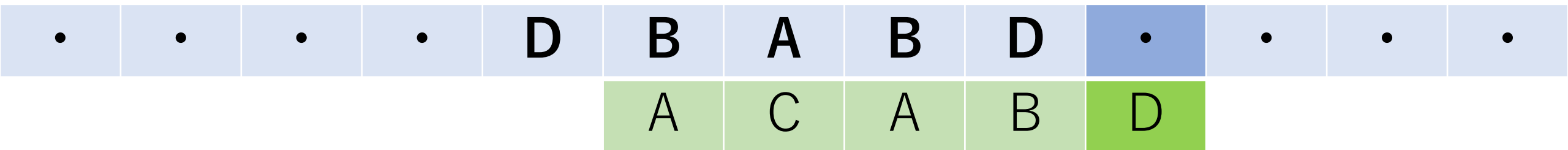
後戻りしないために



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



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

照合パターンが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

頻繁に似たような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？)

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。(なぜ？)

BMH法

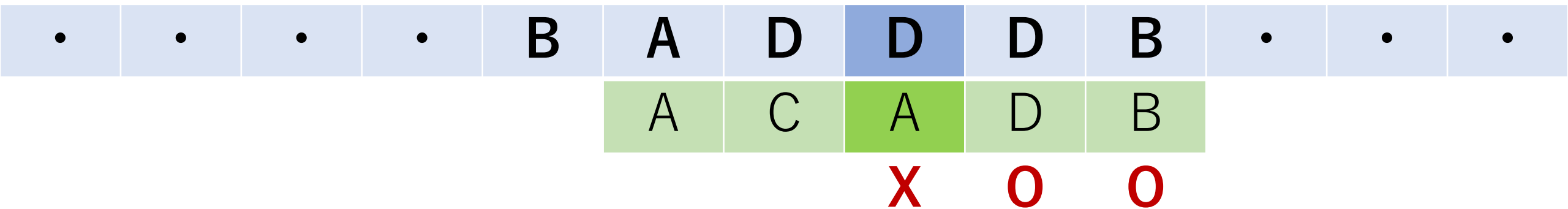
Boyer-Moore-Horspool.

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.

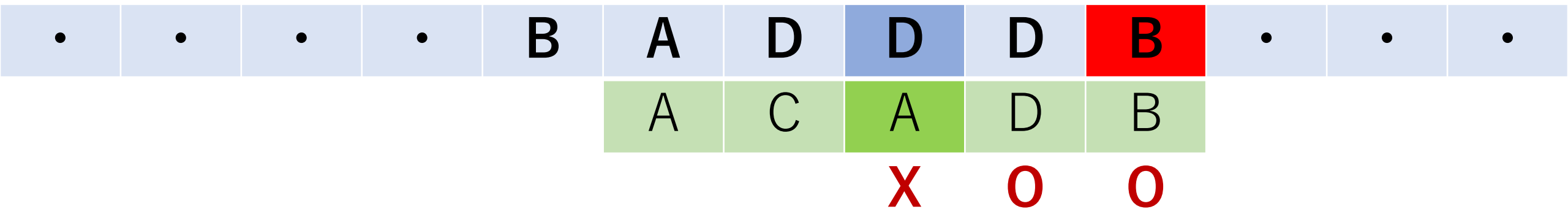
BMH方の考え方



3つ目の文字で照合失敗.

A	B	C	D	E
2	5	3	1	5

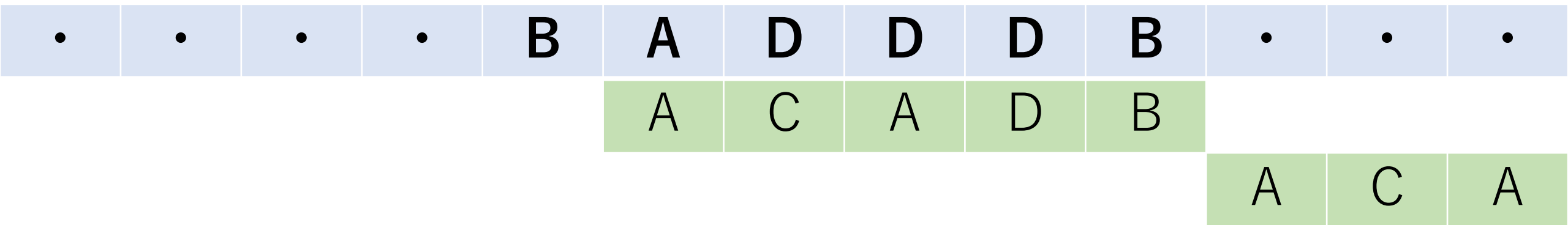
BMH方の考え方



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

BMH方の考え方

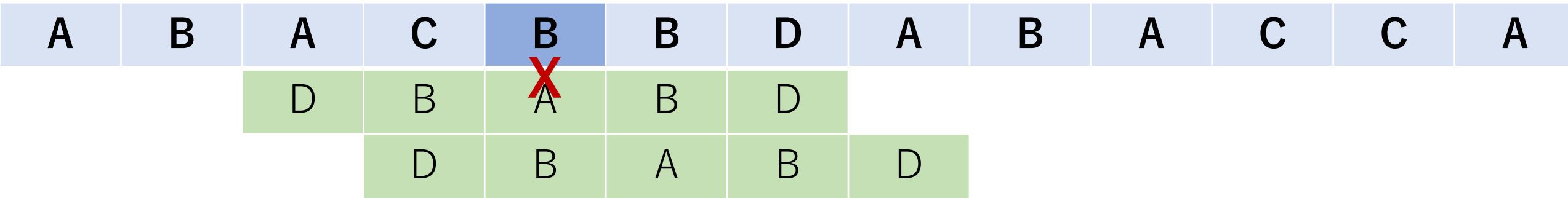


その文字ではなく，一番末尾の文字で移動量を決定.

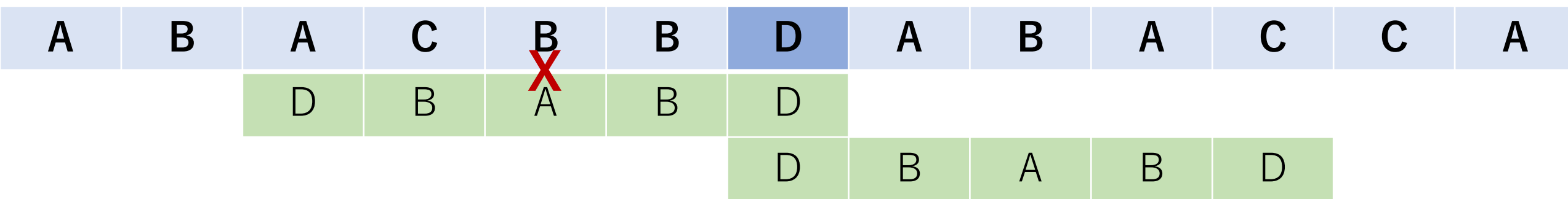
A	B	C	D	E
2	5	3	1	5

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。



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。



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
```

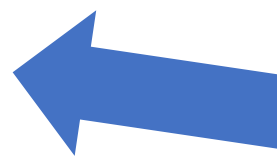
```
        while text[t_i - (p_len - 1 - p_i)] == pattern[p_i]:
```

```
            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_1, p_2, \dots, p_l に対してハッシュ値を計算する。

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

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、

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

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，

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

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．．．

□ーリングハッシュ

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

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

□ーリングハツシユ

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

□ — リングハツシユ

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

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

□ — リングハツシユ

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

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

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

□ — リングハツシユ

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

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

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

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

ローリングハッシュ

よって,

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

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 = 1007    # 基数  
    h = 10**9 + 7  # 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:  $(a^{i-1}p_1 + \dots + a^1p_{i-1} + a^0p_i) \bmod h$ 
```

```
        # i-1番目まで:  $(a^{i-2}p_1 + \dots + a^0p_{i-1}) \bmod h$ 
```

```
        [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和良好的.
```

```
        # ただし, 値がとて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。

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#5-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#5-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。

(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#5 [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.