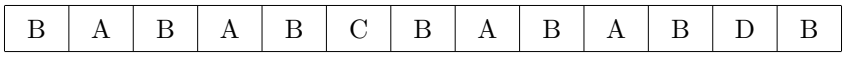
EEIC2024 アルゴリズムの授業で扱った文字列照合のアルゴリズムのシケプリです。扱った内容は以下の通りです。

- 力任せ法
- KMP 法
- BM 法
- BMH 法
- ラビン・カープ法 (ローリングハッシュ)

I. 力任せ法

力任せ法は文字通り文字列において、所望のパターンが見つかるまで前から順に比較していく方法です。以下で text と pattern の文字列が与えられたときに、text の中に pattern が含まれるかどうか考えましょう。文字列の文字を表す記法は text[i] のように 0-index で表します。

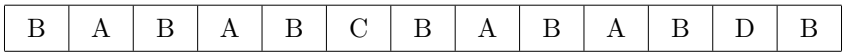
最初は text[0] と pattern[0] を比較すると一致しません。一致しないときは、text の cursor を 1 つ進めて text[1] と pattern[0] を比較します。



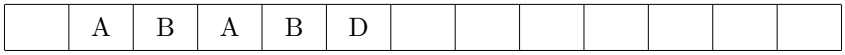
text

pattern

text[1] と pattern[0] を比較すると、一致します。しかし、text[5] と pattern[4] が一致しないので、text の cursor を 1 つ進めて text[2] と pattern[0] を比較します。

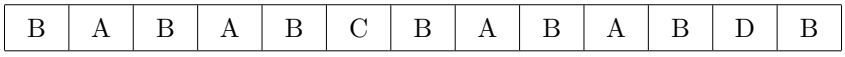


text



pattern

これを一致していくまで続けていくと、以下のようになります。



text

pattern

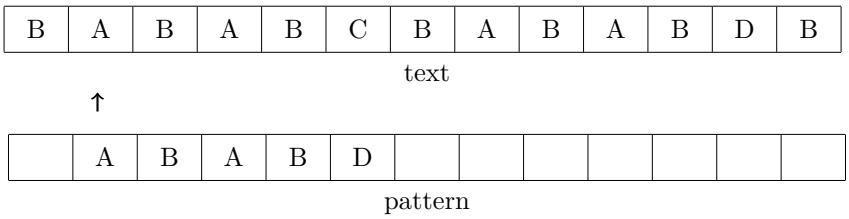
例を通じて力任せ法では、text を 0 から text の長さ - pattern の長さまで動かすことで、pattern が text に含まれるかどうかを判定することができます。C 言語でプログラムを書いてみましょう。

コード 1　力任せの実装

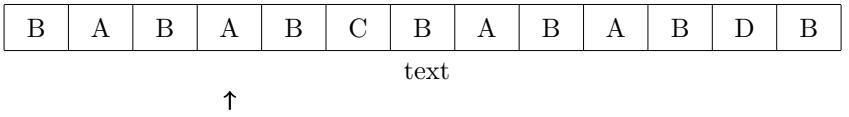
```
def brute_force(text: str, pattern: str) -> int:
    for text_cursor in range(len(text) - len(pattern) + 1):
        pattern_cursor = 0
        moving_text_cursor = text_cursor
        while pattern_cursor < len(pattern) and text[moving_text_cursor]
           == pattern[pattern_cursor]:
            pattern_cursor += 1
            moving_text_cursor += 1

        if pattern_cursor == len(pattern):
            return text_cursor

    return -1
```

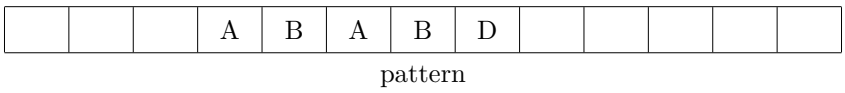
II.　KMP 法

i.　力任せ法の無駄



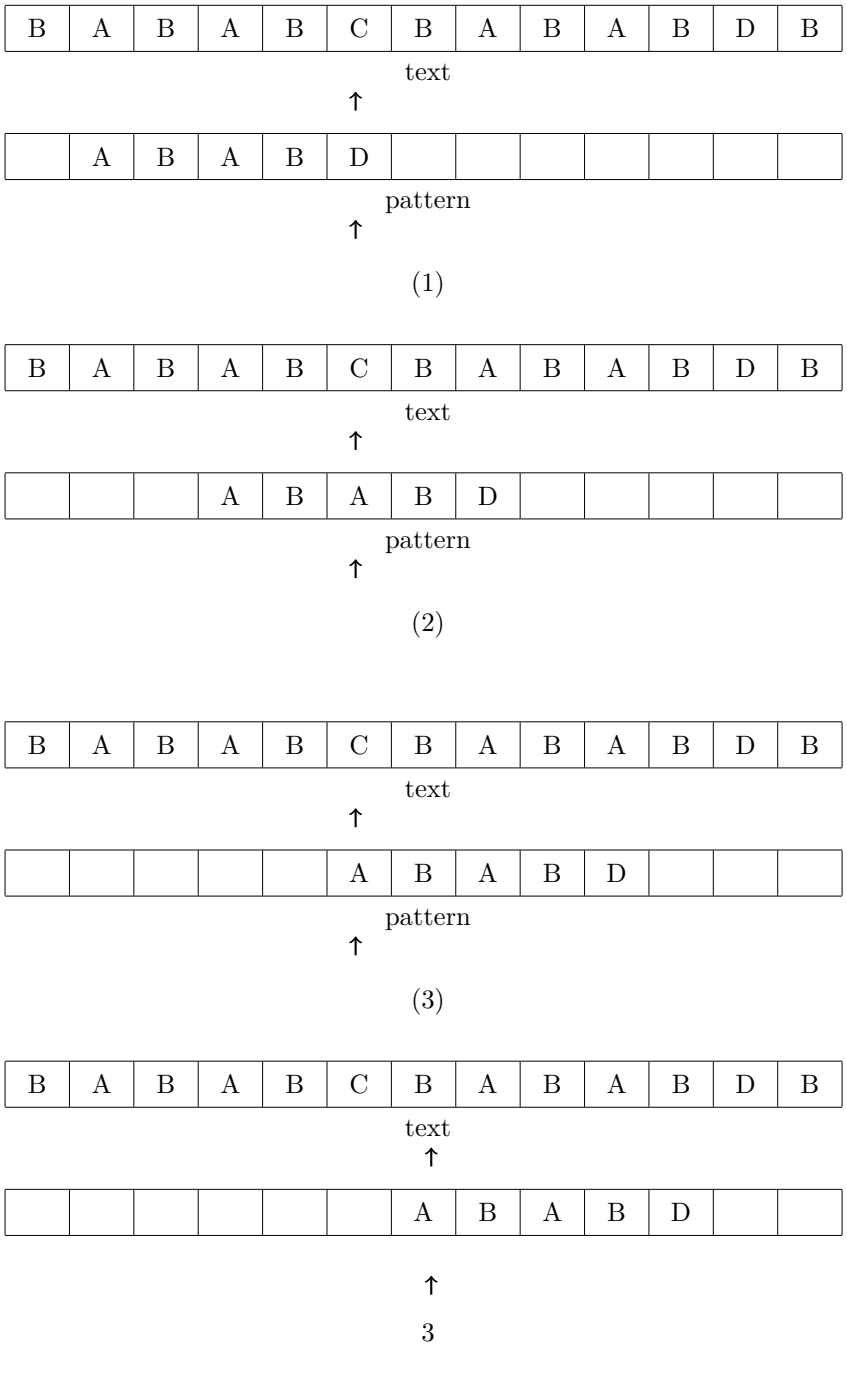
上の例では text[1] から照合を始めて text[5] で照合が失敗していることがわかります。力任せ法では次に text[2] から照合を始めることになりますが、それは text の方の cursor が一度通った場所を再度通ることになります。このように、力任せ法は text の cursor が一度通った場所を再び通ることが多いため、無駄が多いといえます。ただ、上の例では ABAB というパターンがあって、ABAB の部分が繰り返されていることがわかります。このような繰り返し部分がある場合、次に照合を始める位置をスキップすることで無駄が省けそうではないでしょうか？下の例では、text の cursor が後戻りすることなく、pattern の位置も前にスキップされています。





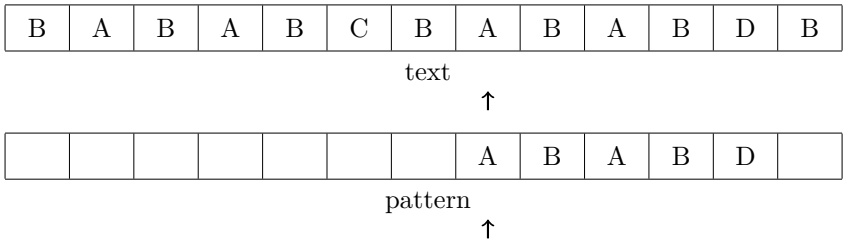
ii.　KMP 法の仕組み

上のスキップを実現する方法の 1 つに KMP 法があります。KMP 法の流れを具体例を見て確認します。



pattern

(4)



(5) 一致

(1) では pattern[4] で照合失敗しており、pattern[0] から pattern[3](ABAB) の前半と text[0] から text[3] の (ABAB) の後半 2 文字が一致しています。つまり、最初の 2 文字は比べなくても一致していることがわかります。そのため (2) では pattern の cursor を patten[2] にして照合を開始しています。

(2) では文字が一致しておらず、pattern をずらしても text[4], ,text[5] には一致しないので、(3) では pattern の cursor をはじめに戻しています。

(3) と (4) では文字が一致しておらず、さらに pattern もこれ以上右にシフトすることができないので、text と pattern ともに最初の位置から cursor を開始しています。

(5) では pattern が text に含まれていることがわかりました。

上の例で見た text と pattern で一致しているときに飛ばせる cursor の数は pattern によって事前に決まります。そのため、文字列照合の前に**スキップテーブル**作成を前処理として行います。

iii. スキップテーブルの作成

スキップテーブルは以下の条件を満たします。

- スキップテーブルの配列の長さは pattern の長さ - 1
- スキップテーブルの i 番目の要素は、pattern の i 番目までの部分文字列の最大の接頭辞と接尾辞の長さ
- スキップテーブルの最初の要素は 0

スキップテーブルを作成するには、pattern 同士をずらして比べていきます。その際に

- 文字がマッチする場合は、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に進める
- 文字がマッチしない場合は、マッチが失敗した位置からパターンの中で可能な部分一致の場所に移動する (KMP 法そのもの)

という手順で作成します。それでは例を見てみましょう。

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

pattern

pattern[1] と pattern[0] から照合を開始します。pattern[1] で照合が失敗しました。pattern[0] で失敗すると pattern を動かせないので skip[1] に 0 を入れて cursor を進めます。

0	0										
0	1	2	3	4	5	6	7	8	9	10	11

skip

次は pattern[2] と pattern[0] を照合します。今回は文字が一致しているため、$0+1$ をして skip[2] に 1 を入れて cursor を進めます。また、pattern[3] と pattern[1]、pattern[4] と pattern[2] も一致しているため、skip[3] と skip[4] にはそれぞれ 2 と 3 を入れます。

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

pattern

0	0	1	2	3							
0	1	2	3	4	5	6	7	8	9	10	11

skip

pattern[5] と pattern[3] は一致していません。マッチが失敗した位置からパターンの中で可能な部分一致の場所に移動するつまり、skip[3-1]　= skip[2] の 1 にします。

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

pattern

比較せずとも pattern[4] と pattern[0] は一致していることに注意をしてください。ただ。pattern[5] と pattern[1] で比較して失敗しているので、skip[1-1] を見て pattern を 0 から開始に移動します。

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

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

pattern

下の pattern を移動できるだけ移動しても一致していないので、skip[5] には 0 を入れて再び cursor を前に移動させます。下の図のように次に移動すると上の pattern[10] までは一致していることがわかるので、skip を同様に埋めてしまいます。

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

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

pattern

0	0	1	2	3	0	1	2	3	4	5	
0	1	2	3	4	5	6	7	8	9	10	11

skip

pattern[11] と pattern[5] を比較すると一致していないので、skip[5-1] = skip[4] を見ると 3 になっているので下の pattern の照合位置を 4 からにします。

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

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

pattern

pattern[11](D) と pattern[3](A) はまたもや一致していないので、skip[3-1] = skip[2] を見ると 1 になっているので、下の pattern の照合位置を 1 からにします。

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

										B	A	B	A	B	C	B	A	B	A	B

pattern

またまた pattern[11] と pattern[1] は一致しておらず、もう pattern の移動する余裕はないので skip[10] には 0 を入れます。

0	0	1	2	3	0	1	2	3	4	5	0
0	1	2	3	4	5	6	7	8	9	10	11

skip

これでスキップテーブルは完成です。好きな文字列と上の例で扱った文字列を照合してみると理解が深まります。

iv.　KMP 法の実装

実装のポイントは以下の 2 つです。

- スキップテーブルの作成
- 前処理で用意したスキップテーブルを使った文字列照合

コード 2　KMP 法の実装

```
    def create_table(pattern: str) -> list[int]:
    skip = [0] * (len(pattern) - 1)
    j = 0

    for i in range(1, len(pattern) - 1):
        if pattern[i] == pattern[j]:
            j += 1
            skip[i] = j
        else:
            while j > 0 and pattern[i] != pattern[j]:
                j = skip[j-1]

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

            skip[i] = j

    return skip

def kmp(text: str, pattern: str) -> int:
  skip = create_table(pattern)

  j = 0

  for i in range(len(text)):
```

```
26      # pattern照合の位置
27      if text[i] == pattern[j]:
28        j += 1
29      else:
30        while j > 0 and text[i] != pattern[j]:
31          j = skip[j-1]
32
33      if j == len(pattern):
34        return i - len(pattern) + 1
35
36    return -1
37
38  text = input()
39  pattern = input()
40
41  index = kmp(text, pattern)
42
43  print(index)
```

III.　BM 法

i.　BM 法の仕組み

BM 法は照合パターンの前からではなくて、後ろから照合を開始します。照合パターンの一番最後つまり D から照合を開始します。C と D は一致していないので、ずらします。

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

BM 法では以下のように最初から多くスキップすることが多いです。

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

また A と D は一致していないのでスキップします。今回は A という文字で照合が失敗していますが、照合パターンにも A があるので先ほどのように一気に飛ばせません。

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

次も同様に考えると以下のようになり、一致します。

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

上の例では、照合パターンの最後の文字から照合を開始しました。その際に text で一致しない文字に応じて text の照合 cursor の位置を動かしていました。では、その動かす大きさはどのように決まるでしょうか？ KMP 法と同様にスキップテーブルを作成します。スキップテーブルを以下のポイントを意識して作成します。照合パターンの長さを l とします。

- パターンに含まれない文字とパターンの一番最後にしかない文字の移動量は l
- 一番最後以外に含まれる文字に関しては、末尾に最も近い出現位置が $i(0 \leq i < l)$ ならば、そのときの移動量は $l - i + 1$
- 移動量は text に出現する文字をすべて網羅する辞書で管理

text 照合中にどの文字で失敗したかに応じて移動量が変化するので、text に登場する文字が文字が予め分かっている必要があります。text に登場する文字に応じて US-ASCII や UTF-8 などの文字コードを使って辞書を作成します。今回は US-ASCII で実装します。

ii.　パターンテーブルだけの実装の問題点

実はパターンテーブルだけでは問題が起こることがあります。プログラムの停止性に関わる問題です。例えば以下のような text と pattern を考えます。

				B	A	D	D	D	B					
				A	C	A	D	B						

マッチテーブルは以下のようです。

A	B	C	D	E
2	5	3	1	5

まず、D で失敗しているのでスキップテーブルの D を見ると 1 になっています。text の照合開始位置を一つ進めます。

				B	A	D	D	D	B					
					A	C	A	D	B					

今度も D で失敗しているので、text の照合開始位置を 1 進めましょう。今回は text の途中までは照合が成功しているので、+1 するのは照合が失敗した index であることに注意してください。

				B	A	D	D	D	B					
				A	C	A	D	B						

これは最初に見たパターンと同じになっているため、無限ループが発生しています。問題の原因は、スキップテーブルだけだと text の照合開始 index が後戻りすることがあるためです。スキップテーブルを用いることで、照合開始地点よりも前から照合してもパターンと一致することがないことが保証されているので、そもそも後戻りする意味はないです。つまり、不一致の場合に次の照合開始地点は前の照合開始地点よりも後ろになって欲しいです。

				D	B	A	B	D						
				A	C	A	B	D						

上の例では照合失敗を検知するまで 4 回比較しており、最後に照合した 1 点の index + 4 の地点から照合を開始しても問題ありません。よってスキップする大きさを決めるには、

$$skip\_size = \max(\text{比較回数、スキップテーブルの値})$$

とすれば良いです。

iii.　BM 法の実装

実装のポイントは以下の通りです。

- スキップテーブルの実装
- スキップテーブルを利用して文字列照合

コード 3　BM 法の実装

```
def bm(text: str, pattern: int) -> int:
  def create_table(pattern: str) -> list[int]:
    skip = [0] * (1 << 7)
    for i in range(1 << 7):
      last_match = -1
      for j in range(len(pattern)):
        if pattern[j] == chr(i):
          last_match = j

      if last_match == -1 or last_match == len(pattern) - 1:
        skip[i] = len(pattern)
      else:
        skip[i] = len(pattern) - last_match - 1
    return skip

  skip = create_table(pattern)

  text_cursor = len(pattern) - 1

  while text_cursor < len(text):
    moving_text_cursor = text_cursor
    pattern_cursor = len(pattern) - 1

    while text[moving_text_cursor] == pattern[pattern_cursor] and
       pattern_cursor > 0:
      moving_text_cursor -= 1
      pattern_cursor -= 1

```

```
28      if pattern_cursor == 0:
29        return text_cursor - len(pattern) + 1
30
31      # スキップテーブルを見てtextの参照開始位置を更新
32      compared_times = len(pattern) - pattern_cursor
33      skip_table_value = skip[ord(text[moving_text_cursor])]
34
35      text_cursor += max(skip_table_value, compared_times)
36
37    return -1
38
39
40  text = input()
41  pattern = input()
42
43  print(bm(text, pattern))
```

IV.　ラビン・カープ法 (ローリングハッシュ)

照合パターンとテキストのハッシュを計算し、そのハッシュが一致するか否かで文字列の一致を判定する方法を紹介します。照合パターンの長さを l とすると、ハッシュを利用することで比較の計算量を $O(l)$ から $O(1)$ へと定数時間にすることが可能です。

ハッシュは以下のように計算します。互いに素な定数 $a, h(1 < a < h)$ を用意して、照合パターン $(p_0 \cdots p_{l-1})$ に対して、

$$H(P) = (a^{l-1}p_0 + a^{l-2}p_1 + \cdots + a^0 p_{l-1}) \mod h \tag{1}$$

テキストの部分文字列 (text[0] から text[l-1]) のハッシュも同様に計算して $H(S, 0, l-1)$ とすると、H(P) と H(S, 0, l - 1) が一致するとき、text[0] から text[l-1] と照合パターンが一致すると考える。もちろんハッシュが衝突する可能性もあるので実際に一致しているか確認する必要がありますが、今回はハッシュが一致していれば文字列も一致しているとします。

しかし、このハッシュの計算をテキストの大きさだけ計算すると、$O(nl)$ の計算量になってしまいます。ここで、テキストのハッシュは $\mod h$ を取ったものであるため、

$$a^{l-1}p_0 + a^{l-2}p_1 + \cdots + a^0 p_{l-1} = H(S, 0, l-1) + Ah(\text{A は整数}) \tag{2}$$

と表せます。

これを利用すると、$H(S, 1, l)$ は前の $H(S, 0, l-1)$ を用いて以下のように表せます。

$$\begin{aligned} H(S, 1, l) &= a^{l-1}s_1 + a^{l-2}s_2 + \cdots + a^0 s_l \\ &= a(a^{l-2}s_1 + a^{l-3}s_2 + \cdots + a^1 s_{l-1}) + s_l \\ &= a(a^{l-1}s_0 + a^{l-2}s_1 + \cdots + a^0 s_{l-1}) - a^l s_0 + a^0 s_l \\ &= aH(S, 0, l-1) - a^l s_0 + a^0 s_l \end{aligned} \tag{3}$$

よって、テキストのそれぞれのハッシュは前のハッシュを利用することで、定数時間で計算できることがわかりました。これを**ローリングハッシュ**といいます。

i.　ラビン・カープ法の実装

実装のポイントは以下通りです。

- a, h を適当に決める
- 文字を数値に変換する
- テキストと照合パターンのハッシュを保持して尺取法のように計算する

コード 4　ラビン・カープ法の実装

```python
def rolling_hash(text: str, pattern: str) -> list[int]:
    a = 31
    h = 998244353

    text_length, pattern_length = len(text), len(pattern)
    text_hash = pattern_hash = 0

    # a^lを先に計算　剰余を取らないと実行時間が非常に長くなる
    a_l = 1
    for i in range(pattern_length):
        a_l = (a_l * a) % h


    # 最初のハッシュを計算する. 式(1)より
    for i in range(pattern_length):
        text_hash = (a * text_hash + ord(text[i])) % h
        pattern_hash = (a * pattern_hash + ord(pattern[i])) % h

    for i in range(text_length - pattern_length + 1):
        if pattern_hash == text_hash:
            return i

        if i < text_length - pattern_length:
            # テキストのハッシュを更新
            text_hash = (a * text_hash - a_l * ord(text[i]) + ord(text[i
                + pattern_length])) % h

            if text_hash < 0:
                text_hash += h

    return -1
```

V.　問題

問題 1 ABC 276 A - Right

英子文字からなる文字列が与えられます。最後に現れる a の index を求める問題です。存在しな

ければ、-1 を出力する問題です。力任せ法と Python の標準ライブラリを使って 2 つの解法を使って解いてみましょう。実際に AtCoder に参加するときなどは、標準ライブラリを使った解法を使うことが多いですが、勉強中は自分で実装することが大切です。

コード 5　問題 1 の解答

```python
def brute_force_last_index(text: str, pattern: str) -> int:
    last_index = -1
    for text_cursor in range(len(text) - len(pattern) + 1):
        pattern_cursor = 0
        moving_text_cursor = text_cursor
        while pattern_cursor < len(pattern) and text[moving_text_cursor]
           == pattern[pattern_cursor]:
            pattern_cursor += 1
            moving_text_cursor += 1

        if pattern_cursor == len(pattern):
            last_index = text_cursor


    return last_index

text = input()
pattern = "a"

last_index = brute_force_last_index(text, pattern)

print(last_index + 1 if last_index != -1 else last_index)

# 標準ライブラリを使った解法
text = input()
pattern = "a"

last_index = text.rfind(pattern)

print(last_index + 1 if last_index != -1 else last_index)
```

問題 1 ABC 276 A - Right