関数から計算へ

繰り返しによる反復計算の定義 --- 代入による変数の更新

```
irb(main):003:0> weight = 104.0
=> 104.0
irb(main):004:0> weight = weight-10
=> 94.0
irb(main):005:0> weight
=> 94.0
irb(main):006:0> weight = weight-10
=> 84.0
```

繰り返しによる和の定義

```
def sum_loop(n)
    s = 0
    for i in 1..n
        s = s+i
    end
    s
end
```

sum_loop.rb

条件を満たす値を探す繰り返し

```
load ("./divisible.rb")


def gd_loop(k,n)
    while !divisible(k,n)
        n = n-1
    end
    n
end
```

gd_loop.rb

再帰による反復計算の定義 --- 再帰による和の定義

```
def sum(n)
    if n >= 2
        sum(n-1) + n
    else
        1
    end
end
```

sum.rb

```
irb(main):004:0> sum(3)
=> 6
irb(main):005:0> sum(10)
=> 55
irb(main):006:0> 1+2+3+4+5+6
=> 55
```

約数の和

```
load ("./divisible.rb")

def sod(k,n)
   if n >= 2
      if divisible(k,n)
         sod(k,n-1) + n
      else
         sod(k,n-1)
      end
   else
      1
   end
end
```

sod.rb

```
irb(main):004:0> sod(10,9)
=> 8
irb(main):005:0> 5+2+1
=> 8
irb(main):006:0> sod(28,27)
=> 28
irb(main):007:0> 14+7+4+2+1
=> 28
```

条件を満たす値を探す

```
load ("./divisible.rb")


def gd(k,n)
    if divisible(k,n)
        n
    else
        gd(k,n-1)
    end
end
```

gd.rb

```
irb(main):008:0> gd(6,5)
=> 3
irb(main):009:0> gd(6,4)
=> 3
irb(main):010:0> gd(6,3)
=> 3
```

練習3.5a)と4.1-4.2

- x が y で割り切れることを判定する関数 divisible(x,y) を定義せよ。（divisible.rb）
- 素数とは、1 と自分自身しか約数がないような数である。上で定義した関数 sod を使って n が素数のときにのみ true, そうでないときに false となるような関数 prime(n) を定義せよ。（prime.rb）
- n 個から k 個を選ぶ組み合わせ数 ${}_nC_k$ を求める combination(n,k) を定義せよ。（combination.rb）

練習4.11c)

- 関数 tnpo(n) は n が偶数なら 1/2, 奇数なら 3 倍して 1 加えた数を求めるものだった。数学者 Collatz はどんな整数 n が与えられたときでも、この関数を使って数を変化させてゆくと、いずれ1 になると予想した。例えば 3 から始めた場合は 3⇒10⇒5⇒16⇒8⇒4⇒2⇒1 といった具合に予想通りになっていることが確められる。そこで n から上の手順で数を変化させて 1 になるまでの回数を collatz(n) とする。例えば collatz(5) = 5, collatz(16) = 4 である。
  A) collatz(n) と collatz(tnpo(n)) の関係を書け。
  B) collatz(n) を求める関数 collatz(n) を定義せよ。

Cantor Set

```
def cantor(n)
    a = make1d(3**n)
    subcantor(a, n, 0)
    a
end

def subcantor(a, n, x)
    if n==0
        a[x] = 1
    else
        subcantor(a, n-1, x)
        subcantor(a, n-1, x+2*3**(n-1))
    end
end
```

大きさ 3**n の配列を作成。
すべて 0 で初期化されている。

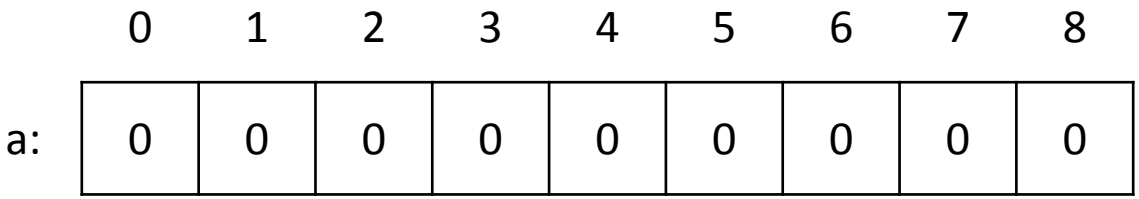
再帰的な手続き subcantor を
座標 0 に対して呼び出す。

配列 a を返す。

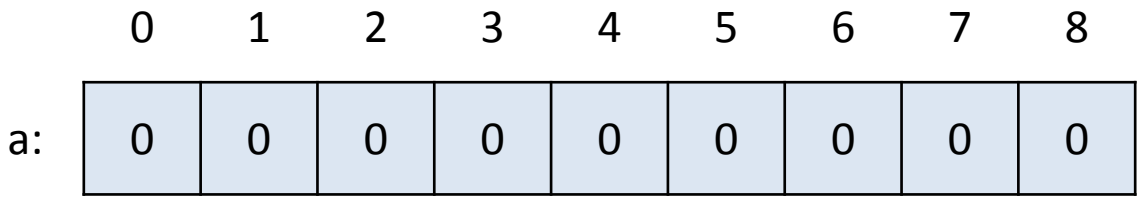
座標 x を起点として、
n 次の Cantor set を
配列 a に設定する。

n が 0 のときは、
単に 1 を設定。

再帰呼び出し。
起点に注意。

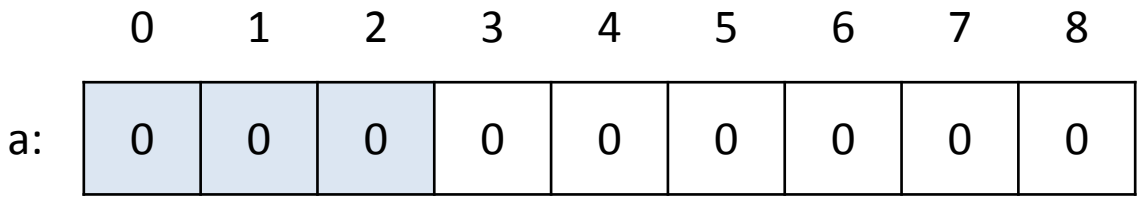


cantor(2)

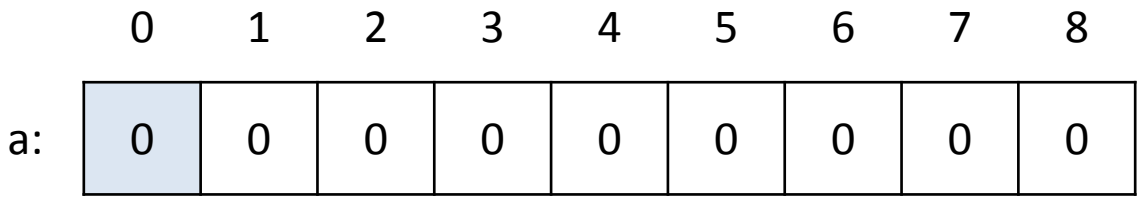


cantor(2)

subcantor(a,2,0)



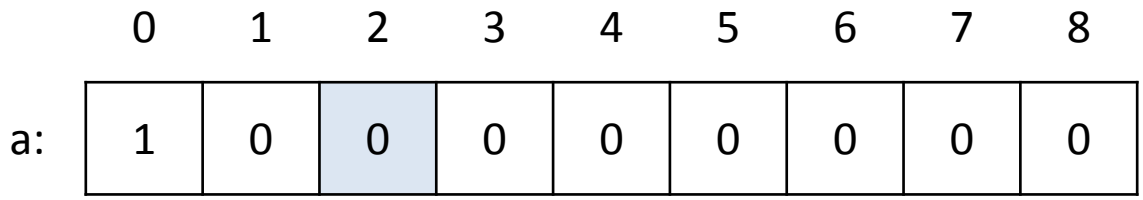

```
cantor(2)
    subcantor(a,2,0)
        subcantor(a,1,0)
```




```
cantor(2)
   subcantor(a,2,0)
      subcantor(a,1,0)
         subcantor(a,0,0)
```

	0	1	2	3	4	5	6	7	8
a:	1	0	0	0	0	0	0	0	0

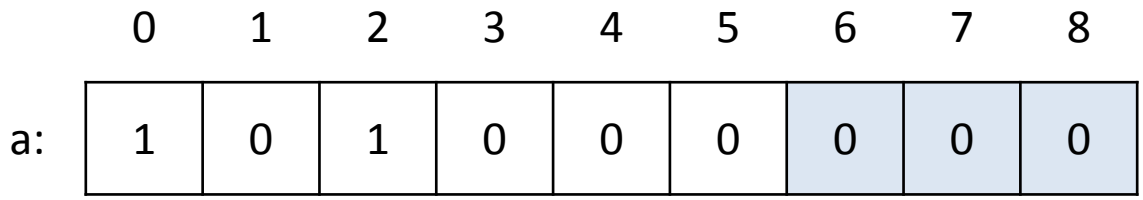
```
cantor(2)
    subcantor(a,2,0)
        subcantor(a,1,0)
            subcantor(a,0,0)
                a[0] = 1
```



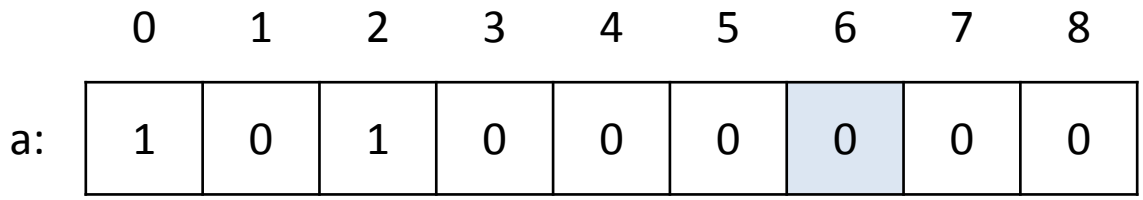

```
cantor(2)
    subcantor(a,2,0)
        subcantor(a,1,0)
            subcantor(a,0,0)
                a[0] = 1
            subcantor(a,0,0+2*3**0)
```

	0	1	2	3	4	5	6	7	8
a:	1	0	1	0	0	0	0	0	0

```
cantor(2)
    subcantor(a,2,0)
        subcantor(a,1,0)
            subcantor(a,0,0)
                a[0] = 1
            subcantor(a,0,0+2*3**0)
                a[2] = 1
```



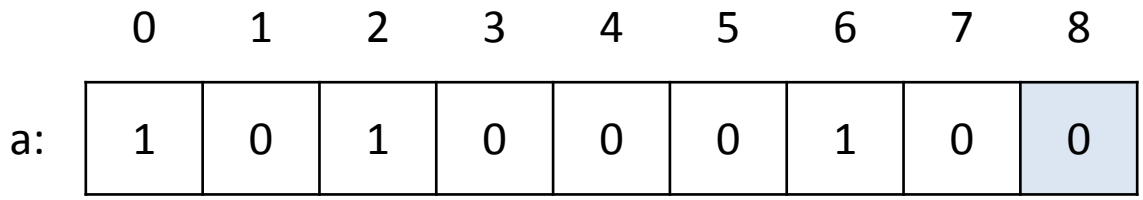

```
cantor(2)
    subcantor(a,2,0)
        subcantor(a,1,0)
            subcantor(a,0,0)
                a[0] = 1
            subcantor(a,0,0+2*3**0)
                a[2] = 1
        subcantor(a,1,0+2*3**1)
```



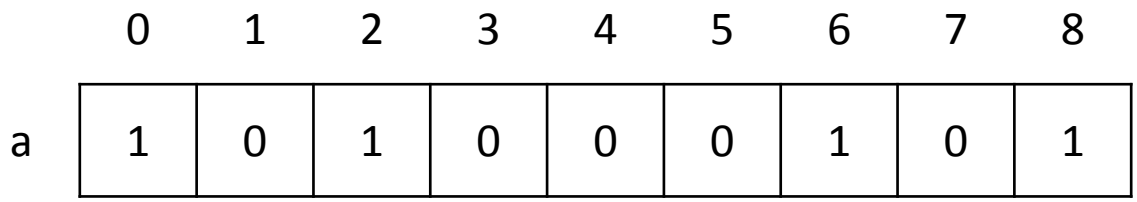

```
cantor(2)
   subcantor(a,2,0)
      subcantor(a,1,0)
         subcantor(a,0,0)
            a[0] = 1
         subcantor(a,0,0+2*3**0)
            a[2] = 1
      subcantor(a,1,0+2*3**1)
         subcantor(a,0,6)
```

	0	1	2	3	4	5	6	7	8
a:	1	0	1	0	0	0	1	0	0

```
cantor(2)
   subcantor(a,2,0)
      subcantor(a,1,0)
         subcantor(a,0,0)
            a[0] = 1
         subcantor(a,0,0+2*3**0)
            a[2] = 1
      subcantor(a,1,0+2*3**1)
         subcantor(a,0,6)
            a[6] = 1
```




```
cantor(2)
   subcantor(a,2,0)
      subcantor(a,1,0)
         subcantor(a,0,0)
            a[0] = 1
         subcantor(a,0,0+2*3**0)
            a[2] = 1
      subcantor(a,1,0+2*3**1)
         subcantor(a,0,6)
            a[6] = 1
         subcantor(a,0,6+2*3**0)
```




```
cantor(2)
   subcantor(a,2,0)
      subcantor(a,1,0)
         subcantor(a,0,0)
            a[0] = 1
         subcantor(a,0,0+2*3**0)
            a[2] = 1
      subcantor(a,1,0+2*3**1)
         subcantor(a,0,6)
            a[6] = 1
         subcantor(a,0,6+2*3**0)
            a[8] = 1
```

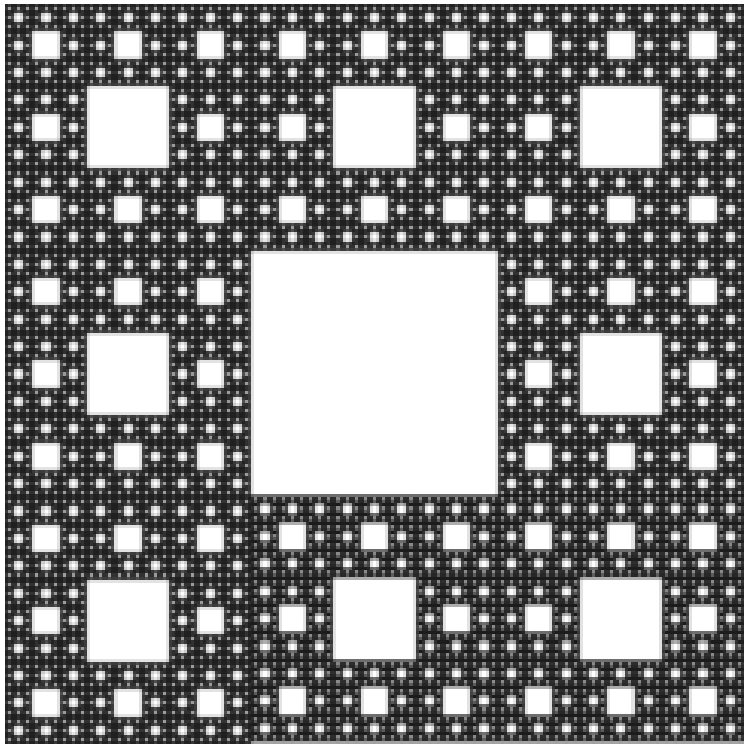
Cantor Set

- 区間 [0,1] の真ん中の 1/3 を除く。
- 残った二つの区間に対して、同じくことを行う。
- 同じことを無限に繰り返す。
- 残った点の集合が Cantor set。
- Cantor set の測度（面積）は 0。
- しかし、濃度（点の数）は、もとの区間 [0,1] と同じ。

練習

Cantor set の 2 次元版。
cf. Cantor dust

- Sierpinski のカーペット
  - n 次のカーペットは、縦横が 3**n で、n-1 次ののカーペットを 8 枚敷き詰めて作られる。真ん中は空いている。0 次のカーペットは、縦横 1 の黒い正方形とする。（実際のプログラムでは、白黒が反転する。）



carpet(n)

```
def carpet(n)
    a = make2d(3**n, 3**n)
    subcarpet(a, n, 0, 0)
    a
end

def subcarpet(a, n, x, y)
    if n==0
        a[y][x] = 1
    else
        ????
        ????
    end
end
```

進捗状況の確認

1. Sierpinski のカーペットができた（できた時点で投票してください）
2. まだ

Sierpinski のカーペットができた人は、練習3.5a)と4.1-4.2、練習4.11c)

配列・文字列と繰り返し --- 配列と繰り返しを使った組み合わせ数の計算

$n \setminus k$	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

表 4.1: 組み合わせ数 ${}_nC_k$ の値

```
load ("./make2d.rb")

def combination_loop(n,k)
    c = make2d(n+1,n+1)
    for i in 0..n
        c[i ][0] = 1
        for j in 1..(i-1)
            c[i][j] = c[i-1][j-1] + c[i-1][j]
        end
        c[i][i] = 1
    end
    c[n][k]
end
```

combination_loop.rb

n ＼ k	0	1	2	3	4	5	6
0	1						
1	1	1					
2							
3							
4							
5							
6							

n ＼ k	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1						
3							
4							
5							
6							

n ＼ k	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2					
3							
4							
5							
6							

n ＼ k	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3							
4							
5							
6							

n ＼ k	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1						
4							
5							
6							

n ＼ k	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3					
4							
5							
6							

n ＼ k	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3				
4							
5							
6							

n ＼ k	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4							
5							
6							

n ＼ k	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5							
6							

n ＼ k	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6							

n ＼ k	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

Pascal の三角形

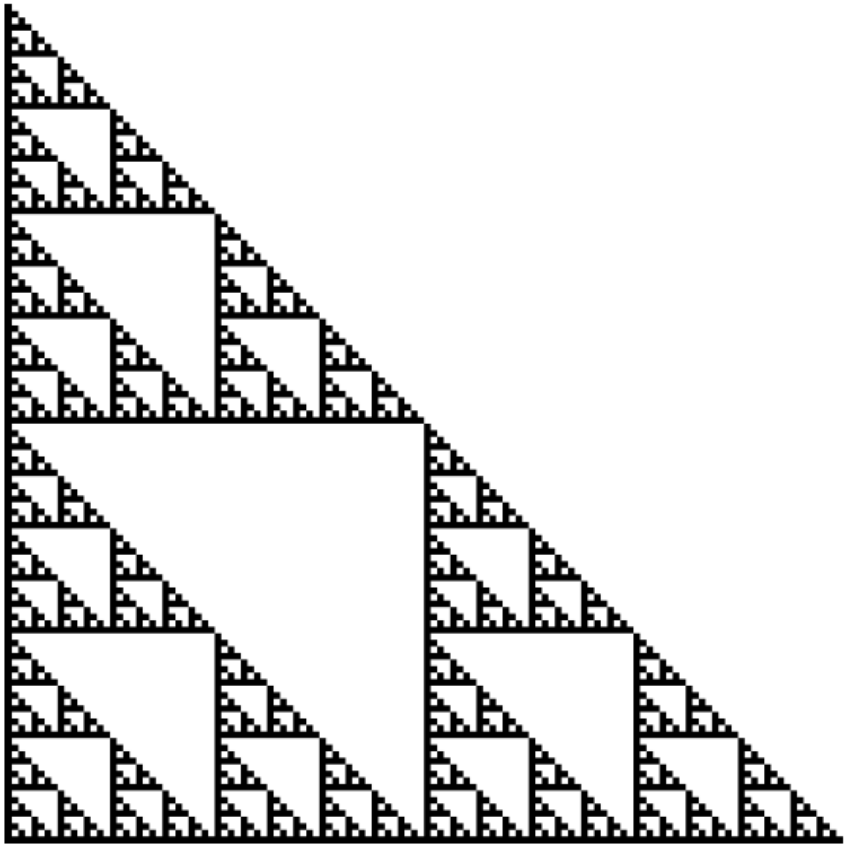
n ＼ k	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

Sierpinski の三角形

n ＼ k	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	0	1				
3	1	1	1	1			
4	1	0	0	0	1		
5	1	1	0	0	1	1	
6	1	0	1	0	1	0	1

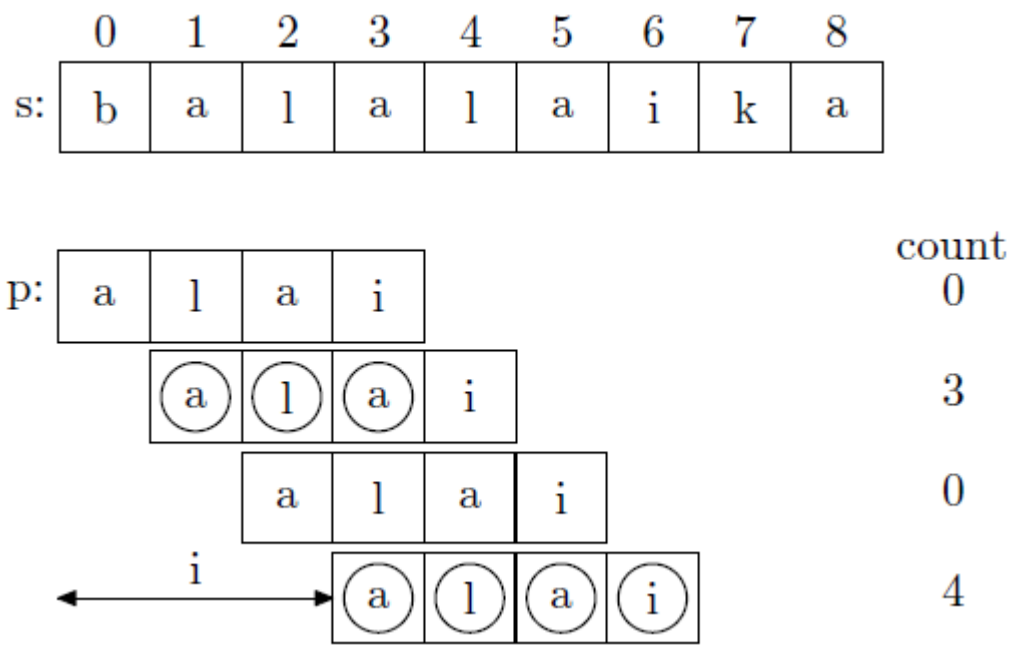
練習4.15

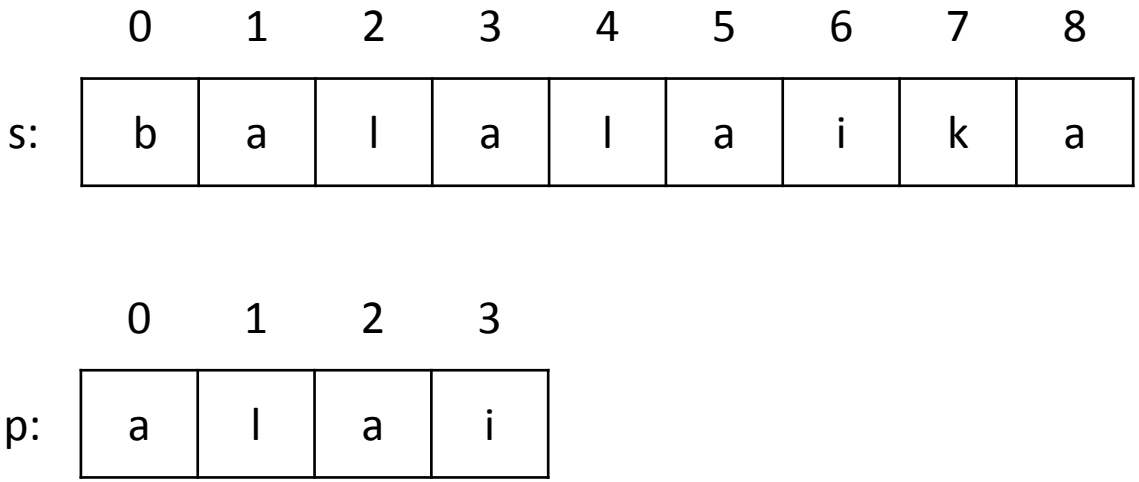
- Sierpinski の三角形
  - 大きさ n*n で、i 行 j 列目が ${}_iC_j$ を 2 で割った余りになっているような配列を作る関数 sierpinski(n) を定義せよ。（見易さのために白黒を逆にしている。）
  - sierpinski.rb に。



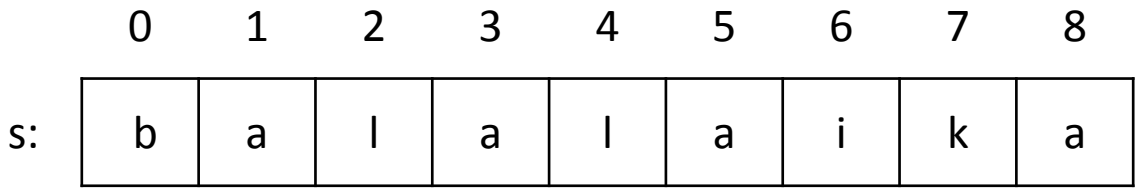
言葉探し

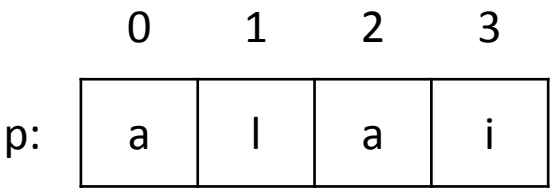
```
def match(s,p)
    i=0
    w=p.length()
    while submatch(s,i,p,w) < w
        i = i + 1
    end
    i
end
```



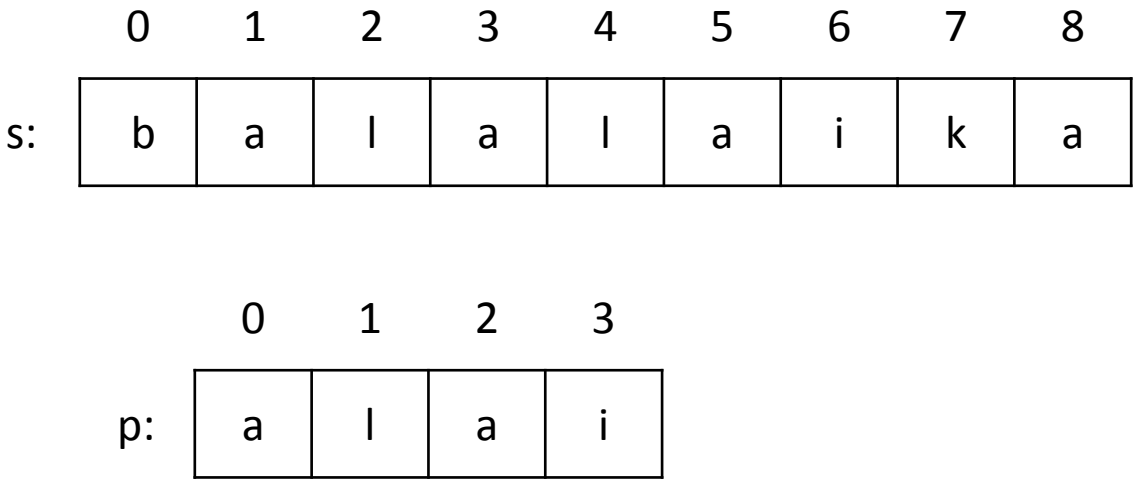


submatch(s, 0, p, 4)

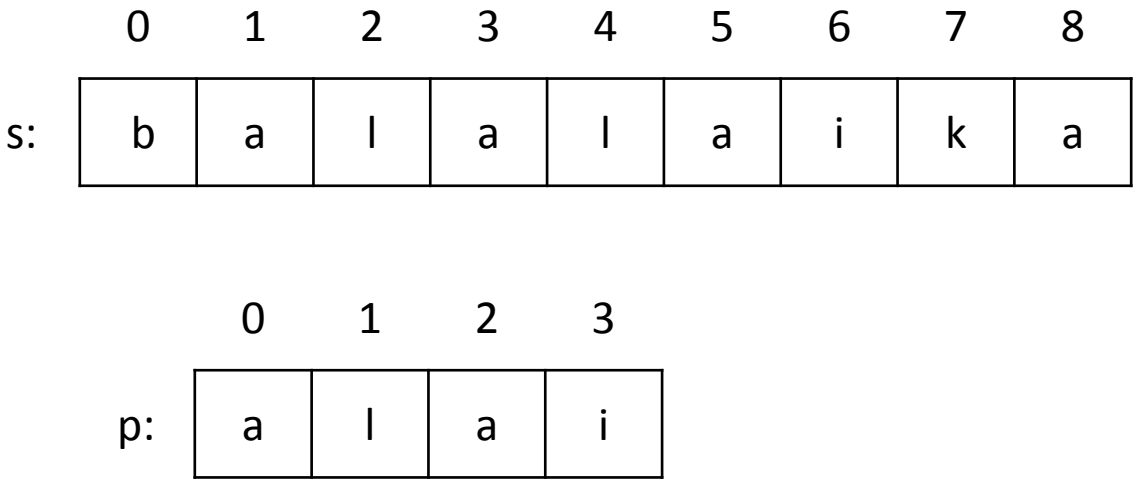




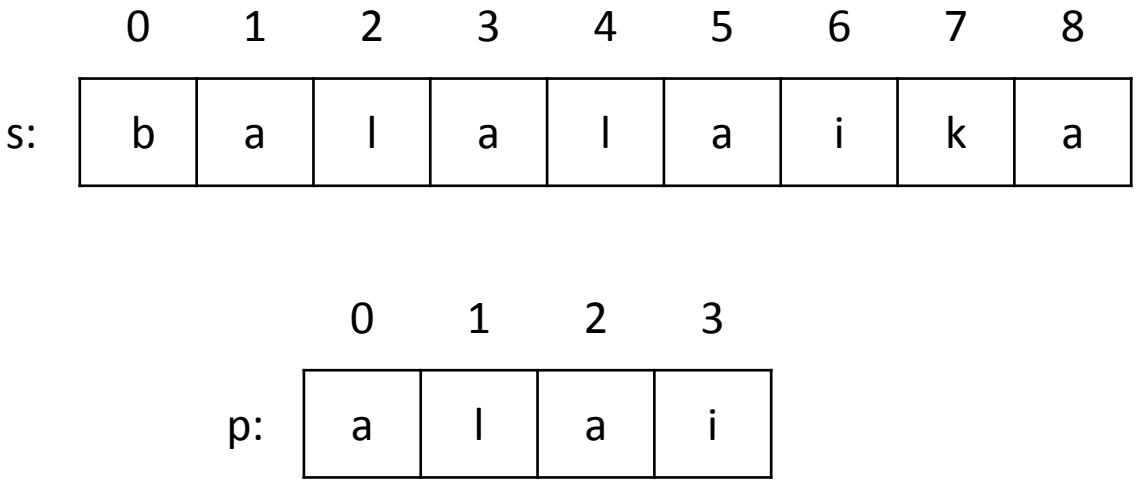
submatch(s, 0, p, 4) => 0



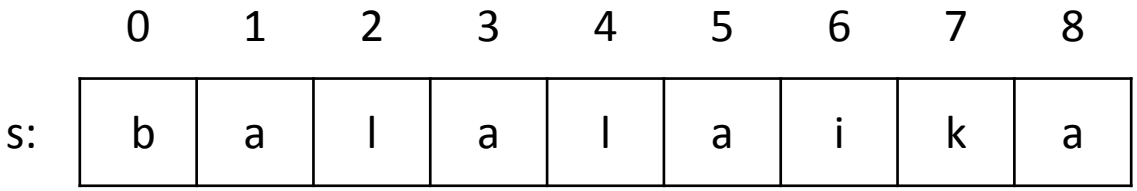
submatch(s, 1, p, 4)

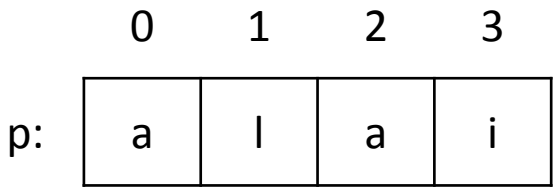


submatch(s, 1, p, 4) => 3

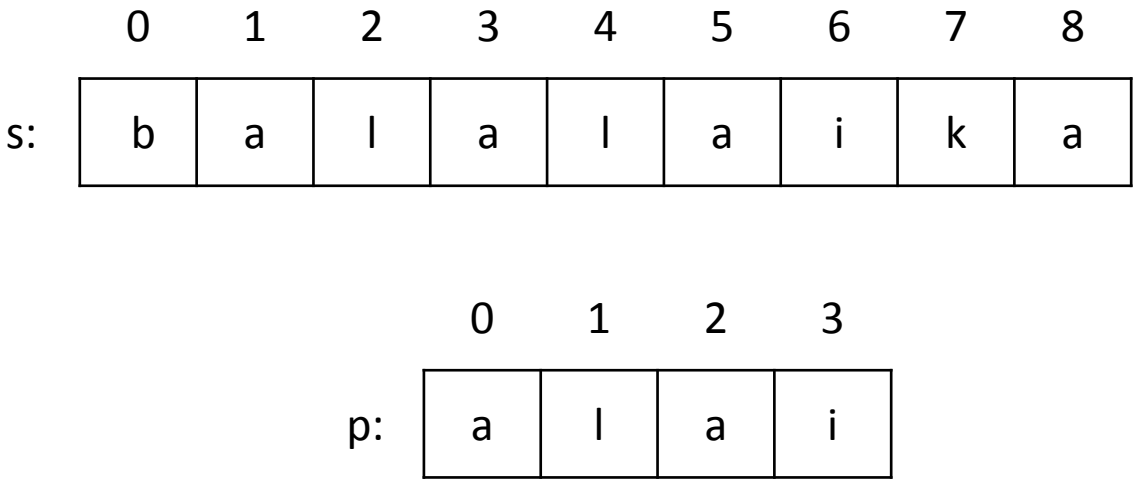


submatch(s, 2, p, 4)

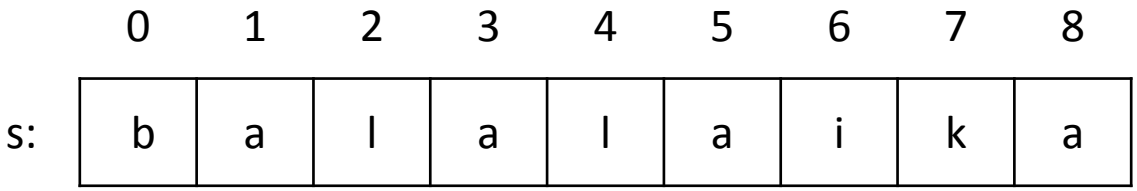


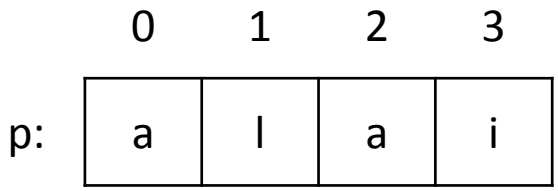


submatch(s, 2, p, 4) => 0

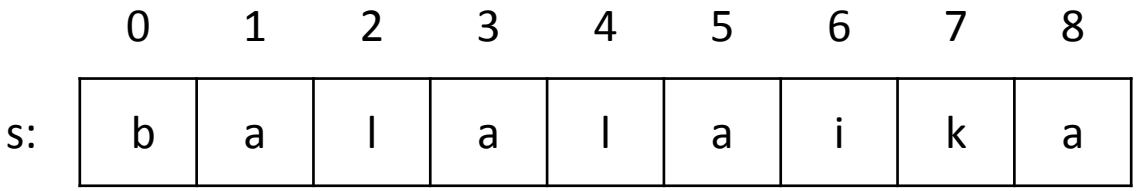


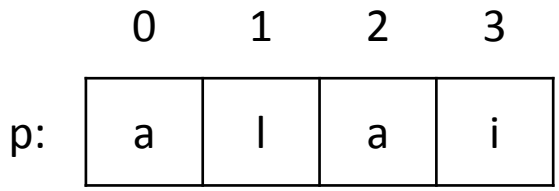
submatch(s, 3, p, 4)





submatch(s, 3, p, 4) => 4

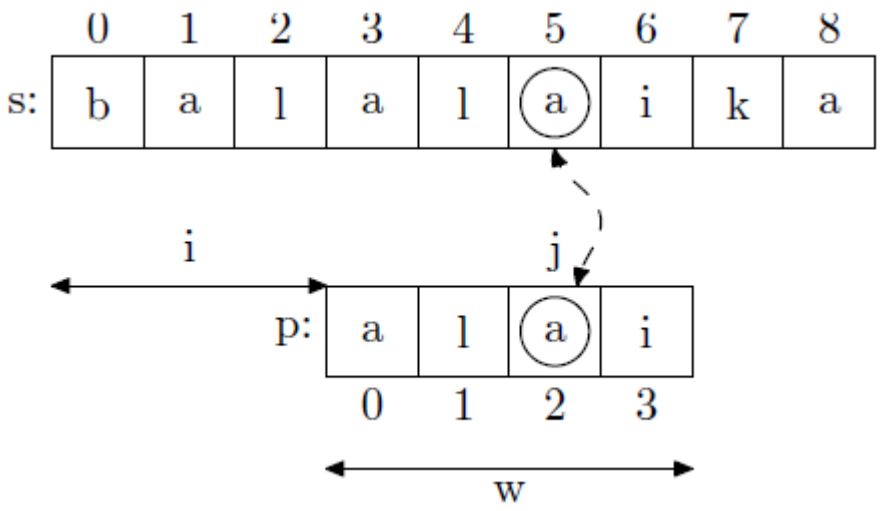


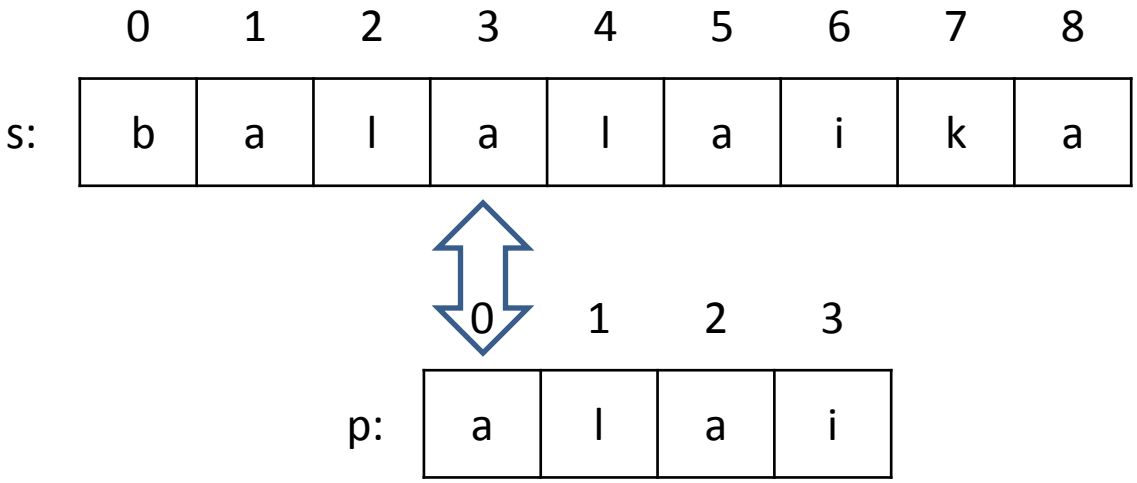


submatch(s, 3, p, 4) => 4

i==3 のときに、submatch(s, i, p, w) < w が false になる

```
def submatch (s,i,p,w)
    j = 0
    while j < w && s[(i+j)..(i+j)] == p[j..j]
        j = j + 1
    end
    j
end
```



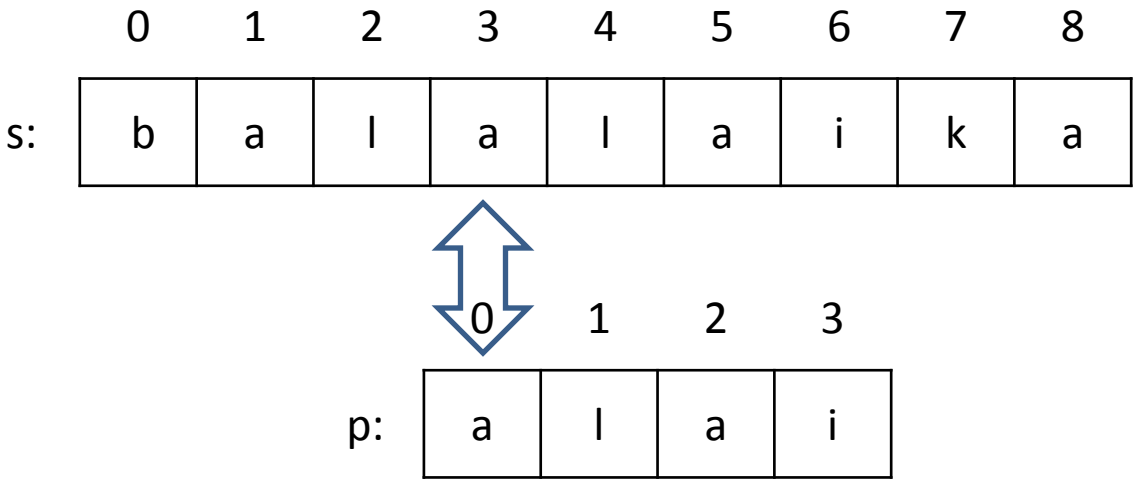


submatch(s, 3, p, 4)

i==3 w==4

j==0

j < w && s[(i+j)..(i+j)] == p[j..j]



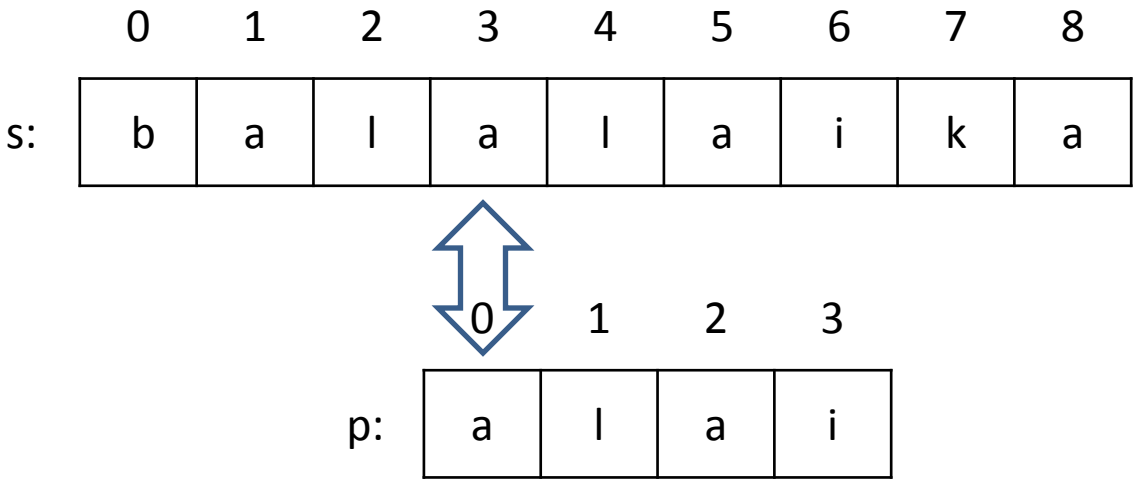
submatch(s, 3, p, 4)

i==3 w==4

j==0

j < w && s[(i+j)..(i+j)] == p[j..j]

=> "a" => "a"

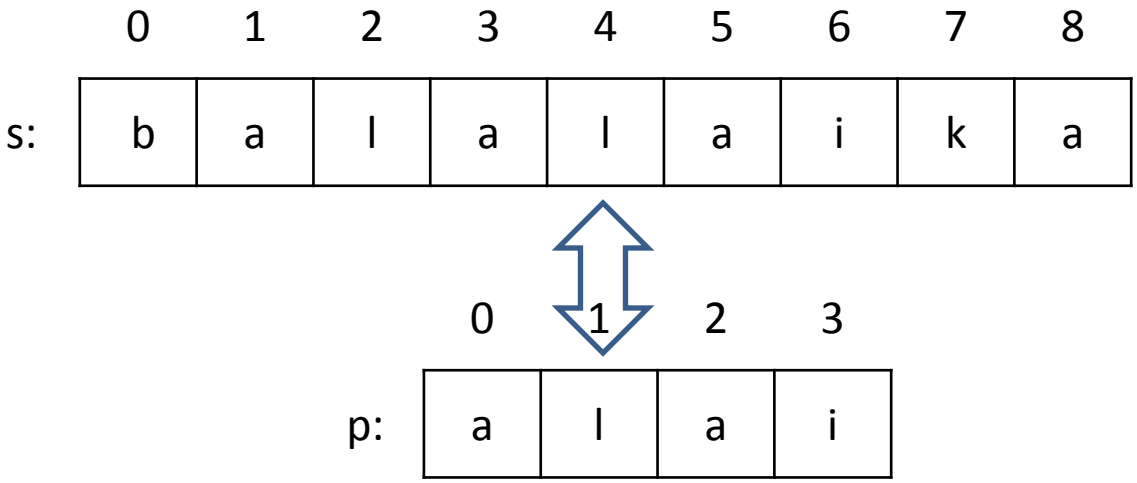


submatch(s, 3, p, 4)

i==3 w==4

j==0

j < w && s[(i+j)..(i+j)] == p[j..j] => true

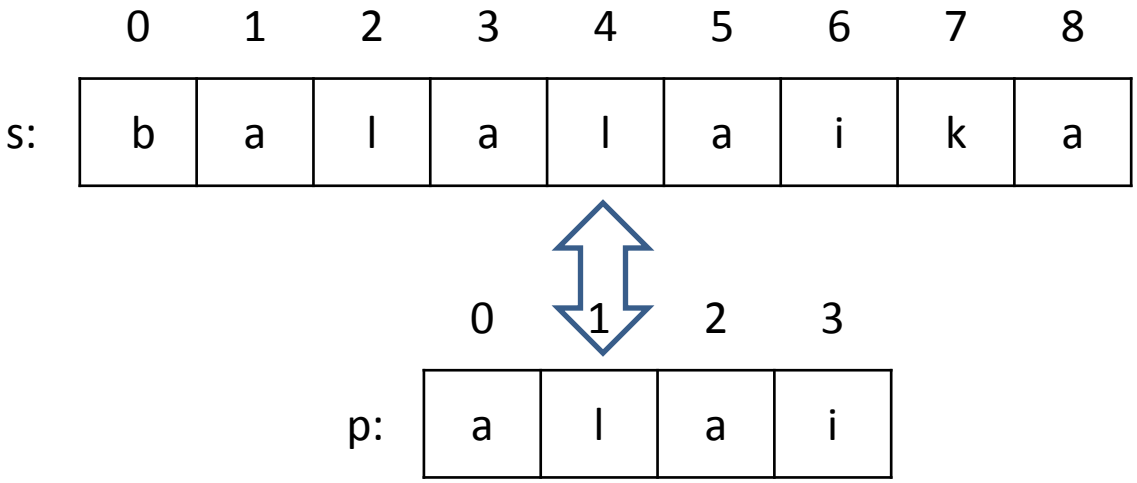


submatch(s, 3, p, 4)

i==3 w==4

j==1

j < w && s[(i+j)..(i+j)] == p[j..j]

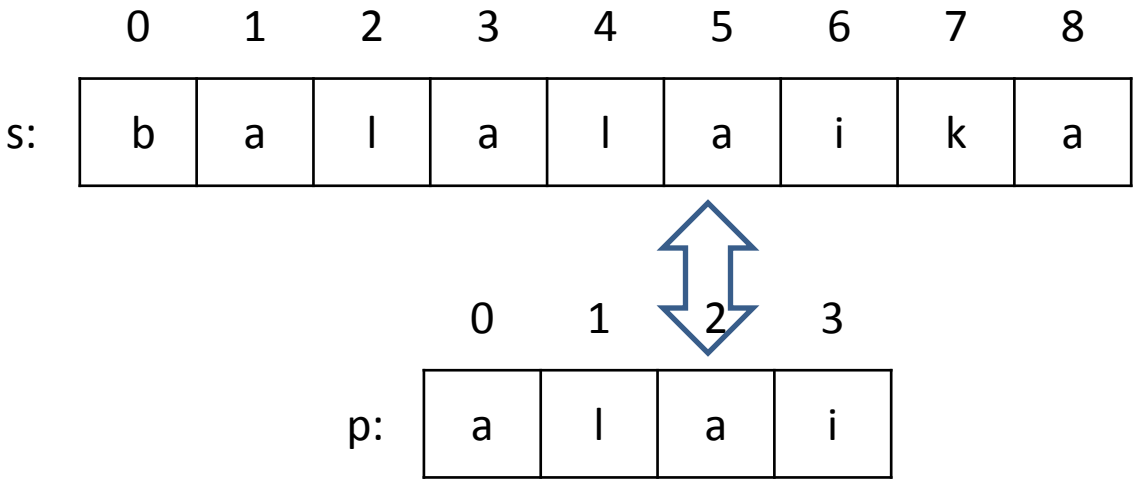


submatch(s, 3, p, 4)

i==3 w==4

j==1

j < w && s[(i+j)..(i+j)] == p[j..j] => true

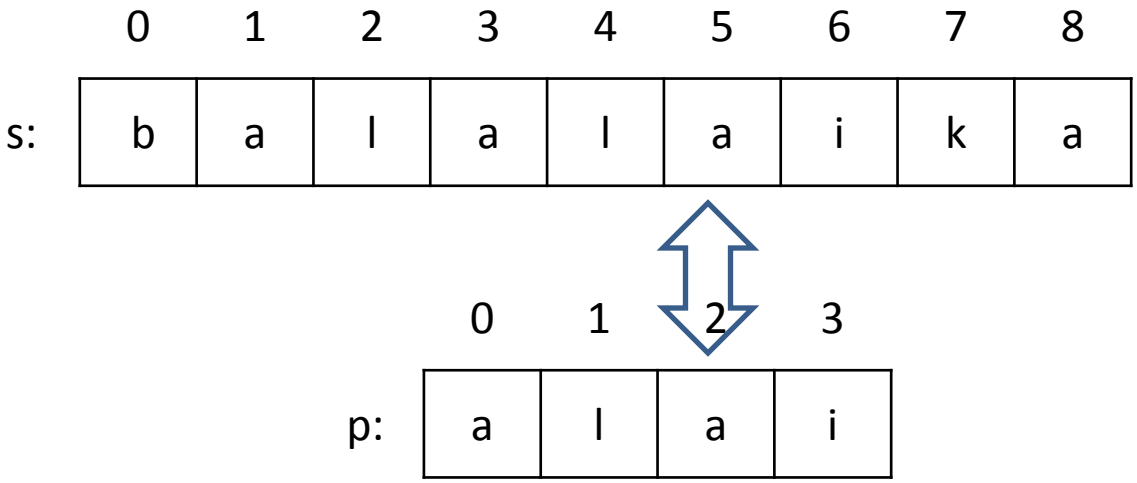


submatch(s, 3, p, 4)

i==3 w==4

j==2

j < w && s[(i+j)..(i+j)] == p[j..j]

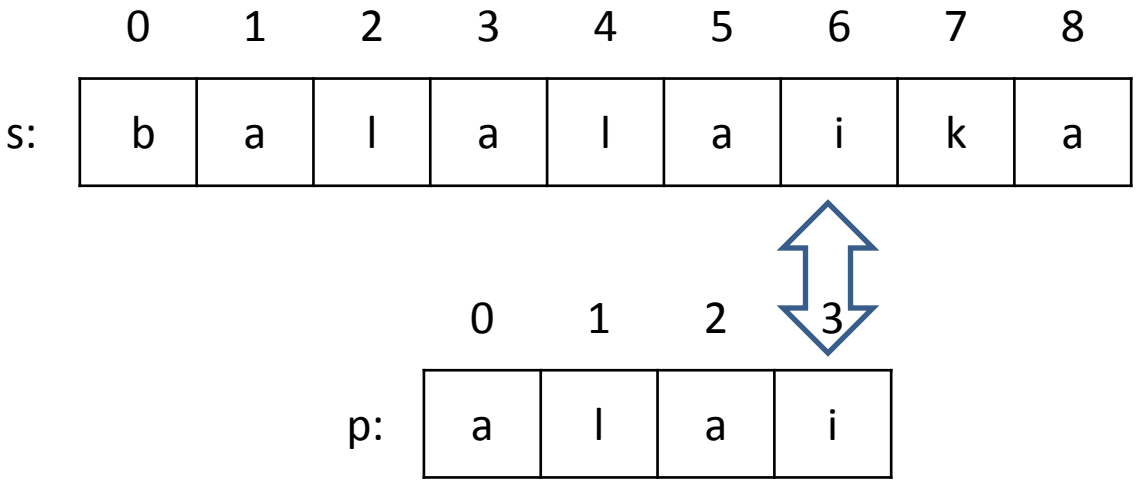


submatch(s, 3, p, 4)

i==3 w==4

j==2

j < w && s[(i+j)..(i+j)] == p[j..j] => true

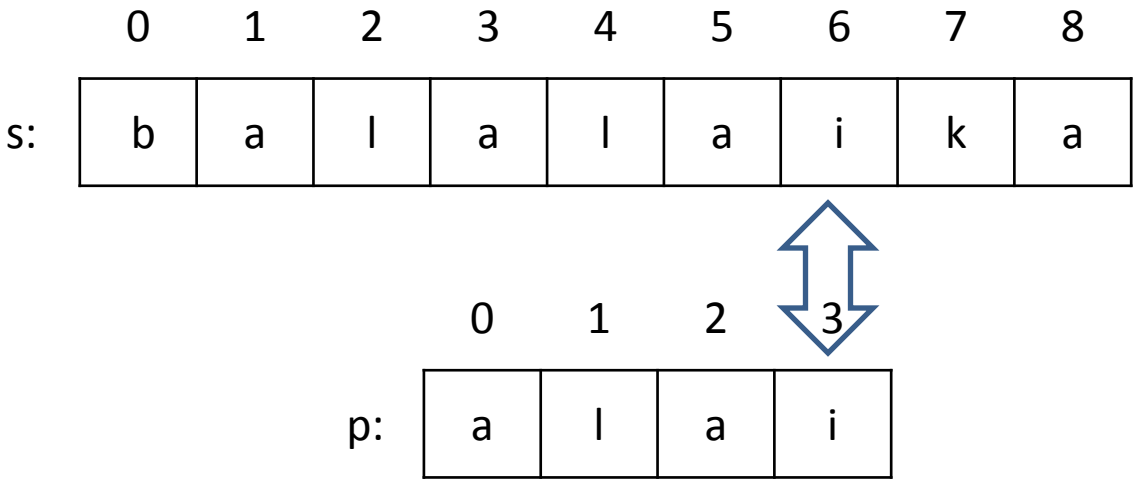


submatch(s, 3, p, 4)

i==3 w==4

j==3

j < w && s[(i+j)..(i+j)] == p[j..j]

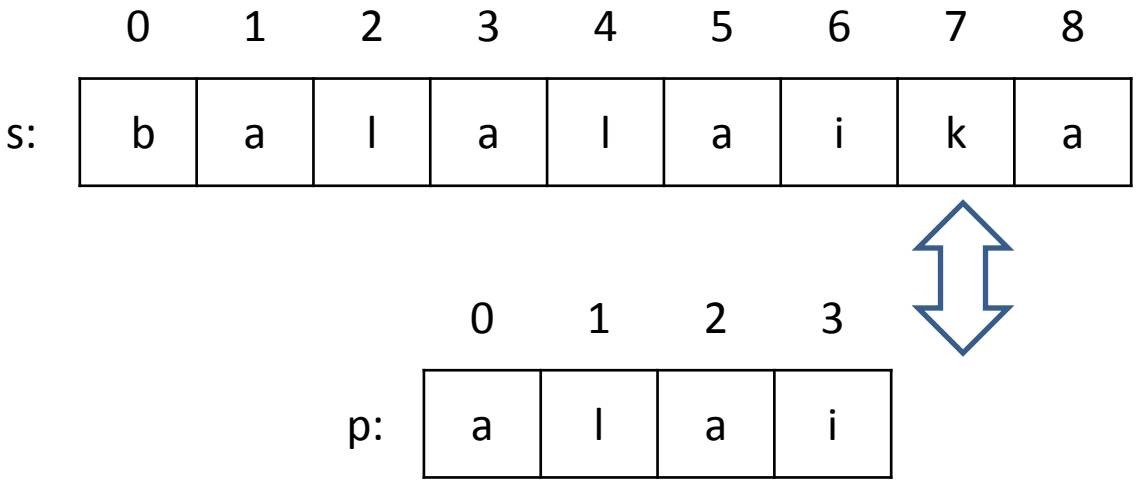


submatch(s, 3, p, 4)

i==3 w==4

j==3

j < w && s[(i+j)..(i+j)] == p[j..j] => true

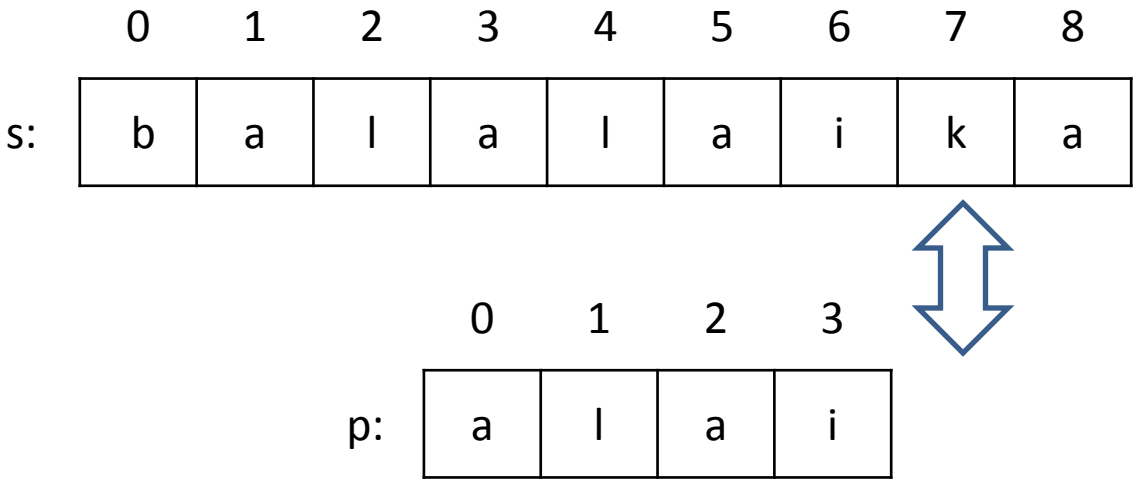


submatch(s, 3, p, 4)

i==3 w==4

j==4

j < w && s[(i+j)..(i+j)] == p[j..j]

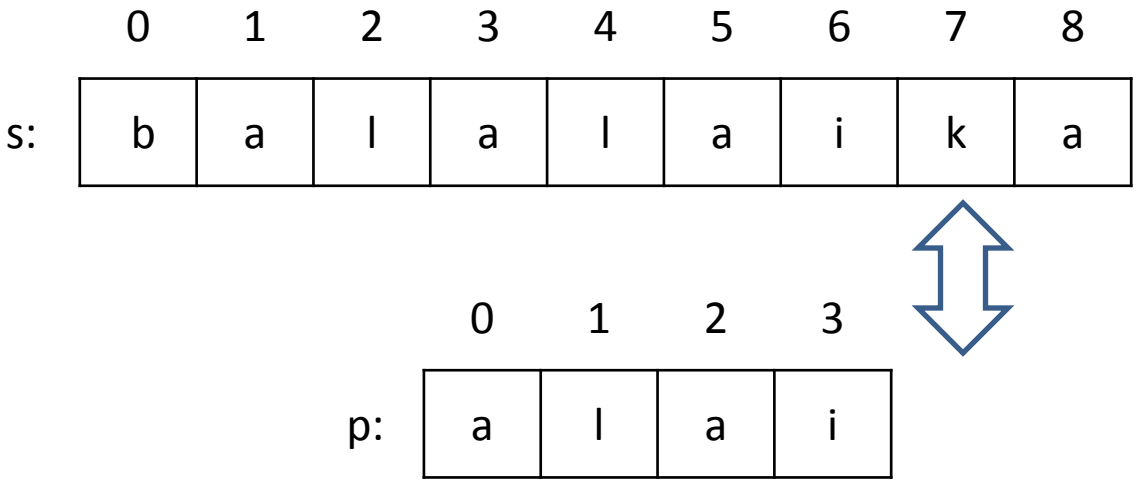


submatch(s, 3, p, 4)

i==3 w==4

j==4

j < w && s[(i+j)..(i+j)] == p[j..j] => false



submatch(s, 3, p, 4)

i==3 w==4

j==4

j < w && s[(i+j)..(i+j)] == p[j..j] => false

j < w が false なので
s[(i+j)..(i+j)] == p[j..j] は
検査されない

練習4.16

- match の定義では s 中に p が必ず現われることを仮定していた。p が現われない場合に -1 と答える match_safe(s,p) を定義せよ。（match_safe.rb）

4.4　定義のまとめ

条件を満たすまでの繰り返し: 式 が成り立つ間 命令$_1$ から 命令$_n$ を毎回実行する繰り返しは次のように書く。

```
while 式
    命令1
    ⋮
    命令n
end
```

練習問題確認プログラム

- ex04.rb（練習4.1-4.3と4.6-4.11と4.13b)と4.14-4.16）の結果を以下のアドレスに送れ。

  is-komaba@lyon.is.s.u-tokyo.ac.jp

- Subject:欄は、

  is-komaba

  としてください。

- 〆切: 12月XX日
  - 練習問題確認プログラムの結果は出席点の一部とします。結果の内容は問いません。（習得状況を知りたい。）