組み合わせの数の話

組み合わせの数

- n 個の中から k 個を選ぶ組み合わせの数を ${}_nC_k$ と書く。
- ${}_nC_k$ は以下のように計算できるのだが…

$$ {}_nC_k = \frac{n \times (n-1) \times \ldots \times (n-k+1)}{k \times (k-1) \times \ldots \times 1} $$

$$ {}_5C_3 = \frac{5 \times 4 \times 3}{3 \times 2 \times 1} = 10 $$

漸化式

- $0<k<n$ ならば、${}_nC_k = {}_{n-1}C_{k-1} + {}_{n-1}C_k$
- $0<k<n$ のとき、n 個からの k 個の選び方は、次の二つの場合に分けることができる。
  - n 個目を選ぶ場合
    - この場合の選び方は、${}_{n-1}C_{k-1}$ に等しい。
    - なぜなら、残りの n-1 個から k-1 個を選ぶので。
  - n 個目を選ばない場合
    - この場合の選び方は、${}_{n-1}C_k$ に等しい。
    - なぜなら、残りの n-1 個から k 個を選ぶので。

境界条件

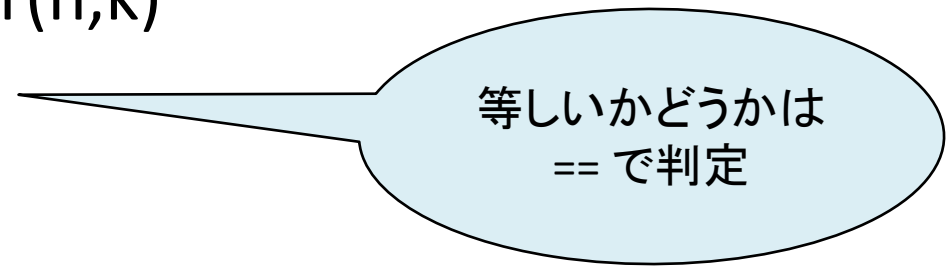
- n 個から一つも選ばない方法は一通りなので、${}_nC_0 = 1$
- n 個から n 個全部選ぶ方法も一通りなので、${}_nC_n = 1$

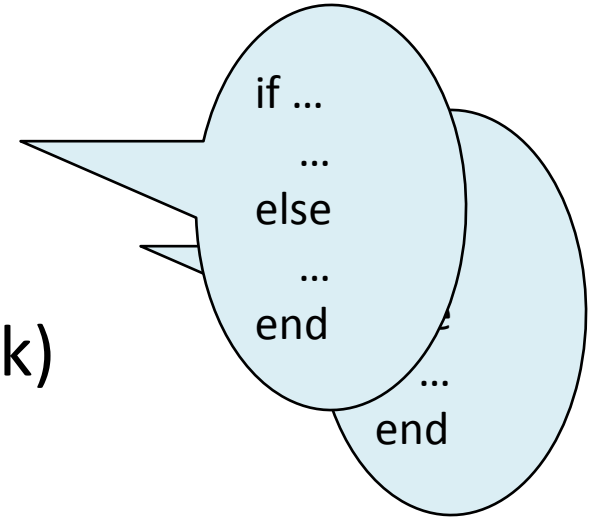
再帰的定義

```
def combr(n,k)
    if k==0
        1
    else
        if k==n
            1
        else
            combr(n-1,k-1)+combr(n-1,k)
        end
    end
end
```

再帰的定義

```
def combr(n,k)
   if k==0
      1
   else
      if k==n
         1
      else
         combr(n-1,k-1)+combr(n-1,k)
      end
   end
end
```





練習

- 指定されたサイトから comb.rb をダウンロード。
  - comb.rb の上で右クリック。
  - 「対象をファイル」に保存を選択。
- isrb を起動。

```
> load("comb.rb")
> combr(10,5)
> combr(20,10)
> combr(22,11)
…
```

なぜ遅い？

- 実は、combr(n,k) の計算は、${}_nC_k$ 自身に比例した時間がかかってしまう。

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

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	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

繰り返しによる定義

```
def comb(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
```

繰り返しによる定義

```
def comb(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
```

(n+1)×(n+1) の
テーブル（配列）を作り
局所変数 c に設定
（最初は全部 0）

i を 0 から n まで動かして
繰り返す

= は
値の
設定

for … in …
…
end

end

テーブルの i 行 j 列

練習

- 先ほどの続き

  > comb(10,5)

  > comb(20,10)

  > comb(40,20)

  > comb(100,50)

  …

  > exit

なぜ速い？

- comb(n,k) の計算は、n × n に比例した時間でできる。
- しかも、0≦m≦n, 0≦j≦m に対して、${}_mC_j$ の値がすべて求まっている。

- なお、Ruby では、いくらでも大きい整数を扱えることに注意。

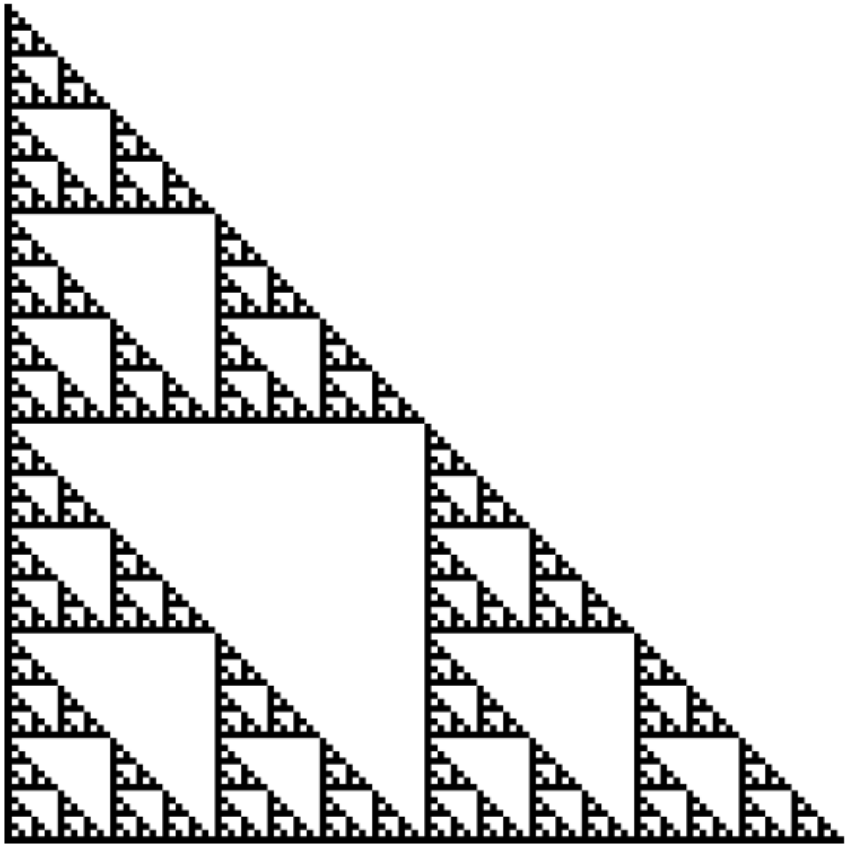
Pascal の三角形

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

Sierpinski の三角形



Sierpinski の三角形の作り方

```
def sierpinski(n)
    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]) % 2
        end
        c[i][i] = 1
    end
    show(c)
end
```

Sierpinski の三角形の作り方

```
def sierpinski(n)
    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]) % 2
        end
        c[i][i] = 1
    end
    show(c)
end
```

2 で割った余り

画像として表示

練習

- isrb を起動して、

  > load("comb.rb")

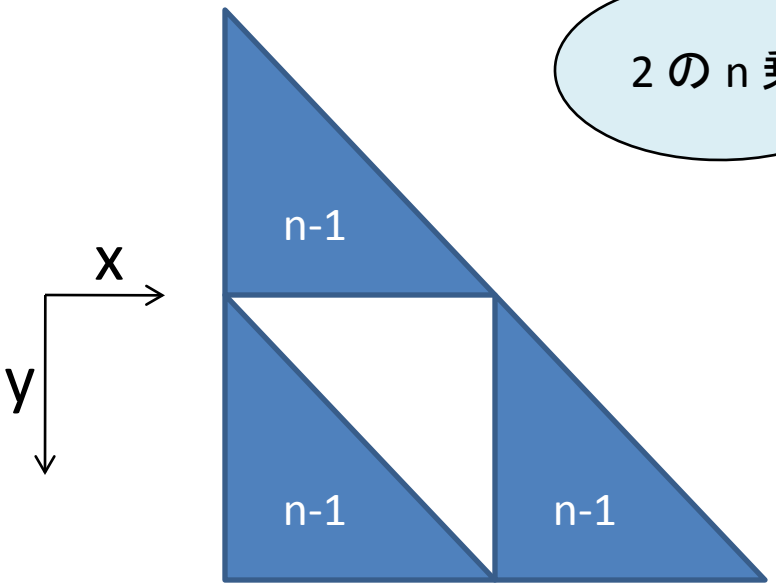
  > sierpinski(127)

  …

Sierpinski の三角形の別の作り方

```
def sier(n)
    a = make2d(2**n, 2**n)
    subsier(a, n, 0, 0)
    show(a)
end
```

2 の n 乗




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

局所変数
m の設定

練習

- 先ほどの続き。

  > sier(0)

  > sier(1)

  > sier(2)

  > sier(3)

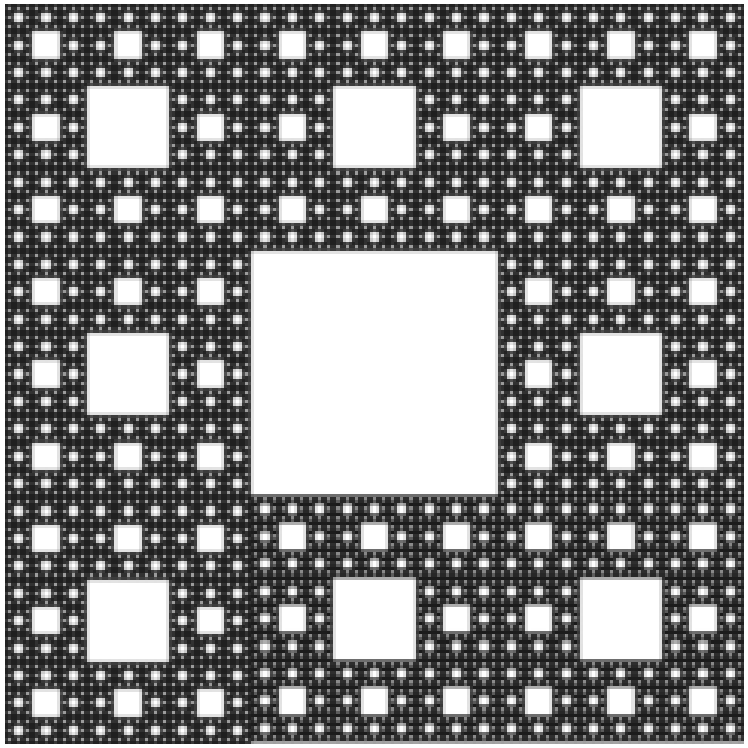
  > sier(4)

  > sier(7)

  …

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)
    show(a)
end
```

```
def subcarpet(a, n, x, y)
    if n==0
        a[y][x] = 1
    else
        m = 3**(n-1)
        #
        # ここを埋めてください
        #
    end
end
```

練習

- 適当なエディタを用いて、comb.rb を編集。
- subcarpet を完成させる。
- 再ロードして、カーペットを表示しよう。

```
> load("comb.rb")
> carpet(4)
> carpet(5)
```

- 進捗により、後回しにします。

二項分布

- コインを n 回投げて、表が k 回出る確率は ${}_nC_k \times (½)^n$
- この分布の平均は $E = n/2$
- この分布の標準偏差は $\sigma = \sqrt{n/4}$
- この分布を $E - w \times \sigma$ から $E + w \times \sigma$ まで表示するプログラム combdist(n,w)

練習

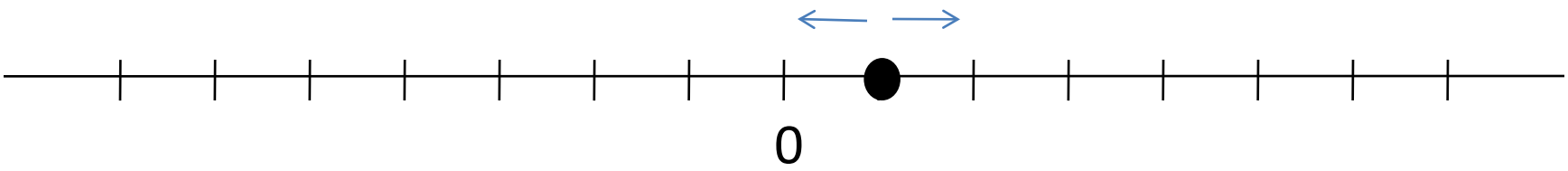
- 引き続き。
- combdist を呼び出そう。

```
> combdist(500,4)
> combdist(500,3)
…
```

- 二項分布は正規分布に近づく。
- 平均値±3σの範囲内に、99.7%が入る。

二項分布とランダムウォーク

- 粒子が1次元格子上を、原点を起点として、確率1/2で +1 あるいは -1 移動する。各粒子が n 回移動した後、格子点上のどの点にいるか。



- n が偶数のとき、粒子が座標 x の点にいる確率は、コインを n 回投げて、表が (x+n)/2 回出る確率に等しい。$_nC_{(x+n)/2} \times (½)^n$

2次元のランダムウォーク

- 粒子が2次元平面上を、原点を起点として、ランダムな方向に距離１ずつ進んで行く。
- 以下を実行してみよう。

```
> rw2(1000,10)
> rw2(1000,100)
```

時間が余ったら

- ウェブページに色々な教材があります。