アルゴリズムとデータ構造

- 第4回講義トピック:
  - 再帰 (recursion)
  - 分割統治 (divide-and-conquer)

Algorithms and Data structures

- Topics of lecture note 4:
  - recursion
  - divide-and-conquer

再帰 (recursion)

- 再帰とは、あるものについて記述する際に、記述しているものそれ自身への参照がその記述中にあらわれることをいう。
- 再帰呼び出しとは関数、あるいは手続き中で再びその手続き自身を呼び出すプログラミング法。
  数学的帰納法(induction)に類似した構造を持つ。
- 再帰の基本構造
  - n次の問題を解決する関数で、自分自身よりも低い部分集合（n - 1次以下）を用いる。さらにその部分集合は、より低次の部分集合を用いて定義するということを繰り返すアルゴリズム構造
  - 自分自身への呼出しが終了するための条件
- 再帰呼び出しは重要な制御構造の1つとして、複雑なアルゴリズムを明快に記述でき、効果的である。

Recursion

- In mathematics
  - Induction: to prove a statement holds for all natural number n
    base case: prove it holds for the first natural number n
    inductive step: prove if it is holds for n, then it holds for n+1
  - Recurrence relation (differential equation)
    given one or more initial terms of a sequence
    the further term is defined as a function of the preceding terms
  - Recursion
    a simple case that does not use recursion produces an answer
    a set of rules that reduce all other cases toward the base case
- Recursion in computer science
  - A programming function includes one or more function callings to itself with smaller data size.
  - It must has a end condition for the recursive calling.

再帰プログラムの一般構造

```
recursive (n)
{
   if (自明なケース) {      /* 終了条件 */
     自明なケースの処理;
   }
   else {                   /* 再帰呼び出し部分 */
     recursive (m);         /* m < n */
   }
}
```

The general structure of a recursive function

```
recursive (n)
{
   if (some base cases) {       /* end condition */
     non-recursive processing for base cases;
   }
   else {                       /* recursive part */
     recursive (m);             /* m < n */
   }
}
```

階乗関数の例

漸化式

$n! = n \cdot (n-1)!$　　$n>=1$

$0! = 1$（終了条件）

関数の形で表すと

$f(n) = n*f(n-1)$

$f(0) = 1$

プログラムによる実現

```
int factorial(int n){
      if (n == 0) return 1; // 終了条件
      return n*factorial(n-1); //再帰呼び出し
}
```

Factorial

Recurrence relation

$n! = n \cdot (n-1)! \quad n>=1$

$0! = 1$ (end condition)

In the manner of programming function

$f(n) = n*f(n-1)$

$f(0) = 1$

An implementation in C

```
int factorial(int n){
     if (n == 0) return 1; // end condition
     return n*factorial(n-1); // recursive calling
}
```

フィボナッチ (fibonacci) 数の例

フィボナッチ数列

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, …

漸化式

f(n) = f(n-1) + f(n-2)　　n >= 2

f(0) = f(1) = 1　　（終了条件）

プログラムによる実現

```
int fibonacci(int n)  {
  if (n <= 1) return 1; //終了条件
  return fibonacci(n-1)+fibonacci(n-2);//再帰呼び出し
  }
```

Fibonacci sequence

Sequence

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,

…

Recurrence relation

f(n) = f(n-1) + f(n-2) n >= 2

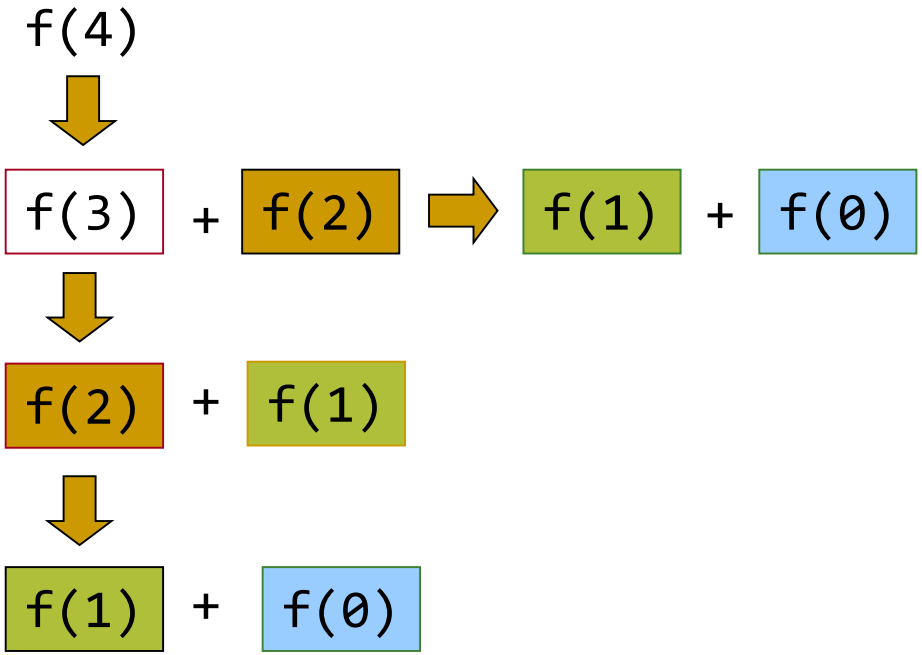
f(0) = f(1) = 1 (end condition)

An implementation in C:

```
int fibonacci(int n)  {
  if (n <= 1) return 1; // end condition
  return fibonacci(n-1)+fibonacci(n-2);//recursive calling
  }
```

フィボナッチ数アルゴリズムの展開

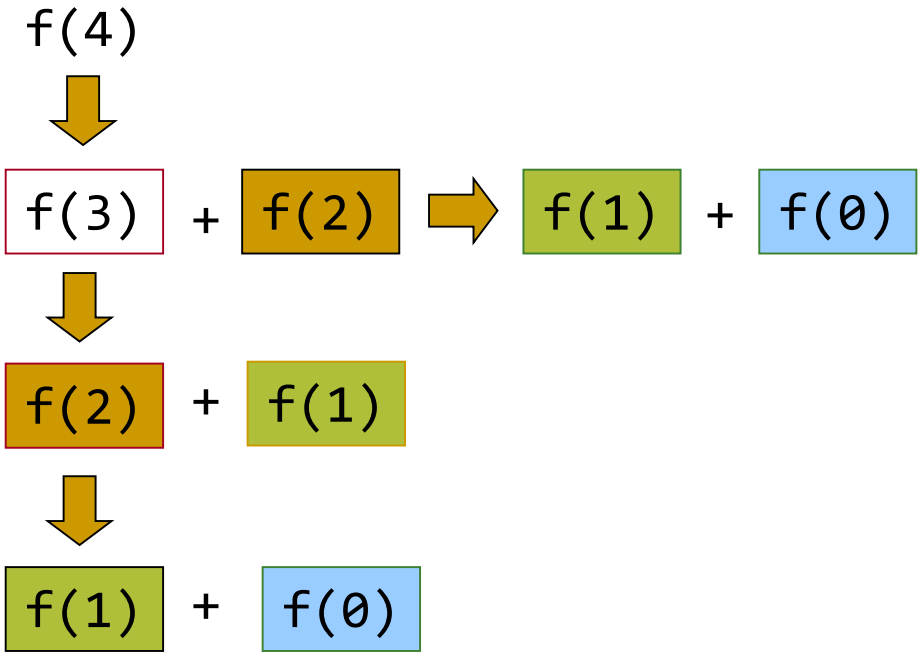
- この再帰アルゴリズムによるフィボナッチの計算は明快ではあるが、重複があって、実際のアルゴリズムとしては優れたものではない。



＊図で、同じ色のブロックは関数呼び出しの繰り返しを示す。

Calling sequence in Fibonacci sequence

- The recursive calling of Fibonacci sequency is simple, but is not an efficient algorithm.



In the figure, the blocks in same colors are repeated calling.

ホーナー法によるn次多項式の例

- 高次多項式を（ax + b）の形に展開していく
  例：$f(x) = x^4 + 2x^3 + 3x^2 + 4x + 5$
  $= (x^3 + 2x^2 + 3x + 4)x + 5$
  $= ((x^2 + 2x + 3)x + 4)x + 5$
  $= (((x + 2)x + 3)x + 4)x + 5$
- 漸化式で表すと
  $f(x, 4) = f(x, 3)*x + 5$, $f(x, 3) = f(x, 2)*x + 4$,
  $f(x, 2) = f(x, 1)*x + 3$, $f(x, 1) = f(x, 0)*x + 2$,
  $f(x, 0) = 1$
- ホーナー法漸化式の一般形
  $f(x,n) = a_0 x^n + a_1 x^{n-1} + \ldots a_{n-1} x + a_n$
  $f(x,0)=a_0$
  $f(x,n)=f(x,n-1)*x+a_n$

Calculate polynomials by Horner's method

- Transforming the polynomial into the form of (ax + b)
- Recurrence relation

  $f(x,n) = a_0 x^n + a_1 x^{n-1} + \ldots a_{n-1} x + a_n$

  $f(x,0)=a_0$

  $f(x,n)=f(x,n-1)*x+a_n$
- For example:

  $f(x) = x^4 + 2x^3 + 3x^2 + 4x + 5$

  $= (x^3 + 2x^2 + 3x + 4)x + 5$

  $= ((x^2 + 2x + 3)x + 4)x + 5$

  $= (((x + 2)x + 3)x + 4)x + 5$
- Recursive function calling

  $f(x, 4) = f(x, 3)*x + 5,\ f(x, 3) = f(x, 2)*x + 4,$

  $f(x, 2) = f(x, 1)*x + 3,\ f(x, 1) = f(x, 0)*x + 2,$

  $f(x, 0) = 1$

再帰によくある問題点

- 階乗の例の場合
  nに-1を与えてfactorial(-1)として呼び出すと、無限ループに入ってしまう。この種のバッグが見つけにくい
- 終了条件が成立しない再帰プログラム

```
int badprogram (unsigned n) {
   if (n==0) return 0
   else return badprogram(n/3 + 1) + n -1;
}
```

- フィボナッチ数の例の場合
  fibonacci(n)がfibonacci(n-1)とfibonacci(n-2)を、fibonacci(n-1)がfibonacci(n-2)とfibonacci(n-3) を、と呼び出し回数が増えるので、nの増加と共に指数関数的に増える。
  f=1.61803...(黄金分割比)のほぼn乗で知られる。

Some common problems in recursion

- Case of factorial:
  if the function is called by factorial(-1), then there will be an infinite loop.
- A case the end condition does not work:

```
int badprogram (unsigned n) {
   if (n==0) return 0
   else return badprogram(n/3 + 1) + n -1;
}
```

- Case of Fibonacci:
  fibonacci(n) calls fibonacci(n-1) and fibonacci(n-2), fibonacci(n-1) calls fibonacci(n-2) and fibonacci(n-3), fibonacci(n-2) calls fibonacci(n-3) and fibonacci(n-4) separately. The total number of calling will increase as exponential of n, for a base f=1.61803...(known as the golden ratio).

再帰を使わないフィボナッチ数アルゴリズム

- 配列を使って以前の計算結果を憶えておくことで、計算の回数を圧倒的に減らすことができる。漸化式の計算に使う典型的な技法の1つ。

```
#define max 50
int fibonacci(int n) {
  int i, f[max];
  f[0] = 1; f[1] = 1;
  for (i=2; i <= max; i++) f[i] = f[i-1]+f[i-2];
  return f[n];
 }
```

＊このアルゴリズムはフィボナッチ数を線形的な時間（nに比例する）で計算できる。

Fibonacci algorithm without recursion

- Use an array to store the calculated Fibonacci numbers can drastically reduce the computation of a large Fibonacci number. A typical technique for calculation using recurrence relation.

```
#define max 50
int fibonacci(int n) {
  int i, f[max];
  f[0] = 1; f[1] = 1;
  for (i=2; i <= max; i++) f[i] = f[i-1]+f[i-2];
  return f[n];
 }
```

note: this algorithm calculates a Fibonacci number with computation in linear time (in order of n).

分割統治(divide-and-conquer)

- 分割統治とは
  - データをいくつかの部分に分割して、その1つ1つについて自分自身を再帰的に適用していく問題解決パラダイム
  - 再帰的アルゴリズムで実現されることが多い。
- 一般的に以下の部分から構成される
  - 入力を2つ以上の部分に分割する
  - それぞれの部分を独立に解く
  - すべての部分を統合する
- 分割する時にフィボナッチ数の例の様な重複を避ける

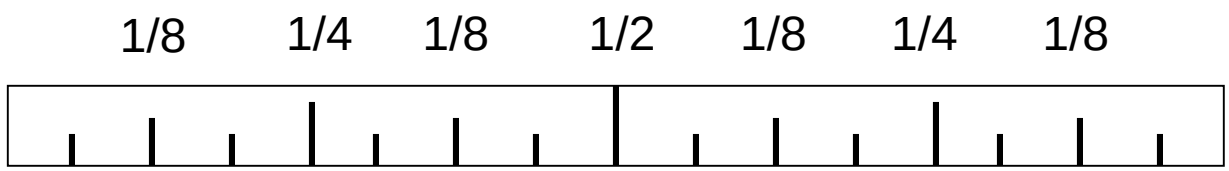
Divide-and-conquer

- A paradigm to divide a problem into sub-problems of same or related type and than solve them by a recursive algorithm.
  - divide the problem into sub-problems
  - solve each sub-problem
  - combine the results of sub-problems
- The sub-problems should have not (or as less as) overlaps (a bad example: Fibonacci number).

目盛を付けるプログラムの例

- 定規の目盛りとして、1/2の点に線をひき、1/4の点に少し短い線をひき、1/8の点にさらに短い線をひき、…

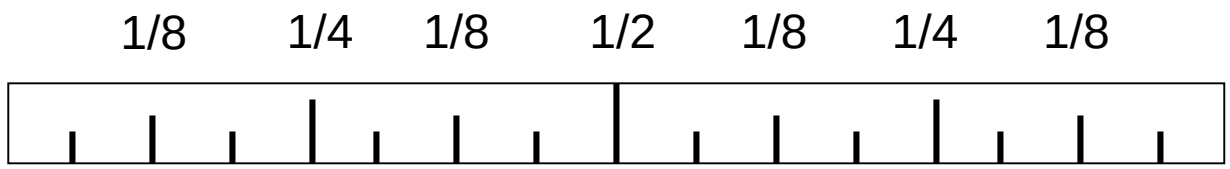
```
void rule(int l, int r, int h) {
  int m = (l+r)/2;    /* 真中の位置 */
  if (h > 0) {
    rule(l, m, h-1); /* 左半分で線高h-1を引く */
    mark(m, h);       /* 真中で線高hを引く */
    rule(m, r, h-1); /* 右半分で線高h-1を引く */
  }
}
```



Marking scales of a ruler

- A recursive algorithm marking scales of a ruler in the order or ½, ¼, 1/8, …, with decreasing line length.

```
void rule(int l, int r, int h) {
  int m = (l+r)/2;   /* middle position */
  if (h > 0) {
    rule(l, m, h-1); /* left ¼ scale (line h-1) */
    mark(m, h);      /* half scale (line h) */
    rule(m, r, h-1); /* right ¼ scale (line h-1) */
  }
}
```

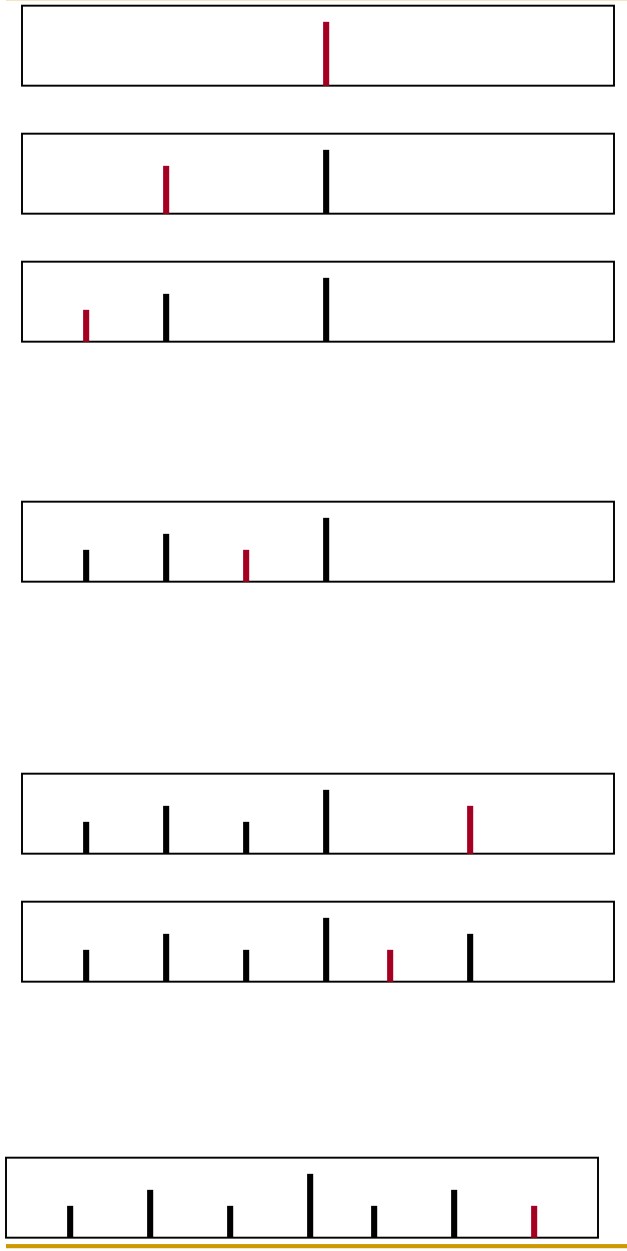


先行順の例

```
void   rule(int   l,
int r, int h){
  int m = (l+r)/2;
  if (h > 0){
    mark(m,h);
    rule(l,m,h-1);
    rule(m,r,h-1);
  }
}
```

*中央の線を先に引き、後で左と右の部分を引く

rule(0,8,3)
mark(4,3)
rule(0,4,2)
mark(2,2)
rule(0,2,1)
mark(1,1)
rule(0,1,0)
rule(1,2,0)
rule(2,4,1)
mark(3,1)
rule(2,3,0)
rule(3,4,0)
rule(4,8,2)
mark(6,2)
rule(4,6,1)
mark(5,1)
rule(4,5,0)
rule(5,6,0)
rule(6,8,1)
mark(7,1)
rule(6,7,0)

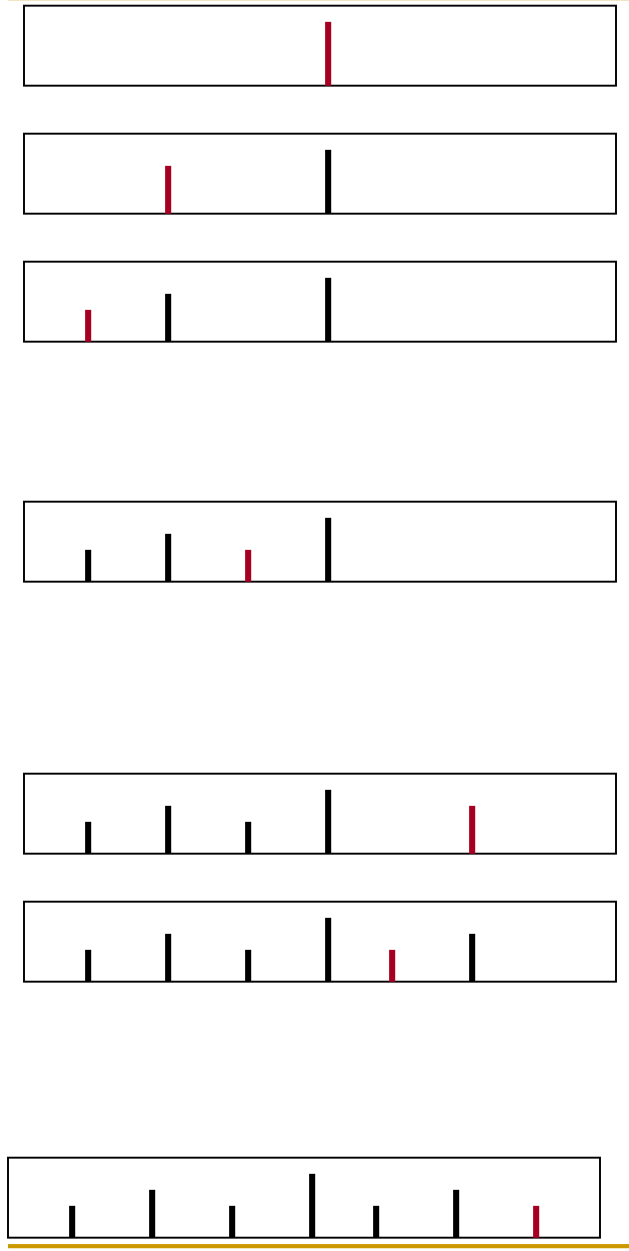


In preorder

```
void   rule(int   l,
int r, int h){
  int m = (l+r)/2;
  if (h > 0){
    mark(m,h);
    rule(l,m,h-1);
    rule(m,r,h-1);
  }
}
```

Mark the middle positon first and then the left and right sides.

rule(0,8,3)
mark(4,3)
rule(0,4,2)
mark(2,2)
rule(0,2,1)
mark(1,1)
rule(0,1,0)
rule(1,2,0)
rule(2,4,1)
mark(3,1)
rule(2,3,0)
rule(3,4,0)
rule(4,8,2)
mark(6,2)
rule(4,6,1)
mark(5,1)
rule(4,5,0)
rule(5,6,0)
rule(6,8,1)
mark(7,1)
rule(6,7,0)



中央順の例

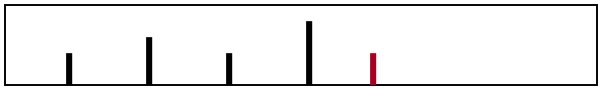
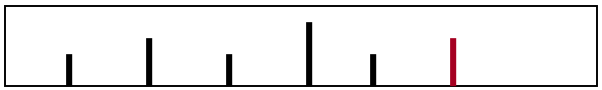
```
void rule(int l,
int r, int h){
  int m = (l+r)/2;
  if (h > 0){
    rule(l,m,h-1);
    mark(m,h);
    rule(m,r,h-1);
  }
}
```

*左の部分を先に引き、次に中央の線を、最後に右の部分を引く

rule(0,8,3)
  rule(0,4,2)
    rule(0,2,1)
      rule(0,1,0)
      mark(1,1)
      rule(1,2,0)
    mark(2,2)
    rule(2,4,1)
      rule(2,3,0)
      mark(3,1)
      rule(3,4,0)
  mark(4,3)
  rule(4,8,2)
    rule(4,6,1)
      rule(4,5,0)
      mark(5,1)
      rule(5,6,0)
    mark(6,2)
    rule(6,8,1)
      rule(6,7,0)
      mark(7,1)





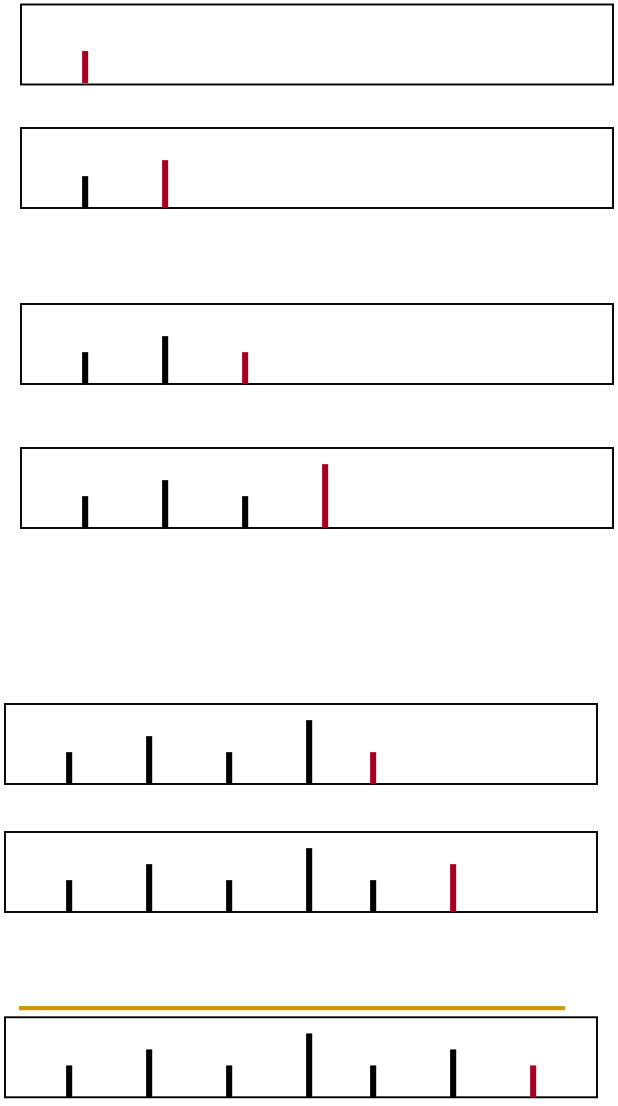




In inorder

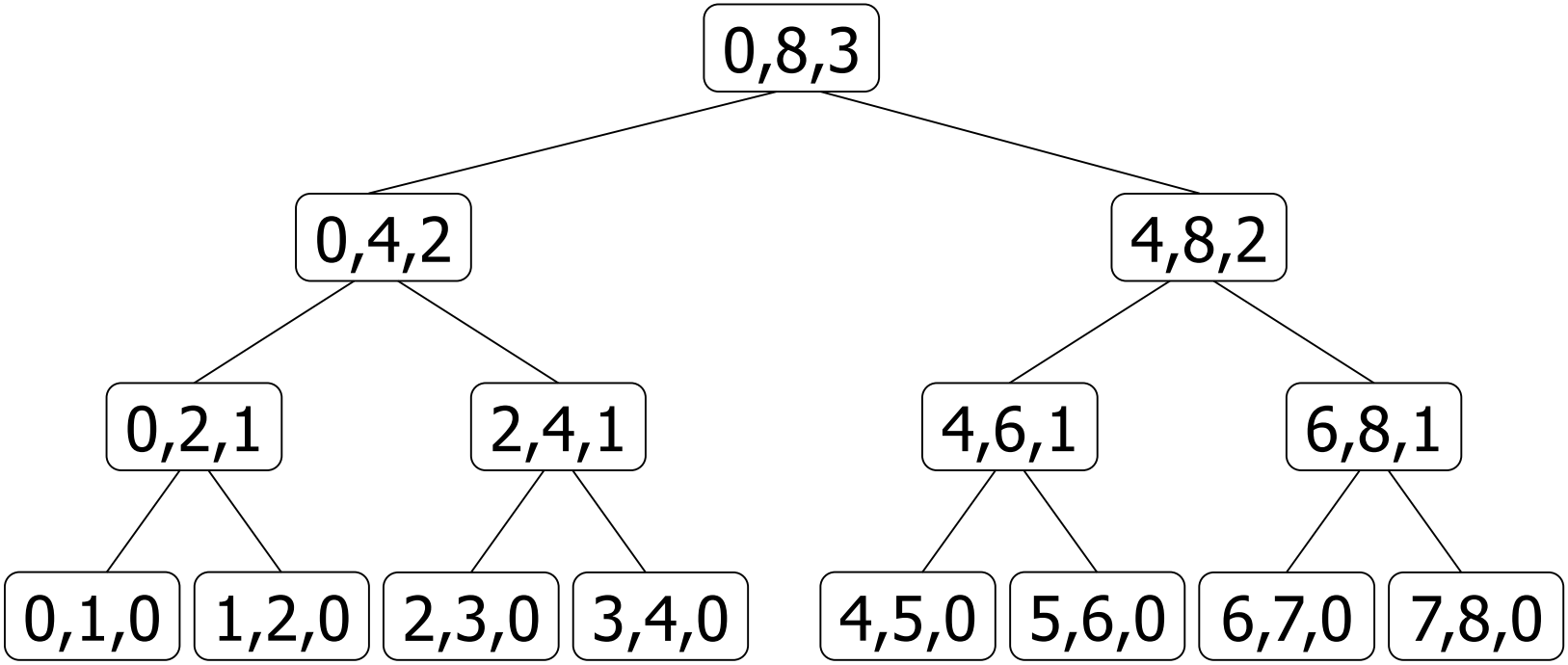
```
void rule(int l,
int r, int h){
  int m = (l+r)/2;
  if (h > 0){
    rule(l,m,h-1);
    mark(m,h);
    rule(m,r,h-1);
  }
}
```

Mark the left scale first then the middle and then the right.

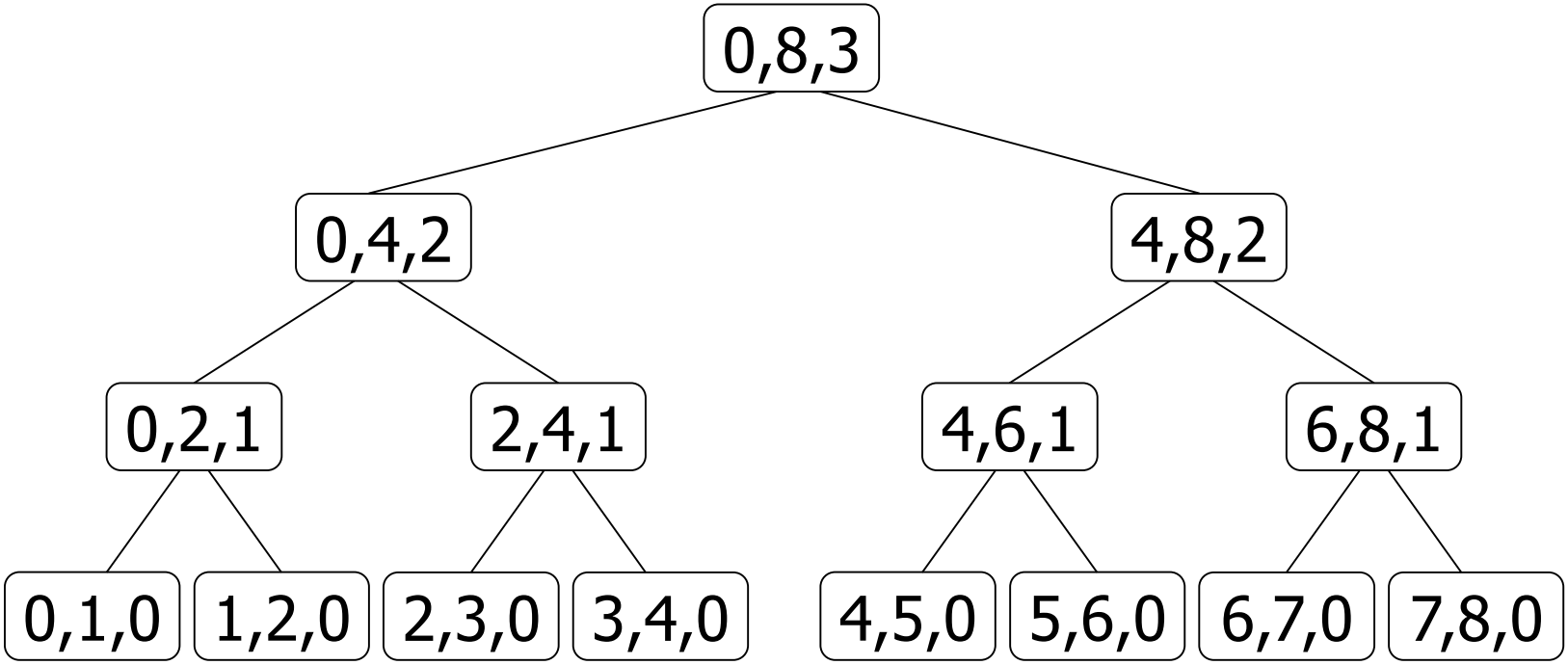
```
rule(0,8,3)
  rule(0,4,2)
    rule(0,2,1)
      rule(0,1,0)
      mark(1,1)
      rule(1,2,0)
    mark(2,2)
    rule(2,4,1)
      rule(2,3,0)
      mark(3,1)
      rule(3,4,0)
  mark(4,3)
  rule(4,8,2)
    rule(4,6,1)
      rule(4,5,0)
      mark(5,1)
      rule(5,6,0)
    mark(6,2)
    rule(6,8,1)
      rule(6,7,0)
      mark(7,1)
```



物差しに目盛りを付ける再帰呼び出しの木



Tree for scale marking recursive algorithm



再帰を使わない目盛を付けるプログラム

- 最も短い線をひき、次に短い線をひき、…

```
rule(int l, int r, int h) {
    int i, j, t;
    for (i = 1, j = 1; i <= h; i++, j+=j)
        for (t = 0; t <= (l+r)/j; t++)
            mark(l+j+t*(j+j),i);
}
```

- この問題解決の方法は分割統治のトップダウンに対して、ボトムアップ型であるといえる。
- このように、自明の小さな問題を解いて最後に大きな問題を解くパラダイムを統合統治(combine-and-conquer)ともいう。

Scale marking by a non-recursive algorithm

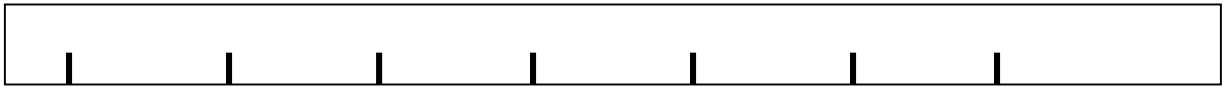
- With a loop for different line length and a loop for different positions

```
rule(int l, int r, int h) {
   int i, j, t;
   for (i = 1, j = 1; i <= h; i++, j+=j)
      for (t = 0; t <= (l+r)/j; t++)
        mark(l+j+t*(j+j),i);
}
```

- This bottom-up type algorithm is called as combine-and-conquer paradigm.

非再帰的な物差しの描画

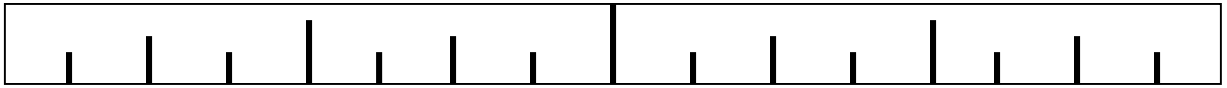
まず、一番短い線を引き、



次に、少し長い線を引き、

さらに、少し長い線を引き、

最後に一番長い線を引く



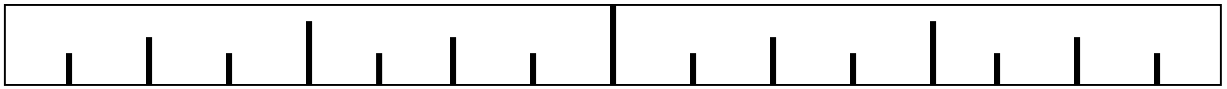
Process of the non-recursive algorithm

Shortest scale lines

Longer scale lines

Even longer scale lines

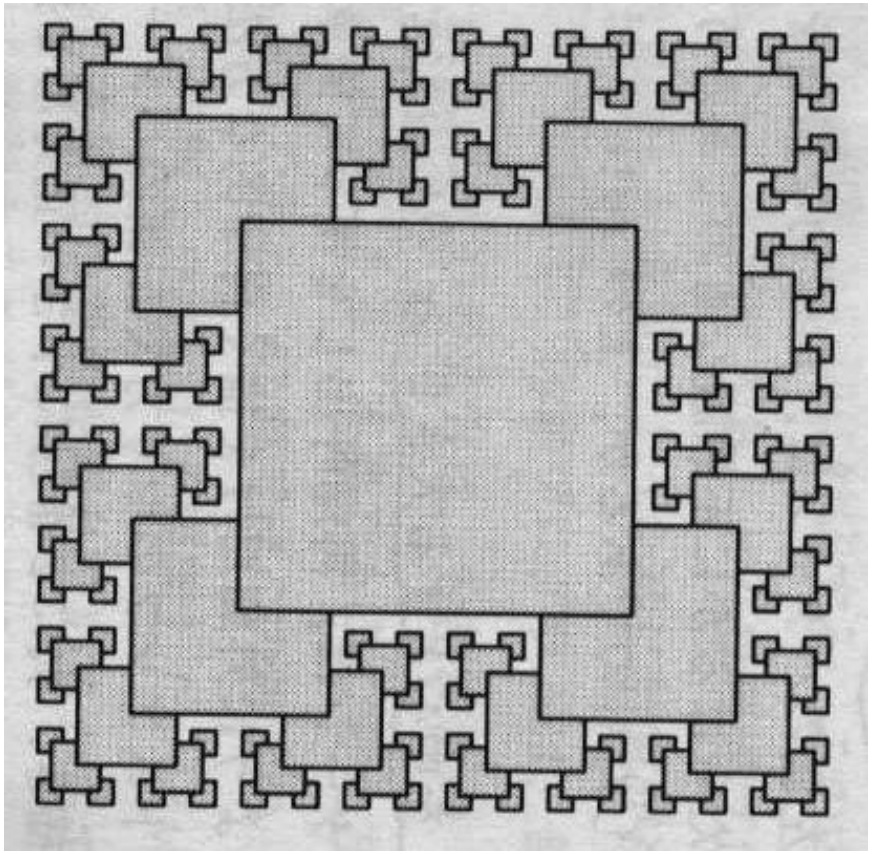
Longest scale lines



フラクタルの星の例

```
star(int x, int y, int r)
  {
      if ( r > 0 ){
        star(x-r, y+r, r/2);
        star(x+r, y+r, r/2);
        star(x-r, y-r, r/2);
        star(x+r, y-r, r/2);
        box(x, y, r);
    }
  }
```

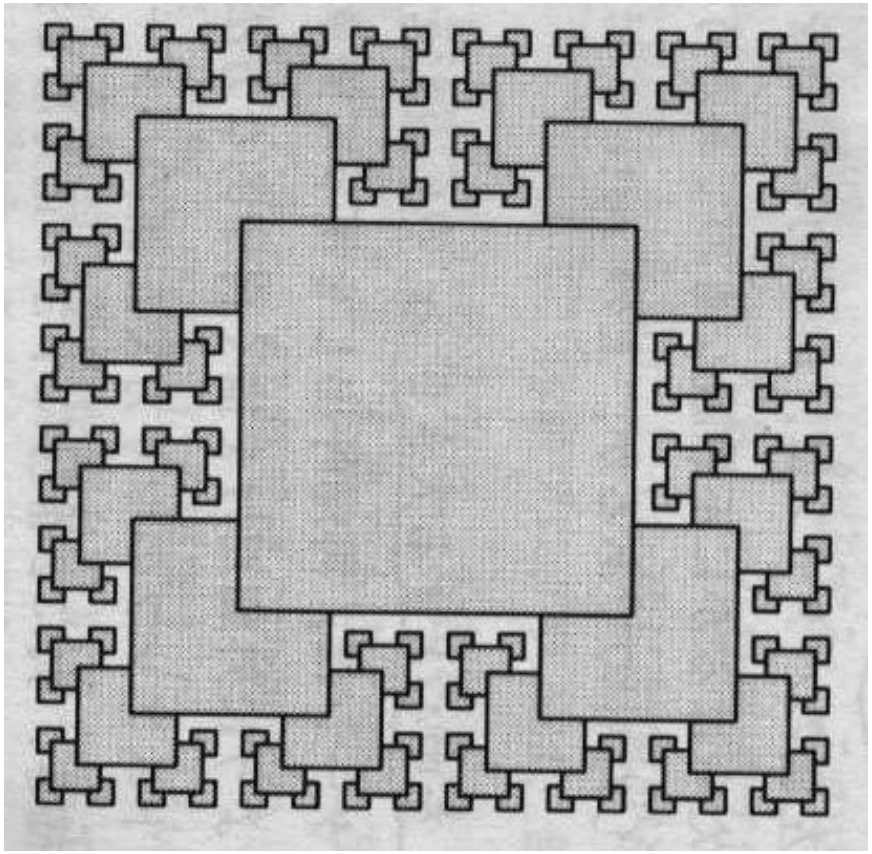
＊ここで、box関数は枠の中を塗つぶすが、line関数だけを使った場合は工夫が必要となる。



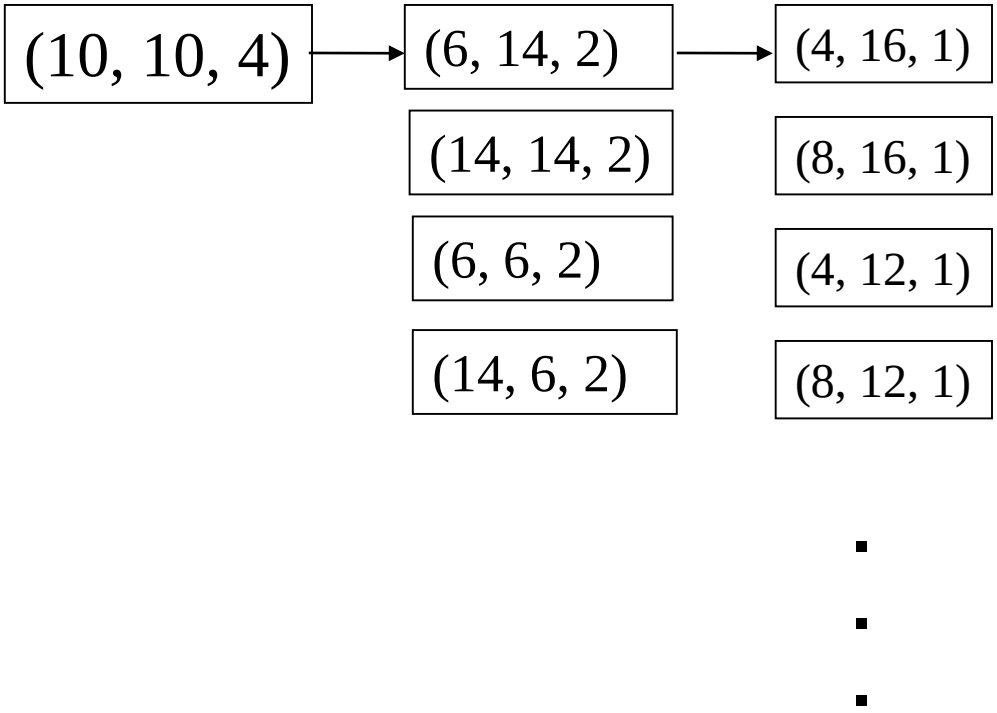
Star fractal

```
star(int x, int y, int r)
  {
      if ( r > 0 ){
        star(x-r, y+r, r/2);
        star(x+r, y+r, r/2);
        star(x-r, y-r, r/2);
        star(x+r, y-r, r/2);
        box(x, y, r);
    }
  }
```

The box function has to fill the area. If you use line function only you will need some techunique.



再帰呼び出しの様子（一部分）



Recursive calling (in part)