アルゴリズムとデータ構造

第８回　再帰的アルゴリズム

第8回 再帰的アルゴリズム

- 今日の内容
  - 再帰的アルゴリズム
    (これまでにも再帰は出てきましたが、改めて)
  - 再帰的アルゴリズムと、
    ループを使ったアルゴリズムの対応
  - 再帰的アルゴリズムの時間計算量の解析

- ポイント
  - 再帰は、強力なツールなので、使いこなそう

再帰呼出し

再帰呼出し

関数がその定義の中でそれ自身を呼び出すこと

再帰的アルゴリズム

再帰呼出しを用いて記述されたアルゴリズム

再帰的アルゴリズムの例（その1）

自然数 n の階乗 n! を求めるアルゴリズム

$$n! = \begin{cases} 1 & (n = 1 \text{ の場合}) \\ n \times (n-1)! & (\text{その他の場合}) \end{cases}$$

```
int factorial(int n) {
  if (n==1) return 1;
  else      return n*factorial(n-1);
}
```

例

$4! = 4 \times 3! = 24$

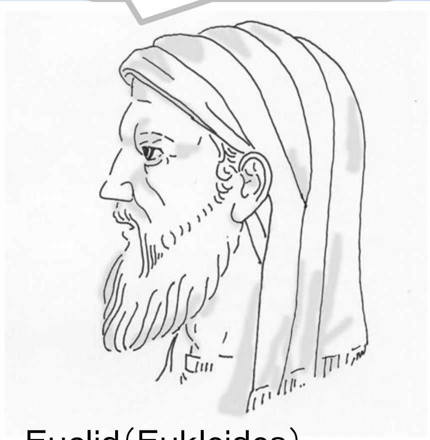
$3! = 3 \times 2! = 6$

$2! = 2 \times 1! = 2$

$1! = 1$

再帰的アルゴリズムの例（その2）

9と6のgcd
6と3のgcd



Euclid（Eukleides）
ユークリッド（エウクレィデス）
（紀元前300年ごろ）
Wikipedia より

最大公約数を求めるアルゴリズム（ユークリッドの互除法）

2つの自然数 m, n (m≧n) の最大公約数 gcd(m, n)

```
int gcd(int m, int n)
{
    int  r = m%n;
    if (r==0) return n;
    else        return gcd(n,r);
}
```

実は、もっと簡潔に以下のようにも書ける

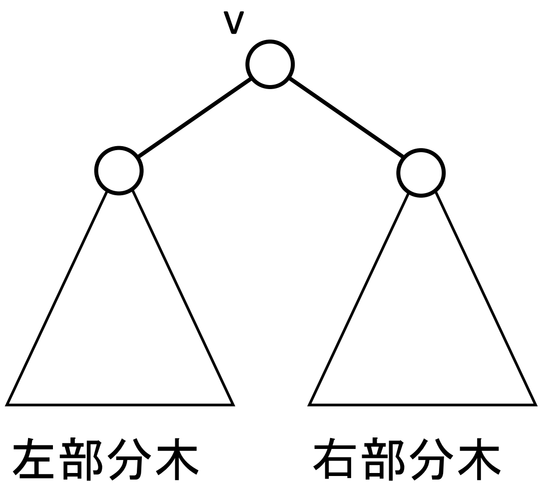
```
int gcd(int m, int n)
{
    if (n ==0) return m;
    else         return gcd(n, m%n);
}
```

再帰的アルゴリズムの例（その3）

木の巡回（第6回）の復習

```
void preorder(node *v)
{
    if (v==null) return;
    v を出力する
    preorder(v->left);
    preorder(v->right);
}

void main( )
{
    preorder(root);
}
```



preorder の順番
v　左部分木　右部分木

再帰的アルゴリズムの例（その4）

自然数 a, n に対して a^n を求めるアルゴリズム

```
int power (int a, int n)
{
    int m, b;
    if (n == 1)  return a;
    m = n / 2;        // 小数点以下切り捨て
    b = power( a, m );
    b = b * b;
    if ( n % 2 == 0 )  return b;
    else               return b * a;
}

void main( )
{
    power(a, n) を出力;
}
```

n = 100 の時には、

$a^{100} = a^{50} \times a^{50}$

n = 101 の時には、

$a^{101} = a^{50} \times a^{50} \times a$

を計算する

a を n回かけ続けるよりも、
ずっと速く計算できる

（このアルゴリズムの
計算時間の解析は後ほど）

再帰呼出しを使うことのメリット

- 記述が簡潔になる
- 理解しやすくなる
- アルゴリズムの正しさの証明がしやすくなる
- 計算量の解析が容易になる

休憩

- ここで、少し休憩しましょう。
- 深呼吸したり、肩の力を抜いてから、
次のビデオに進んでください。

再帰的アルゴリズムの作り方

重要

作り方のコツ：数学的帰納法で証明を書くつもりで！

1. （漸化式を立てる）解くべき問題を、同じ問題で
   よりサイズの小さなものを解くことに帰着させる

   例） m と n (m ≧ n) の最大公約数は、n と m % n の最大公約数と同じ
   gcd(m, n) = gcd(n, m % n) for n > 0

2. （終端条件を示す）最小サイズの問題の解を示す

   例) m と n (m ≧ n) の最大公約数は、m % n = 0 のとき n

再帰呼出しを使わない方法も …

関数の最初または最後に
１回だけ再帰呼出しされる場合

⇒

ループを用いて比較的容易に
書ける場合が多い

```
gcd(int m, int n)
{
    if (n==0) return m;
    else      return gcd(n, m%n);
}
```

再帰呼出しを
使わないで書く

⇒

```
gcd(int m, int n)
{
    while (n != 0) {
        int tmp;
        m = m % n;
        tmp = m; m = n; n = tmp;
    }
    return m;
}
```

m, n の交換

○ 記述がスッキリ

× 記述は少し複雑
○ メモリ使用量が少ない
(実行時に使うスタック領域が少ない)
○ 計算時間も短い
(関数呼出し、復帰処理がない)

再帰呼出しを使わない方法も …

関数の最初または最後に
１回だけ再帰呼出しされる場合

⇨

ループを用いて比較的容易に
書ける場合が多い

```
gcd( int m, int n )
{
    if ( n == 0 ) return m;
    else          return gcd( n, m % n );
}
```

再帰呼出しを
使わないで書く

⇨

```
gcd( int m, int n )
{
    while ( n != 0 ) {
        int tmp;
        m = m % n;
        tmp = m; m = n; n = tmp;
    }
    return m;
}
```

m, n の交換

例

gcd(76, 30)
gcd(30, 16)
gcd(16, 14)
gcd(14, 2)
gcd(2, 0)

× 記述は少し複雑
○ メモリ使用量が少ない
(実行時に使うスタック領域が少ない)
○ 計算時間も短い
(関数呼出し、復帰処理がない)

再帰呼出しを使わない方法も …

木の巡回（第6回）の復習

```
void preorder(node *v)
{
    if (v==null) return;
    v を出力する
    preorder(v->left);
    preorder(v->right);
}

void main( )
{
    preorder(root);
}
```

再帰呼出しを
使わないで書く

⇨

```
void main( )
{
    Stack S;  // 空のスタック
    S.push(root);
    while (! S.isEmpty( )) {
        node v = S.pop( );
        if (v != null) {
            v を出力する
            S.push(v.right);
            S.push(v.left);
        }
    }
}
```

○ 記述がスッキリして、
動作が理解しやすい

再帰呼出しの時間計算量

重要

再帰的アルゴリズムの時間計算量

= 再帰呼出しの時間計算量 ＋ 再帰呼出し以外の時間計算量

例）

```
int factorial(int n) {
  if (n==1) return 1;
  else       return n*factorial(n-1);
}
```

factrial(n) の実行時間を T(n) とおくと

再帰呼出し factrial(n-1) の時間計算量
= T(n-1)

再帰呼出し以外の時間計算量 $\leqq$ c (定数)

よって　$T(n) \leqq T(n-1) + c,\ T(1) \leqq c$

したがって　$T(n) \leqq c\,n = O(n)$

再帰呼出しの時間計算量

重要

再帰的アルゴリズムの時間計算量

＝ 再帰呼出しの時間計算量 ＋ 再帰呼出し以外の時間計算量

例）

```
int factorial(int n) {
  if (n==1) return 1;
  else      return n*factorial(n-1);
}
```

factrial(n) の実行時間を T(n) とおくと

再帰呼出し factrial(n-1) の時間計算量
＝ T(n-1)

再帰呼出し以外の時間計算量 ≦ c (定数)

よって　$T(n) \leqq T(n-1) + c, \ T(1) \leqq c$

$$
\begin{aligned}
T(n) &\leqq T(n-1) + c \\
&\leqq T(n-2) + 2c \\
&\leqq T(n-3) + 3c \\
&\cdots \\
&\leqq T(1) + (n-1)c \\
&\leqq cn
\end{aligned}
$$

したがって　$T(n) \leqq cn = O(n)$

再帰呼出しの時間計算量

再帰的アルゴリズムの時間計算量

= 再帰呼出しの時間計算量 ＋ 再帰呼出し以外の時間計算量

例）

```
int power (int a, int n)
{
    int m, b;
    if (n == 1)  return a;
    m = n / 2;          // 小数点以下切り捨て
    b = power( a, m );
    b = b * b;
    if ( n % 2 == 0 )  return b;
    else               return b * a;
}
```

$T(n) \leqq T(n/2) + c,\ \ T(1) \leqq c$

したがって

$$
\begin{aligned}
T(n) &\leqq T(n/2) + c \\
&\leqq T(n/4) + 2c \\
&\leqq T(n/8) + 3c \\
&\quad \cdots \\
&\leqq T(1) + c \log_2 n
\end{aligned}
$$

となり、$T(n) = O(\log n)$ と分かる

再帰呼出しの時間計算量

再帰的アルゴリズムの時間計算量

＝ 再帰呼出しの時間計算量 ＋ 再帰呼出し以外の時間計算量

n を
半分の半分の半分の … とやって
何回で 1 になる？ → $\log_2 n$ 回

言い方を変えると、

1 を
2倍の2倍の2倍の … とやって
何回で n になる？ → $\log_2 n$ 回

だって、$2^{\log_2 n} = n$ だから

$T(n) \leqq T(n/2) + c$, $T(1) \leqq c$

したがって

$$
\begin{aligned}
T(n) &\leqq T(n/2) + c \\
&\leqq T(n/4) + 2c \\
&\leqq T(n/8) + 3c \\
&\cdots \\
&\leqq T(1) + c \log_2 n
\end{aligned}
$$

となり、$T(n) = O(\log n)$ と分かる

ハノイの塔

ハノイの塔は、フランスの数学者 E･リュカ (Edouard Lucas) が1883年に考えたものである。リュカは、インドに次のような伝説があると説明している。

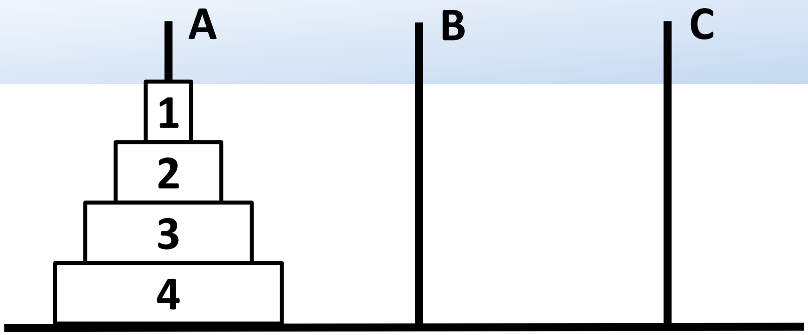
ブラフマーの塔

インドのガンジス河の畔のベナレス（ヴァラナシ）に世界の中心を表すという聖堂がある。そこには3本の大理石の柱（ダイヤモンドの針との説もあり）が立てられており、そのうちの1本には、当初64枚の黄金の円盤が大きい円盤から順に重ねられていたという。バラモン僧たちはそこで、一日中円盤を別の柱に移し替える作業を行っている。そして、全ての円盤の移し替えが終わったときに、この世は崩壊し終焉を迎えると言われている。

もちろん、これはリュカの作り話であるが、64枚の円盤を移動させるには、最低でも 18,446,744,073,709,551,615 回かかり、1枚移動させるのに1秒かかったとして、約 5,845 億年かかる（なお、ビッグバンは今から約137億年前の発生とされている）。

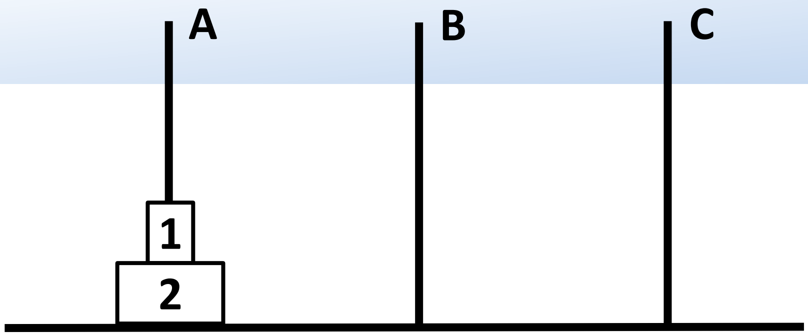
ハノイの塔（パズル）

パズルの説明を読んで、n = 2 の時、n = 3 の時に、どの円盤をどの杭からどの杭に移動させていうか、検討してください。さらに、一般の n では、どうすべきか検討してみてください。



- 3本の杭 A, B, C と、n 枚の円盤 1, 2, …, n がある
- 初期状態では、円盤はすべて杭 A にあり、数字の小さい円盤が上になるように順に積み重ねられている
- 以下のルールにより、すべての円盤を杭 B に移動させる
  - いずれかの杭の一番上にある円盤を一つ選び、他の杭に移動させる。これを1回の操作と数える
  - 移動先の杭に円盤がある場合には、その円盤がいま移動させようとしている円盤よりも大きな数字でなければならない

ハノイの塔（パズル）

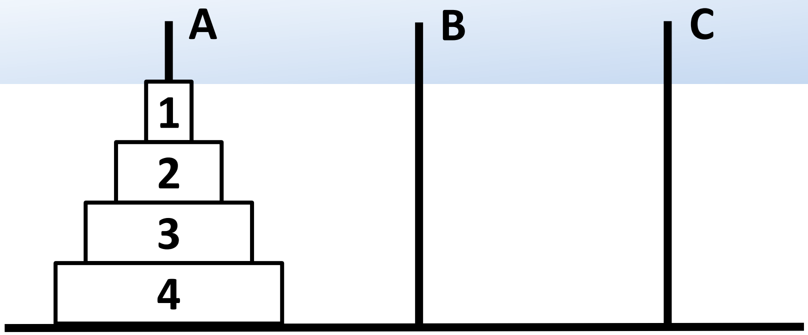


n = 2 の場合

目標: 円盤 1 〜 2 を A → B (杭 A から杭 B に移動させる)

- 円盤 1 を A → C
  - ここで、円盤 2 は A → C とは動かせない
- 円盤 2 を A → B
  - これができれば、あとは円盤 1 を B に持ってくるだけ
- 円盤 1 を C → B

ハノイの塔（パズル）

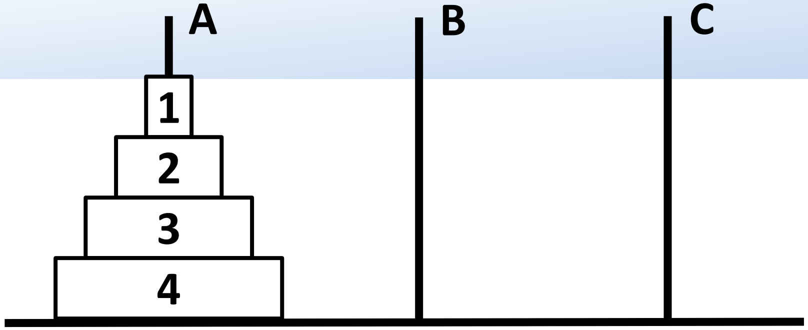


一般の n の場合

目標: 円盤 1 ～ n を A → B

- 円盤 1 ～ n-1 を A → C
  - どう移動させるかは、再帰ですね
- 円盤 n を A → B
  - これは、1回の操作でできます
- 円盤 1 ～ n-1 を C → B
  - どう移動させるかは、これまた再帰ですね

ハノイの塔（パズル）



一般の n の場合

目標: 円盤 1 ～ n を A → B

円盤 1 ～ n を動かしたい

円盤の目的地

円盤が今ある杭

途中で使っても ok な杭

hanoi(n, ’A’ , ’B’ , ’C’)
と呼び出してみる？

- 円盤 1 ～ n-1 を A → C
  - どう移動させるかは、再帰ですね
- 円盤 n を A → B
  - これは、1回の操作でできます
- 円盤 1 ～ n-1 を C → B
  - どう移動させるかは、これまた再帰ですね

hanoi(int k, char x,
char y , char z)
の内容は？

終端条件として、n = 1 の場合を忘れずに

ハノイの塔（パズル）

- 前のページで作ったアルゴリズムに対して、
  その時間計算量（円盤の操作回数）を解析しなさい
  - 円盤が n 枚の時の操作回数を T(n) とする

（おまけ）

- 上記の解析結果を使って、もともとのリュカのパズル（n = 64）の場合には操作回数が何回になるか、確認できるだろうか？

パズル（興味のある人はどうぞ）

- ハノイの塔で、杭が 3 本ではなく 4 本の場合には、
  円盤の操作回数はどのようになるか、求めなさい
- 杭が k 本の場合には、どうなるだろうか？

第8回 再帰的アルゴリズム

- 今日の内容
  - 再帰的アルゴリズム
    (これまでにも再帰は出てきましたが、改めて)
  - 再帰的アルゴリズムと、
    ループを使ったアルゴリズムの対応
  - 再帰的アルゴリズムの時間計算量の解析

- ポイント
  - 再帰は、強力なツールなので、使いこなそう