計算機プログラミングI

第7回　2002年11月28日(木)

- 再帰
  – 帰納的定義
  – 再帰的なメソッド定義
  – 再帰的なメソッドの実行
  – 応用
- 課題の提出方法

なぞなぞ

1, 1, 2, 3, ?

なぞなぞ

1, 1, 2, 3, 5, ?

なぞなぞ

1, 1, 2, 3, 5, 8, ?

なぞなぞ

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

直前の2つの数の和

なぞなぞ

- $f(0)=1$
- $f(1)=1$
- $f(2)=f(1)+f(0)=2$
- $f(3)=f(2)+f(1)=3$

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

- $f(n)=$?

(ただし$n>1$)

なぞなぞ

- $f(0)=1$
- $f(1)=1$
- $f(2)= f(1)+ f(0)=2$
- $f(3)= f(2)+ f(1)=3$

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

- ベースケースと
- 1つ進めた場合

で定義 ——

帰納的定義

(inductive)

- $f(n)= f(n-1)+ f(n-2)$ (ただし$n>1$)

帰納的定義

関数や処理を定義する方法

- パラメータ x に注目して
  - ベースケース
  - 一つ進めた場合

を定義し、全ての場合を定義する

- 例: フィボナッチ数列

  x (番目のフィボナッチ数) に関する帰納的定義
  - ベースケース: $x=0$, $x=1$ の場合
  - 一つ進めた場合: x番目をx-1, x-2番目で定義

    →x>0 全ての場合に定義される

フィボナッチ数の帰納的定義

$$f(n) = \begin{cases} 1 & (n=0, n=1) \\ f(n\text{-}1) + f(n\text{-}2) & (n>1) \end{cases}$$

全ての自然数nについて定義されている。

- 0,1については定義されている
- 0からkまで定義されているとする(帰納法の仮定)とk+1についても定義されている

再帰的なメソッド定義

$$f(n) = \begin{cases} 1 & (n=0, n=1) \\ f(n\text{-}1) + f(n\text{-}2) & (n>1) \end{cases}$$

帰納的な定義は、再帰的なメソッド呼出によって、ほとんどそのままメソッド定義にできる

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

再帰的なメソッド定義

$$f(n) = \begin{cases} 1 & (n=0, n=1) \\ f(n\text{-}1) + f(n\text{-}2) & (n>1) \end{cases}$$

帰納的な定義は、再帰的なメソッド呼出によって、
ほとんどそのままメソッド定義にできる

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

再帰的なメソッド定義

$$f(n) = \begin{cases} 1 & (n=0, n=1) \\ f(n\text{-}1) + f(n\text{-}2) & (n>1) \end{cases}$$

帰納的な定義は、再帰的なメソッド呼出によって、
ほとんどそのままメソッド定義にできる

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

再帰的なメソッド定義

$$f(n) = \begin{cases} 1 & (n=0, n=1) \\ f(n\text{-}1) + f(n\text{-}2) & (n>1) \end{cases}$$

帰納的な定義は、再帰的なメソッド呼出によって、
ほとんどそのままメソッド定義にできる

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

再帰的なメソッド定義

$$f(n) = \begin{cases} 1 & (n=0, n=1) \\ f(n\text{-}1) + f(n\text{-}2) & (n>1) \end{cases}$$

帰納的な定義は、再帰的なメソッド呼出によって、
ほとんどそのままメソッド定義にできる

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

再帰的なメソッド定義

$$f(n) = \begin{cases} 1 & (n=0, n=1) \\ f(n\text{-}1) + f(n\text{-}2) & (n>1) \end{cases}$$

帰納的な定義は、再帰的なメソッド呼出によって、
ほとんどそのままメソッド定義にできる

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

ほとんどそのまま
対応している

補足

- クラスメソッドの定義
- 返り値の型
- 引数の型
- 引数の値がしまわれる変数名

- クラスメソッドの呼出(Fibonacci.f(n-1)と同じ同一クラス中ではクラス名が省略できる)
- 計算結果を返す文

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

```
public class Fibonacci {
  …
  public static int f(int n) {
    if (n == 0 || n == 1) {
      return 1;
    } else {
      return Fibonacci.f(n-1) + Fibon
    }
  }
  …
}
```

補足

- クラスメソッドの定義
- 返り値の型
- 引数の型
- 引数の値がしまわれる変数名

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

- クラスメソッドの呼出
  (Fibonacci.f(n-1)と同じ
  同一クラス中では
  クラス名が省略できる)
- 計算結果を返す文

```
public class Fibonacci {
  ...
  public static int f(int n) {
    if (n == 0 || n == 1) {
      return 1;
    } else {
      return Fibonacci.f(n-1) + Fibon
    }
  }
  ...
}
```

補足

- クラスメソッドの定義
- 返り値の型
- 引数の型
- 引数の値がしまわれる変数名

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

- クラスメソッドの呼出 (Fibonacci.f(n-1)と同じ 同一クラス中では クラス名が省略できる)
- 計算結果を返す文

```
public class Fibonacci {
  ...
  public static int f(int n) {
    if (n == 0 || n == 1) {
      return 1;
    } else {
      return Fibonacci.f(n-1) + Fibon
    }
  }
  ...
}
```

補足

- クラスメソッドの定義
- 返り値の型
- 引数の型
- 引数の値がしまわれる変数名

- クラスメソッドの呼出
(Fibonacci.f(n-1)と同じ
同一クラス中では
クラス名が省略できる)
- 計算結果を返す文

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

```
public class Fibonacci {
  ...
  public static int f(int n) {
    if (n == 0 || n == 1) {
      return 1;
    } else {
      return Fibonacci.f(n-1) + Fibon
    }
  }
  ...
}
```

補足

- クラスメソッドの定義
- 返り値の型
- 引数の型
- 引数の値がしまわれる変数名

- クラスメソッドの呼出(Fibonacci.f(n-1)と同じ
  同一クラス中では
  クラス名が省略できる)
- 計算結果を返す文

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

```
public class Fibonacci {
  ...
  public static int f(int n) {
    if (n == 0 || n == 1) {
      return 1;
    } else {
      return Fibonacci.f(n-1) + Fibon
    }
  }
  ...
}
```

補足

- クラスメソッドの定義
- 返り値の型
- 引数の型
- 引数の値がしまわれる変数名

- クラスメソッドの呼出 (Fibonacci.f(n-1)と同じ 同一クラス中では クラス名が省略できる)
- 計算結果を返す文

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

```
public class Fibonacci {
  …
  public static int f(int n) {
    if (n == 0 || n == 1) {
      return 1;
    } else {
      return Fibonacci.f(n-1) + Fibon
    }
  }
  …
}
```

補足

- クラスメソッドの定義
- 返り値の型
- 引数の型
- 引数の値がしまわれる変数名

- クラスメソッドの呼出 (Fibonacci.f(n-1)と同じ 同一クラス中では クラス名が省略できる)
- 計算結果を返す文

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

```
public class Fibonacci {
  ...
  public static int f(int n) {
    if (n == 0 || n == 1) {
      return 1;
    } else {
      return Fibonacci.f(n-1) + Fibon
    }
  }
  ...
}
```

補足

再帰的なメソッド呼出:
同じメソッドを呼出している
こと以外は変わらない

- クラスメソッドの定義
- 返り値の型
- 引数の型
- 引数の値がしまわれる変数名

- クラスメソッドの呼出(Fibonacci.f(n-1)と同じ
  同一クラス中では
  クラス名が省略できる)
- 計算結果を返す文

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

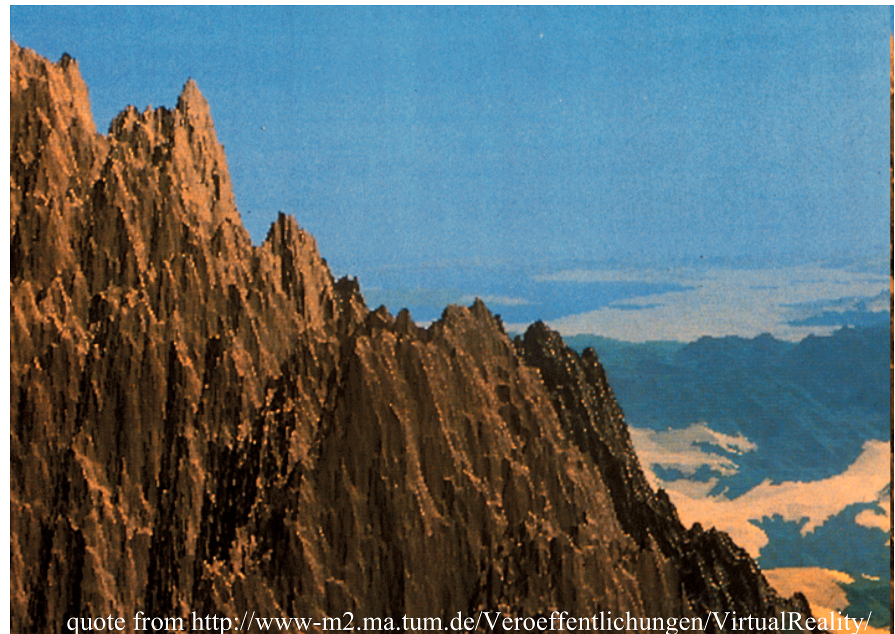
```
public class Fibonacci {
  ...
  public static int f(int n) {
    if (n == 0 || n == 1) {
      return 1;
    } else {
      return Fibonacci.f(n-1) + Fibona
    }
  }
  ...
}
```

練習

- 7-1*: 冪乗と同じような形式で書けばよい
- 7-2:
- 7-3*: 難しい場合は、まず2の30乗だけ
  (参考: 第6回のSqrtクラス)
- 7-4:

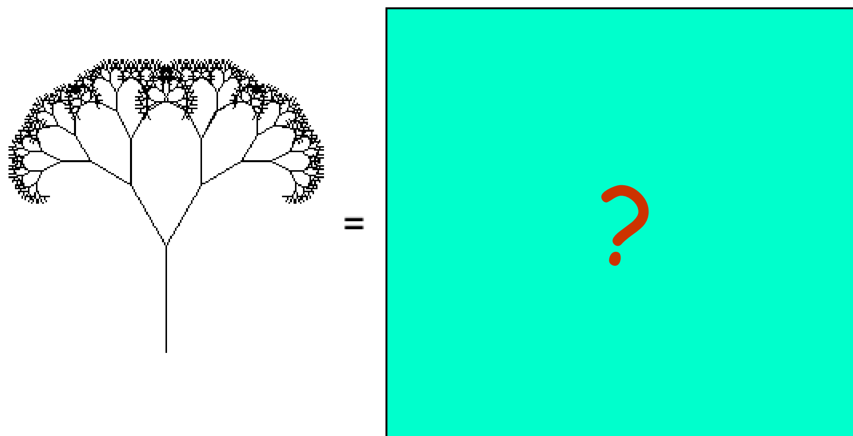
フラクタル図形

- 自己相似形とも言う
- 自然界の形状によく見られる
- 木の形・山の形・・・


quote from http://www-m2.ma.tum.de/Veroeffentlichungen/VirtualReality/

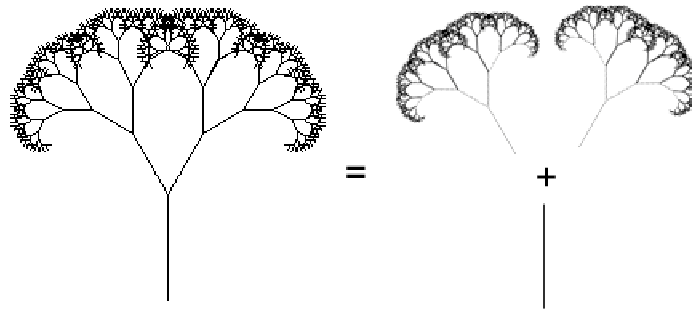
フラクタル図形による木

木 = ?



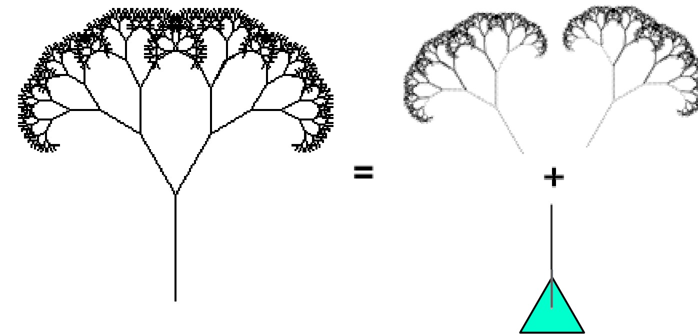
フラクタル図形による木

木 = 幹 + (小さく傾いた木) × 2



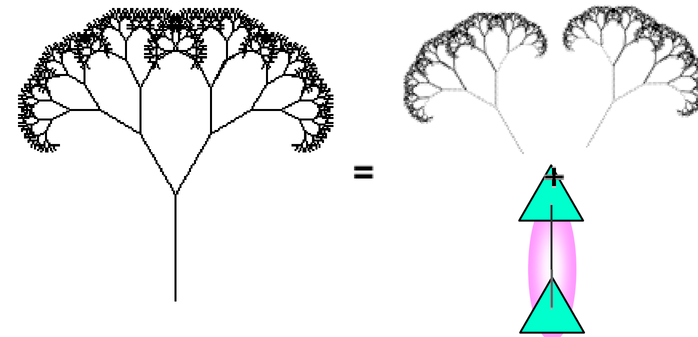
タートルグラフィクスによる木

- 高さ*l*の木を描く:
  - l/3前進する
  - 30度左に回転
  - 高さ2l/3の木を描く
  - 60度右に回転
  - 高さ2l/3の木を描く
  - 30度左に回転
  - l/3後退する



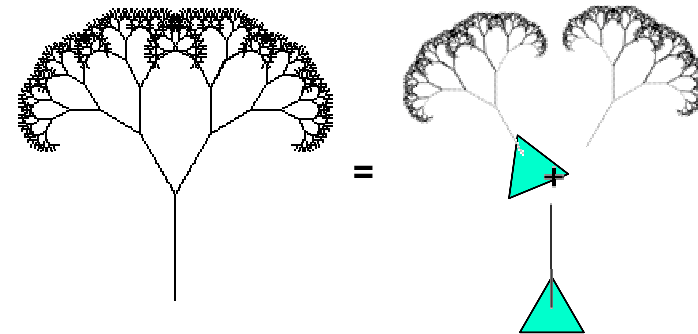
タートルグラフィクスによる木

- 高さ*l*の木を描く:
  - 1/3前進する
  - 30度左に回転
  - 高さ2l/3の木を描く
  - 60度右に回転
  - 高さ2l/3の木を描く
  - 30度左に回転
  - 1/3後退する



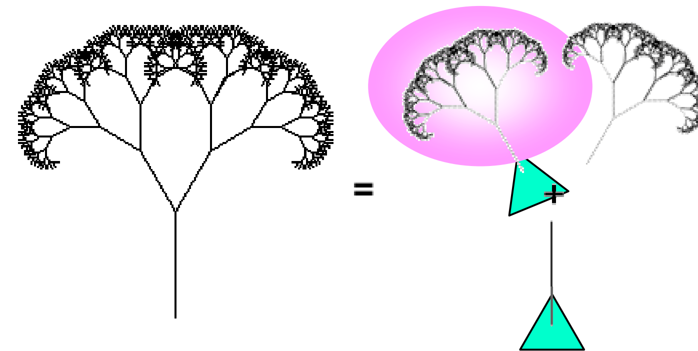
タートルグラフィクスによる木

- 高さ*l*の木を描く:
  - 1/3前進する
  - 30度左に回転
  - 高さ2l/3の木を描く
  - 60度右に回転
  - 高さ2l/3の木を描く
  - 30度左に回転
  - 1/3後退する



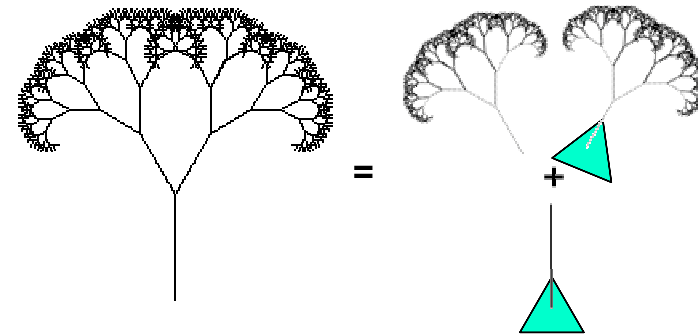
タートルグラフィクスによる木

- 高さ*l*の木を描く:
  - l/3前進する
  - 30度左に回転
  - 高さ2l/3の木を描く
  - 60度右に回転
  - 高さ2l/3の木を描く
  - 30度左に回転
  - l/3後退する



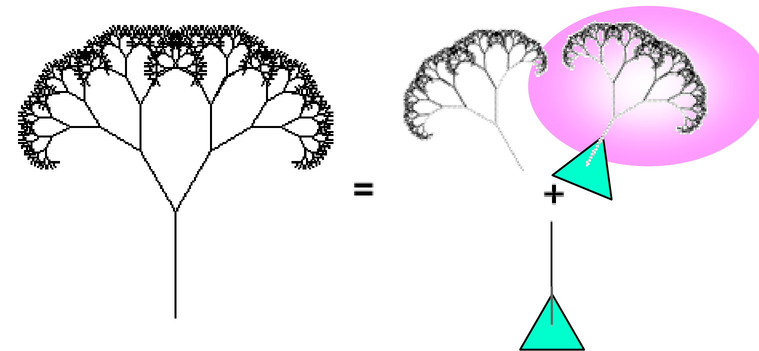
タートルグラフィクスによる木

- 高さ*l*の木を描く:
  - l/3前進する
  - 30度左に回転
  - 高さ2l/3の木を描く
  - 60度右に回転
  - 高さ2l/3の木を描く
  - 30度左に回転
  - l/3後退する



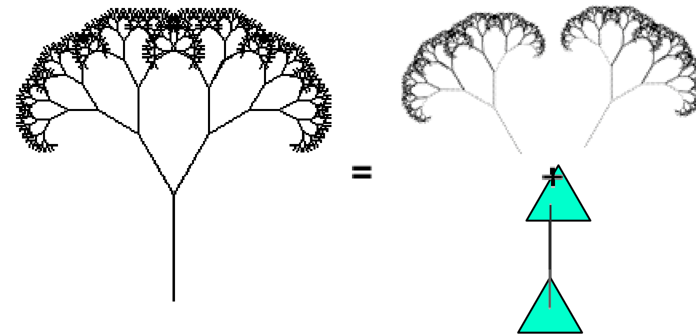
タートルグラフィクスによる木

- 高さ*l*の木を描く:
  - l/3前進する
  - 30度左に回転
  - 高さ2l/3の木を描く
  - 60度右に回転
  - 高さ2l/3の木を描く
  - 30度左に回転
  - l/3後退する



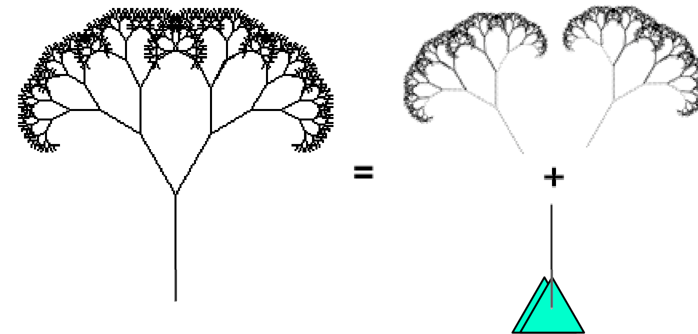
タートルグラフィクスによる木

- 高さ*l*の木を描く:
  - 1/3前進する
  - 30度左に回転
  - 高さ2l/3の木を描く
  - 60度右に回転
  - 高さ2l/3の木を描く
  - 30度左に回転
  - 1/3後退する



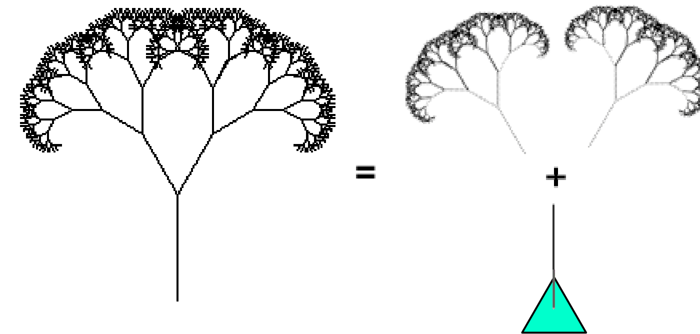
タートルグラフィクスによる木

- 高さ*l*の木を描く:
  - l/3前進する
  - 30度左に回転
  - 高さ2l/3の木を描く
  - 60度右に回転
  - 高さ2l/3の木を描く
  - 30度左に回転
  - l/3後退する



タートルグラフィクスによる木

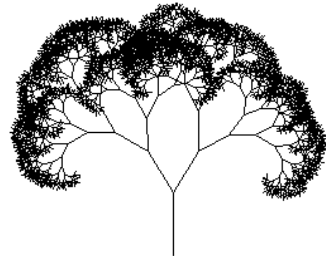
- 高さ*l*の木を描く:
  - 1/3前進する
  - 30度左に回転
  - 高さ2l/3の木を描く
  - 60度右に回転
  - 高さ2l/3の木を描く
  - 30度左に回転
  - 1/3後退する

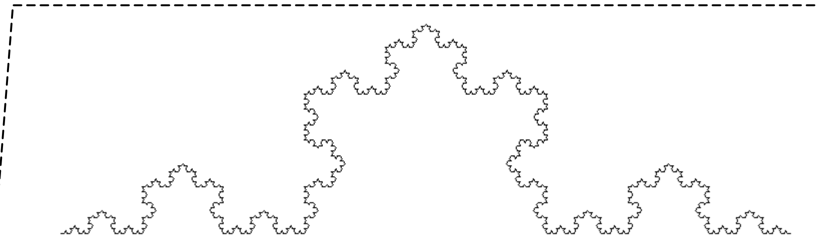


再帰的なメソッド呼出

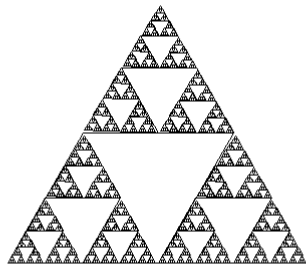
練習

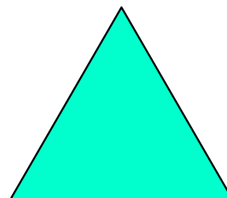
- 7-5* Tree.javaをダウンロード
  drawTree(m, 長さ); —再帰的な呼出し
- 7-6 (b)



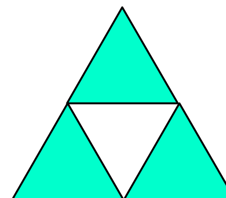


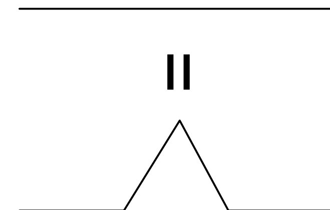
- 7-7





=





再帰的メソッドの実行

- f(2) が実行される:
  途中でf(1)とf(0)が実行される
- 呼出す側・呼出される側の実行は全く別のもの

f(2)を呼び出し

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

実行1: n=2

f(n-1)つまりf(1)を呼出し

値と場所を覚える

再帰的メソッドの実行

- f(2) が実行される:
  途中でf(1)とf(0)が実行される
- 呼出す側・呼出される側の実行は全く別のもの

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

実行1: n=2 ここを実行中

return 1 で終了

再帰的メソッドの実行

- f(2) が実行される:
  途中でf(1)とf(0)が実行される
- 呼出す側・呼出される側の実行は全く別のもの

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

実行1: n=2　ここを実行中

実行2: n=1

return 1 で終了

再帰的メソッドの実行

- f(2) が実行される:
  途中でf(1)とf(0)が実行される
- 呼出す側・呼出される側の実行は全く別のもの

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

実行1: n=2　ここを実行中

実行2: n=1

return 1 で終了

場所と値を思い出す

再帰的メソッドの実行

- f(2) が実行される:
  途中でf(1)とf(0)が実行される
- 呼出す側・呼出される側の実行は全く別のもの

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

実行1: n=2 この次から実行再開

f(n-1)の結果=1

f(n-2)つまりf(0)を実行

値と場所を覚える

再帰的メソッドの実行

- f(2) が実行される:
  途中でf(1)とf(0)が実行される
- 呼出す側・呼出される側の実行は全く別のもの

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

実行1: n=2 これの結果は1
ここを実行中

return 1 で終了

再帰的メソッドの実行

- f(2) が実行される:
  途中でf(1)とf(0)が実行される
- 呼出す側・呼出される側の実行は全く別のもの

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

実行1: n=2　これの結果は1
ここを実行中

実行3: n=0

return 1 で終了

再帰的メソッドの実行

- f(2) が実行される:
  途中でf(1)とf(0)が実行される
- 呼出す側・呼出される側の実行は全く別のもの

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

実行1: n=2　これの結果は1
ここを実行中

実行3: n=0

return 1 で終了

場所と値を思い出す

再帰的メソッドの実行

- f(2) が実行される:
  途中でf(1)とf(0)が実行される
- 呼出す側・呼出される側の実行は全く別のもの

```
public static int f(int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return f(n-1) + f(n-2);
  }
}
```

実行1: n=2　これの結果は1
ここから再開

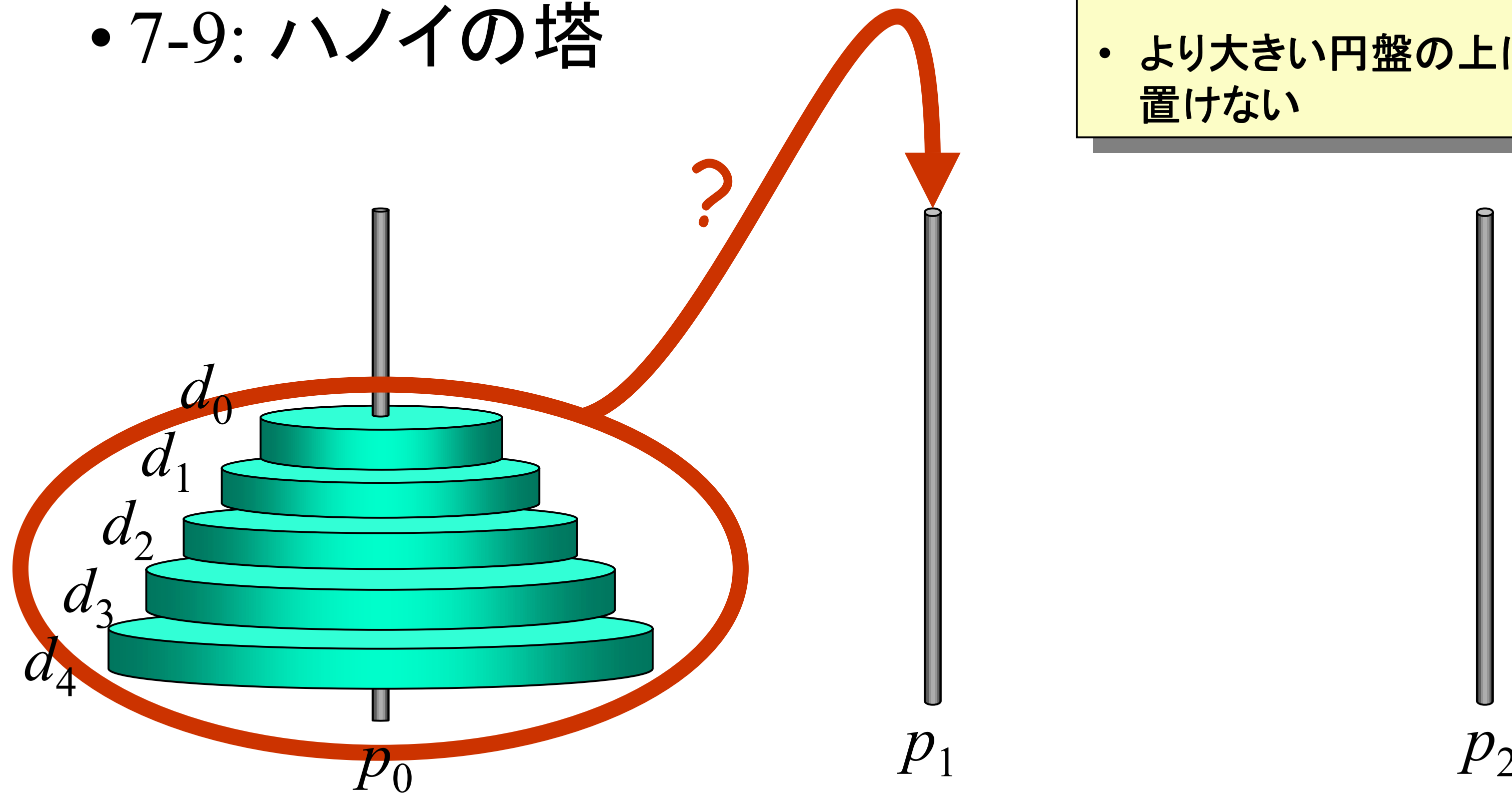
f(n-2)の結果=1

1+1を計算して終了

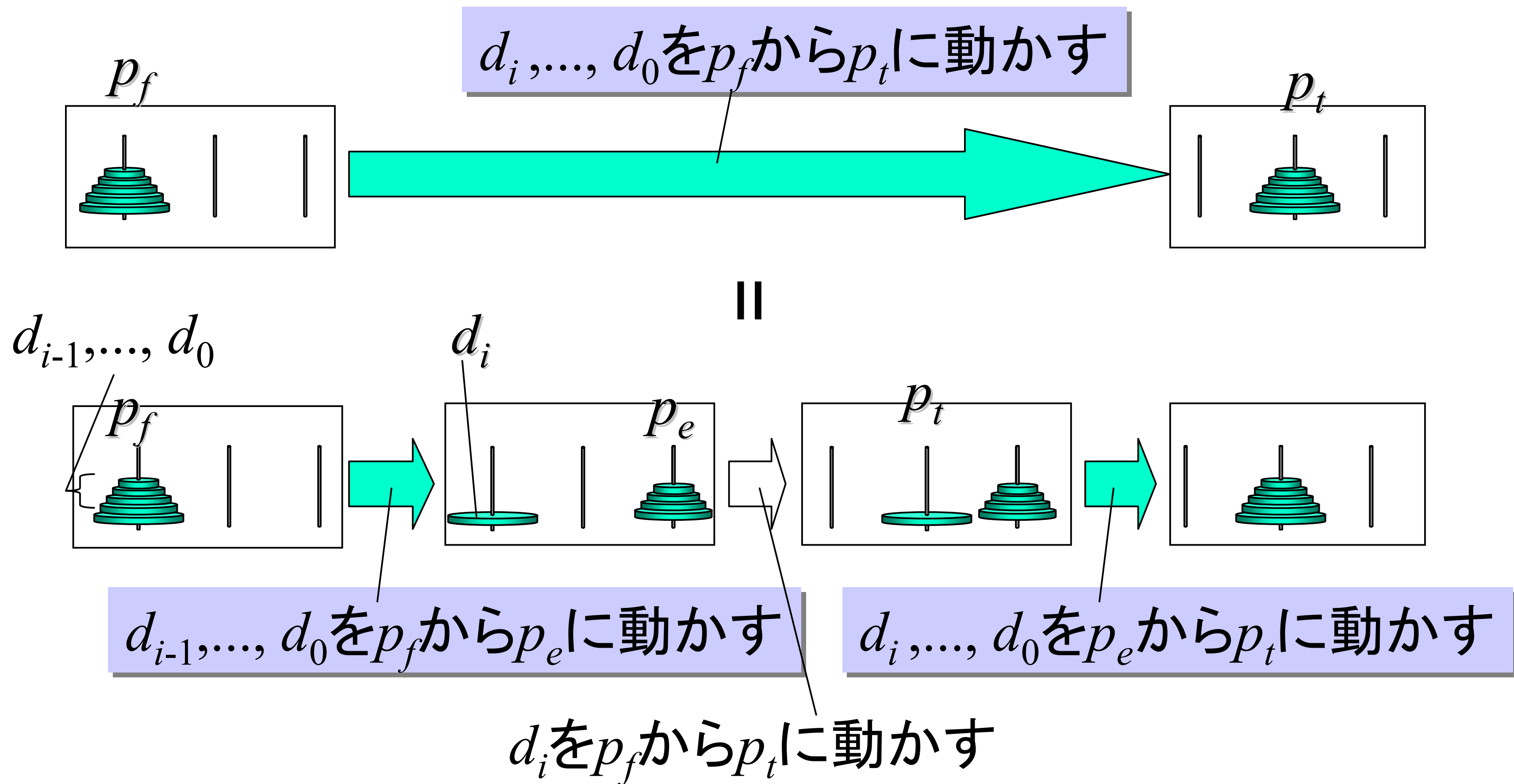
再帰的メソッドの応用

- 7-8: 冪乗
- 7-9: ハノイの塔

- 一番上の円盤のみ動かせる
- より大きい円盤の上にしか置けない



ハノイの塔



課題レポートの提出方法