テキスト処理 第4回 (2007-05-15)

田中哲
産業技術総合研究所
情報技術研究部門
akr@isc.senshu-u.ac.jp
http://staff.aist.go.jp/tanaka-akira/textprocess-2007/

今日の内容

- 端末とコマンドライン
- メソッド呼び出しのしくみ
- 再帰
  - 階乗
  - フィボナッチ数
  - クイックソート
  - 木構造
- レポート

端末とコマンドライン

- GUI でなく、文字だけでコンピュータと対話する方法がある
- キーボードからコマンドを入力して画面に結果を出力する
- GUI 環境で、そういう方法を提供するソフトウェアを端末エミュレータという
  - Windows のコマンドプロンプト
  - Unix (X Window System) の xterm
- エミュレータではない本物の端末は現在ではまず使われない

VT100

(Unix初期によく使われた端末)

端末での対話

- プロンプトの後にコマンドラインを入力する
- コマンドが実行される
- コマンドが終わるとまたプロンプトが出る

- この対話を行うプログラムをシェルという
  - sh
  - csh
  - zsh

```
prompt% コマンドライン入力
コマンドの出力
prompt% 
```

コマンドライン

- コマンドラインはコマンド名と引数からなる

  prompt% コマンド名 引数1 引数2 ...

  例:
  prompt% egrep foo filename

- 出力をファイルに向けたり、複数のコマンドを接続するパイプなどの機能もある

シングルクォート

- 引数に空白を含めるには引用符で括る
- パイプとかと解釈されないようにするときにも使う

- egrep 'foo bar' filename

  引数1

  f, o, o, 空白, b, a, r の 7文字

- シングルクォート自身は中に入れられない
- この規則は Unix のシェルの規則

ダブルクォート

- 引数に空白を含めるには引用符で括る
- パイプとかと解釈されないようにするときにも使う

- egrep "foo bar" filename

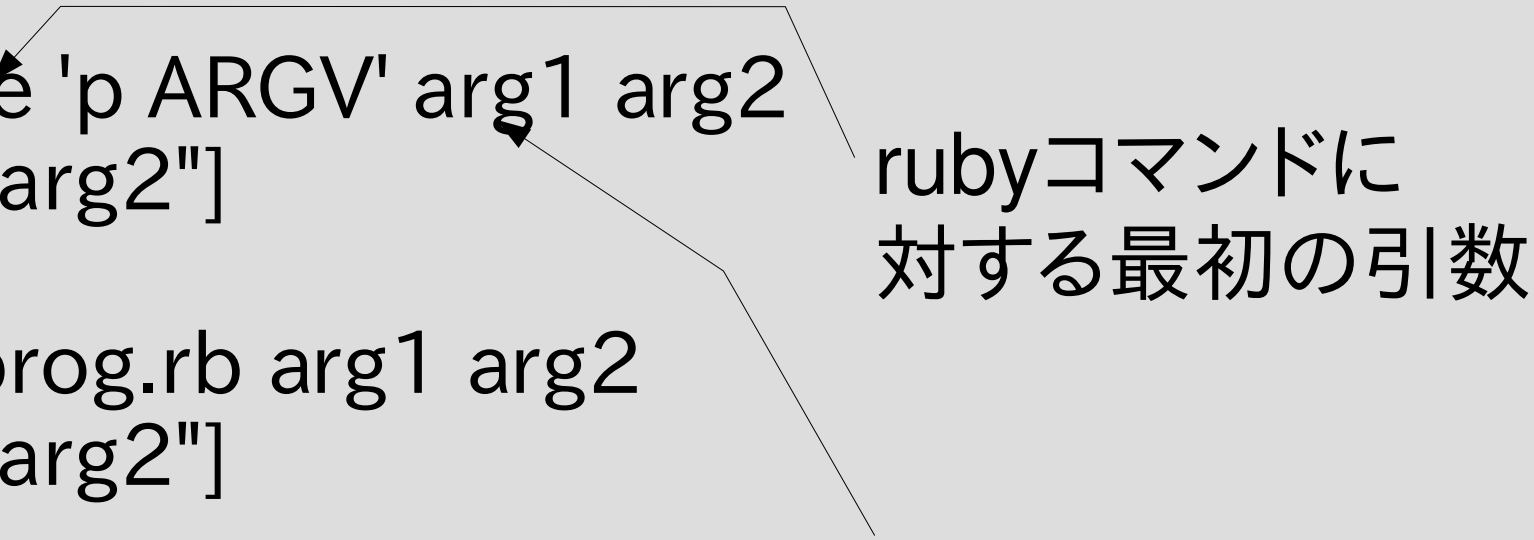
  引数1

  f, o, o, 空白, b, a, r の 7文字

- $, `, ", ￥を入れるときは￥を前置する
- たとえば " 自身を入れるときは "...¥"..." とする
- この規則は Unix のシェルの規則

Ruby とコマンドライン引数

- コマンドラインの引数のうち Ruby でかかれたプログラムの指定を除いた残りが ARGV になる
- % ruby -e 'p ARGV' arg1 arg2
  ["arg1", "arg2"]
- % ruby prog.rb arg1 arg2
  ["arg1", "arg2"]



メソッド呼び出し

- 定義

```
def func(arg1, arg2, ...)
  式1
  式2
  ...
  式n
end
```

引数1の値がarg1,
引数2の値がarg2, ...
となった状態で
式1, 式2, ... が順に実行される

- 呼び出し

```
func(引数1, 引数2, ...)
```

メソッドの返り値

- def func(arg1, arg2, ...)
  式1
  式2
  ...
  式n
  end

最後の式の値が返値になる

- def func(...)
  return 式 if ...
  ...
  end

途中で値を返したいときは return も使える

メソッド呼び出しのしくみ

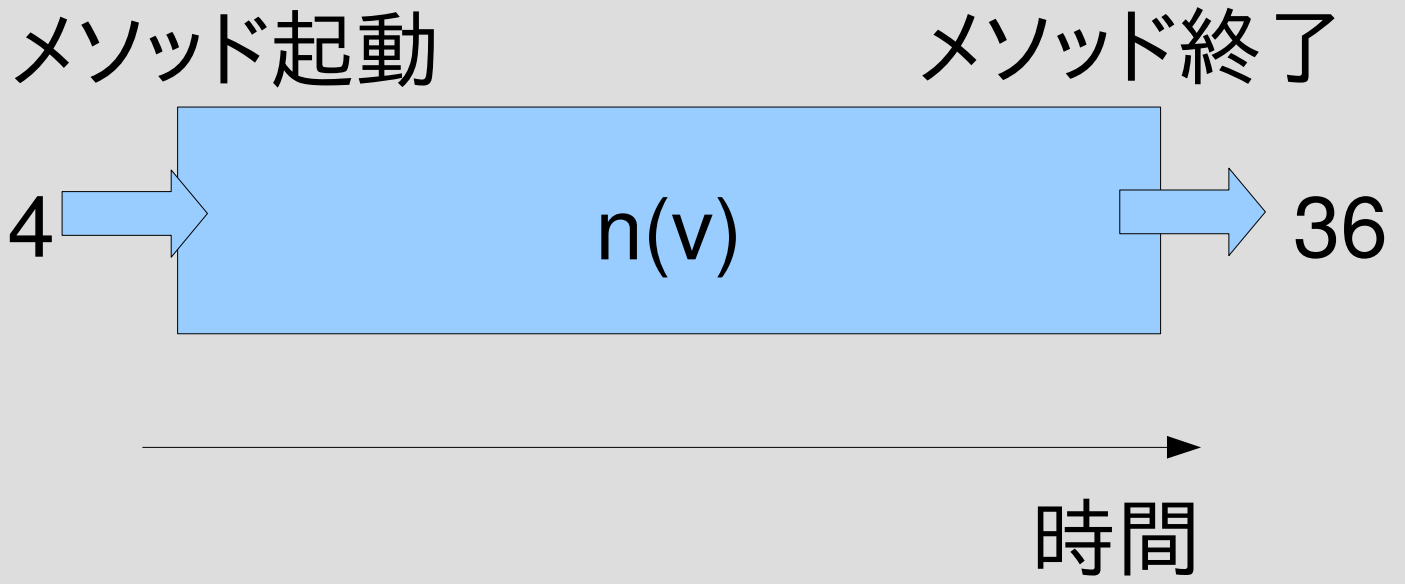
- def m(v)
  v + 1
  end
  def n(v)
  m(v*2)*v
  end

- p n(4) #=> 36

- n(4)=
  m(4*2)*4=
  m(8)*4=
  (8+1)*4=
  9*4=
  36

メソッド呼び出しのしくみ

- def m(v)
  v + 1
  end
  def n(v)
  m(v*2)*v
  end

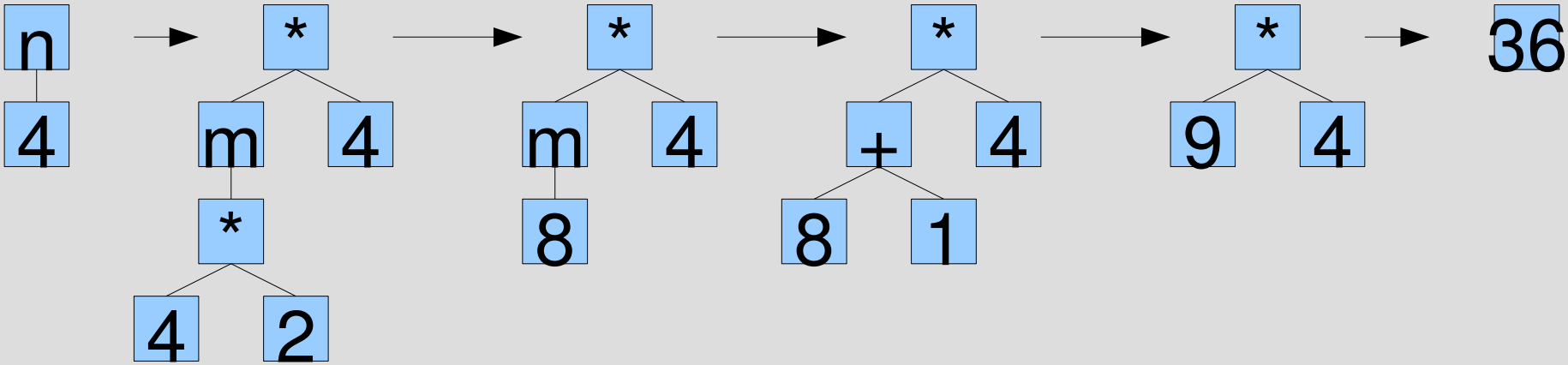


n(v) の中身は実際にはどう動作するのか？

- p n(4) #=> 36

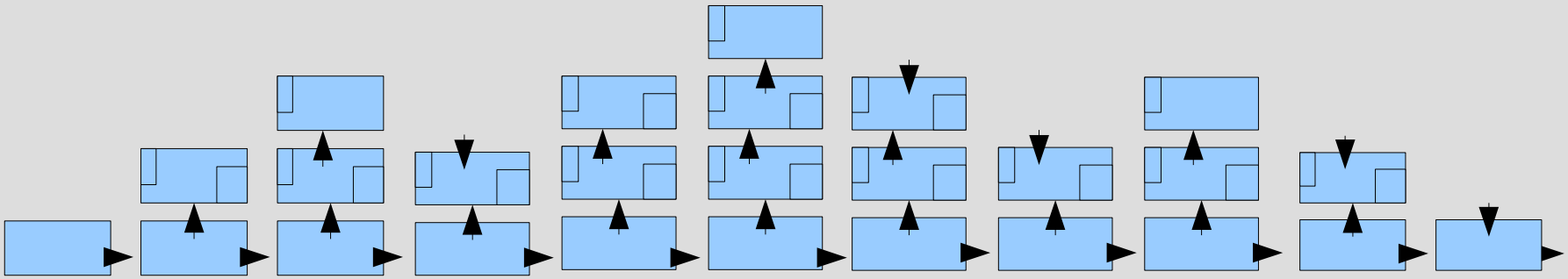
n(4)の計算: 項書換え

- 案1: 式をデータ構造で表現して変形していく
  - コンピュータ上には木構造を表現できる
  - 式の変形のとおりに木構造を書き換えていく
  - そういう言語もある: Haskell とか
  - でも、Ruby を含む多くの言語は直接にはそうしない
  - C も Java も Python も Perl も PHP も

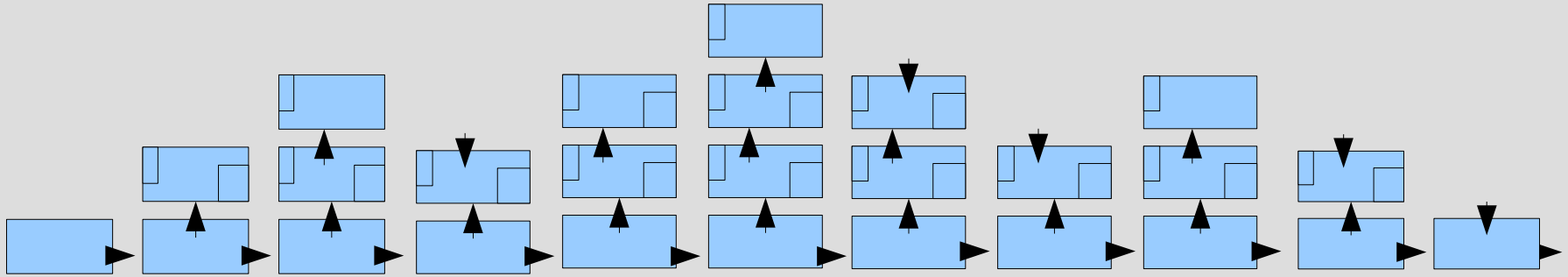


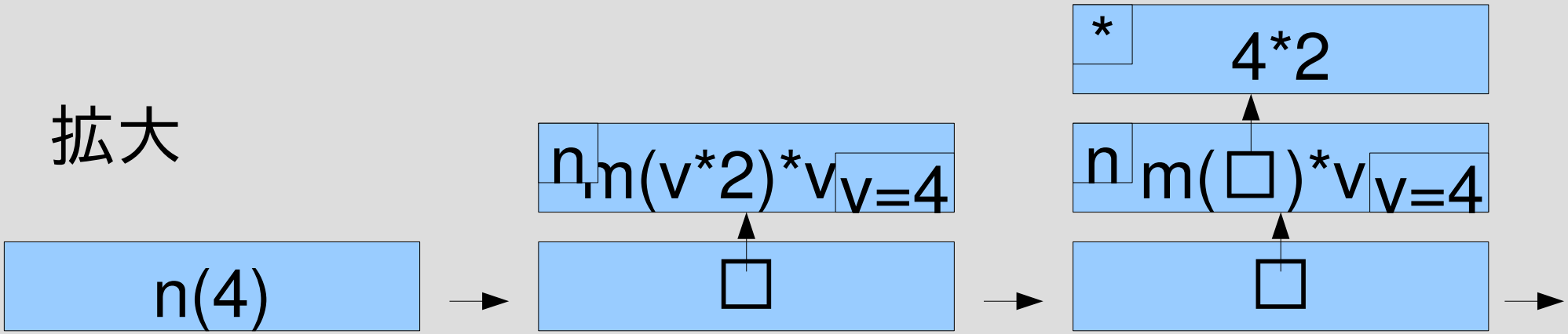
n(4) の計算: スタック

- 案2: スタックを使って計算する
  - メソッドが呼び出されるたびにスタックフレームというメモリを確保してスタックにプッシュする
  - スタックフレームに戻り先とローカル変数を記録する
  - メソッドが終わったらその戻り先に戻ってスタックフレームをスタックからポップする
  - こういう最後に入れたものを最初に使うデータ構造を一般にスタックという
  - メソッド呼び出し用のスタックは制御スタックなどと呼ぶこともあるが、ここでは単にスタックと呼ぶ

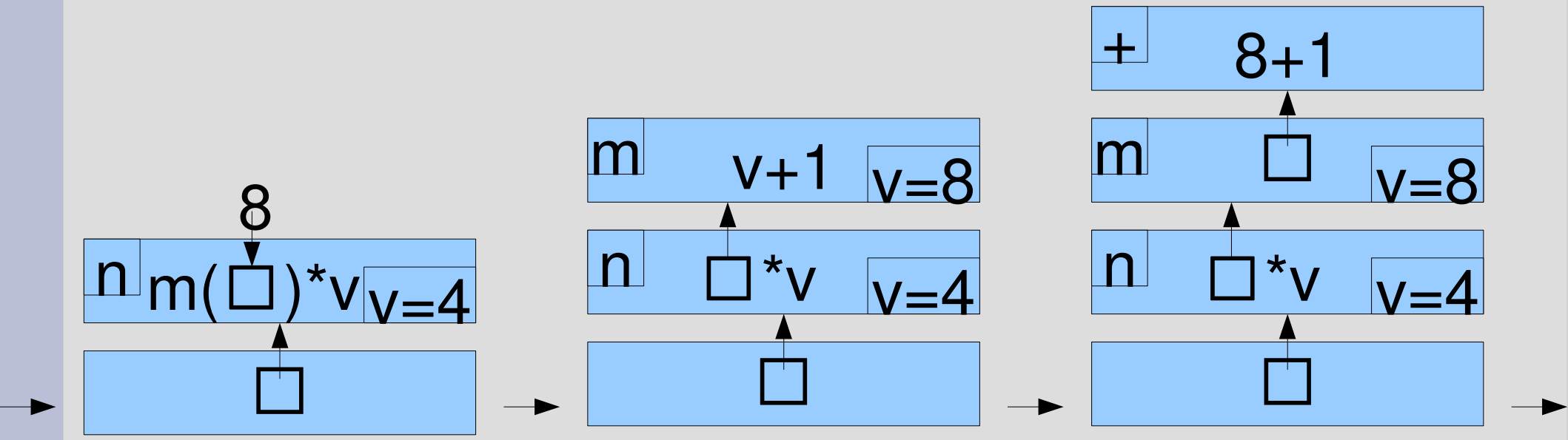


スタックの動作 (1)

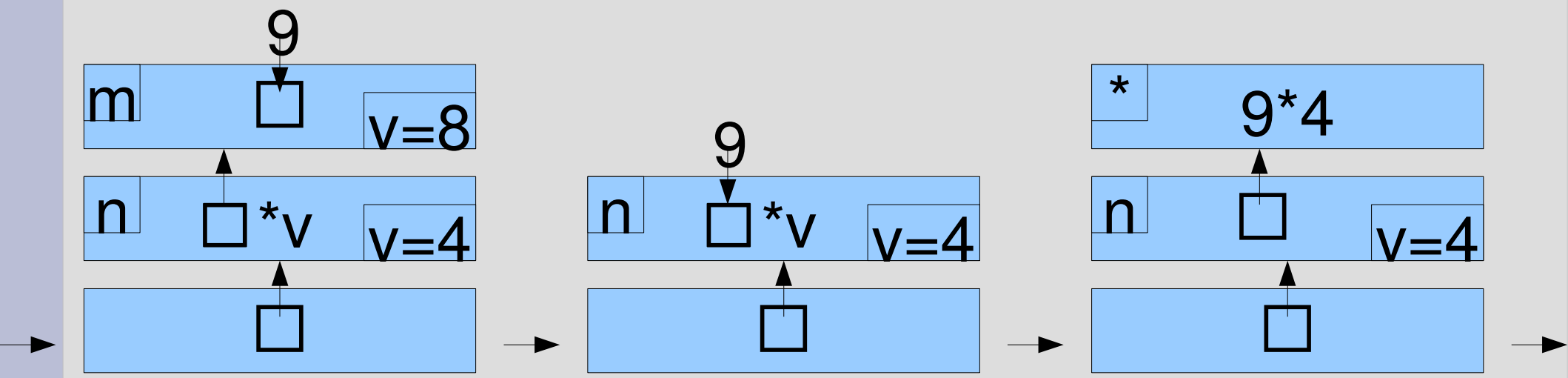




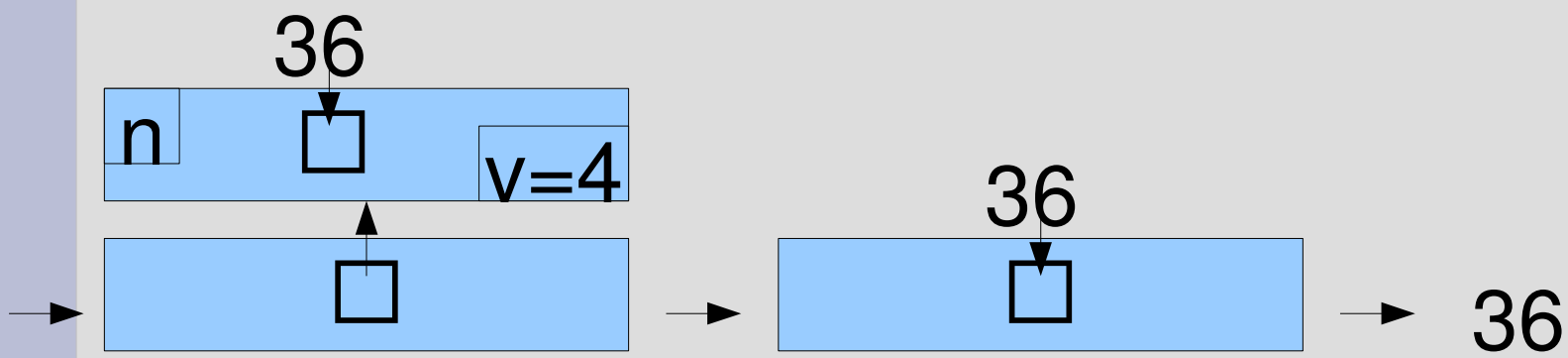
スタックの動作 (2)



スタックの動作 (3)

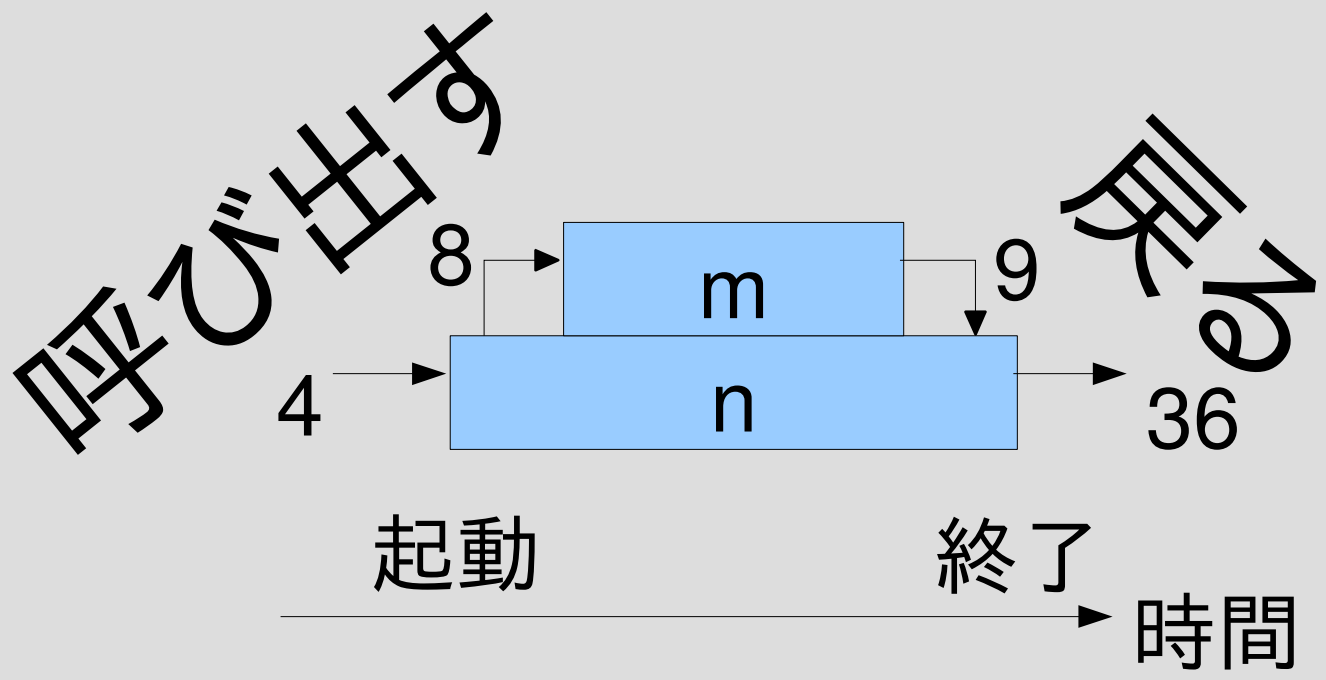


スタックの動作 (4)



おおざっぱな呼び出しの時系列

- def m(v)
  v + 1
  end
  def n(v)
  m(v*2)*v
  end



- p n(4) #=> 36

再帰

- ある関数がその関数自身を呼ぶこと

階乗(factorial)

- n! = 1*2*3*...*(n-2)*(n-1)*n

- 0! = 1　　(0! は便宜的にこう決める)
- 1! = 1
- 2! = 1*2 = 2
- 3! = 1*2*3 = 6
- 4! = 1*2*3*4 = 24
- 5! = 1*2*3*4*5 = 120
- 6! = 1*2*3*4*5*6 = 720
- ...

階乗: 数学版

- 帰納的定義
  - 0! = 1　　基底段階
  - n! = n(n-1)!　帰納段階

- 0! は定義の基底段階により 1 と決まっている
- 1! は帰納段階より 1*(1-1)!=1*0! でつまり 1*1=1
- 2! は帰納段階より 2*(2-1)!=2*1! でつまり 2*1=2
- 3! は帰納段階より 3*(3-1)!=3*2! でつまり 3*2=6
- 4! は帰納段階より 4*(4-1)!=4*3! でつまり 4*6=24
- 5! は帰納段階より 5*(5-1)!=5*4! でつまり 5*24=120
- 以下同様にして非負整数nに対し n! が決定される

階乗: 非再帰版

- ループによる実装例
- def fact(n)

```
  ret=1
  1.upto(n) {|i|
    ret *= i
  }
  ret
end
```

階乗: 再帰版

- 再帰を使った実装例
- def fact(n)
  if n == 0
  1
  else
  n * fact(n-1)
  end
  end

- fact(m) は fact(m-1) を呼ぶ
- fact(m-1) は fact(m-2) を呼ぶ
- ...
- fact(2) は fact(1) を呼ぶ
- fact(1) は fact(0) を呼ぶ
- fact(0) は 1 を返す
- fact(1) は 1*1 を返す
- fact(2) は 2*1*1 を返す
- ...
- fact(m) は m*...*1 を返す

帰納的定義と再帰的実装

- 帰納的定義

$0! = 1$

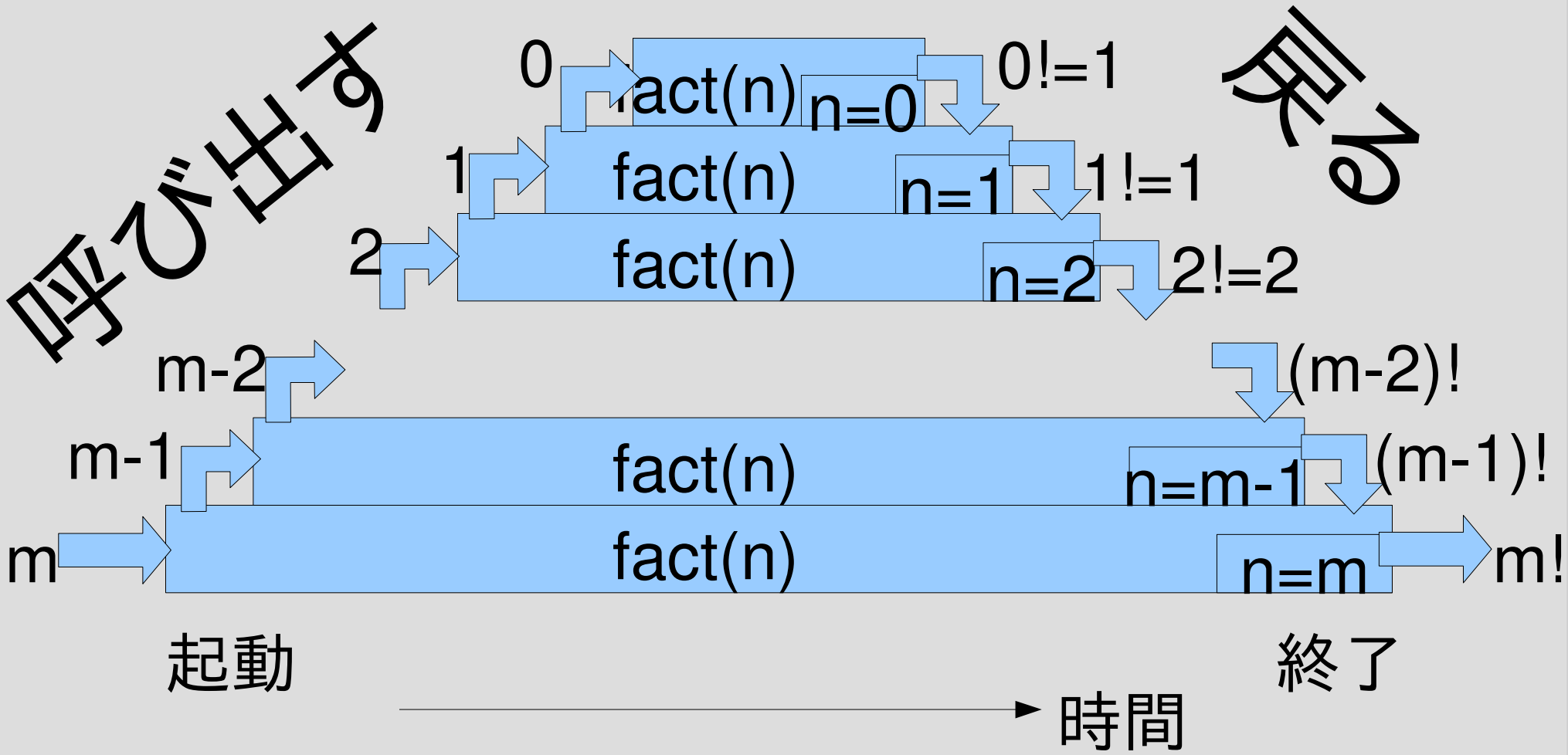
$n! = n(n-1)!$

- 再帰を使った実装例

```
def fact(n)
  if n == 0
    1
  else
    n * fact(n-1)
  end
end
```

帰納的定義と再帰的実装は同じ構造
数学的な記述をプログラムで表現するときに再帰は便利

呼び出しの時系列


呼び出す
戻る
0
fact(n)
n=0
0!=1
1
fact(n)
n=1
1!=1
2
fact(n)
n=2
2!=2
m-2
(m-2)!
m-1
fact(n)
n=m-1
(m-1)!
m
fact(n)
n=m
m!
起動
終了
時間

フィボナッチ数

- F(1) = 1
- F(2) = 1
- F(n+2) = F(n) + F(n+1)

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, ...

- 再帰で素朴に書くと遅いのでベンチマークによく使われる
- 再帰を使わずに書くのも難しくない

フィボナッチ数の実装

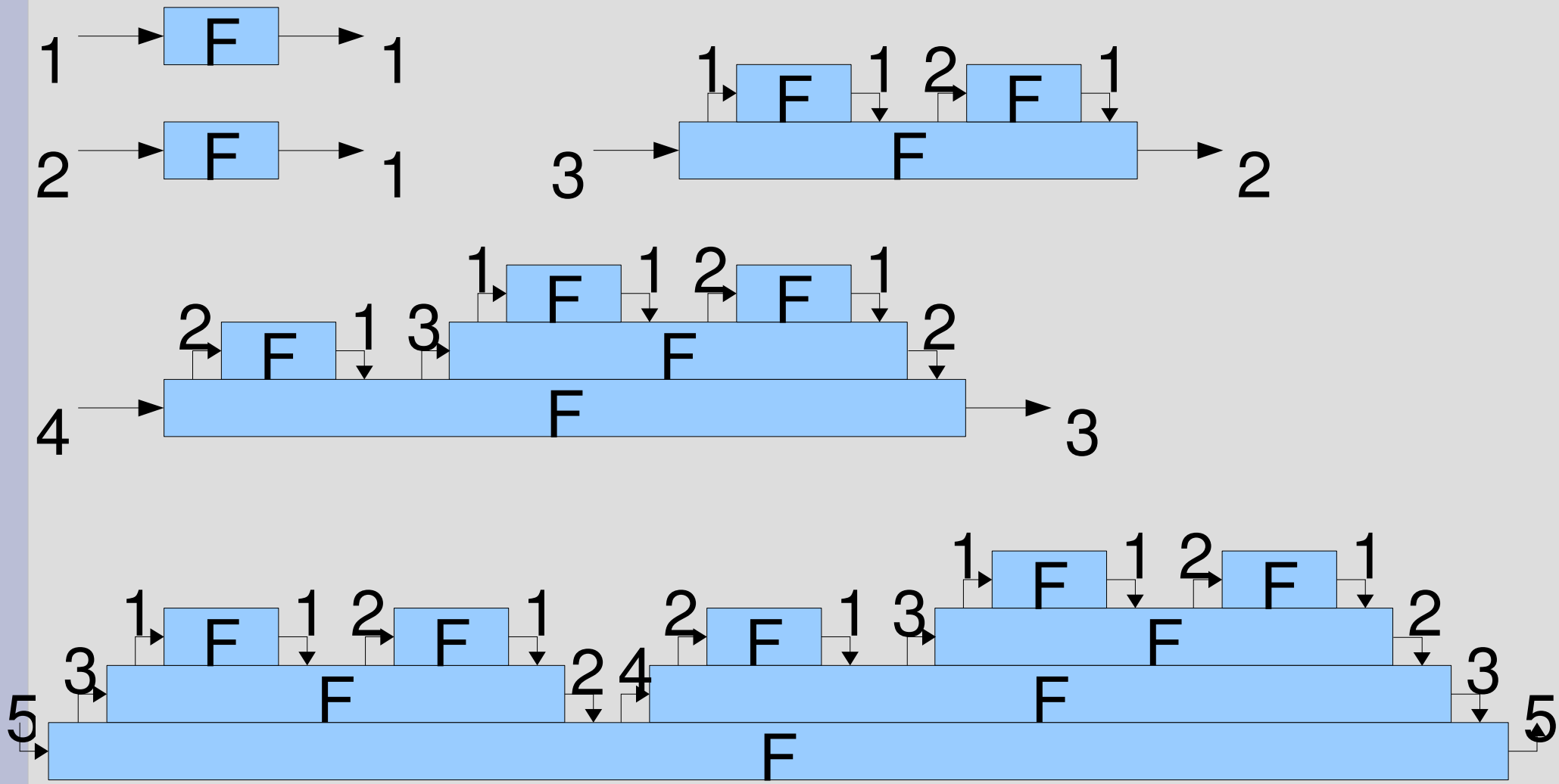
- 帰納的定義

  $F(1) = F(2) = 1$

  $F(n+2) = F(n) + F(n+1)$

- ```
  def fib(n)
    if n <= 2
      1
    else
      fib(n-2) + fib(n-1)
    end
  end
  ```
  ```

フィボナッチの呼び出しの時系列



ソート

- ソート: 配列の要素を順番に並べ替える
  [16, 5, 13, 29, 15, 21, 17, 12, 18, 6]

  ⇒[5, 6, 12, 13, 15, 16, 17, 18, 21, 29]
- さまざまなアルゴリズムがある
  - 選択ソート
  - 挿入ソート
  - バブルソート
  - クイックソート　速くて実用的で再帰を使う
  - ヒープソート
  - マージソート
  - etc.

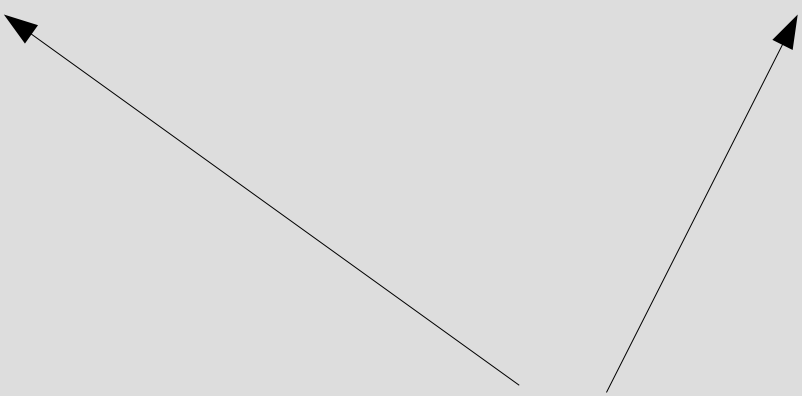
クイックソートのアルゴリズム

- 配列の長さが 1以下であればそれはすでに順番になっているのでおしまい
- そうでなければ配列から適当にひとつ要素を取り出す (pivot と呼ぶ)
- 配列の残りを pivot よりも小さい要素と大きい要素に分割する
- 分割したそれぞれについて再帰的にクイックソートを行う

- 平均 $O(n \log n)$、最悪 $O(n^2)$
- 再帰を使わずに書くのは難しい

クイックソートの実装

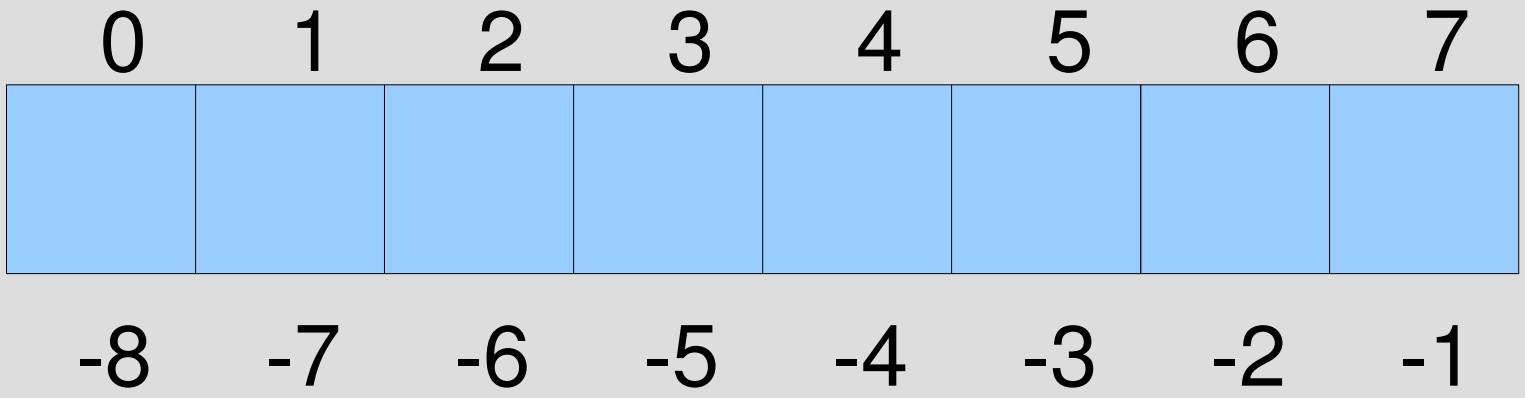
```
def qsort(ary)
  return ary if ary.length <= 1
  pivot = ary[0]
  smaller, bigger = ary[1..-1].partition {|v| v < pivot }
  qsort(smaller) + [pivot] + qsort(bigger)
end
```



再帰: qsort が qsort を呼んでいる

ary[1..-1]

- ary[m..n] : 配列の部分配列を取り出す
- ary[m] から ary[n] まで (inclusive : 両端を含む)
  ["a", "b", "c", "d"][1..2] #=> ["b", "c"]
- m, n には -len から -1 までも使用できる
  これは右端からの位置を表す



- ary[1..-1] は最初の要素を除いたそれ以降

クイックソートの ary[1..-1]

```
def qsort(ary)
  return ary if ary.length <= 1
  pivot = ary[0]
  smaller, bigger = ary[1..-1].partition {|v| v < pivot }
  qsort(smaller) + [pivot] + qsort(bigger)
end
```

Array#partition {|elt| condition }

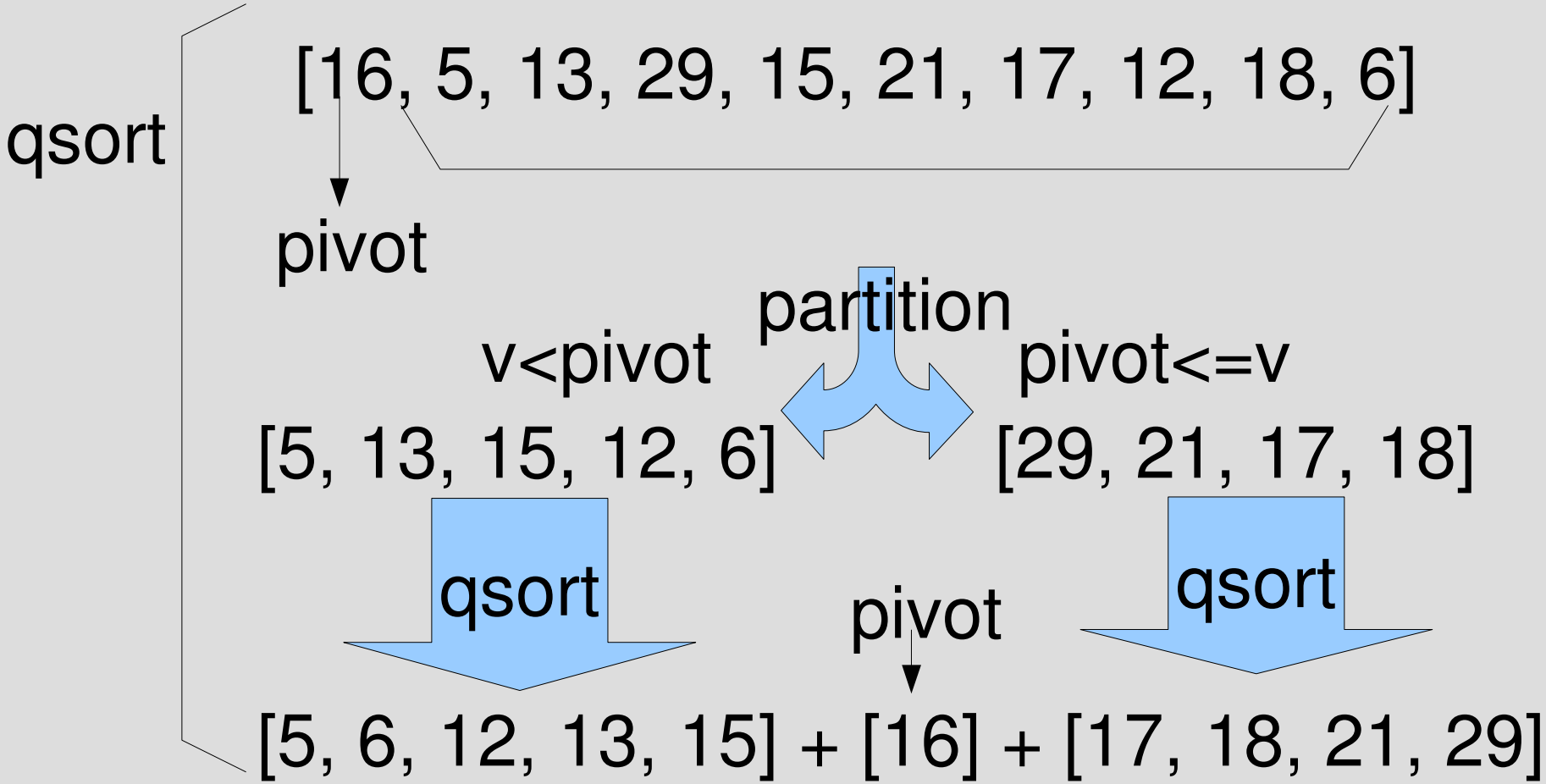
- 配列の要素を条件を満たすかどうかで分類する
- 条件はブロックで指定する
- 返り値は 2要素の配列で、
  最初のが条件を満たした要素からなる配列、
  後のが条件を満たさなかった要素からなる配列

- [0,1,2,3,4,5].partition {|e| e % 3 == 0 }
  #=> [[0, 3], [1, 2, 4, 5]]　3の倍数とそうでないもの
- [5,13,29,15,21,17,12,18,6].partition {|v| v < 16 }
  #=> [[5, 13, 15, 12, 6], [29, 21, 17, 18]]
  16未満　16以上

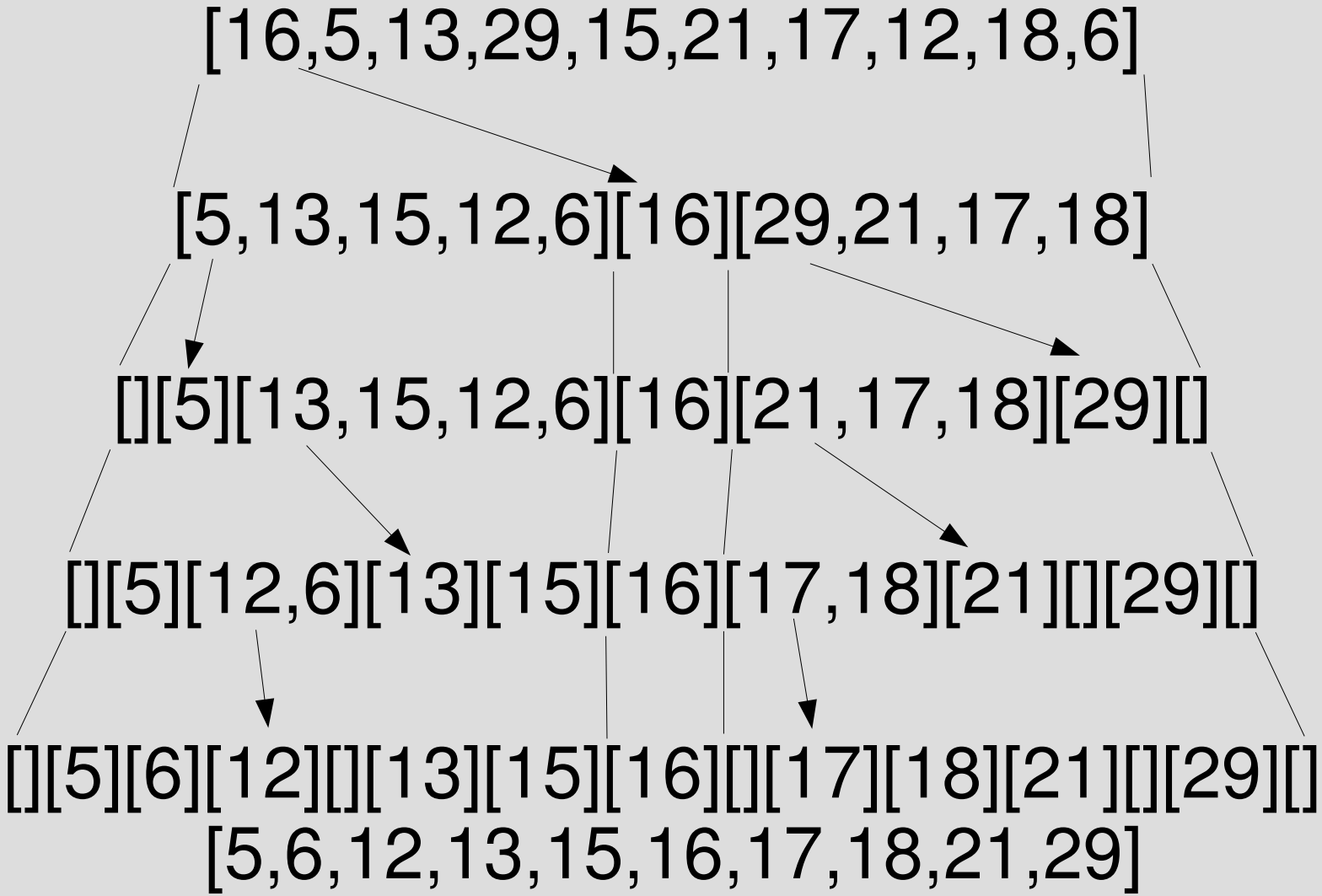
クイックソートの partition

- def qsort(ary)
  return ary if ary.length <= 1
  pivot = ary[0]
  smaller, bigger = ary[1..-1].partition {|v| v < pivot }
  qsort(smaller) + [pivot] + qsort(bigger)
  end

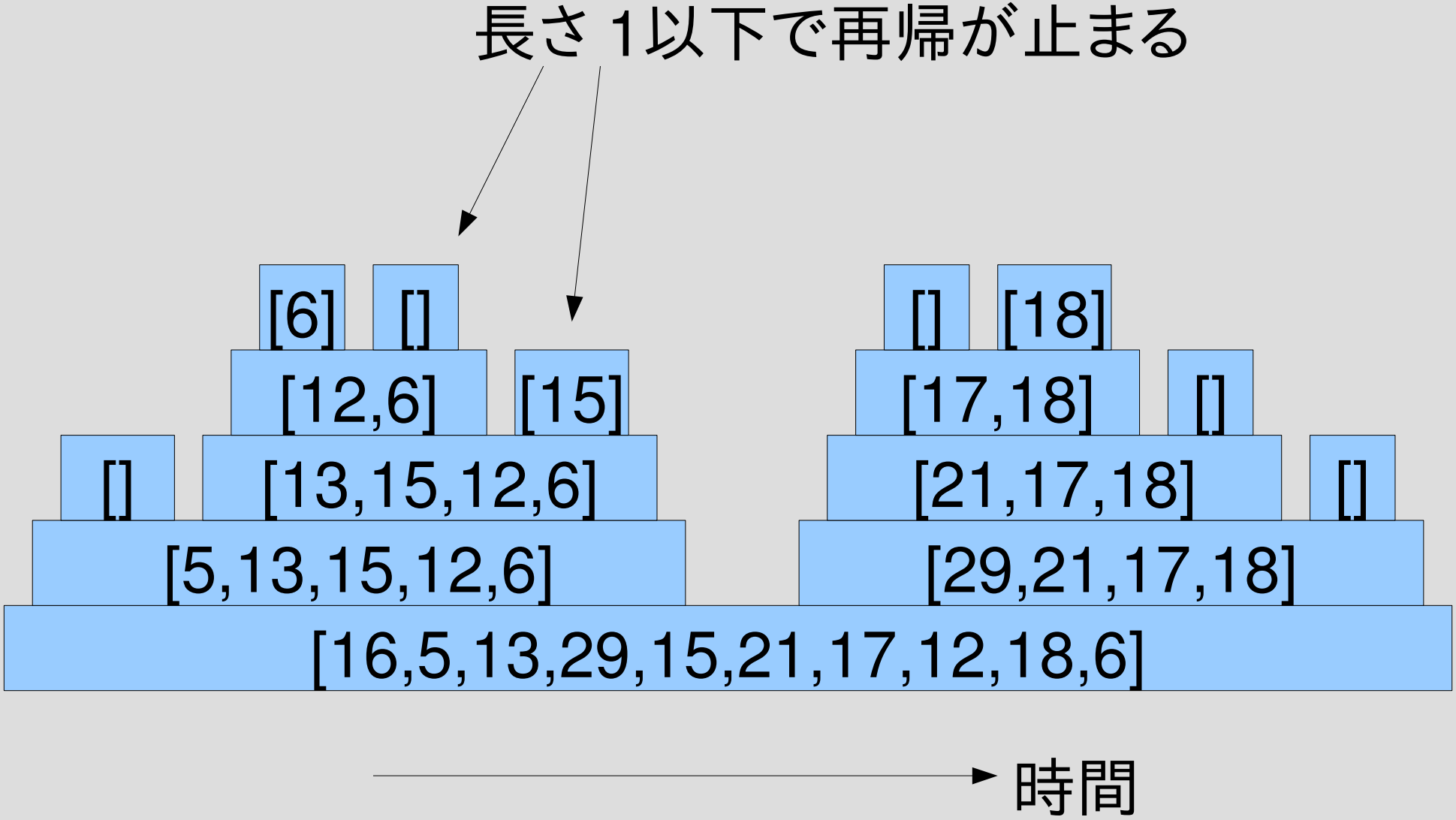
クイックソートの構造


qsort
[16, 5, 13, 29, 15, 21, 17, 12, 18, 6]
pivot
partition
v<pivot
pivot<=v
[5, 13, 15, 12, 6]
[29, 21, 17, 18]
qsort
qsort
pivot
[5, 6, 12, 13, 15] + [16] + [17, 18, 21, 29]

クイックソートの動作全体



qsort の呼び出し時系列



レポート

- 以下の関数を実装してその内容を解説せよ
  - compact
  - アッカーマン関数ack
- レポートには以下の内容を含むものとする
  - 実装
  - 動作の様子
  - 解説
- ∠切 2007-05-22 16:20
- HIPLUS
- 拡張子が txt なテキストファイルが望ましい

compact

- Array#compact に似た機能を持つ関数
- 配列を引数として受け取り、nil を除いた新しい配列を返す
- def compact(ary) ... end と定義して ... を埋める
- Array#compact を使うのは禁止
- 動作の例

```
compact([])          #=> []
compact([1,nil,2])   #=> [1,2]
compact([nil,nil])   #=> []
```

アッカーマン関数 ack

- 有名な再帰関数
- 帰納的定義
  - ack(m,n) = n + 1　　if m = 0
  - ack(m,n) = ack(m-1,1)　　if n = 0
  - ack(m,n) = ack(m-1, ack(m,n-1))　　otherwise
- 関数の値の例
  - ack(0,0)　　#=> 1
  - ack(1,1)　　#=> 3
  - ack(1,2)　　#=> 4
  - ack(2,1)　　#=> 5
  - ack(2,2)　　#=> 7
  - ack(3,4)　　#=> 125

まとめ

- 端末とコマンドライン
- メソッド呼び出しのしくみ
- 再帰の解説
  - 階乗
  - フィボナッチ数
  - クイックソート
- レポートを出した