アルゴリズムとデータ構造入門

2.データによる抽象の構築

2.5 汇用演算システム

奥 乃 博
大学院情報学研究科 知能情報学専攻
知能メディア講座 音声メディア分野
http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/07/IntroAlgDs/
okuno@i.kyoto-u.ac.jp

1月8日・本日のメニュー

- **2.5 Systems with Generic Operations**
  - **2.5.1 Generic Arithmetic Operations**
  - 2.5.2 Combining Data of Different Types
  - 2.5.3 Example: Symbolic Algebra
- Sorting（整列）

1. 内部整列
2. 外部整列

来週の講義の最後の15分間はアンケートを行います.

汎用演算システムの構造

図形言語

`add sub mul div`

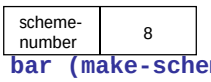
汎用算術演算パッケージ
Genetic arithmetic package

`add-rat sub-rat mul-rat div-rat`	`add-complex sub-complex mul-complex div-complex`	`+ - * /`
有理数算術演算 Rational arithmetic	**複素数算術演算 Complex arithmetic**	**通常算術演算 Ordinary arithmetic**
	直交座標表現 Rectangular / **極座標表現 Polar**	

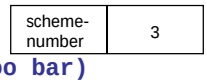
リスト構造と基本マシン算術演算

Scheme number パッケージの使用法

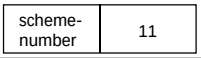
```
(define (make-scheme-number n)
   ((get 'make 'scheme-number) n) )
(define foo (make-scheme-number 8))
```




```
(define bar (make-scheme-number 3))
```



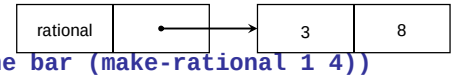

```
(add foo bar)
((get 'add '(scheme-number scheme-number))
  (contents foo) (contents bar) )
(+ 8 3)
```



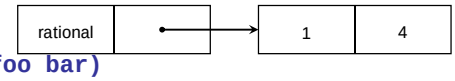
11

Rational number パッケージの使用法

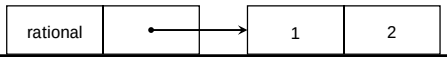
```
(define (make-rational n d)
   ((get 'make 'rational) n d)) )
(define foo (make-rational 3 8))
```




```
(define bar (make-rational 1 4))
```




```
(add foo bar)
((get 'add '(rational rational))
 (contents foo) (contents bar) )
(add-rat (contets foo) (contents bar))
```



16

Complex number パッケージ

```
(define(install-complex-package)
  ;; i imported procedures from rectangular and polar
   packages

   (define (make-from-real-imag x y)
      ((get 'make-from-real-imag
            'rectangular) x y ))
   (define (make-from-mag-ang r a)
      ((get 'make-from-mag-ang 'polar)
       r a ))
```

17

Complex number パッケージ(続)

```
(define(install-complex-package)
  ;; internal procedures
  (define (add-complex z1 z2)
     (make-from-real-imag
       (+ (real-part z1) (real-part z2))
       (+ (imag-part z1) (imag-part z2))))
  (define (sub-complex z1 z2)
     (make-from-real-imag
       (- (real-part z1) (real-part z2))
       (- (imag-part z1) (imag-part z2))))
  (define (mul-complex z1 z2)
     (make-from-mag-ang
       (* (magnitude z1) (magnitude z2))
       (+ (angle z1) (angle z2))))
```

18

Complex number パッケージ(続々)

```
(define(install-complex-package)
  ;; internal procedures

  (define (div-complex z1 z2)
     (make-from-mag-ang
       (/ (magnitude z1) (magnitude z2))
       (- (angle z1) (angle z2))))

  ;; internal procedures
  (define (tag z) (attach-tag 'complex z))
  (put 'add '(complex complex)
       (lambda (z1 z2)
          (tag (add-complex z1 z2)) ))
```

19

Complex number パッケージ(4)

```
(define(install-complex-package)
  ;; internal procedures

  (put 'sub '(complex complex)
       (lambda (z1 z2)
          (tag (sub-complex z1 z2)) ))
  (put 'mul '(complex complex)
       (lambda (z1 z2)
          (tag (mul-complex z1 z2)) ))
  (put 'div '(complex complex)
       (lambda (z1 z2)
          (tag (div-complex z1 z2)) ))
```

20

Complex number パッケージ(5)

```
(define(install-complex-package)
  ;; internal procedures

  (put 'make-from-real-imag 'complex
       (lambda (x y)
          (tag (make-from-real-imag x y))
       ))
  (put 'make-from-mag-ang 'complex
       (lambda (r a)
          (tag (make-from-mag-ang r a))))
  'done )
```

21

Ex.2.77 Complex number パッケージ

```
(define (make-complex-from-real-imag x y)
   ((get 'make-from-real-imag 'complex)
    x y ))

(define (make-complex-from-mag-ang r a)
   ((get 'make-from-mag-ang 'complex)
    r a ))
```

- Complex number の genetic operations

```
(put 'real-part '(complex) real-part)
(put 'imag-part '(complex) imag-part)
(put 'magnitude '(complex) magnitude)
(put 'angle '(complex) angle)
```

22

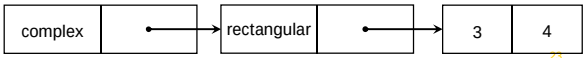
Complex number パッケージの使用法

```
(define (make-complex-from-real-imag x y)
   ((get 'make-from-real-imag 'complex)
    x y ))

(define (make-complex-from-mag-ang r a)
   ((get 'make-from-mag-ang 'complex)
    r a ))

(define foo
   (make-complex-from-real-imag 3 4) )
```

3+4i



23

Ex.2.78 Ordinary number パッケージ

- Scheme-number を効率化したい
- 基本手続きは、内部でタイプチェックをしている
- symbol? number? pair? などを使用
- scheme-number では、タイプチェックをシステムに任せて、高速化したい。

24

scheme-number の効率化

```
(define (attach-tag type-tag contents)
  (if (eq? type-tag 'scheme-number)
      contents
      (cons type-tag contents) ))
(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      'scheme-number ))
(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      datum ))
```

- タイプ(型)チェックはシステムに任せた！

26

1月8日・本日のメニュー

- 2.5 Systems with Generic Operations
  - 2.5.1 Generic Arithmetic Operations
  - **2.5.2 Combining Data of Different Types**
  - 2.5.3 Example: Symbolic Algebra
- Sorting（整列）

1. 内部整列
2. 外部整列

29

2.5.2 Combining Data of Different Types

- 異なるタイプ同士に算術演算を拡張する。
- 引数のタイプにより適切な演算を行う手続きを適用
- 案1はどうか？

```
;; to be included in the complex package
(define (add-complex-to-schemenum z x)
   (make-from-real-imag
      (+ (real-part z) x)
       (imag-part z)))
(put 'add '(complex scheme-number)
   (lambda (z x)
     (tag (add-complex-to-schemenum z x))
   ))
```

30

2.5.2 Combining Data of Different Types

- 異なるタイプ同士に算術演算を拡張する。
- 手続きを適用する前に、対応するタイプでない引数についてはそのタイプに変換する。

強制型変換（*Coercion*）と呼ぶ。

```
(+ 3 3.1) ⇒ (+ 3.0 3.1)

(* 3+4i 2) ⇒ (+ 3+4i 2+0i)
```

31

Coercion（強制型変換）

- 異なるタイプ同士に算術演算を拡張する。
- 引数のタイプにより適切な演算を行う手続きを適用
- 手続きを適用する前に、対応するタイプでない引数についてはそのタイプに変換する。

```
(define (scheme-number->complex n)
   (make-complex-from-real-imag
       (contents n) 0 ))
(put-coercion
    'scheme-number 'complex
    scheme-number->complex )
```

32

Coercion（強制型変換）

```
(define (apply-generic op . args)
 (let ((type-tags (map type-tag args)))
   (let ((proc (get op type-tags)))
     (if proc
        (apply proc (map contents args))
        (if (= (length args) 2)
            (let ((type1 (car type-tags))
                  (type2 (cadr type-tags))
                  (a1 (car args))
                  (a2 (cadr args)) )
              (let ((t1->t2 (get-coercion type1 type2))
                    (t2->t1 (get-coercion type2 type1)) )
                (cond (t1->t2
                       (apply-generic op (t1->t2 a1) a2))
                      (t2->t1
                       (apply-generic op a1 (t2->t1 a2)))
                      (else
                       (error "No method for these types"
                              (list op type-tags) )))))
            (error "No method for these types"
                   (list op type-tags) ))))))
```

33

Coercion（強制型変換、改）

```
(define (apply-generic op . args)
 (let* ((type-tags (map type-tag args))
        (proc (get op type-tags)) )
   (if proc
       (apply proc (map contents args))
       (if (= (length args) 2)
           (let* ((type1 (car type-tags))
                  (type2 (cadr type-tags))
                  (a1 (car args))
                  (a2 (cadr args))
                  (t1->t2 (get-coercion type1 type2))
                  (t2->t1 (get-coercion type2 type1)) )
             (cond (t1->t2
                    (apply-generic op (t1->t2 a1) a2))
                   (t2->t1
                    (apply-generic op a1 (t2->t1 a2)))
                   (else
                    (error "No method for these types"
                           (list op type-tags) )))))
           (error "No method for these types"
                  (list op type-tags) ))))))
```

34

let と let*

```
(let ((x 1)
      (y 3))
  (let ((x 8)
        (z (+ x y)) )
    (display (list x y z))    8 (+ x y) ))
  ))
```

の出力は? **(8 3 4)**

```
(let ((x 1)
      (y 3))
  (let* ((x 8)
         (z (+ x y)) )
    (display (list x y z))
  ))
```

の出力は? **(8 3 11)**

```
(let ((x 1)
      (y 3))
   ((lambda (x z)
      (display (list x y z)))
    8 (+ x y) ))
```

と等価 (syntax sugar)

```
(let ((x 1)
      (y 3))
  ((lambda (x)
     ((lambda (z)
        (display (list x y z)))
      (+ x y) ))
   8 ))
```

と等価 (syntax sugar)

Hierarchies of types（型階層）

- Coercionではどの型へ変換するかが重要。
- 単純な場合：

- *Tower of types*（型の塔）
- 演算結果の単純化には既約化だけでなく、型階層を低位に簡略化することも含まれる。
- 例： 4.0+3.7i + 5.0-3.7i
- 結果は複素数ではなく、実数

36

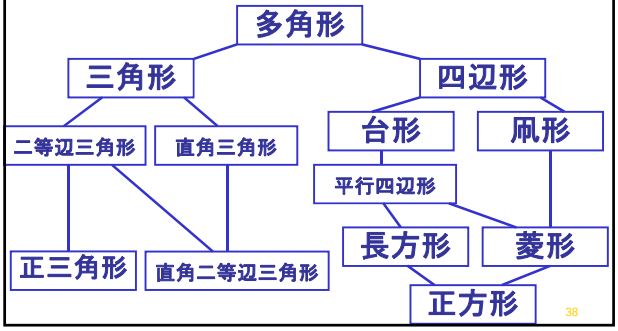
Inadequencies of hierachies

- 演算結果の単純化には既約化だけでなく、型階層を低位に簡略化することも含まれる。
- 例： 4.0+3.7i + 5.0-3.7i
- 結果は複素数ではなく、実数
- Ex2.83, 84 raise の設計
- Ex2.85 drop の設計

37

Hierarchies of types（型階層）

- 複雑な場合： 両者に共通の上位に変換



38

Abstraction barrier の効用

1. インタフェースの手続き(generic nameで)を定義しておけば、その手続きをどのように実装するかの決定は遅延できる。
2. インタフェースの実装はタイプにより規定。
3. 複数のタイプに対して手続きを適用する前には、対応するタイプに強制型変換を行う。
4. 同じインタフェースを複数のタイプで実装することも可能。

- 汎用演算(generic operations)
- 情報隠蔽(information hiding)

41

2.5.3 例:記号代数(Symbolic algebra)

- 設計の課題
- どれだけ数学的な性質を満足するか。

$$f(x) = 5x^2 + 3x - 8$$

$$g(y) = 5y^2 + 3y - 8$$

- 数学的には、上記の2つの式は等価
- 本講義では、表現が異なると捉える。
- 以下、加算と乗算のみを考える。

42

多項式型(poly)の加算と乗算

```
(define (add-poly p1 p2)
  (if (same-variable? (variable p1)
                      (variable p2) )
      (make-poly
        (variable p1)
        (add-terms (term-list p1)
                   (term-list p2)) )
      (error "Polys not in same var - ADD-POLY"
             (list p1 p2) )))
(define (mul-poly p1 p2)
  (if (same-variable? (variable p1)
                      (variable p2) )
      (make-poly
        (variable p1)
        (mul-terms (term-list p1)
                   (term-list p2) ))
      (error "Polys not in same var - MUL-POLY"
             (list p1 p2) )))
```

43

Polynomial パッケージ

```
(define (install-polynomial-package)
    ;; internal procedures
    ;; representation of poly
  (define (make-poly variable term-list)
     (cons variable term-list))
  (define (variable p) (car p))
  (define (term-list p) (cdr p))
  (define (variable? x) (symbol? x))
  (define (same-variable? v1 v2)
     (and (variable? v1) (eq? v1 v2)) )

  ;; representation of terms (項) and term lists
    <下記で定義する adjoin-term ··· coeff>
```

44

Polynomial パッケージ(続)

```
(define (install-polynomial-package)

    ;; 先ほど定義した多項式の加算と乗算
  (define (add-poly p1 p2)
     (if (same-variable? (variable p1) (variable p2))
         (make-poly (variable p1)
            (add-terms (term-list p1) (term-list p2)) )
         (error "Polys not in same var – ADD-POLY"
                (list p1 p2) )))

  (define (mul-poly p1 p2)
     (if (same-variable? (variable p1) (variable p2))
         (make-poly (variable p1)
             (mul-terms (term-list p1)(term-list p2)) )
         (error "Polys not in same var – MUL-POLY"
                (list p1 p2) )))
```

45

Polynomial パッケージ(続)

```
(define (install-polynomial-package)
  ;; interface to rest of the system
  (define (tag p)
     (attach-tag 'polynomial p) )
  (put 'add '(polynomial polynomial)
       (lambda (p1 p2)
          (tag (add-poly p1 p2)) ))
  (put 'mul '(polynomial polynomial)
       (lambda (p1 p2)
          (tag (mul-poly p1 p2)) ))
  (put 'make 'polynomial
       (lambda (var terms)
          (tag (make-poly var terms)) ))
  'done )
```

46

加算での term list の処理

```
(define (add-terms L1 L2)
  (cond ((empty-termlist? L1) L2)
        ((empty-termlist? L2) L1)
        (else
          (let ((t1 (first-term L1))
                (t2 (first-term L2)) )
            (cond ((> (order t1) (order t2))
                   (adjoin-term t1
                     (add-terms (rest-terms L1) L2)) )
                  ((< (order t1) (order t2))
                   (adjoin-term t2
                     (add-terms L1 (rest-terms L2)) ))
                  (else
                   (adjoin-term
                     (make-term (order t1)
                          (add (coeff t1) (coeff t2)) )
                     (add-terms (rest-terms L1)
                                (rest-terms L2) ))))))))
```

47

Polynomial packageを追加すると

- add-poly と mul-poly を add, mul として、汎用算術演算システムに追加
- 当然、次の多項式は計算できる。

$$\left[5x^2+(3+5i)x-8\right]\cdot\left[x^4+\frac{2}{3}x^3+(9+5i)\right]$$

- さらに、次の多項式も計算可能。

$$\left[(y+1)x^2+(y^2+3)x+(y-8)\right]\cdot\left[(y-2)x+(y^3+9)\right]$$

49

term list（項リスト）の表現

- 項の次数を鍵（key）とした係数の集合
- 集合の表現法を思い出そう。

1. 多項式のほとんどの次数の項で非零係数を持つ⇒密（dense）

$$3x^8-4x^7+10x^6+x^5-3x^4+9x^3+5x^2+3x-8$$

```
(3 -4 10 1 -3 9 5 3 -8)
```

2. 多項式が多くの零の項を持つ⇒疎（sparse）

$$5x^{100}+3x^7-1$$

```
((100 5) (7 3) (0 -1))
```

本講義では sparseを仮定

50

term list（項リスト）の表現

```
(define (adjoin-term term term-list)
   (if (=zero? (coeff term))
       term-list
       (cons term term-list) ))

(define (the-empty-termlist) '())
(define (first-term term-list) (car term-list))
(define (rest-terms term-list) (cdr term-list))
(define (empty-termlist? term-list)
   (null? term-list))

(define (make-term order coeff)
   (list order coeff))
(define (order term) (car term))
(define (coeff term) (cadr term))
```

51

term list（項リスト）の表現

```
(define (=zero? x) (apply-generic '=zero? x))
;; install-scheme-number-package に追加
(define (=zero-schemenum? x) (= x 0))
(put '=zero? '(scheme-number) =zero-schemenum?)
;; install-rational-number-package に追加
(define (zero-rat? x) (=zero? (numer x)))
(put '=zero? '(rational) =zero-rat?)
;; install-polynomial-package に追加
(define (zero-poly? p)
   (empty-termlist? (term-list p)) )
(put '=zero? '(polynomial) =zero-poly?)

;; 多項式を作成
(define (make-polynomial var terms)
   ((get 'make 'polynomial) var terms))
```

52

A Happy New Year

- 最後の宿題は、次の計1問：
- **Ex.2.81**
- **1月15日正午締切**

1月8日・本日のメニュー

- 2.5 Systems with Generic Operations
  - 2.5.1 Generic Arithmetic Operations
  - 2.5.2 Combining Data of Different Types
  - 2.5.3 Example: Symbolic Algebra
- **Sorting（整列）**

1. 内部整列

2. 外部整列

Java によるデモ

http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/07/IntroAlgDs/

55

Sorting（整列）

- **内部整列（internal sorting）**
  - データはすべて主記憶上に置いて整列

  1. 作業領域を極力減らす。
  2. 比較回数を極力減らす。
- **外部整列（external sorting）**
  - 外部の記憶装置を用いて整列

  3. 主記憶と補助記憶との間でのデータ転送回数を極力減らす。

56

Internal Sorting（内部整列）

- 逐次入力型
  1. 挿入ソート（insertion sort）
  2. ヒープソート（heap sort）
- バッチ型
  1. クイックソート（quick sort）
  2. バブルソート（bubble sort）
- その他（外部整列と共通）
  1. マージソート（merge sort）

57

Internal Sorting（内部整列）

- 逐次入力型
  - **挿入ソート（insertion sort）**
  - ヒープソート（heap sort）
- バッチ型
  - **クイックソート（quick sort）**
  - バブルソート（bubble sort）
- その他（外部整列と共通）
  - マージソート（merge sort）

58

安定整列（stable sorting）

- 同じデータ間のもともとの順序が整列後も保存されている整列のこと。
- 基数ソート（radix sort）では重要な性質。
- 辞書式順序で整列で基数ソートが使われる。

59

挿入ソート（insert sort）

```
(define (insert-sort-pred pred records)
  (if (null? records)
      '()
      (insert-elem pred (car records)
         (insert-sort-pred pred (cdr records))
      )))
(define (insert-elem pred elem ordered-rec)
  (cond ((null? ordered-rec) (cons elem '()))
        ((pred elem (car ordered-rec))
         (cons elem ordered-rec) )
        (else
         (cons (car ordered-rec)
               (insert-elem pred elem
                     (cdr orderd-rec) )))))
(define (insert-sort records . args)
  (insert-sort-pred
     (if (null? args) > (car args)))
    records ))
```

60

挿入ソート(insert sort)の実行トレース

```
(insert-sort-pred > '(2 3 1 6 4))
                      ↑
2
3  2
1
6  3  2  1
   4  3  2  1
```

61

挿入ソート(insert sort)の計算量

1. 最悪の場合(*worst case*) (predが≧とする)
   - 大きなものから順に入ってくる
   - 比較回数は $\sum_{i=1}^{n}(i-1)=\frac{1}{2}n(n-1)$ $\Theta(n^2)$
2. 最良の場合(*best case*)
   - 小さいものから順に入ってくる
   - 比較回数は $\sum_{i=2}^{n}1=(n-1)$ $\Theta(n)$
3. 平均の場合(*average case*)
   - すでに入っているデータの半分だけ比較
   - 比較回数は $\sum_{i=1}^{n}\frac{1}{2}(i-1)=\frac{1}{4}n(n-1)$ $\Theta(n^2)$

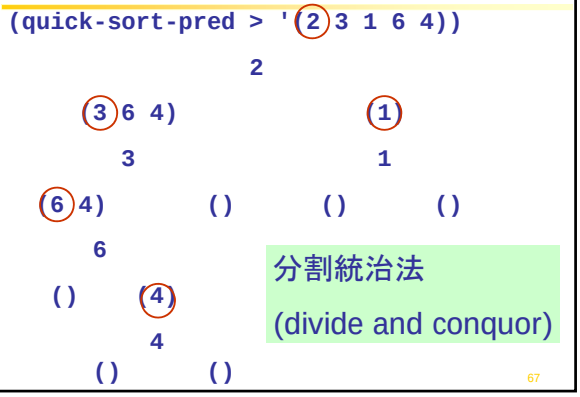
65

クイックソート(quick sort)

```
(define (quick-sort records . args)
  (quick-sort-pred (if (null? args) > (car args))
                   records ))
(define (quick-sort-pred pred records)
   (if (null? records)
       '()
       (let* ((pivot (car records))
              (division (partition pred pivot
                                   (cdr records) '() '() )))
         (append (quick-sort-pred pred (car division))
                 (cons pivot
                       (quick-sort-pred pred
                                        (cdr division) ))))))
(define (partition pred pivot records left right)
  (cond ((null? records) (cons left right))
        ((pred pivot (car records))
         (partition pred pivot (cdr records) left
                    (cons (car records) right) ))
        (else
         (partition pred pivot (cdr records)
                    (cons (car records) left) right ))))
```

66

quick sortの実行トレース（まとめ）



quick sortの実行トレース1

```
1.  (quick-sort-pred > '(2 3 1 6 4))
2.  (partition > 2 '(3 1 6 4) '() '())
      ((3 6 4) (1))
3.  (append (quick-sort-pred > '(3 6 4))
        (cons 2 (quick-sort-pred > '(1)) )

3-1. (partition > 3 '(6 4) '() '())
       ((6 4) ())
3-2. (append (quick-sort-pred > '(6 4))
           (cons 3 ()) )
4-1.     (partition > 6 '(4) '() '())
             (() (4))
4-2.     (append ()
             (cons 6 (quick-sort-pred > '(4))) )
5-1.       (partition > 4 '() '() '())
             (() ())
4-3.     (append () (cons 6 (append () (cons 4 ()))))
           (6 4)
4-4.     (6 4)
```


quick sortの実行トレース2

```
3-1.    (partition > 3 '(6 4) '() '())
          ((6 4) ())
3-2.    (append (quick-sort-pred > '(6 4))
            (cons 3 ()) )
3-3.    (append '(6 4) (3))
          (6 4 3)
3-4. (append '(6 4 3)
           (cons 2 (quick-sort-pred > '(1)) )
4-1.     (partition > 1 '() '() '())
4-2. (append '(6 4 3)
           (cons 2 '(1)) )
     (6 4 3 2 1)
```

クイックソート（quick sort）の計算量

1．最悪の場合（*worst case*）（predが≧とする）
- 小さいものから順に入ってくる
- partitionでの走査回数は

$$\sum_{i=1}^{n}(n-i)=n^2-\frac{1}{2}n(n+1)=\frac{1}{2}n(n-1) \qquad \Theta(n^2)$$

2．最良の場合（*best case*）
- 分割がバランスしている
- partitionの呼ばれる回数は

$\log n$ $\qquad \Theta(n\log n)$

3．平均の場合（*average case*）

70

クイックソート（quick sort）の計算量

3．平均の場合（*average case*）
- データはすべて異なる。あらゆる順列が等確率。
- 途中での分割でも同様の仮定が成立するとする。

- *n*要素のquick sort に要する時間：*T(n)* とする
- 左右の結果の統合に要する時間: *cn*（*c*: 定数）
- *n*要素が*i*要素と*n-i-1*要素に分割されたとすると

$$T(n)\le T(i)+T(n-i-1)+cn$$

71

クイックソート（quick sort）の計算量

- *n*要素の分割、*(0,n-1), (1, n-2),…, (n-1,0)* が等確率で生ずるとすると、次の漸化式を得る

$$T(n)=cn+\frac{1}{n}\sum_{i=0}^{n-1}T(i) \qquad (n\ge 2)$$
$$T(1)=0$$
$$T(0)=1$$

- 帰納法で証明すると

$T(n)\approx 2n\log n$ $\qquad \Theta(n\log n)$

72