コンピュータサイエンス第２
ソート（整列）アルゴリズム：演習

南出 靖彦

第 3 回

ソート（整列）

問題：

- 入力： 数列 $\langle a_1, a_2, \ldots, a_n \rangle$
- 出力: 入力を並べ替えた $\langle b_1, b_2, \ldots, b_n \rangle$
  - $b_1 \leq b_2 \leq \cdots \leq b_n$ が成り立つ

例：

- $\langle 8, 3, 1, 4, 6 \rangle \Longrightarrow \langle 1, 3, 4, 6, 8 \rangle$
- $\langle 8, 3, 1, 3, 6 \rangle \Longrightarrow \langle 1, 3, 3, 6, 8 \rangle$
  - 入力に同じ値が複数個あってもよい.

アルゴリズム (algorithm)

アルゴリズムとは

- 計算手順
- 目標を計算を達成する計算方法
- 良いアルゴリズム $\Leftrightarrow$ 効率の良いアルゴリズム
  - 計算時間が短い

例えば，ソートには色々なアルゴリズムがある．数列の長さを n とする．

- バブルソート：計算時間が n^2 に比例
- マージソート：計算時間が $n \log n$ に比例
- ...

マージ

$\mathrm{merge}(\langle a_1, \ldots, a_i \rangle, \langle b_1, \ldots, b_j \rangle)$

- 引数：二つの整列された数列
- 結果：$\langle a_1, \ldots, a_i \rangle$, $\langle b_1, \ldots, b_j \rangle$ を併合した整列された数列

例:

$$\mathrm{merge}(\langle 1, 3, 3, 7 \rangle, \langle 2, 4, 6, 7 \rangle) = \langle 1, 2, 3, 3, 4, 6, 7, 7 \rangle$$

アルゴリズム

1. 両方の数列を先頭から調べて，小さい方を結果の数列に追加して行く
   - ここでは，$\langle a_1, \ldots, a_i \rangle$ を優先
2. 片方の数列を最後まで調べたら，残り配列の要素を結果に追加して行く

a	$\langle 1, 3, 3, 4 \rangle$	$\langle 1, 3, 3, 4 \rangle$	$\langle 1, 3, 3, 4 \rangle$	$\langle 1, 3, 3, 4 \rangle$
b	$\langle 2, 3, 6, 8 \rangle$	$\langle 2, 3, 6, 8 \rangle$	$\langle 2, 3, 6, 8 \rangle$	$\langle 2, 3, 6, 8 \rangle$
結果	$\langle \rangle$	$\langle 1 \rangle$	$\langle 1, 2 \rangle$	$\langle 1, 2, 3 \rangle$

- 赤が今調べているところ

マージ

アルゴリズム

1. 両方の数列を先頭から調べて，小さい方を結果の数列に追加して行く
   - ここでは，$\langle a_1, \ldots, a_i \rangle$ を優先
2. 片方の数列を最後まで調べたら，残り配列の要素を結果に追加して行く

a	$\langle 1,3,3,4 \rangle$	$\langle 1,3,3,4 \rangle$	$\langle 1,3,3,4 \rangle$	$\langle 1,3,3,4 \rangle$
b	$\langle 2,3,6,8 \rangle$	$\langle 2,3,6,8 \rangle$	$\langle 2,3,6,8 \rangle$	$\langle 2,3,6,8 \rangle$
結果	$\langle \rangle$	$\langle 1 \rangle$	$\langle 1,2 \rangle$	$\langle 1,2,3 \rangle$
a		$\langle 1,3,3,4 \rangle$	$\langle ,3,3,4 \rangle$	
b	$\cdots$	$\langle 2,3,6,8 \rangle$	$\langle 2,3,6,8 \rangle$	$\langle 2,3,6,8 \rangle$
結果		$\langle 1,2,3,3,3 \rangle$	$\langle 1,2,3,3,3,4 \rangle$	$\langle 1,2,3,3,3,4,6 \rangle$
			b の残りを追加	

マージソート

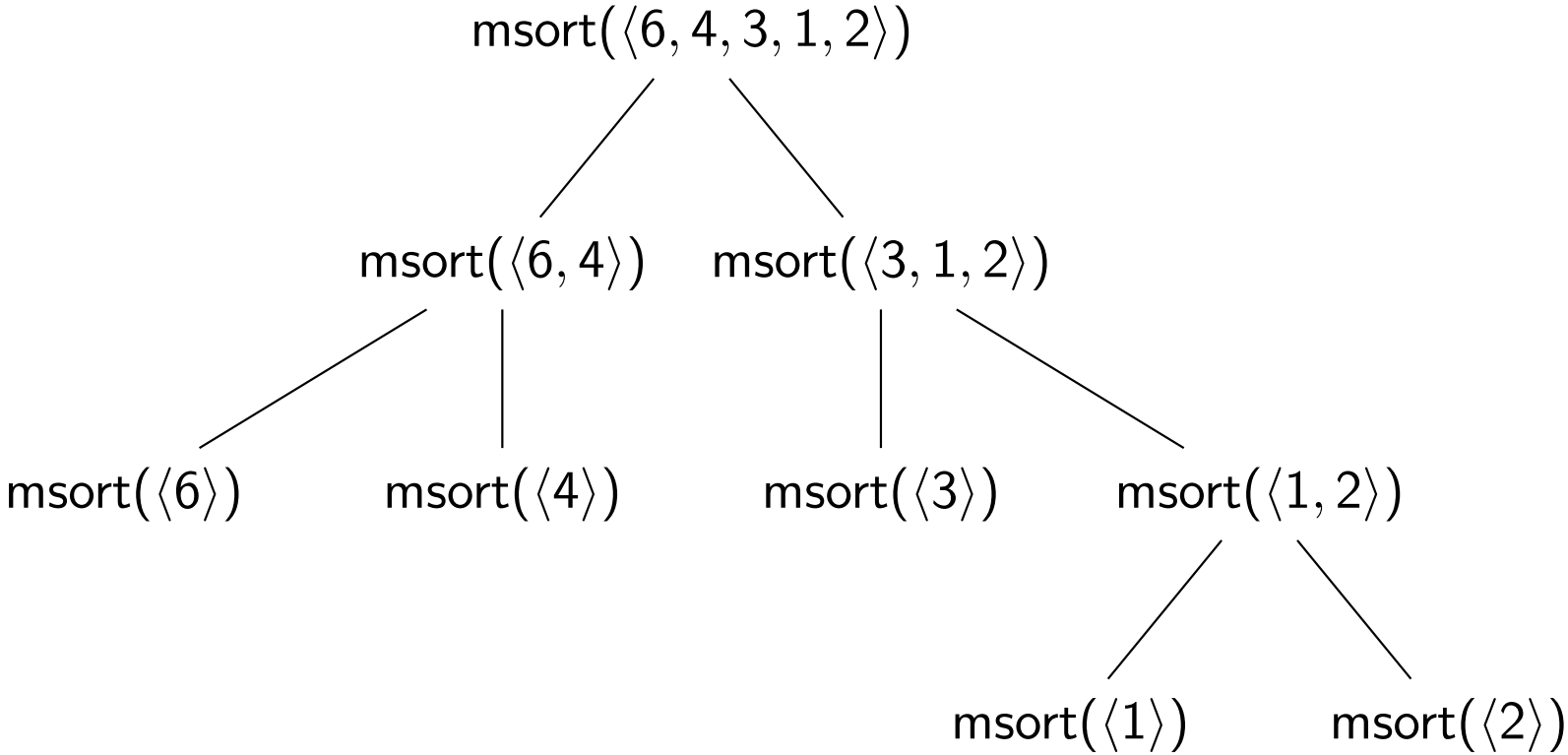
$\mathrm{merge\_sort}(\langle a_1, a_2, \ldots, a_n \rangle)$

- $n \leq 1$：整列されている．列 $\langle a_1, a_2, \ldots, a_n \rangle$ をそのまま返す．
- $n > 1$
  - $\langle a_1, a_2, \ldots, a_n \rangle$ を半分の長さの二つの列に分ける．例えば，
    - $\vec{b} = \langle a_1, a_2, \ldots, a_{n/2} \rangle$
    - $\vec{c} = \langle a_{n/2+1}, a_2, \ldots, a_n \rangle$

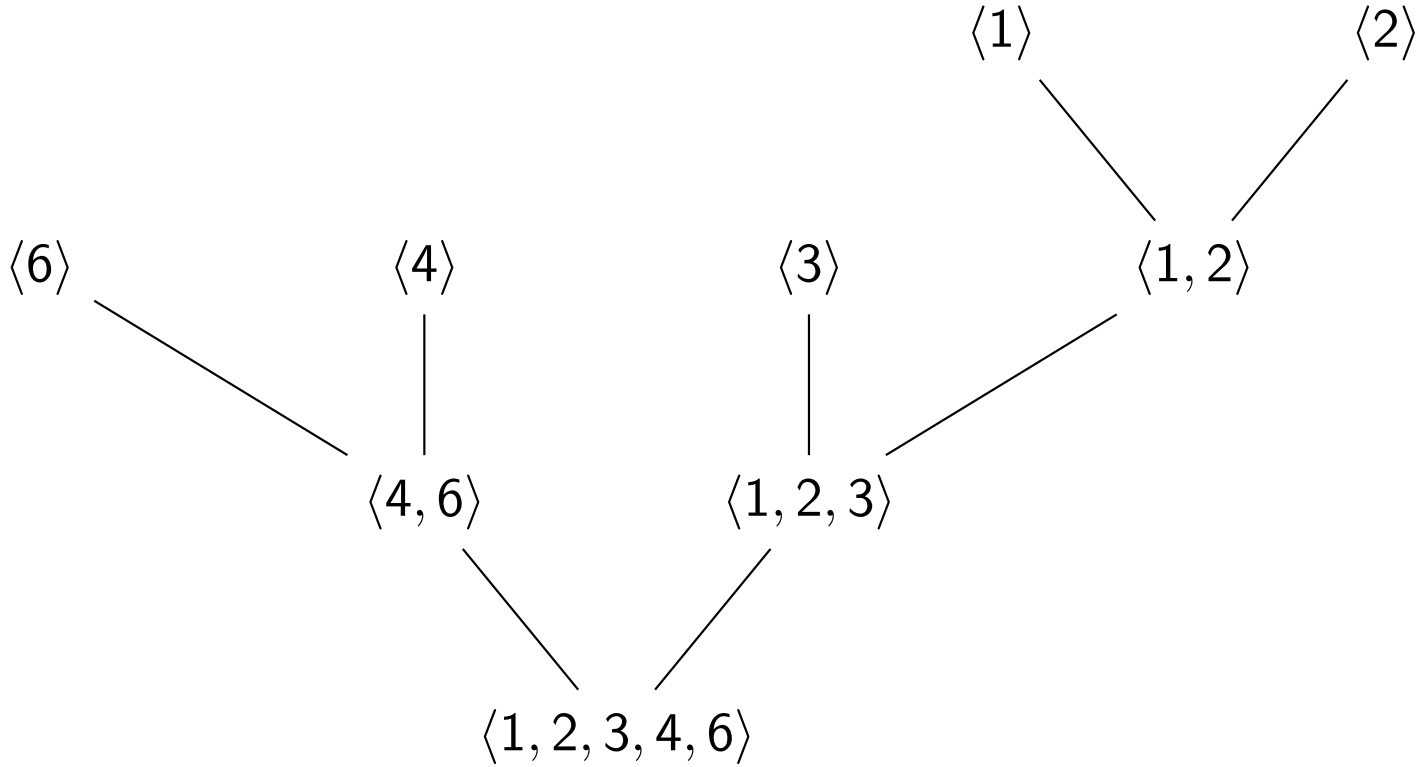
    に分ける．
  - 再帰的にマージソートし，マージする

$$\mathrm{merge}(\mathrm{merge\_sort}(\vec{b}), \mathrm{merge\_sort}(\vec{c}))$$

マージソート: 例


msort(⟨6, 4, 3, 1, 2⟩)
msort(⟨6, 4⟩)
msort(⟨3, 1, 2⟩)
msort(⟨6⟩)
msort(⟨4⟩)
msort(⟨3⟩)
msort(⟨1, 2⟩)
msort(⟨1⟩)
msort(⟨2⟩)

マージソート: 例



マージソートの比較回数

長さ n と m の数列のマージ

- 最大 $n + m - 1$ 回の比較が必要

$T(n)$: 整列する場合に最大何回比較が必要か

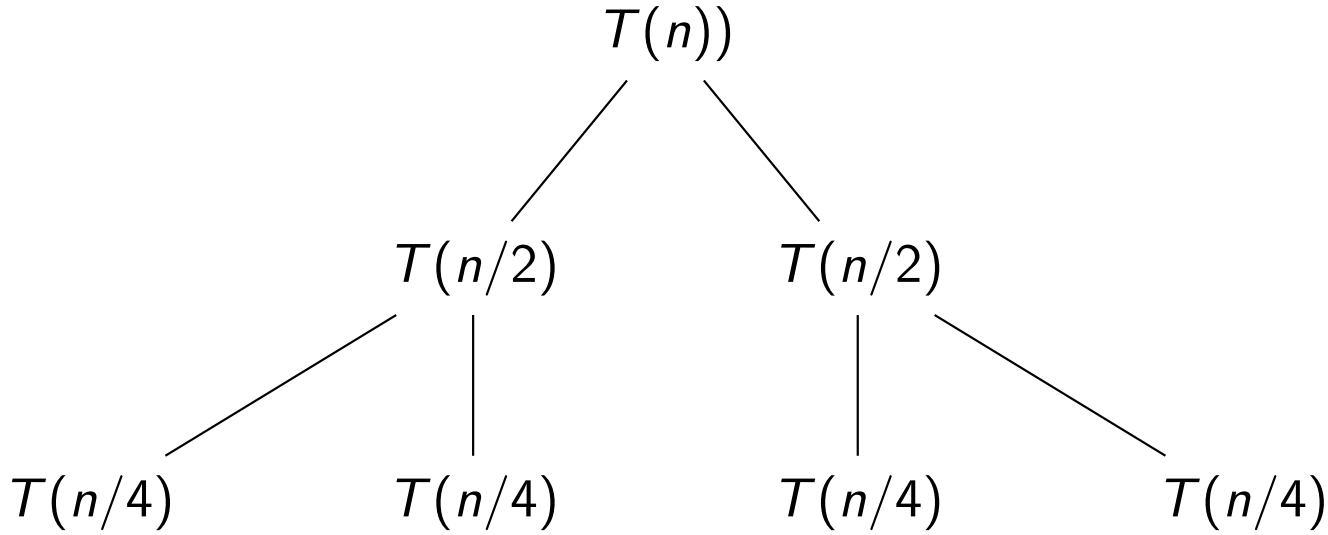
$$\begin{aligned} T(1) &= 0 \\ T(n) &\approx (n/2 + n/2 - 1) + 2T(n/2) \\ &\approx n + 2T(n/2) \end{aligned}$$

これを解くと

$$T(n) \approx n \log n$$

マージソートの比較回数

高さ $\log n$ の木になる



$$T(n) = n + 2T(n/2)$$

$$2T(n/2) = 2\frac{n}{2} + 4T(n/4) = n + 4T(n/4)$$

$$4T(n/4) = 4\frac{n}{4} + 8T(n/8) = n + 8T(n/8)$$

比較回数: 高さ $\times\ n = n \log n$

配列に対するプッシュ

x.push(y): 配列 x の末尾に要素 y を追加する

```
irb(main):002:0> [1,2,3].push(4)
=> [1, 2, 3, 4]
```

使用例：

```
def rev(a)
  r = []                  # 空の配列
  i = a.length - 1
  while i >= 0
    r.push(a[i])
    i = i - 1
  end
  return r
end
```

マージ：プログラム

```
def merge(a, b)
  alen = a.length
  blen = b.length
  r = []
  i = 0
  j = 0
  while i < alen && j < blen
    if a[i] <= b[j]
      r.push(a[i])
      i = i + 1
    else
      r.push(b[j])
      j = j + 1
    end
  end
  while i < alen
    r.push(a[i])
    i = i + 1
  end
  while j < blen
    r.push(b[j])
    j = j + 1
  end
  return r
end
```

Ruby：部分配列

部分配列: a[start, length]

- 配列 a の部分配列
- インデックス start から長さ length

```
irb(main):018:0>  a = [1,2,3,4]
=> [1, 2, 3, 4]
irb(main):020:0> a[1,2]
a[1,2]
=> [2, 3]
```

マージソート：プログラム

```
def msort(a)
  len = a.length
  if len <= 1
    return a
  else
    return merge(msort(a[0,len/2]),
                 msort(a[len/2,len - len/2]))
  end
end
```

クイックソート: $\mathrm{quick\_sort}(\langle a_1, a_2, \ldots, a_n \rangle)$

- $n \leq 1$: 数列をそのまま返す
- $n > 1$:
  1. $\langle a_2, \ldots, a_n \rangle$ を二つに分割する
     - $\vec{b}$: a_1 より小さい要素の列
     - $\vec{c}$: a_1 以上の要素の列
  2. 再帰的にソートして，以下のように連結した数列を返す

$$\mathrm{quick\_sort}(\vec{b}), a_1, \mathrm{quick\_sort}(\vec{c})$$

Ruby: 配列の連結

配列同士の連結:

$\langle a_1, a_2, \ldots, a_i \rangle + \langle b_1, b_2, \ldots, b_j \rangle = \langle a_1, a_2, \ldots, a_i, b_1, b_2, \ldots, b_j \rangle$

```
irb(main):001:0> [1,3,5,7] + [2,4,6]
=> [1, 3, 5, 7, 2, 4, 6]
irb(main):002:0> [1,3,5,7] + []
=> [1, 3, 5, 7]
```

注意：新し配列を返す

```
irb(main):003:0> a = [1,3,5,7]
=> [1, 3, 5, 7]
irb(main):004:0> a + [2,4,6]
=> [1, 3, 5, 7, 2, 4, 6]
irb(main):005:0> a
=> [1, 3, 5, 7]
```

クイックソート: プログラム

```
def qsort(a)
  if a.length <= 1
    return a
  else
    b = []
    c = []
    for i in 1..a.length-1     # 配列 a を a[0] で分割
      if a[i] < a[0]
        b.push(a[i])
      else
        c.push(a[i])
      end
    end
    return quick_sort(b)+[a[0]]+quick_sort(c)
  end
end
```

課題用のプログラム

cs2-kadai1.zip を授業ページからダウンロードする。

- 配列を作る関数 arrays.rb
- バブルソート bubble.rb
- 挿入ソート　insertion.rb
- マージソート merge.rb
- クイックソート quick.rb

課題用のプログラム: arrays.rb

```
# 大きさ n の正順（昇順）に整列された配列
def increasing_array(n)

# 大きさ n の逆順（降順）に整列された配列
def decreasing_array(n)

# 大きさ n のランダムな要素の配列
def random_array(n)
```

実行例

```
irb(main):004:0> increasing_array(5)
=> [0, 1, 2, 3, 4]
irb(main):002:0> decreasing_array(5)
=> [4, 3, 2, 1, 0]
irb(main):007:0> random_array(10)
=> [4, 2, 0, 6, 0, 8, 0, 9, 6, 2]
irb(main):008:0> random_array(10)
=> [2, 6, 3, 6, 0, 9, 6, 1, 2, 9]
```

課題プログラム：merge.rb

```
# 引数：配列 [n_0, n_1, ... ,n_(k-1)]
# マージソートを大きさ n_i 配列に対してテストする
def test_merge(test_array)
  k = test_array.length
  for i in 0..k-1
    n = test_array[i]
    a = increasing_array(n)     # テストしたい配列の種類を選ぶ
    # a = decreasing_array(n)
    # a = random_array(n)
    $ncomp = 0
    start_time = Time.now
    b = merge_sort(a)           # マージソートを実行
    puts "大きさ #{n}"
    puts "実行時間 #{Time.now - start_time}秒"
    puts "比較回数 #{$ncomp}"
    sorted(b)
  end
  return
end
```

プログラム実行方法

例：マージソートを長さ 100 と 200 に対して実行する

```
irb(main):007:0> load "arrays.rb"
irb(main):007:0> load "merge.rb"
irb(main):008:0> test_merge([100,200])
大きさ 100
実行時間 0.000154 秒
比較回数 540
大きさ 200
実行時間 0.000327 秒
比較回数 1278
```

以下の関数が用意してある

- 正順: `test_merge_inc`
- 逆順: `test_merge_dec`
- ランダム: `test_merge_random`

課題２ (a)

ソートする列の長さを適切に設定して各アルゴリズムと列種類について複数回実行して，長さと比較回数と実行時間をエクセルファイル (`sort-exam.xlsx`) に記入せよ。

得られたデータ全体から読み取れる各アルゴリズムの特徴を考察せよ.

- 長さの増加に対して比較回数や実行時間がどのような関数で増えていくか。
- 同じ長さで列種類を変えたり同じ列でアルゴリズムを変えると，比較回数や実行時間はどのように変わるか。

加点項目：グラフ

課題２ (a)：続き

乱数列の場合，長さが同じでも実行毎に異なる列をソートすることになるので，同じ長さで５回実行して比較回数のばらつきを記録し，考察せよ．

- ばらつきを比べるために，変動係数を計算せよ．

$$\text{変動係数} = \text{標準偏差}/\text{平均}$$

- バブルソートは，比較回数は変化しないので除いて良い．

これらの特徴がよくわかるような長さを適切に選ぶこと。列が短すぎると特徴がわからないし，実行回数が少なすぎても特徴がわからない。

課題２ (b)：オプション

- 長さ１億の乱数列をソートするのにかかる実行時間が最も短いプログラムと最も長いプログラムを挙げて，それぞれの実行時間を推測せよ。
- 同様に正順列と逆順列に関しても，それぞれ長さ１億の列をソートする最短時間と最長時間のプログラムを挙げ，それぞれの時間を推測せよ。

提出方法等：課題 2(a),(b)

- 提出期限：12 月 26 日 午前 10:40
  - 午後 9:00 までの提出は採点しますが，大きく減点します.
- 提出するもの
  - レポート
    - 課題 2(a)：通常のフォントの大きさ (11pt まで) で 1 ページ程度を想定しています.
      (細かく読む時間はないと思いますが，長くなっても構いません．)
  - エクセルファイル (`sort-exam.xlsx`)

発展課題：オプション

今回使用したプログラムやアルゴリズムを改良して実験してみよ。

- 提出期限：1 月 30 日 午前 10:40
- 提出方法：OCWi
- 提出するもの：プログラムとレポート（PDF）