

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

準備

- ▶ Documents フォルダの下に CS2 フォルダに移動
 - ▶ `cd Documents/CS2`
- ▶ 講義のウェブページから `day3.zip` をダウンロードする.
- ▶ Terminal で `day3` を `CS2` に移動
 - ▶ `mv ~/Downloads/day3 ./`
 - ▶ `cd day3`

ソート（整列）

問題：

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

例：

- ▶ $\langle 8, 3, 1, 4, 6 \rangle \implies \langle 1, 3, 4, 6, 8 \rangle$
- ▶ $\langle 8, 3, 1, 3, 6 \rangle \implies \langle 1, 3, 3, 6, 8 \rangle$
 - ▶ 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

$\text{merge}(\langle a_1, \dots, a_i \rangle, \langle b_1, \dots, b_j \rangle)$

- ▶ 引数：二つの**整列**された数列
- ▶ 結果： $\langle a_1, \dots, a_i \rangle, \langle b_1, \dots, b_j \rangle$ を併合した**整列**された数列

例:

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

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いく
 - ▶ ここでは、 $\langle a_1, \dots, a_i \rangle$ を優先
2. 片方の数列を最後まで調べたら、残りの配列の要素を結果に追加していく

a	$\langle \textcolor{red}{1}, 3, 3, 4 \rangle$	$\langle 1, \textcolor{red}{3}, 3, 4 \rangle$	$\langle 1, \textcolor{red}{3}, 3, 4 \rangle$	$\langle 1, 3, \textcolor{red}{3}, 4 \rangle$
b	$\langle \textcolor{red}{2}, 3, 6, 8 \rangle$	$\langle \textcolor{red}{2}, 3, 6, 8 \rangle$	$\langle 2, \textcolor{red}{3}, 6, 8 \rangle$	$\langle 2, 3, \textcolor{red}{6}, 8 \rangle$
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いく
▶ ここでは、 $\langle a_1, \dots, a_i \rangle$ を優先
2. 片方の数列を最後まで調べたら、残りの配列の要素を結果に追加していく

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

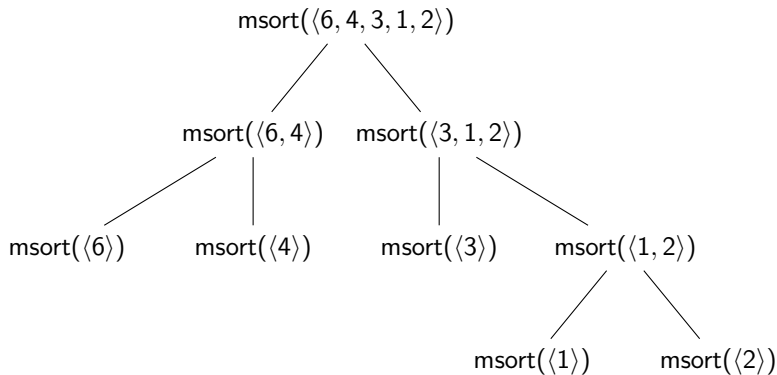
マージソート

`merge_sort($\langle a_1, a_2, \dots, a_n \rangle$)`

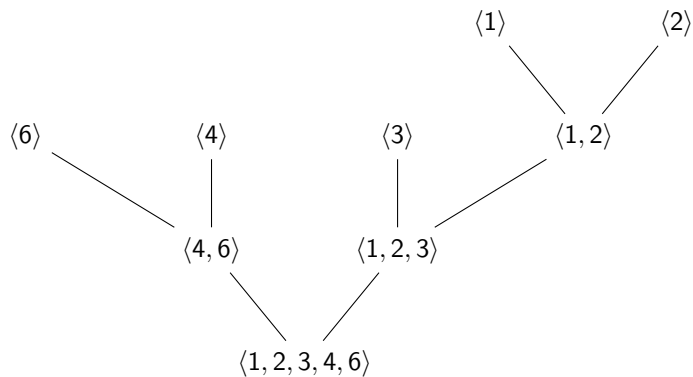
- ▶ $n \leq 1$: 整列されている. 列 $\langle a_1, a_2, \dots, a_n \rangle$ をそのまま返す.
- ▶ $n > 1$
 - ▶ $\langle a_1, a_2, \dots, a_n \rangle$ を半分の長さの二つの列に分ける. 例えば,
 - ▶ $\vec{b} = \langle a_1, a_2, \dots, a_{n/2} \rangle$
 - ▶ $\vec{c} = \langle a_{n/2+1}, a_{n/2+2}, \dots, a_n \rangle$に分ける.
 - ▶ 再帰的にマージソートし, マージする

`merge(merge_sort(\vec{b}), merge_sort(\vec{c}))`

マージソート: 例



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

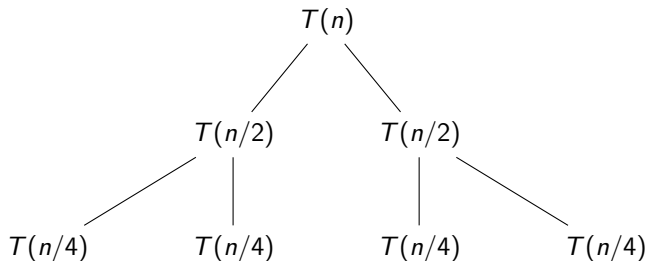
$$\begin{aligned}T(1) &= 0 \\T(n) &\approx (n/2 + n/2 - 1) + 2T(n/2) \\&\approx n + 2T(n/2)\end{aligned}$$

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



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

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

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

比較回数: 高さ $\times n = n \log n$

配列に対する追加 (append)

`x.append(y)`: 配列 `x` の末尾に要素 `y` を追加する

```
>>> a = [1,2,3]
>>> a.append(4)
>>> a
[1, 2, 3, 4]
```

使用例 :

```
def rev(a):
    r = []                # 空の配列
    i = len(a) - 1
    while i >= 0:
        r.append(a[i])
        i = i - 1
    return r
```

マージ：プログラム

```
def merge(a, b):  
    alen = len(a)  
    blen = len(b)  
    c = []  
    i = 0  
    j = 0  
    while i < alen and j < blen:  
        if a[i] <= b[j]:  
            c.append(a[i])  
            i = i + 1  
        else:  
            c.append(b[j])  
            j = j + 1
```

```
    while i < alen:  
        c.append(a[i])  
        i = i + 1  
    while j < blen:  
        c.append(b[j])  
        j = j + 1  
    return c
```

Python : 部分配列

部分配列: `a[start : end]`

- ▶ 配列 `a` の部分配列
- ▶ インデックス `start` から `end - 1`

```
>>> a = [1,2,3,4,5]
```

```
>>> a[1:3]
```

```
[2, 3]
```

```
>>> a[0:5]
```

```
[1, 2, 3, 4, 5]
```

マージソート : プログラム

```
def merge_sort(a):  
    alen = len(a)  
    if alen <= 1:  
        return a  
    else:  
        return merge(merge_sort(a[0 : alen//2]),  
                      merge_sort(a[alen//2 : alen]))
```

クイックソート: 考え方

数列 \vec{a} をクイックソートで整列

1. 基準値 (ピボット) を選ぶ. 例えば, 先頭の値.
2. 数列を以下のように分割する
 - ▶ \vec{b} : ピボットより小さい値
 - ▶ \vec{c} : ピボット以上の値
3. \vec{b} , \vec{c} を再帰的にクイックソートで整列. 整列した結果を \vec{b}' , \vec{c}' とする.
4. \vec{b}' と \vec{c}' を連結する.

クイックソート: $\text{quick_sort}(\langle a_1, a_2, \dots, a_n \rangle)$

ピボット: 先頭の値

- ▶ $n \leq 1$: 数列をそのまま返す
- ▶ $n > 1$:
 1. $\langle a_2, \dots, a_n \rangle$ 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

$\text{quick_sort}(\vec{b}), a_1, \text{quick_sort}(\vec{c})$

Python: 配列の連結

配列同士の連結:

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

```
>>> [1,3,5,7] + [2,4,6]
```

```
[1, 3, 5, 7, 2, 4, 6]
```

```
>>> [1,3,5,7] + []
```

```
[1, 3, 5, 7]
```

注意：新しい配列を返す

```
>>> a = [1,3,5,7]
```

```
>>> a + [2,4,6]
```

```
[1, 3, 5, 7, 2, 4, 6]
```

```
>>> a
```

```
[1, 3, 5, 7]
```

クイックソート: プログラム

```
def quick_sort(a):  
    if len(a) <= 1:  
        return a  
    else:  
        b = []  
        c = []  
        for i in range(1, len(a)):    # 配列 a を a[0] で分割  
            if a[i] < a[0]:  
                b.append(a[i])  
            else:  
                c.append(a[i])  
        return quick_sort(b)+[a[0]]+quick_sort(c)
```

課題用のプログラム: sort.py

補助関数

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 の乱数列の配列

```
def random_array(n):
```

実行例

```
$ python -i sort.py
```

```
>>> increasing_array(5)
```

```
[0, 1, 2, 3, 4]
```

```
>>> decreasing_array(5)
```

```
[4, 3, 2, 1, 0]
```

```
>>> random_array(10)
```

```
[0, 0, 4, 1, 7, 2, 9, 5, 9, 9]
```

```
>>> random_array(10)
```

```
[2, 0, 0, 5, 8, 2, 5, 9, 0, 8]
```

整列の実装

```
def bubble_sort(a):
```

```
def insertion_sort(a):
```

```
def merge_sort(a):
```

```
def quick_sort(a):
```

テストのための関数

整列関数 `sortf` を `mkarray(n)` によってできる配列に対してテストする.

```
def test_sort(sortf, mkarray, n):  
    global ncomp  
    ncomp = 0                      # 比較回数をリセット  
    a = mkarray(n)  
    start_time = time.time()  
    b = sortf(a)  
    print(f"長さ {n}")  
    print(f"実行時間 {time.time() - start_time:.6f}秒")  
    print(f"比較回数 {ncomp} 回")  
    sorted(b)                      # 整列できているか確認
```

授業で説明していない Python の言語機能を用いている.

- ▶ 高階関数: 関数 `test_sort` は, 関数 `sortf`, `mkarray` を引数として
いる.

テストの実行例

```
$ python -i sort.py
>>> test_sort(merge_sort, increasing_array, 1000)
長さ 1000
実行時間 0.005348 秒
比較回数 4932 回
>>> test_sort(quick_sort, random_array, 1000)
長さ 1000
実行時間 0.005446 秒
比較回数 11357 回
```

ある状況ではプログラムの実行が途中で打ち切られる場合がある

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．

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 25 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 ～ 2 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 29 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）

プログラムでテスト結果を集計しよう

CSV(Comma-Separated Values) 形式で実行結果を出力

整列関数 `sortf` を `mkarray(n)` によってできる配列に対してテストする.

CSV 形式 (カンマ区切り) で出力する.

```
def test_sort_csv(sortf, mkarray, n):  
    ...  
    print(f"{n},{ncomp},{time.time() - start_time:.4f}")
```

バブルソートを昇順配列に対してテストする関数の雛形

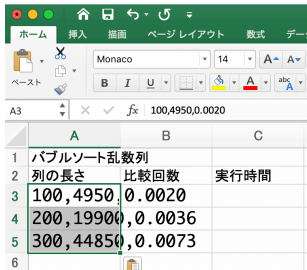
```
def test_bubble_increasing():  
    test_sort_csv(bubble_sort, increasing_array, 100)  
    test_sort_csv(bubble_sort, increasing_array, 200)  
    test_sort_csv(bubble_sort, increasing_array, 300)
```

実行例 :

```
% python -i sort.py  
>>> test_sort_csv(bubble_sort, random_array, 1000)  
1000,499500,0.1099  
>>> test_bubble_increasing()  
100,4950,0.0020  
200,19900,0.0036  
300,44850,0.0073
```

プログラムでテスト結果を集計しよう

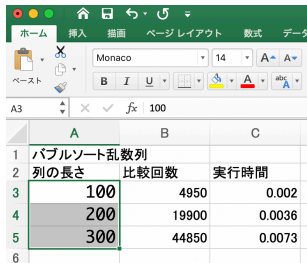
CSV 形式の出力を Excel ファイルにカット&ペースト



The screenshot shows an Excel spreadsheet with the following data:

	A	B	C
1	バブルソート乱数列		
2	列の長さ	比較回数	実行時間
3	100,4950,0.0020		
4	200,19900,0.0036		
5	300,44850,0.0073		
6			

The formula bar shows the formula for cell A3: `=100,4950,0.0020`.



The screenshot shows the same Excel spreadsheet after editing. The data in column A is now separated by commas:

	A	B	C
1	バブルソート乱数列		
2	列の長さ	比較回数	実行時間
3	100	4950	0.002
4	200	19900	0.0036
5	300	44850	0.0073
6			

The formula bar shows the formula for cell A3: `=100`.

- ▶ データ → 区切り位置 → 次へ
- ▶ 区切り文字: カンマ → 完了