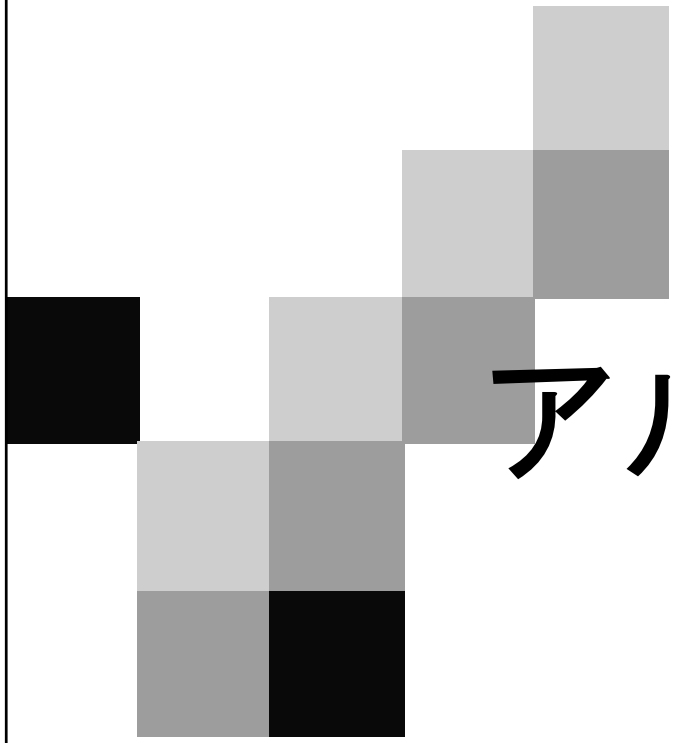


アルゴリズムとデータ構造
整列のアルゴリズム

塩浦昭義
情報科学研究科　准教授
shioura@dais.is.tohoku.ac.jp
http://www.dais.is.tohoku.ac.jp
/~shioura/teaching

整列（ソート, Sorting）

- 全順序つきの集合の要素を順番に並べること
- 例1：数字の整列（小さい数から大きい数へ）

  51、23、46、9、30

  ⇒ 9、23, 30, 46、51
- 例2：名前の五十音順による整列

  しおうら、たなか、とくやま、すずき

  ⇒ しおうら、すずき、たなか、とくやま
- この講義では数字の整列を扱う

ソートのアルゴリズム
Algorithms for Sorting

- 様々なアルゴリズムが存在
  - バブルソート
  - 挿入ソート
  - ヒープソート
  - マージソート
  - クイックソート
  - などなど

バブルソート, 挿入ソート: 最悪計算時間 $O(n^2)$

ヒープソート, マージソート: 最悪計算時間 $O(n \log n)$

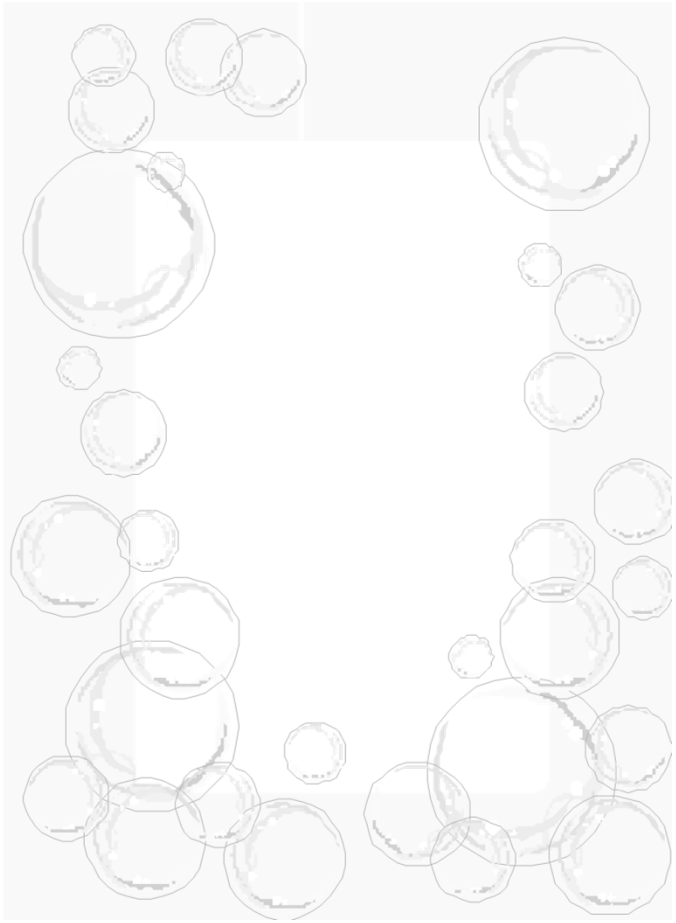
クイックソート: 平均計算時間 $O(n \log n)$

などなど: 最悪$O(n^2)$, 実用的には高速

バブルソート

Bubble Sort

（教科書87ページ）



バブルソートの動き（その1）

Behavior of Bubble Sort

A[i] = 配列の i 番目の要素 (i = 1, 2, …, n)

1回目の反復

■A[7]とA[8]の大小を比較

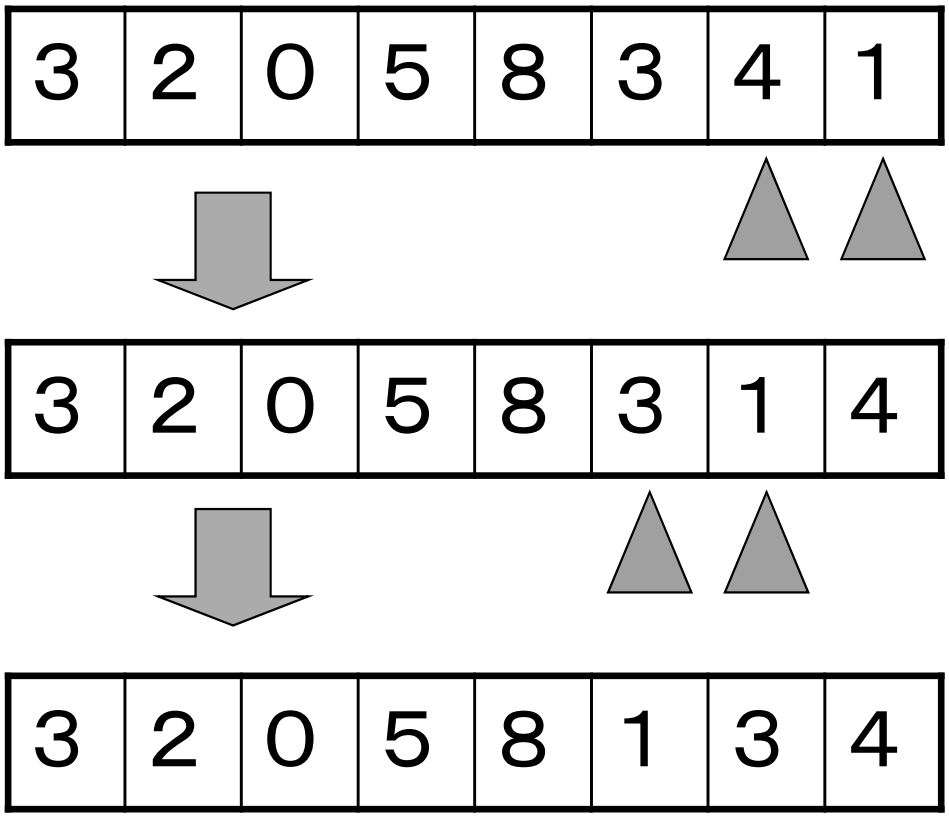
A[7] > A[8]

⇒ 2つの要素を入れ替え

■A[6]とA[7]の大小を比較

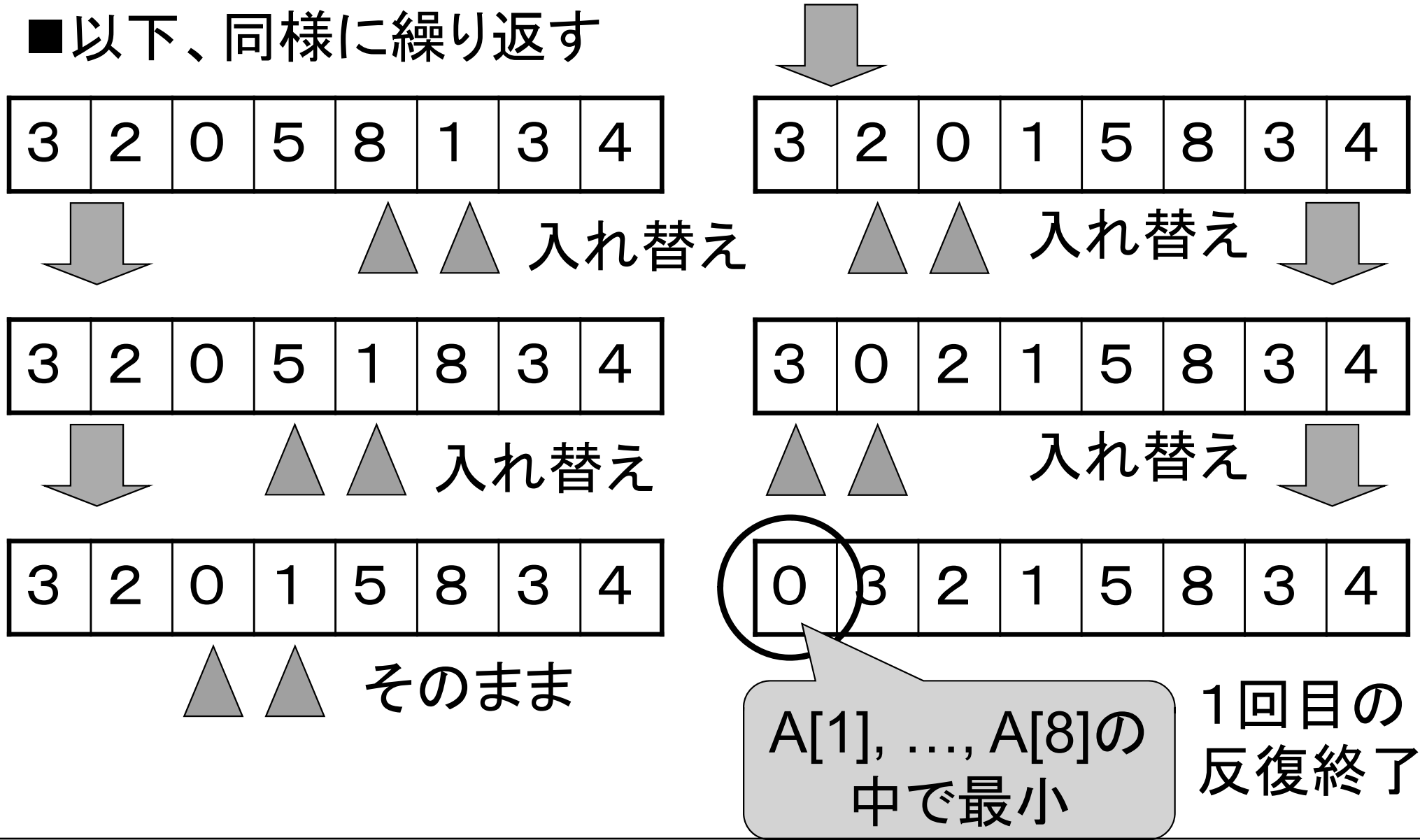
A[6] > A[7]

⇒ 2つの要素を入れ替え



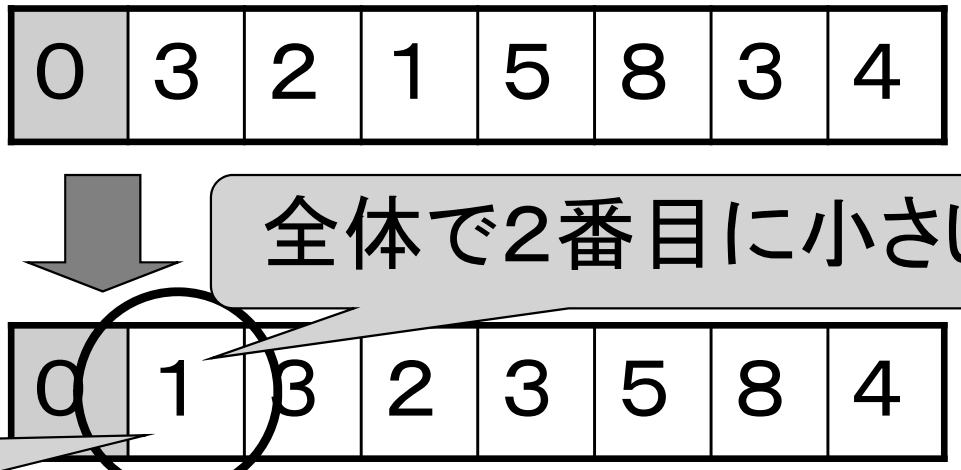
バブルソートの動き（その2）

■以下、同様に繰り返す

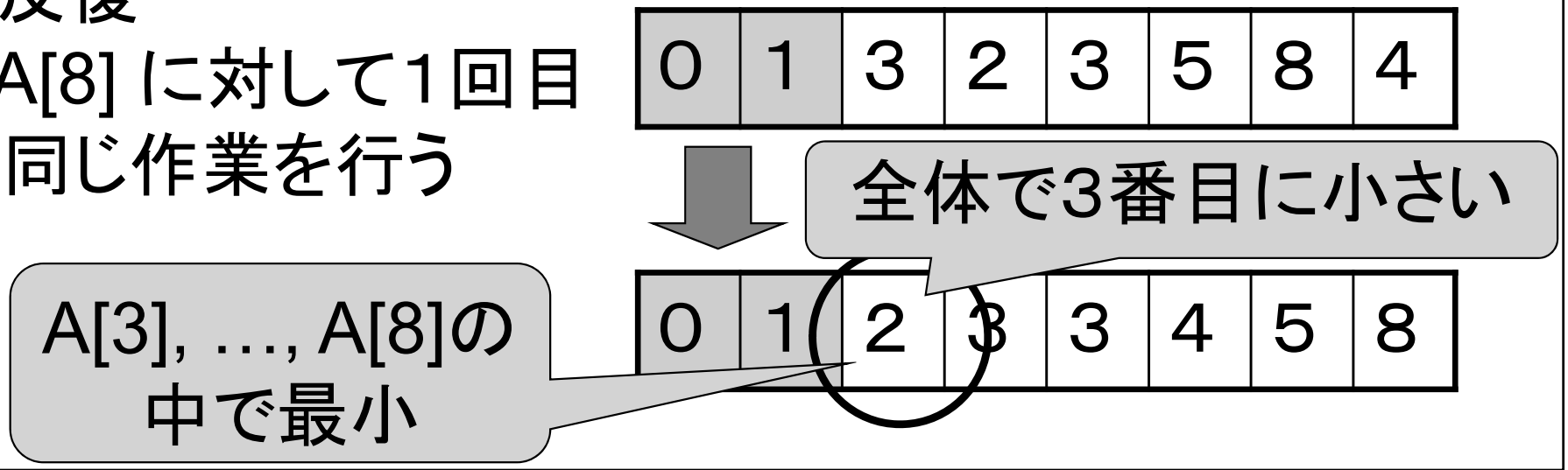


バブルソートの動き（その3）

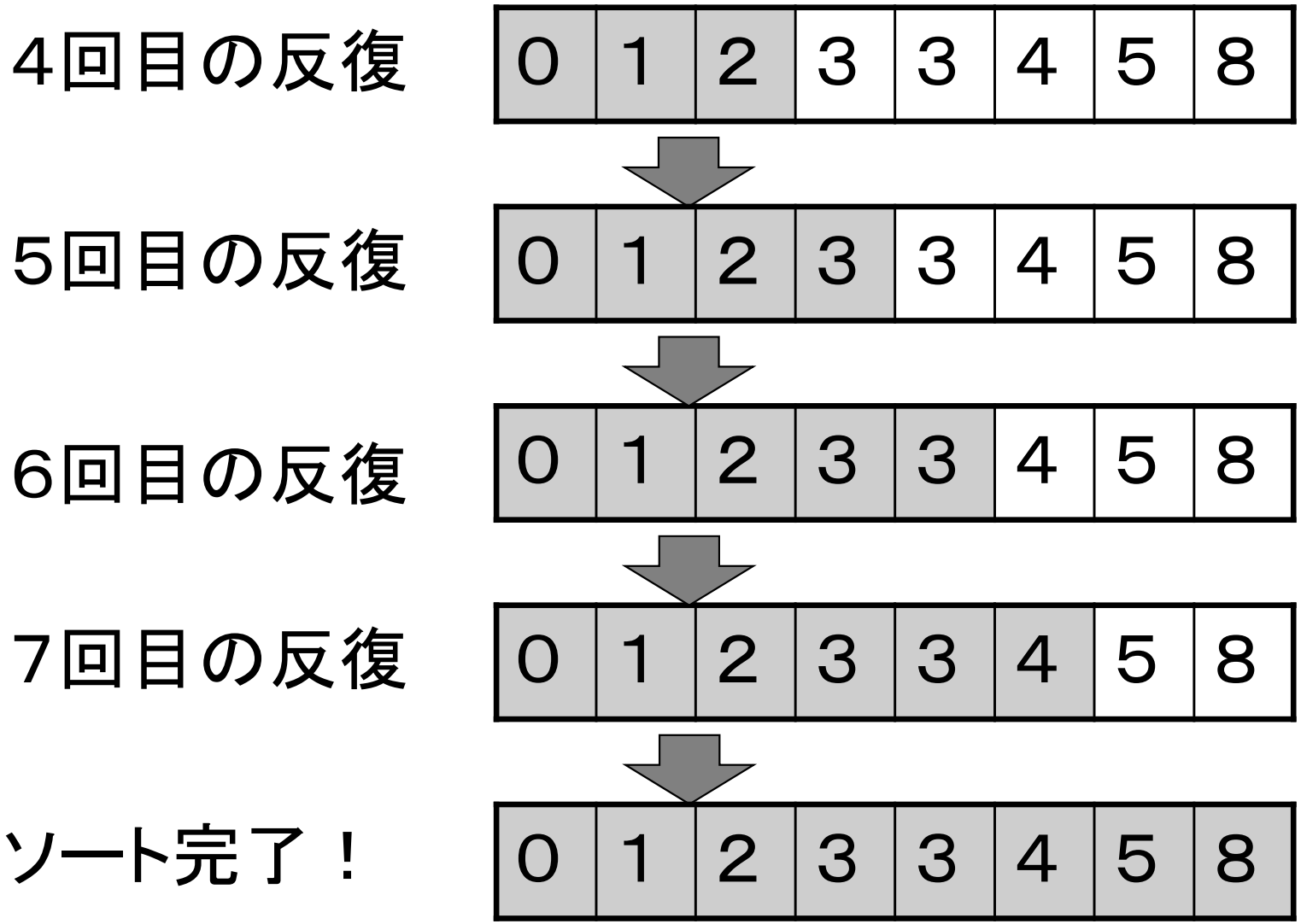
2回目の反復
A[2], …,A[8] に対して1回目の反復と同じ作業を行う



3回目の反復
A[3], …,A[8] に対して1回目の反復と同じ作業を行う



バブルソートの動き（その４）



バブルソートの計算時間（その１）
Time Complexity of Bubble Sort

- 一回目の反復
  A[n-1]とA[n]の比較、入れ替え
  A[n-2]とA[n-1]の比較、入れ替え
  ⋮
  A[1]とA[2]の比較、入れ替え
  ⇒　c (n-1) 時間　（c は定数）
- ２回目の反復：　c (n – 2) 時間
- ３回目の反復：　c (n – 3) 時間

バブルソートの計算時間(その2)

- 一般に、k 回目の反復 : c(n − k) 時間

∴ バブルソートの計算時間は

$$c \times \{(n-1) + (n-2) + \cdots + 2 + 1 + 0\} = c(n-1)(n-2)/2 = O(n^2)$$

※途中でソートが終わっていたら
以降の反復を省略できる
(例:4回目の反復以降)

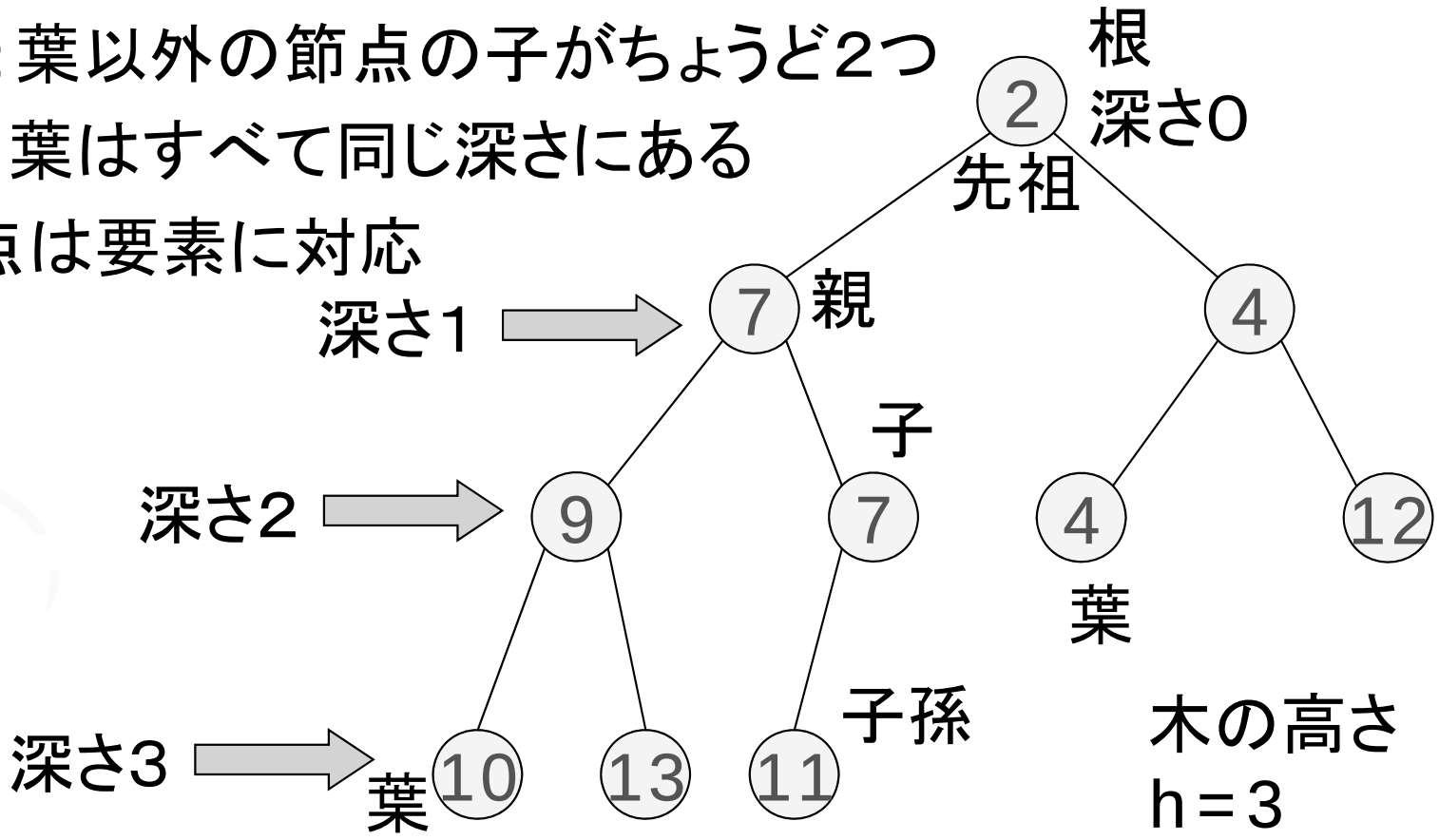
ヒープソート

Heap Sort

（教科書９４ページ）

ヒープ

- 最も小さい（大きい）要素を常に削除するときに使うデータ構造
  - 新しい要素の追加，最小要素の削除は O(log n) 時間
- 2分木により表現される
  - 2分木：葉以外の各節点が高々2つの子をもつ根付き木
  - 完全2分木：葉以外の節点の子がちょうど2つ
    葉はすべて同じ深さにある
- 2分木の各節点は要素に対応



ヒープソート

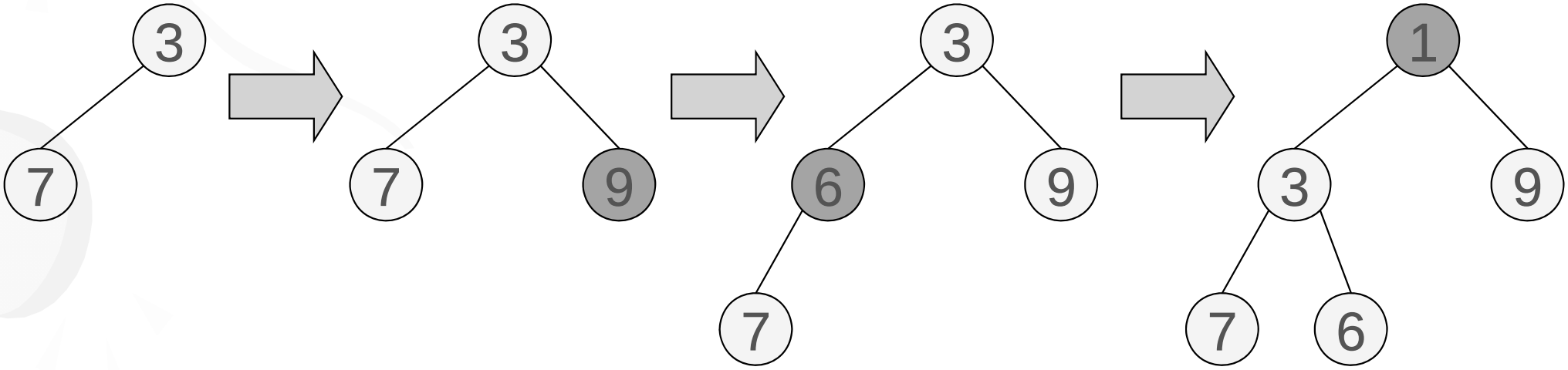
- ヒープを使った整列アルゴリズム
- アルゴリズムの手順

  (1)与えられた数の集合を順番にヒープに追加(n 回)

  (2)ヒープから最小要素を次々に削除，並べる(n 回)

  ➔ O(n log n) 時間

7，3，9，6，1 を順に追加



ヒープソート

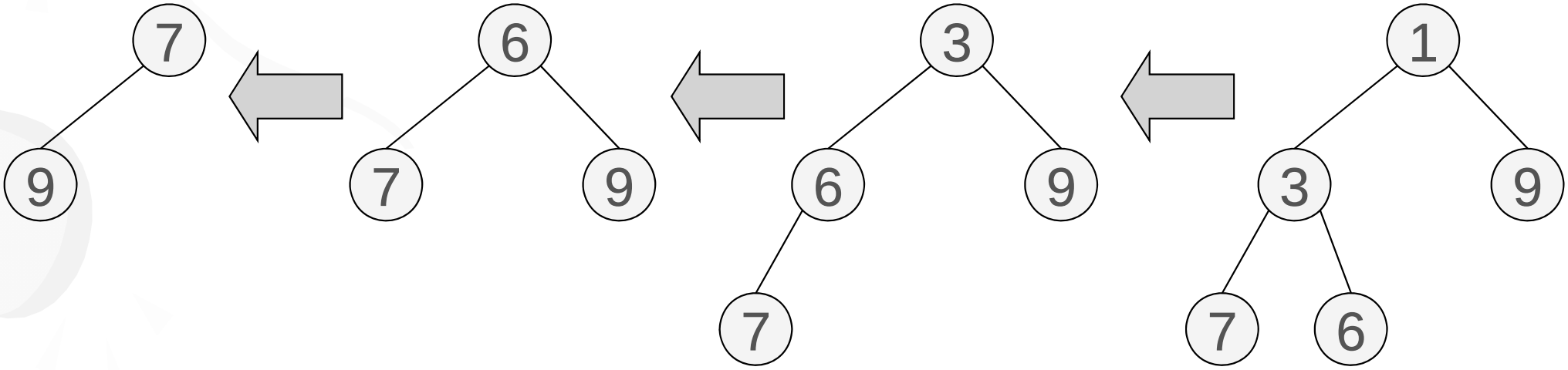
- ヒープを使った整列アルゴリズム
- アルゴリズムの手順

  （1）与えられた数の集合を順番にヒープに追加（n 回）

  （2）ヒープから最小要素を次々に削除，並べる（n 回）

  ➔ O(n log n) 時間

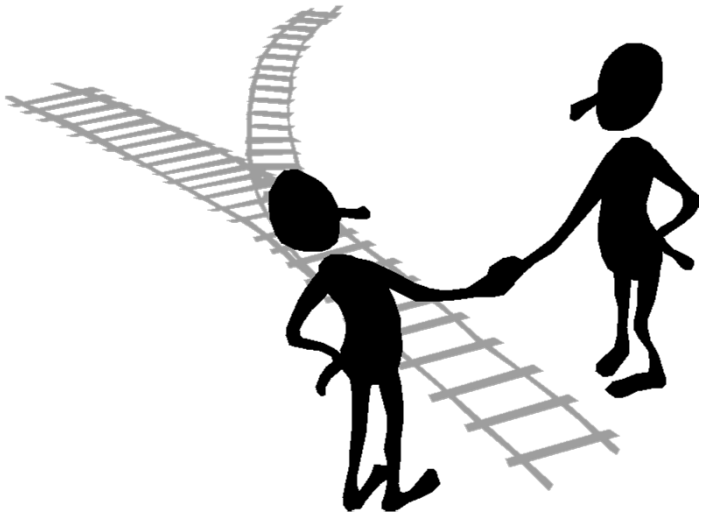
最小要素を順に削除　1　3　6　7　9



マージソート

Merge Sort

（教科書120ページ）



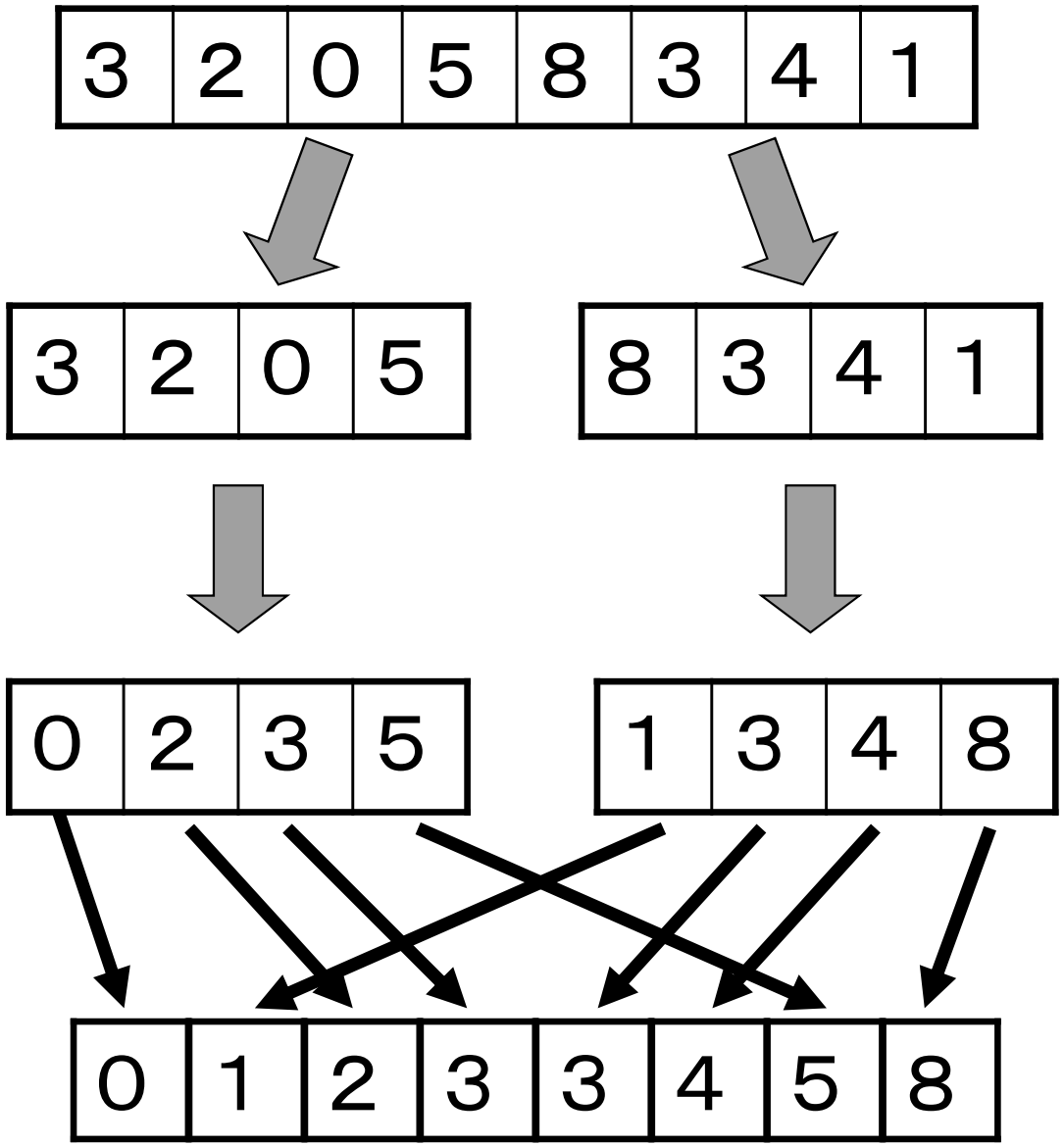
マージソートのアイディア
Idea of Merge Sort

- アイディア: 分割統治法
(divide-and-conquer method)

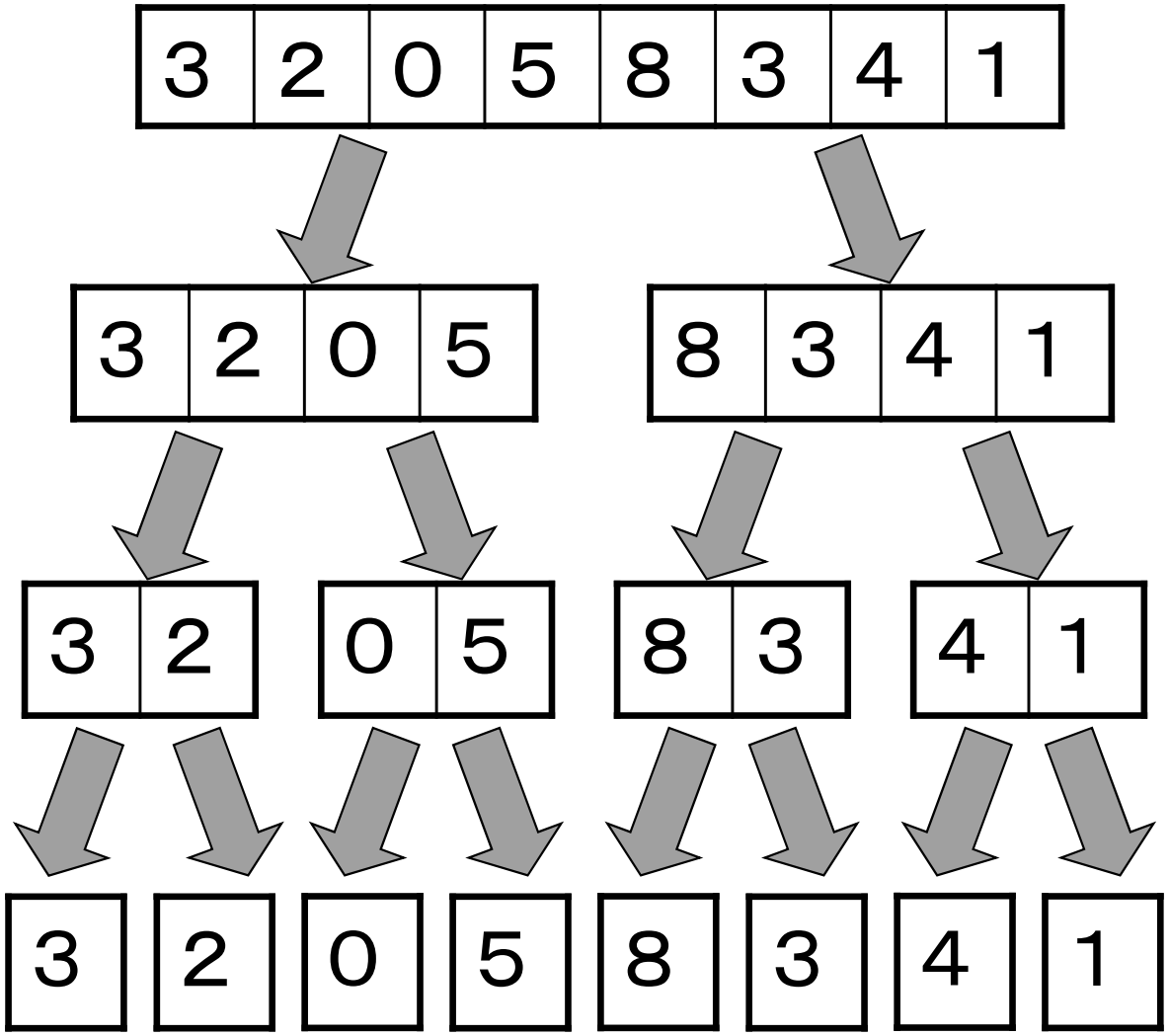
①与えられた配列を2分割

②2分割された配列をそれぞれ再帰的にソート

③ソートされた2つの配列をマージ（統治）



マージソートの動き（前半）
Behavior of Merge Sort

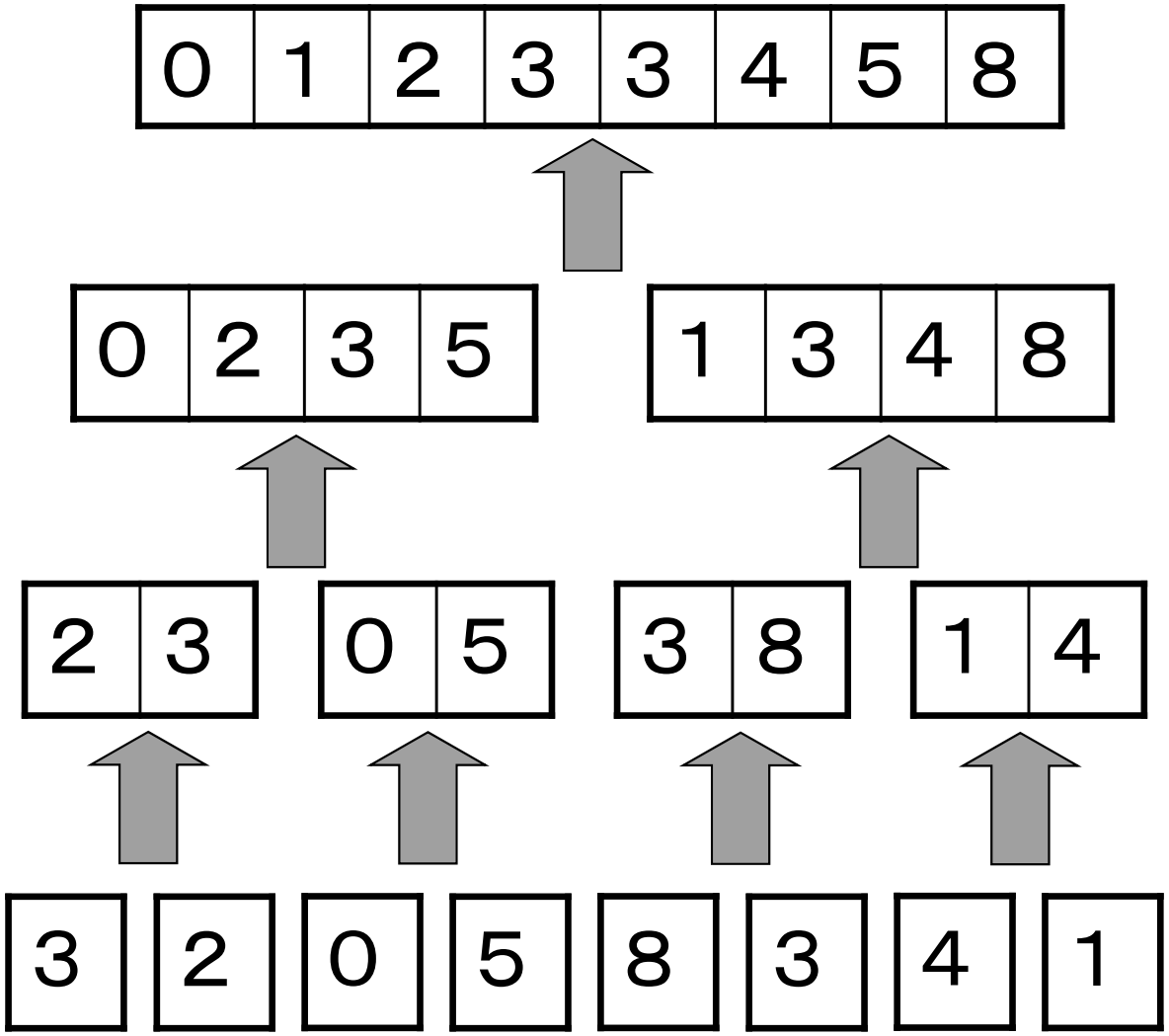


配列を2分割
（大きさ：8→4）

配列を2分割
（大きさ：4→2）

配列を2分割
（大きさ：2→1）

マージソートの動き（後半）



ソート列をマージ
（大きさ：4→8）

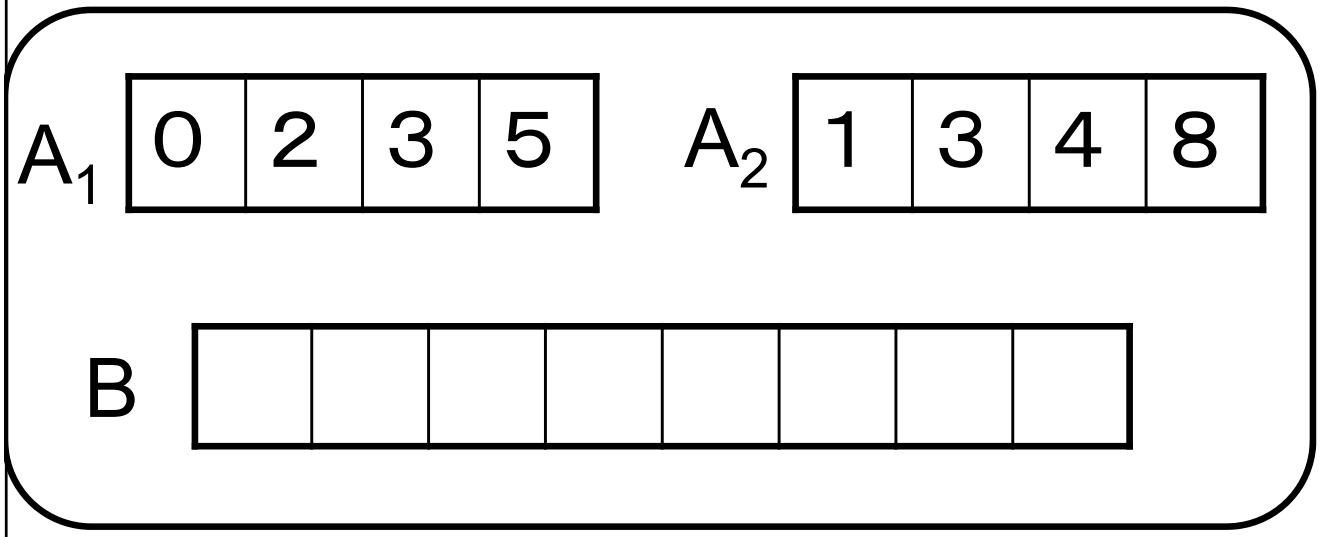
ソート列をマージ
（大きさ：2→4）

ソート列をマージ
（大きさ：1→2）

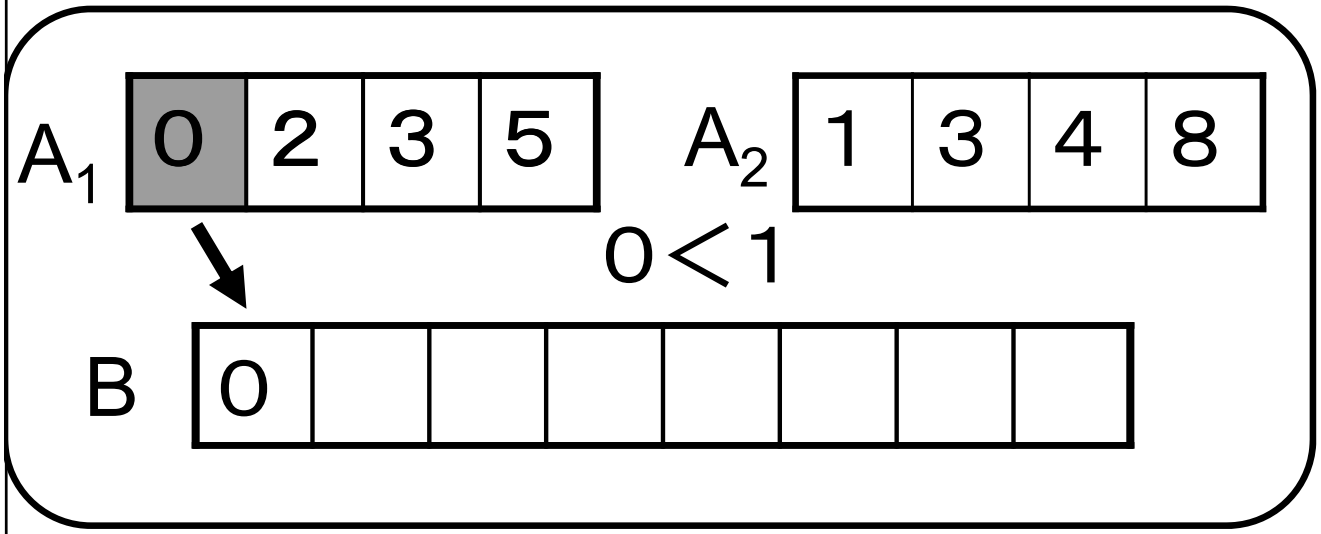
ソート列のマージ（その１）

Merge of Sorted Sequences

ソート列の併合（マージ）は線形時間で出来る

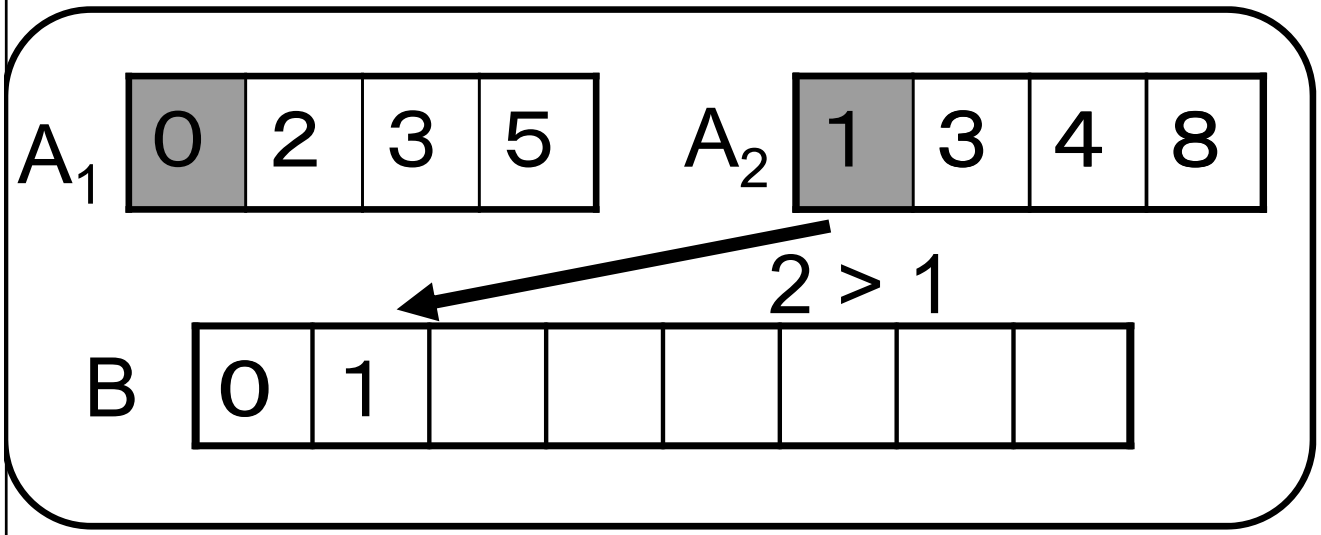


マージした結果を格納する配列Bを用意

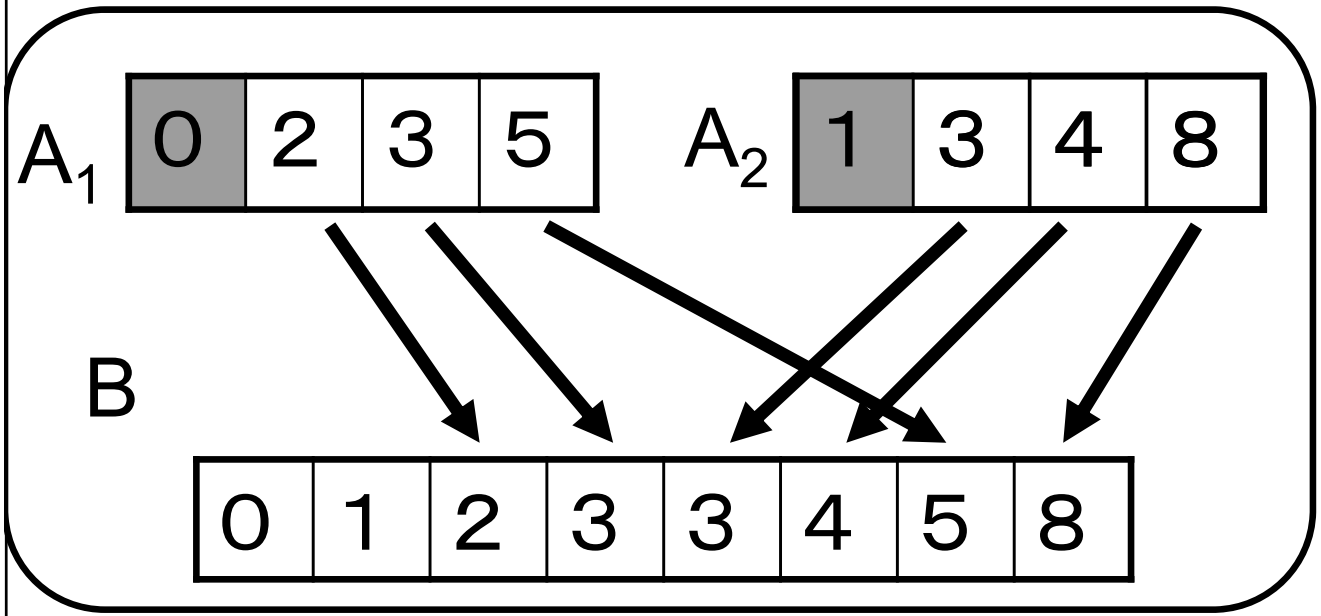


A_1 と A_2 の先頭の数字を比較
小さいほうを B の空欄の先頭へ移動

ソート列のマージ（その2）



A_1 と A_2 の先頭の数字を比較
小さいほうを B の空欄の先頭へ移動



この作業を繰り返す

ソート列のマージ（その3）

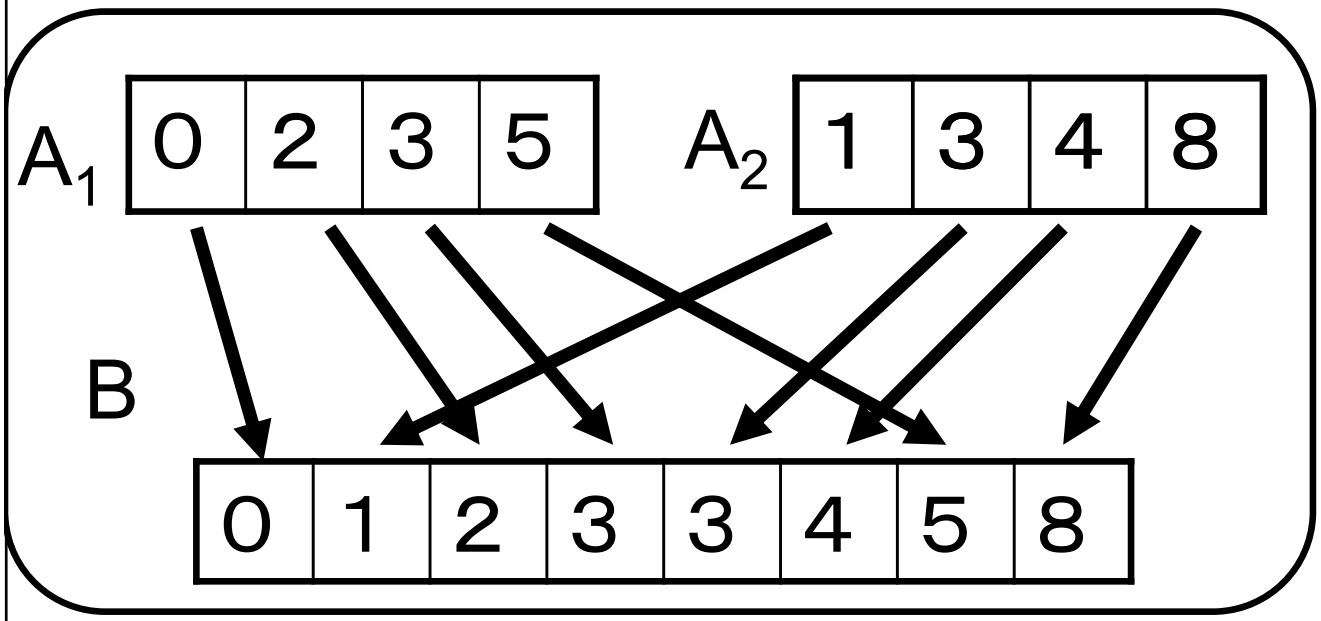
計算時間の解析

$n_1 = A_1$ の要素数、$n_2 = A_2$ の要素数

A_1 と A_2 の要素ひとつひとつに対して、

他の要素との比較
B に要素を書き込む

} 定数時間 c



c (n_1+n_2) 時間
で実行可能

マージソートの計算時間（その1）

Time Complexity of Merge Sort

T(n) = n 個の要素のソートの時間（n は2のべき乗と仮定）

①与えられた配列を2分割

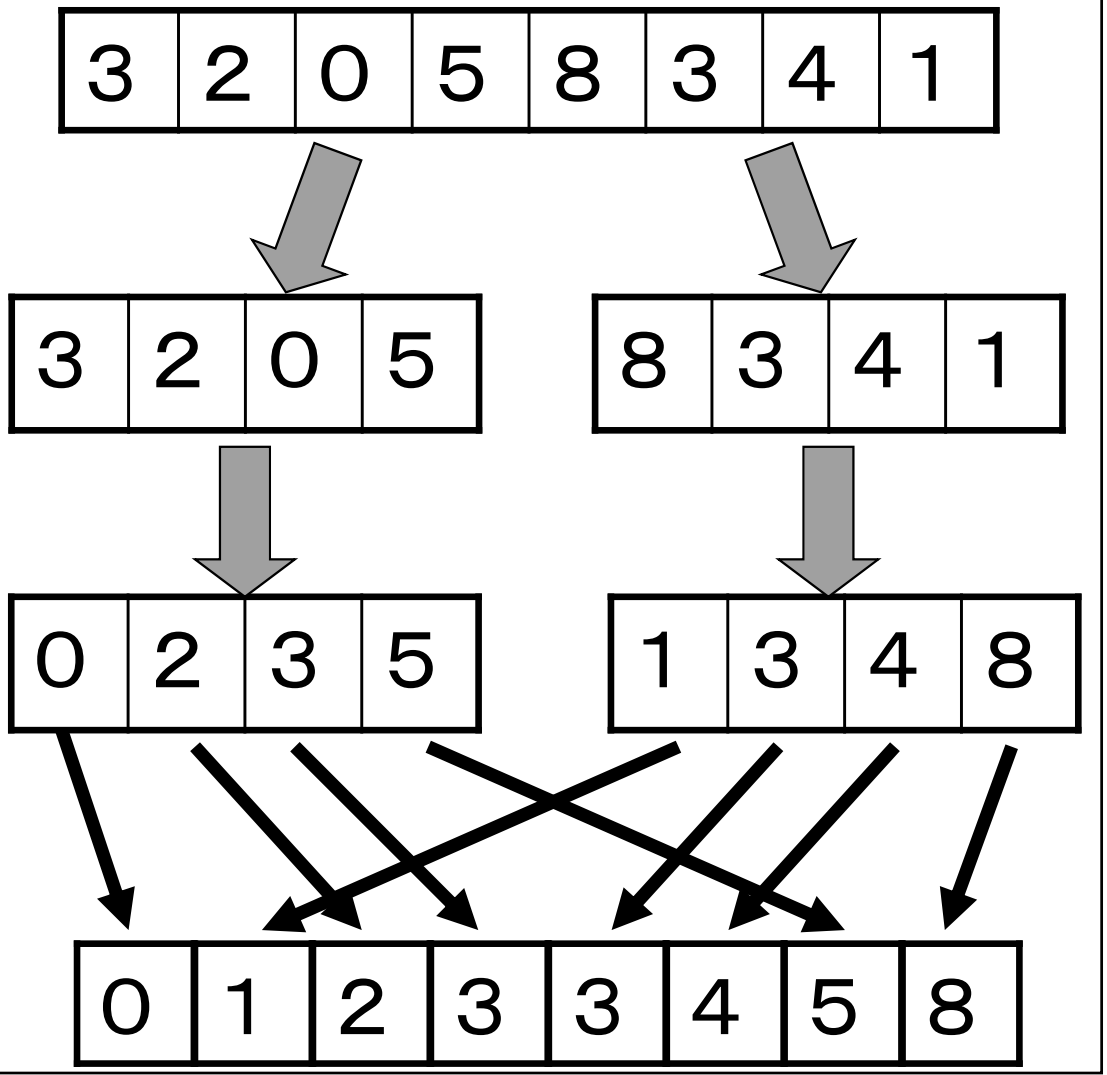
c’ n 時間

②2分割された配列をそれぞれ再帰的ソート

T(n/2) × 2　時間

③ソートされた2つの配列をマージ （統治）

c n 時間



マージソートの計算時間(その2)

T(n) = n 個の要素のソートの時間

T(n) = 2 T(n/2) + (c+c') n　が成り立つ

⇒ 解は　$T(n) = (c+c') n \log_2 n = O(n \log n)$

※ n が2のべき乗でないときも，解析を少し修正すれば O(n log n) の時間計算量が証明できる

クイックソート

Quick Sort

（教科書96ページ）

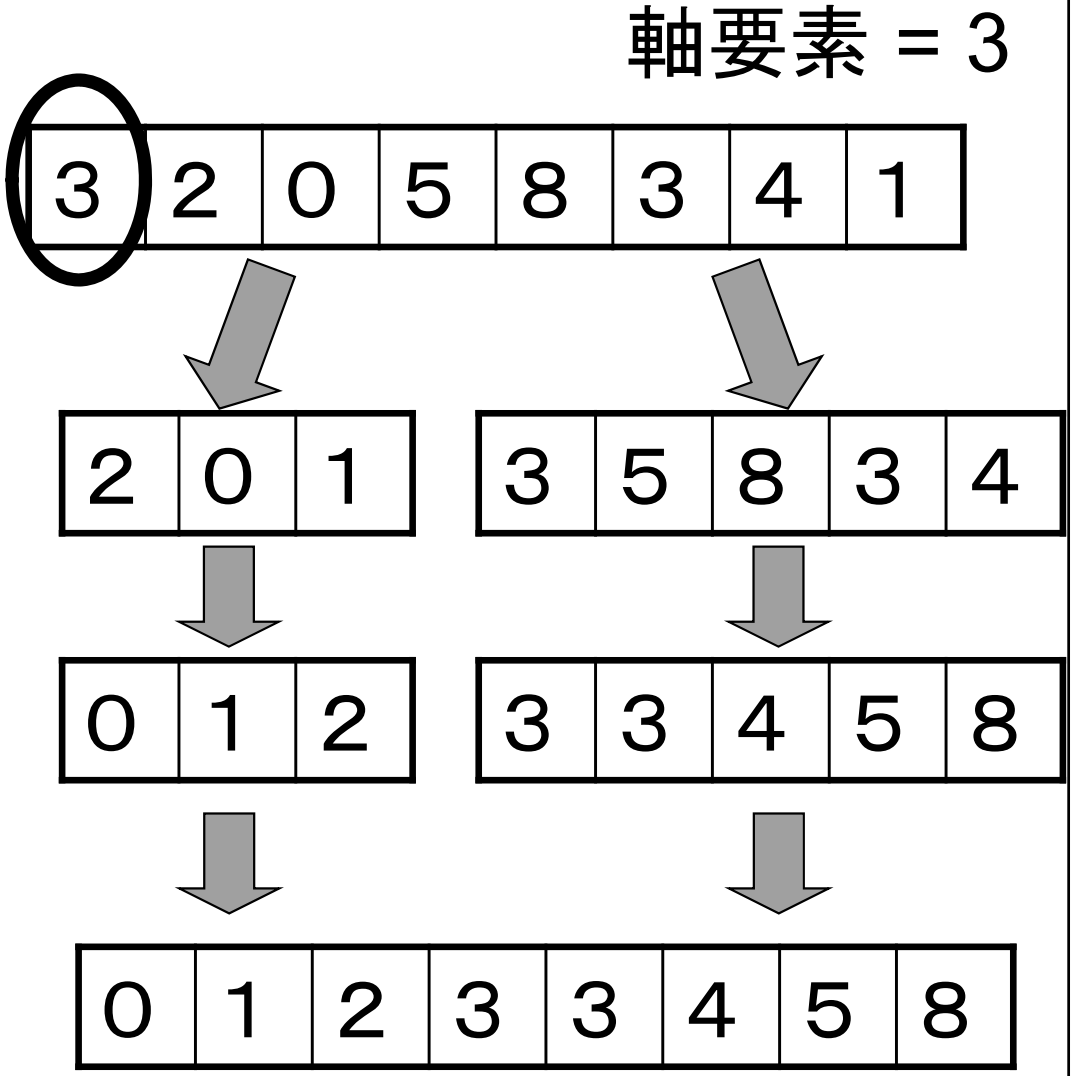
クイックソートのアイディア
Idea of Quick Sort

①A[1], …, A[n]から
ひとつの値（軸要素, pivot）
　を選ぶ

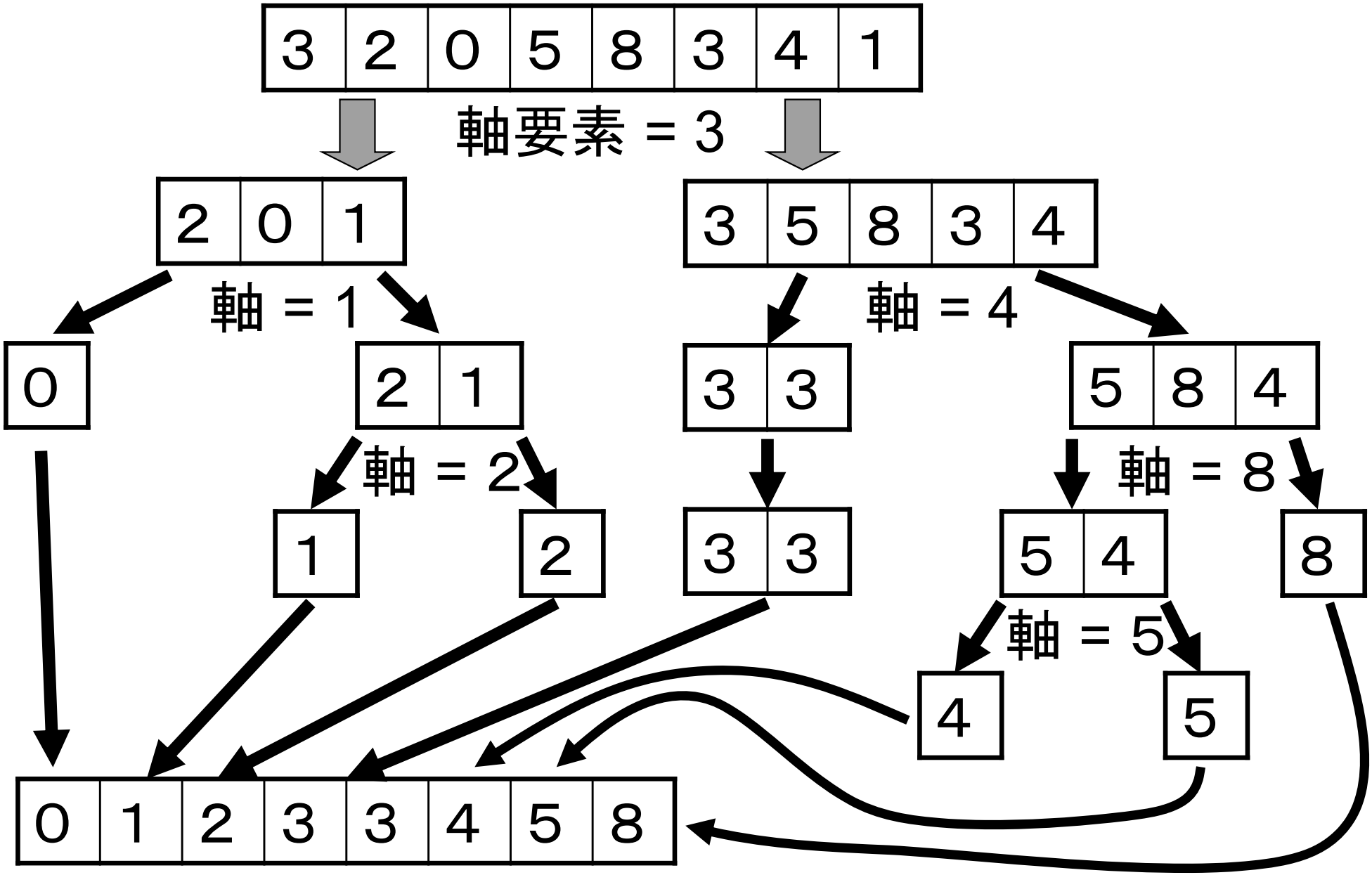
② 軸要素未満の要素と
それ以外に分割

③2分割された配列を
それぞれ再帰的にソート

④ソートされた2つの配列
　をつなげる



クイックソートの動き Behavior of Quick Sort



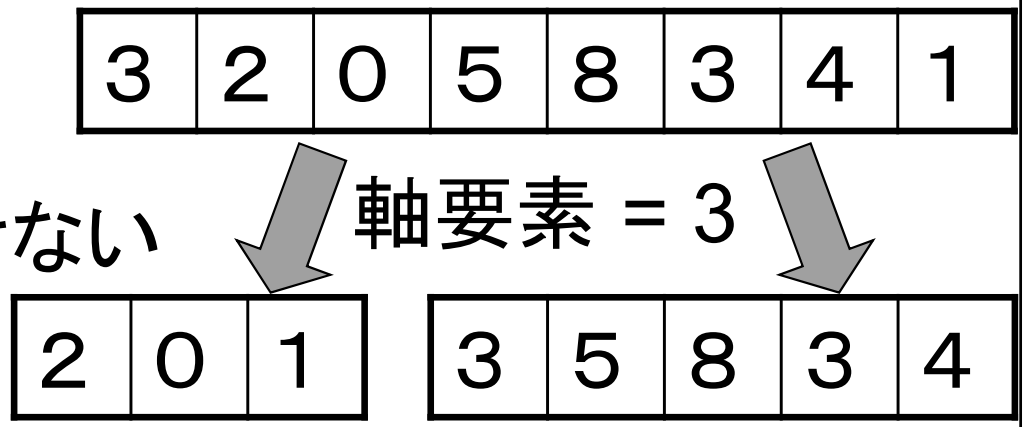
軸要素の選び方(その1)

How to Choose Pivot

良い選び方:

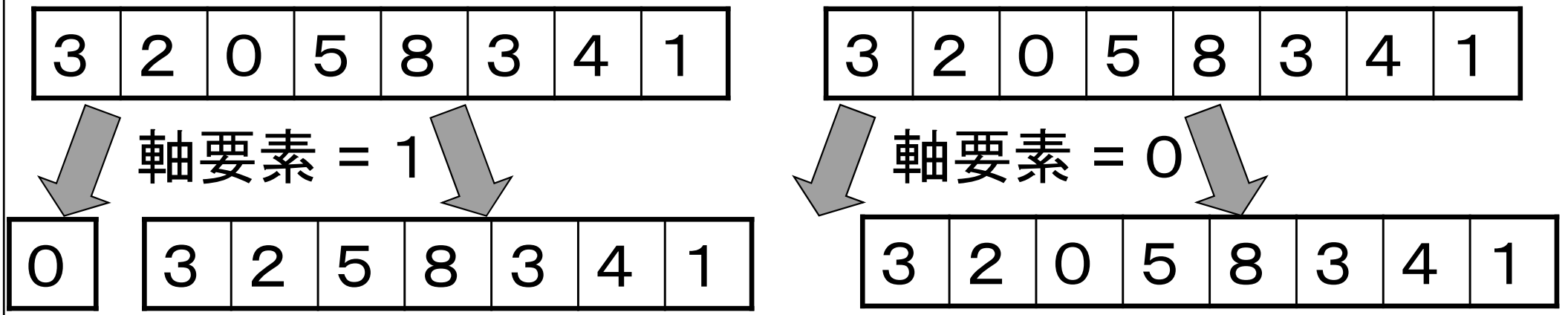
配列をほぼ二等分する

軸要素の選択に時間をかけない



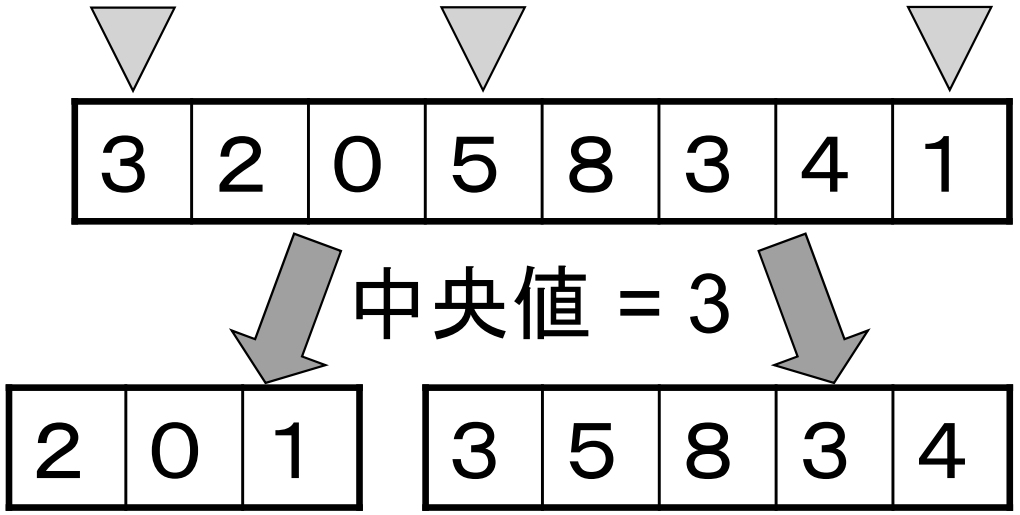
悪い選び方:

2分された配列の大きさがアンバランス



軸要素の選び方(その2)

よく使われる選び方:
(a) ランダムにひとつ選ぶ
(b) 左端、右端、真中の3要素の中央値

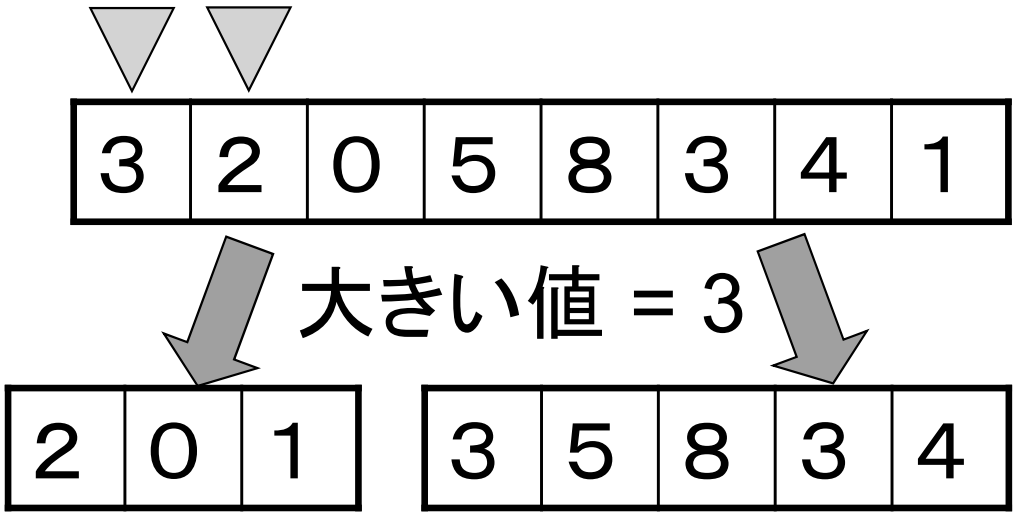


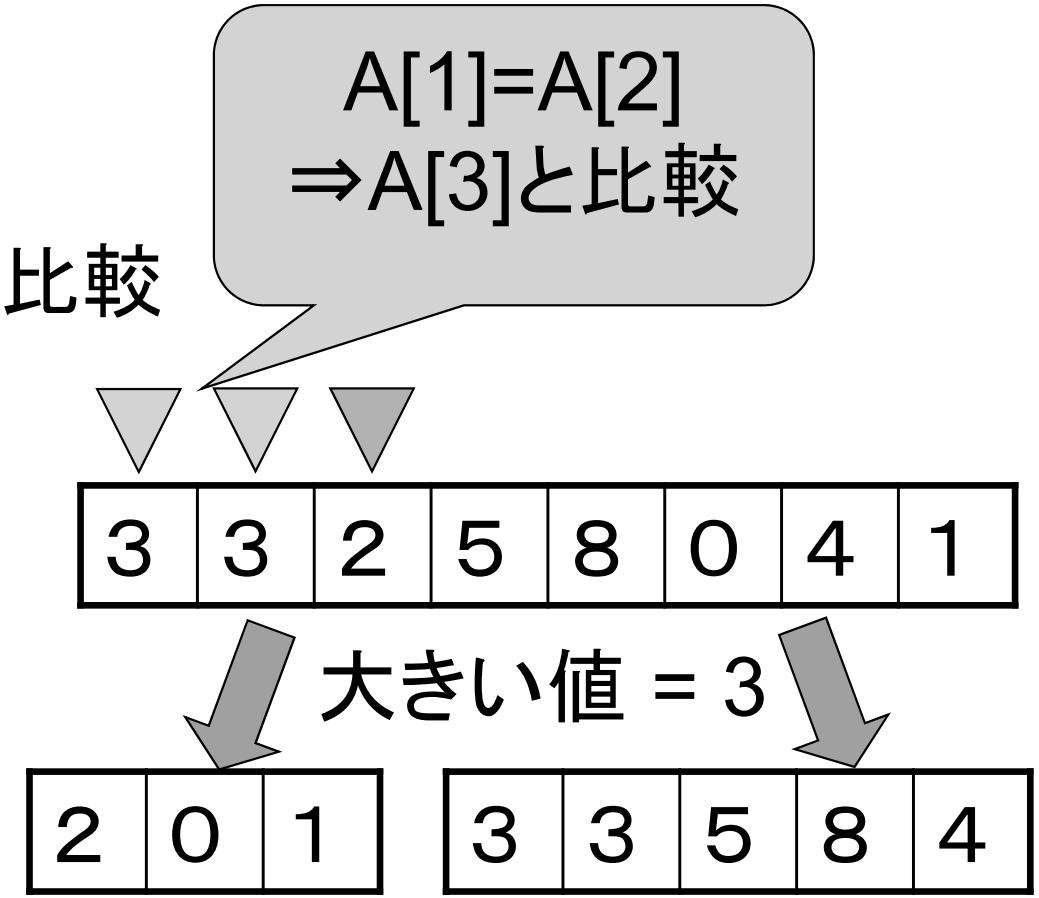
軸要素の選び方（その3）

よく使われる選び方：

（c）A[1], A[2]のうち、大きい値

　　同じ場合は次の異なる値と比較





クイックソートの計算時間

T(n) = n 個の要素のソートの計算時間

一般の場合 : k 個と n – k 個に分割される (1 < k < n)

⇒ $T(n) \leqq T(k) + T(n-k) + c\,n$ (c: 定数)

⇒ 帰納法により、解は $T(n) \leqq 2c\,n^2$

∴最悪計算時間は $O(n^2)$

実際には，数列がほぼ半分に分割される

⇒ 実用上の計算時間は$O(n \log n)$ に近い

平均時間も $O(n \log n)$ （解析はちょっと難しい）