アルゴリズムとデータ構造

バケットソート，基数ソート
ソートの下界，第p要素の選択

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

バケットソート
Bucket Sort

- 整列の対象となっている整数の範囲が事前にわかっている（例：各入力値a_iが$0 \leqq a_i \leqq m-1$を満たす）
- 整数の範囲(mの大きさ)があまり大きくない

→ バケットソートにより高速に整列が可能
時間計算量 O(m+n)

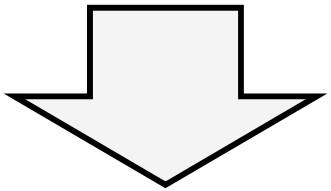
バケット（bucket）

＝ バケツ

バケットソートの利点

- バケットソートでは，同じ数字が多く出てくる可能性大
  - 同じ数字であれば，元々の順番に並んで欲しい
  - 例：学籍番号順
- バケットソートでは，同じ数字は元の順番通りに並べられる

学籍番号1111	学籍番号1234	学籍番号1300	学籍番号1414
得点　8	得点　5	得点　5	得点　8



学籍番号1234	<	学籍番号1300	学籍番号1111	<	学籍番号1414
得点　5		得点　5	得点　8		得点　8

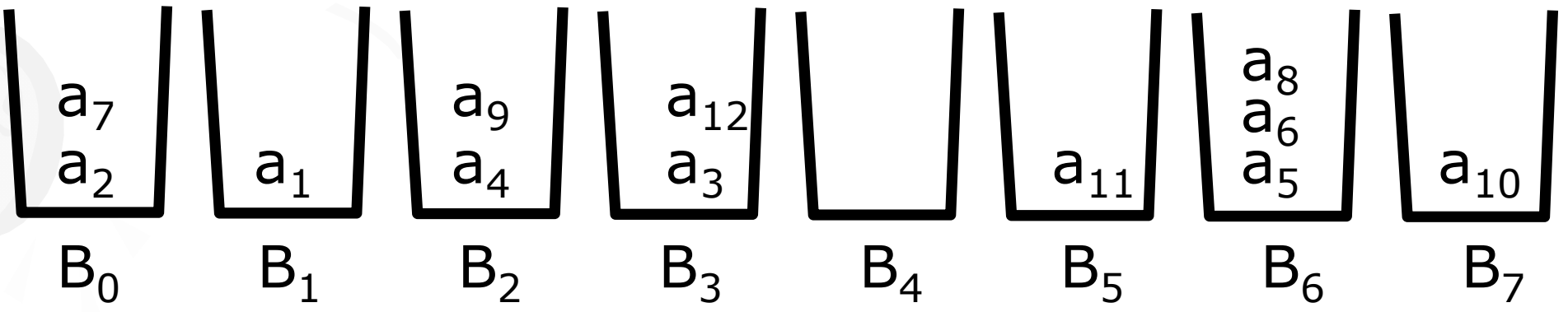
バケットソートのアイディア
Idea of Bucket Sort

$0 \leqq a_i \leqq m-1$を満たす整数$a_1, a_2, \ldots, a_n$ を整列したい

- 各 j (0≦ j ≦m-1)に対し，空のバケツB_jを用意
- 各 a_i に対し，$a_i=j$ ならば a_i をバケツB_jに入れる

0以上7以下の整数

1	2	3	4	5	6	7	8	9	10	11	12
1	0	3	2	6	6	0	6	2	7	5	3



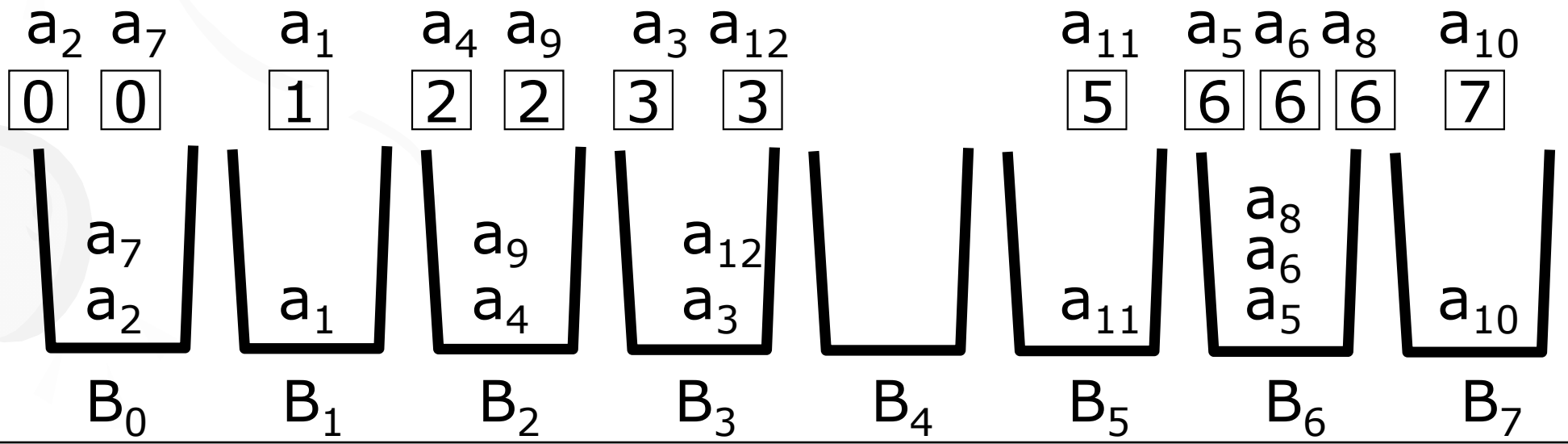
バケットソートのアイディア（続き）
Idea of Bucket Sort

- バケツの中身をB_{m-1}, B_{m-2}, …, B_1の順番に取り出す
- 取り出すときは，後に入れた要素を先に取り出す

➔ソートが完了

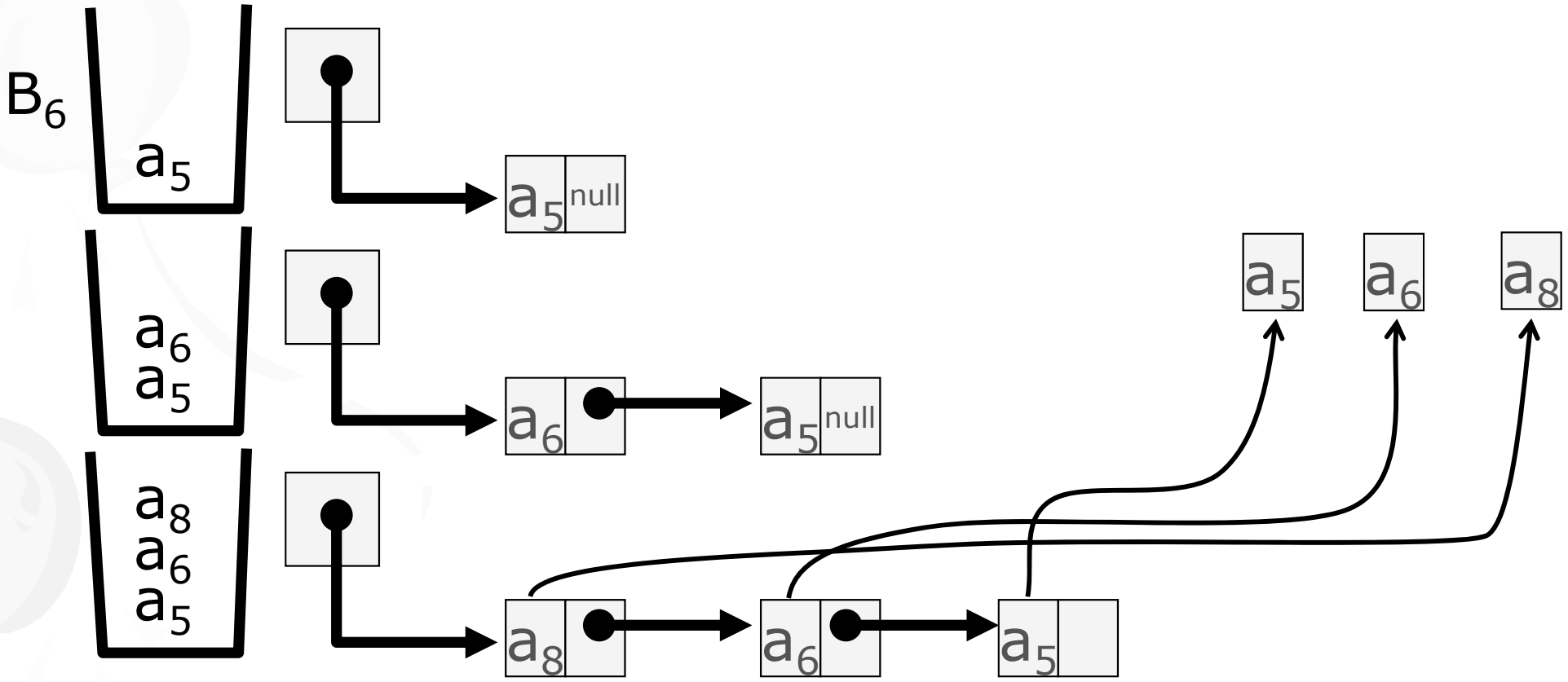
[重要]同じバケツの中の要素は
元の順番通りに並べられている

1	2	3	4	5	6	7	8	9	10	11	12
1	0	3	2	6	6	0	6	2	7	5	3



バケットソートの実現
Implementation of Bucket Sort

- 各バケツは連結リストで実現
- 中身を取り出すときに，後ろから前に並べる
  - ➔ 同じバケツの要素は元の順番通りに並ぶ



バケットソートの計算時間
Time Complexity of Bucket Sort

- m個のバケツを準備➔$O(m)$時間
- n個の要素をバケツに挿入➔$O(1) \times n = O(n)$
- バケツに入れた要素を取り出す➔O(m+n)時間

∴ 全体でO(m + n) 時間
　空間計算量もO(m + n)

※ 整数の範囲が狭いときには有効
　m が大きいときは時間計算量も空間計算量も膨大

基数ソート：カードを並べる方法(p91-94)
How to Sort Cards: Radix Sort

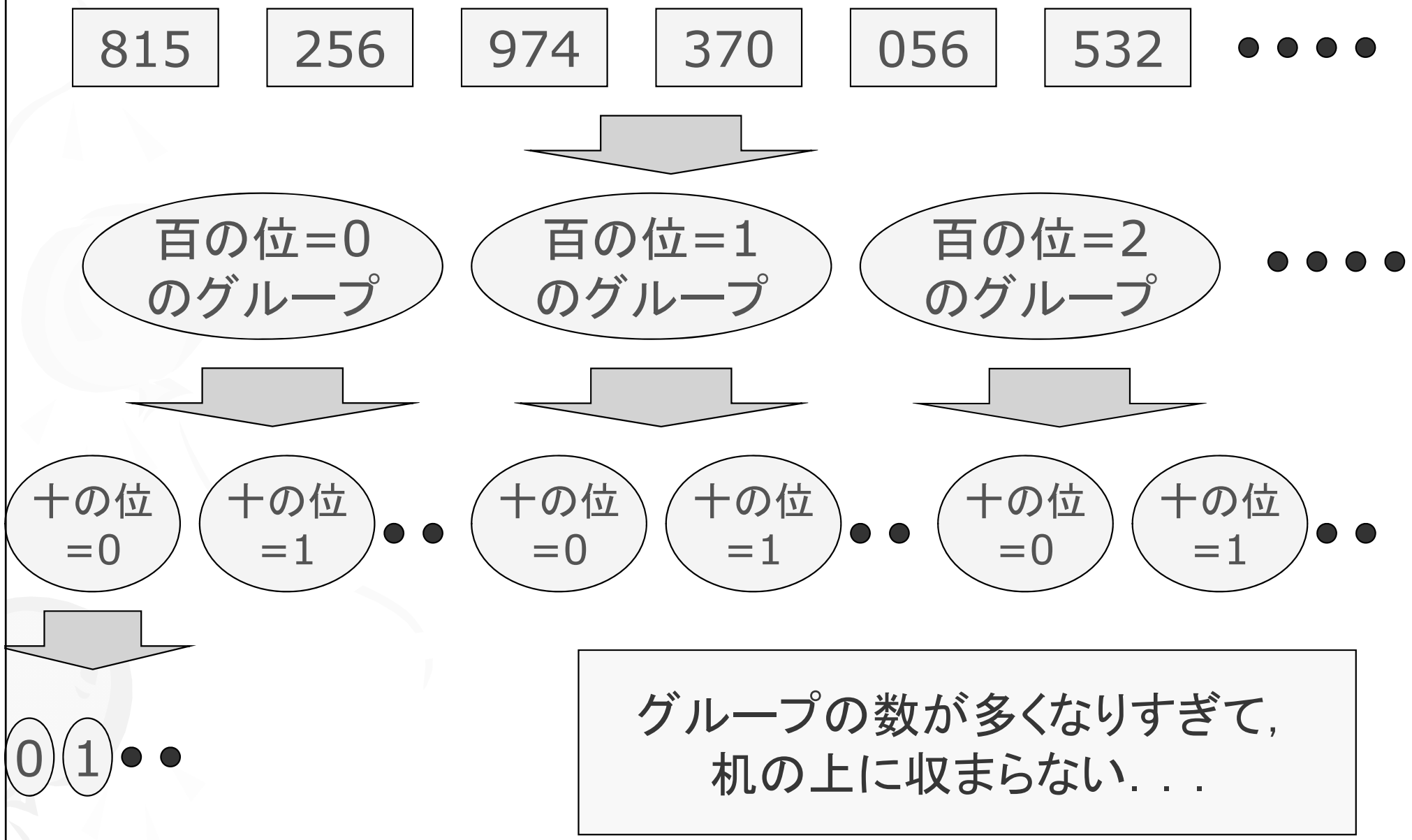
- 例：3桁の数字からなる学籍番号の書かれたカードを番号順に（机の上で）並べたい

815	256	974	370	056	532	●●●●

- よく使われる方法：
  - まず百の位の値によってグループ分け
  - 次に各グループを十の位によってグループ分け
  - 最後に各グループを一の位によってソート
  - ソートされたカードをまとめる

机の上で実現可能か？

カードを並べる
Sorting Cards



基数ソートのアイディア
Idea of Radix Sort

- よく使われる方法:
  - まず百の位の値によってグループ分け
  - 次に各グループを十の位によってグループ分け
  - 最後に各グループを一の位によってソート
  - ソートされたカードをまとめる

- 基数ソートの手順:
  - まず一の位の値によってバケットソート
  - 次に十の位によってバケットソート
  - 最後に百の位によってバケットソート

なぜこの方法で
ソートが
できるのか？

基数ソートの例
Example of Radix Sort

- 簡単のため，各桁の数字は0,1,2,3のみとする

123	013	322	102	021	311	222	110	200

まず一の位の値によってバケットソート

110	200	021	311	322	102	222	123	013

次に各グループを十の位によってバケットソート

200	102	110	311	013	021	322	222	123

バケットソートを利用

➔十の位が同じ数字ならば，元の順番（一の位に関する昇順）通りに並ぶ

➔下2桁に関して昇順に並んでいる

基数ソートの例
Example of Radix Sort

次に各グループを十の位によってバケットソート

200	102	110	311	013	021	322	222	123

下2桁に関して昇順に並んでいる

最後に各グループを百の位によってバケットソート

013	021	102	110	123	200	222	311	322

バケットソートを利用

➔百の位が同じ数字ならば，元の順番（下2桁に関する昇順）通りに並ぶ

➔下3桁に関して昇順に並んでいる

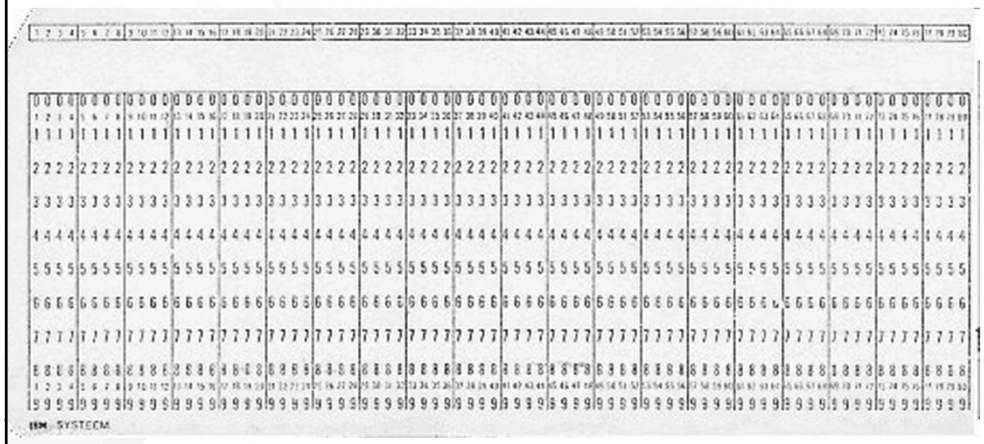
基数ソートの計算時間
Time Complexity of Radix Sort

- 桁数が K 以下の n 個の（非負）整数をソートする場合
  - バケットソートを K 回実行
  - 一回のバケットソートでは10個(0,1,2,..,9)個のバケツを使用➔ O(10 + n) =O(n)時間
  - 全体では O(n) x K = O(Kn)時間
- K 桁以下の m 進数をソートする場合は O(K(m + n)) 時間

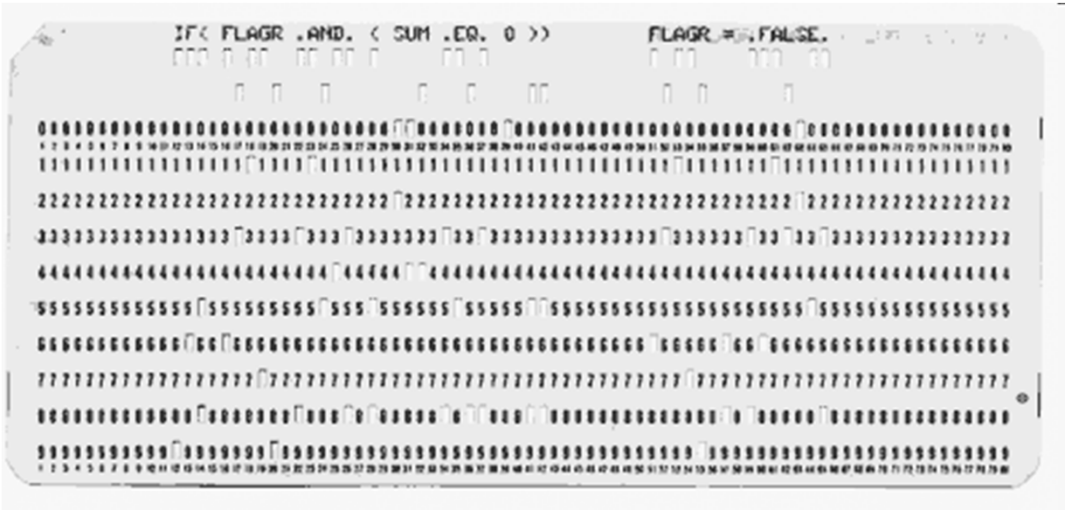
- K文字以下のアルファベットで書かれた名前・単語のソートも可能
  - 文字の種類が m, 名前・単語の数が n ならば
    O(K(m+n)) 時間

実際に使われていた基数ソート
Radix Sort Used in Real Life

- 基数ソートは実際にカードを機械を使ってソートするときに使われていた
- カードには，各桁ごとに対応する数字のところに穴を開ける
- 基数ソートをするときには，各桁ごとに0,1,2,…,9に対応する穴を参照する



http://www.museumwaalsdorp.nl/computer/en/punchcards.html



http://smoto.mii.kurume-u.ac.jp/~smoto/edu/ala/history/p0.html

ソートアルゴリズムの計算量の下界

- ソートアルゴリズムの時間計算量
  - バブルソート，クイックソートは $O(n^2)$
  - マージソート，ヒープソートは $O(n \log n)$

2要素の比較のみを使ったアルゴリズム

  - バケットソート，基数ソートは $O(n)$
    （バケツの数，桁数などを定数と見なしたとき）

- 数値情報を利用
- 数値の範囲が限定されている

- ソートのために最低限必要な計算時間は？

定理：2要素の大小比較に基づくソートアルゴリズムは $\Omega(n \log n)$ の時間計算量を必要とする.

ソートアルゴリズムの決定木（その1）

- 2要素の比較に基づくソートアルゴリズムの進行状況を決定木により表現
  - 比較を1回行なう度に状況がどのように変化するかを表す
- 例：A[1] = a, A[2] = b, A[3] = c に対するバブルソート

a, b, c の値および大小関係は未知

簡単のため，全ての値は異なると仮定

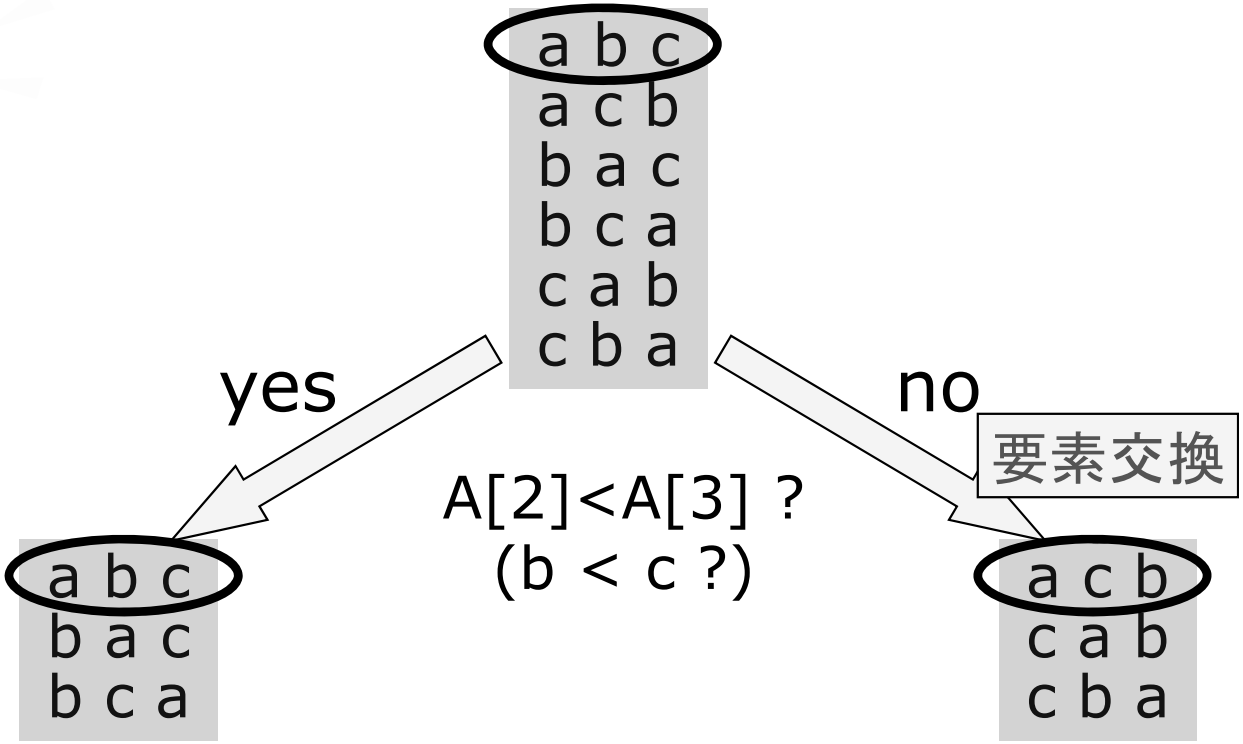
➔ソート結果の候補は

abc, acb, bac, bca, cab, cba の6通り

比較を行なう度に，ソート結果の候補数が減少

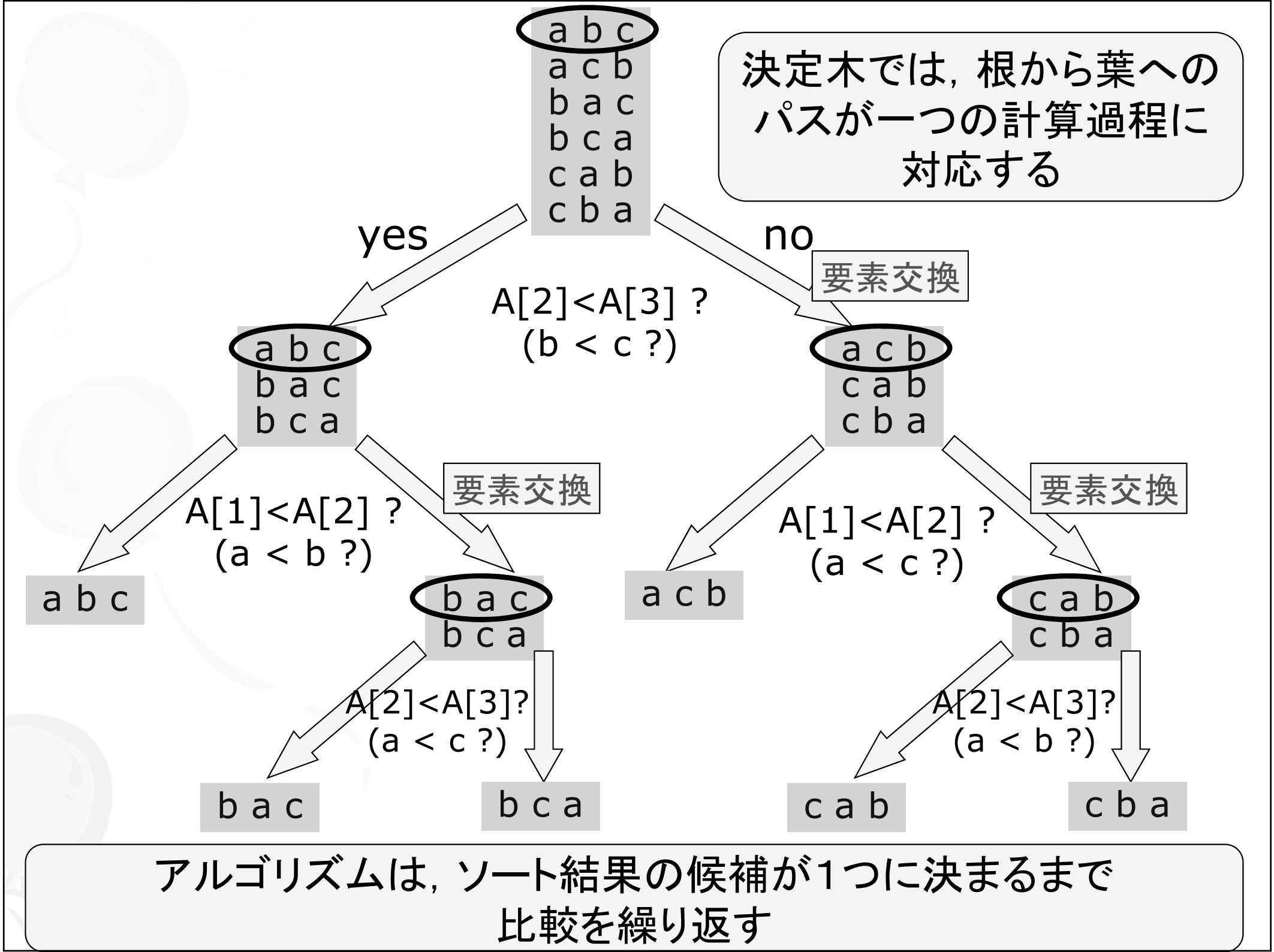
ソートアルゴリズムの決定木（その2）

- バブルソートでは，最初に A[2]=b と A[3]=c を比較
  - b<c の場合，
    - ソート結果の候補は abc, bac, bca の3通り
    - バブルソートでは要素の入れ替えをしない➔ A: abc
  - b>c の場合，
    - ソート結果の候補は acb, cab, cba の3通り
    - バブルソートではA[2]とA[3]を入れ替え➔ A:acb
    - 以下，この場合を考える


a b c
a c b
b a c
b c a
c a b
c b a
yes
no
要素交換
A[2]<A[3] ?
(b < c ?)
a b c
b a c
b c a
a c b
c a b
c b a

ソートアルゴリズムの決定木（その3）

- バブルソートでは，次に A[1]=a と A[2]=c を比較
  - a<c の場合，
    - ソート結果の候補は acb の1通り
    - バブルソートでは要素の入れ替えをしない➔ A: acb
    - バブルソートでは，この後（不要な）A[2]とA[3]の比較をして終了➔A: acb をソート結果として出力
  - a>c の場合，
    - ソート結果の候補は cab, cba の2通り
    - バブルソートではA[1]とA[2]を入れ替え➔ A:cab
    - バブルソートでは，この後A[2]とA[3]の比較を行なう.


a b c
a c b
b a c
b c a
c a b
c b a
決定木では，根から葉へのパスが一つの計算過程に対応する
yes
no
A[2]<A[3] ?
(b < c ?)
要素交換
a b c
b a c
b c a
a c b
c a b
c b a
A[1]<A[2] ?
(a < b ?)
要素交換
A[1]<A[2] ?
(a < c ?)
要素交換
a b c
b a c
b c a
a c b
c a b
c b a
A[2]<A[3]?
(a < c ?)
A[2]<A[3]?
(a < b ?)
b a c
b c a
c a b
c b a
アルゴリズムは，ソート結果の候補が1つに決まるまで比較を繰り返す

比較回数の下界値

- 決定木を使って，アルゴリズムでの比較回数の下界値を調べる
- アルゴリズムでの比較の回数の最大値
  ≧根から葉へのパスの長さの最大値 ＝決定木の高さ h

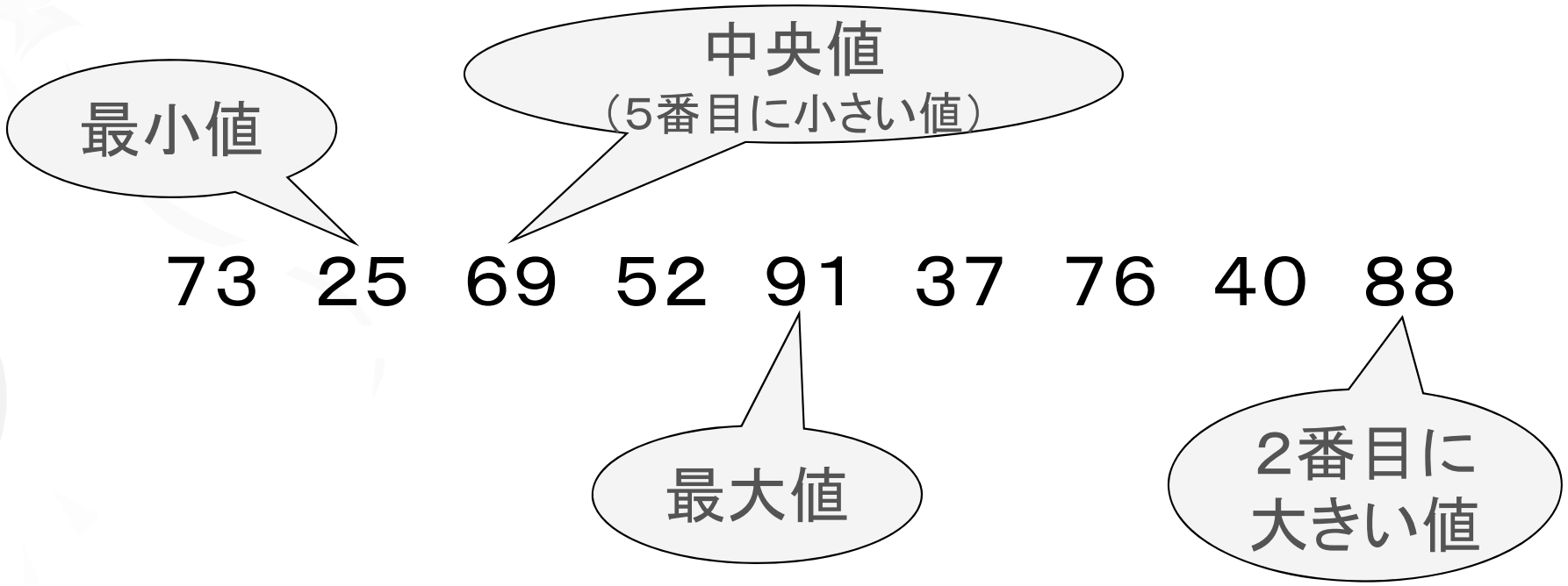
- 決定木の葉は n 要素の順列（ソート列）に対応
  ➔ n! 個の葉が存在する
  - 高さ h の二分木の葉の数 ≦ 2^h
    ➔ $n! \leqq 2^h$ ➔ $h \geqq \log (n!)$
  - スターリングの近似公式より
    $\log (n!) \fallingdotseq n \log n - n$ （n が十分に大きいとき）

この値を
知りたい！

定理：2要素の大小比較に基づくソートアルゴリズムは $\Omega(n \log n)$ の時間計算量を必要とする.

第p要素の選択

- n 個の実数の中から p 番目に小さい(大きい)ものを選ぶ
  - p=1➔n個の実数の中の最小値
  - p=n➔n個の実数の中の最大値
  - p= $\lceil n/2 \rceil$ または $\lfloor n/2 \rfloor$ ➔n個の実数の中の中央値(median)



第p要素の選択の時間計算量

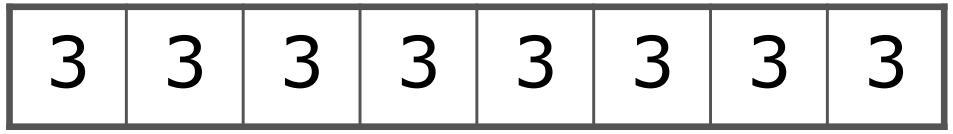
- 最小値(最大値)は簡単に O(n) 時間で求めることが可能
  - 最小値の暫定値を b =A[1] とする
  - 各 i = 2, 3, …, n に対し A[i] と b を比較,
    A[i] の方が小さければ b = A[i] とする
  - 最後に b を最小値として出力
- p番目(1 < p < n)に小さい値を求める際の時間計算量は？
  - 簡単なやり方: n 個の実数を昇順にソート
    ➔p番目の要素を選ぶ ➔ O(n log n)時間
- 今回紹介する方法---ソートを使わないアルゴリズム
  - QUICKSELECT: 平均時間計算量 O(n), 実用上高速
  - SELECT: 最悪時間計算量 O(n), 実用上は少し劣る
  - (LAZYSELECT: 平均時間計算量 O(n), 実用上高速)

QUICKSELECTの手順（その1）

- クイックソートと同様のアイディアを用いたアルゴリズム

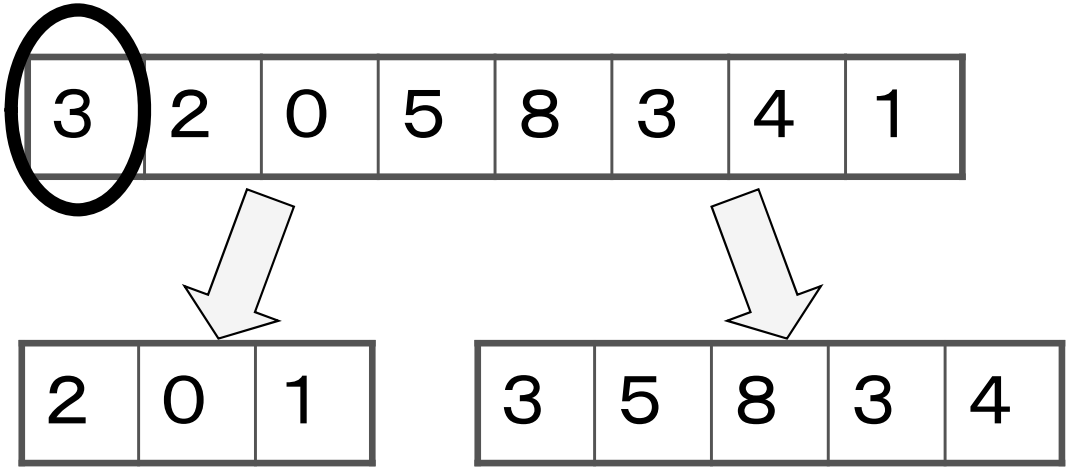
(0) A[1], …, A[n]が全て同じ要素➔
いずれかの要素を出力，終了

p=5 の場合



(1) A[1], …, A[n]から
ひとつの値（軸要素）を選ぶ

(2) 軸要素未満の要素と
それ以外に分割
k = 軸要素未満の要素数



QUICKSELECTの手順（その2）

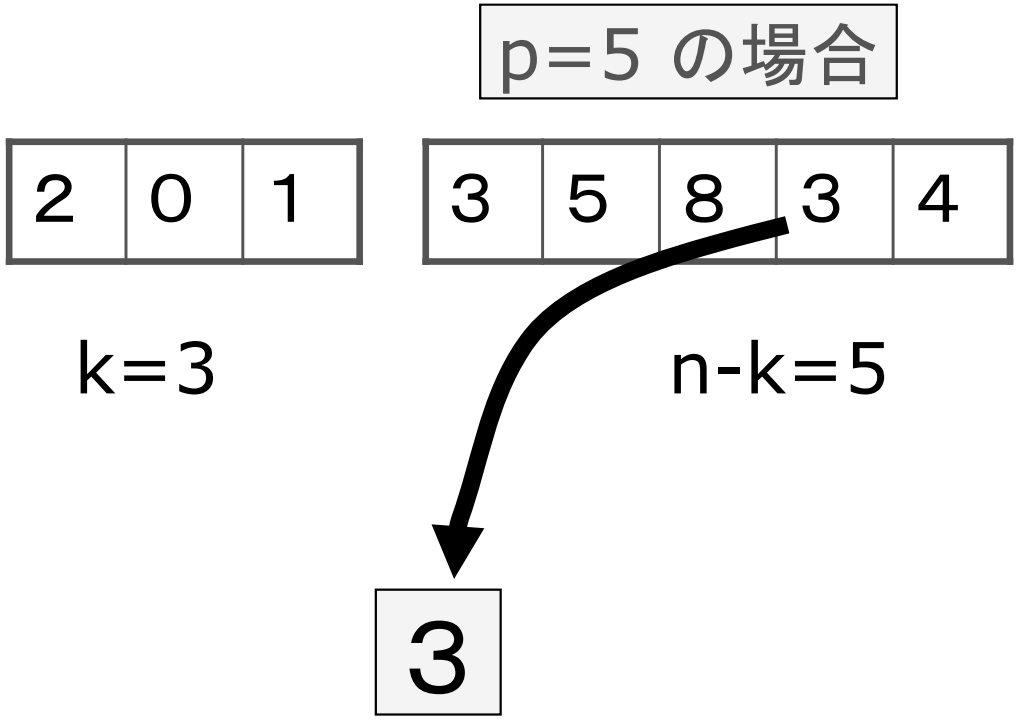
(3) p≦k のとき，

軸要素未満の配列から第 p 要素を再帰的に選択

p > k のとき，

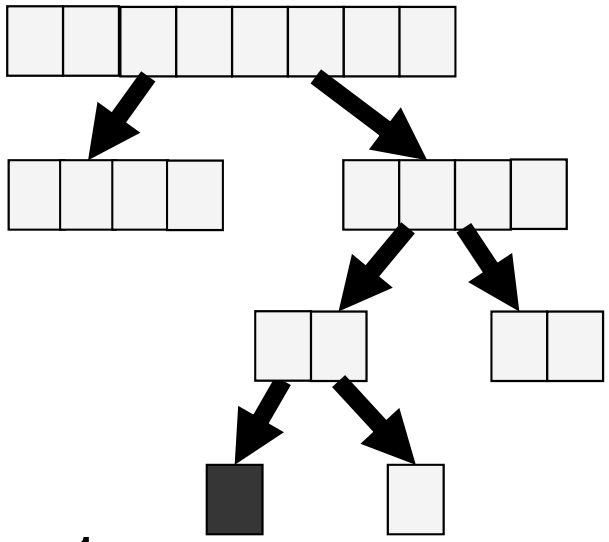
軸要素以上の配列から第 p-k 要素を再帰的に選択

p=5>3=k なので
右の配列から
第 5-3=2 要素を
再帰的に選択



QUICKSELECTの時間計算量（その1）

- 大きさ n の配列の分割は cn 時間で可能
- 分割後の配列の大きさは
  軸要素の選び方に依存



- 運がよい場合：毎回の分割で
  配列の大きさが半分になる

➔ $cn + c(n/2) + c(n/4) + \ldots + c\cdot 2 + c\cdot 1$
$\leqq cn \times 2 = O(n)$ 線形時間

- 普通の場合：毎回の分割で配列の大きさが
  (1-ε)倍以下になる（εは定数，$0<\varepsilon<1$）

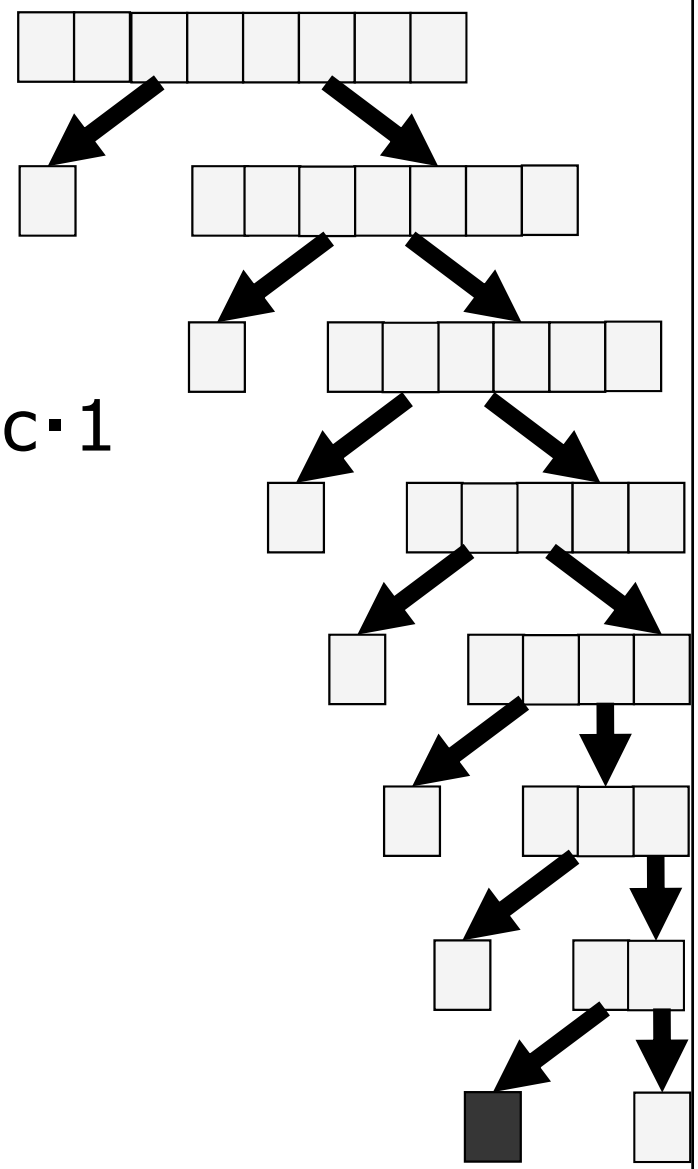
➔ $cn + c(1-\varepsilon)n + c(1-\varepsilon)^2 n + c(1-\varepsilon)^3 n$
$+ \ldots + c\, n/(1-\varepsilon) + c\cdot 1$
$\leqq cn \times 1/(1-\varepsilon) = O(n)$ 線形時間

QUICKSELECTの時間計算量（その2）

- 大きさ n の配列の分割は cn 時間で可能

- 最悪の場合（運が悪い場合）：毎回の分割で
  配列の大きさが1ずつ減少
  ➔ $cn + c(n-1) + c(n-2) + \dots + c \cdot 2 + c \cdot 1$
  $\leqq cn(n-1)/2 = O(n^2)$

- 軸の選び方をランダムにする
  ➔ 平均的な時間計算量は O(n)
  解析はかなり面倒なので省略



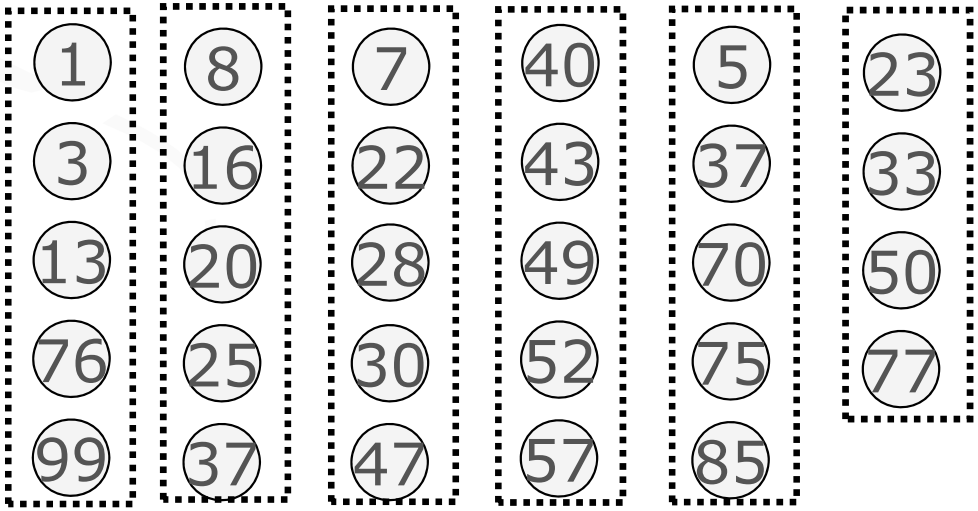
SELECTの概要

- QUICKSELECTで時間計算量がO(n)になるケース
  - 毎回の分割で2つの配列の大きさが(1-ε)倍以下になる
    （εは定数，0<ε<1)
- どんな入力に対しても，このケースが起こるように
  軸要素の選び方を工夫する
    ➔アルゴリズムSELECT　(ε= 1/4)

SELECTでの軸要素の選び方

- 説明を簡単にするため，全ての要素が異なると仮定
  - A[i] と A[k] が等しい場合は，
    配列の番号i, k によって大小関係を決めればよい
    （i<k ならば A[i]が「小さい」, i ＞ k ならばA[k]が「小さい」）

手順1： n 個の要素を5個ずつのグループに分ける.
5個未満のグループが一つ出来たら，それは別にしておく.
➔ ちょうど5個のグループの数は $\lfloor n/5 \rfloor$



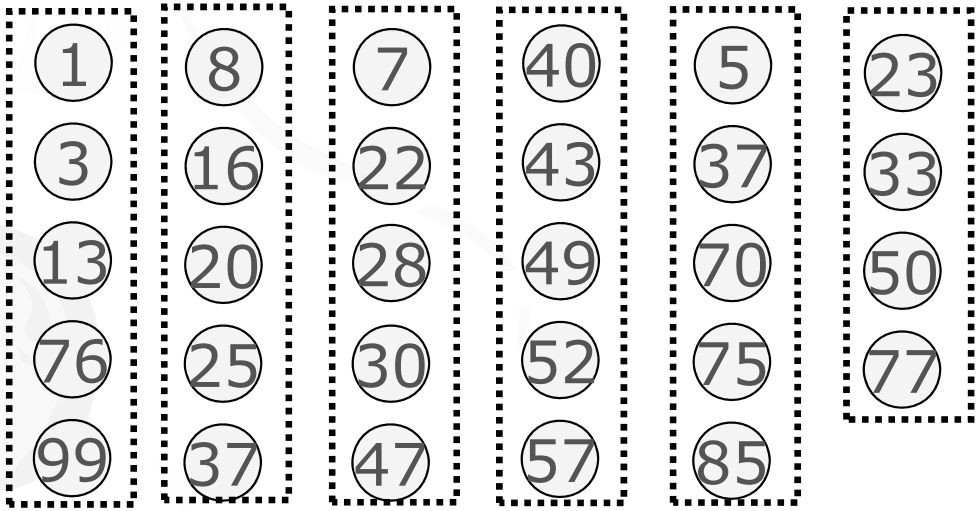
$n = 29$

$\lfloor n/5 \rfloor = 5$

SELECTでの軸要素の選び方

手順2：$\lfloor n/5 \rfloor$個の各グループにおいて，中央値（3番目に小さい値）を計算し，取り出す．

手順3：手順2で取り出した$\lfloor n/5 \rfloor$個の要素に対し，中央値（$\lceil \lfloor n/5 \rfloor /2 \rceil = \lfloor (n+5)/10 \rfloor$番目に小さい値）を計算し，軸要素 a とする



$n = 29$

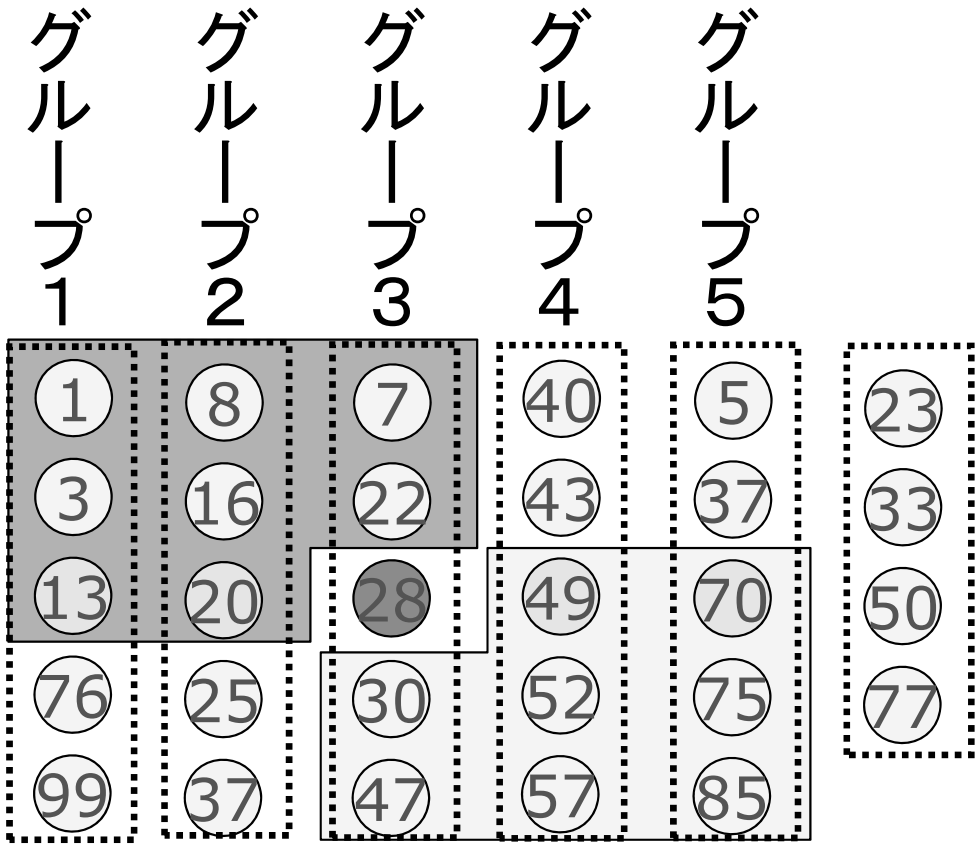
$\lfloor n/5 \rfloor = 5$

中央値の集合
13, 49, 70, 28, 20
この中の中央値は
28← 軸要素 a

各反復での要素数の減少率

グループ1，2のそれぞれの中央値は軸要素 a 未満
➔グループ1，2には a未満の要素が3つ以上存在
➔a 未満の要素数
　≧3×3グループ-1＝8

グループ4，5のそれぞれの中央値は軸要素aより大きい
➔グループ4，5には aより大きい要素が3つ以上存在
➔a より大きい要素の数
　≧3×3グループ-1＝8



軸要素 a より大きい要素の割合は？
軸要素 a より小さい要素の割合は？

各反復での要素数の減少率

軸要素 a は，各グループの中央値の中の中央値，$\lceil \lfloor n/5 \rfloor /2 \rceil = \lfloor (n+5)/10 \rfloor$ 番目に小さい要素

「グループの中央値＜a」となるグループ数は $\lfloor (n+5)/10 \rfloor - 1$

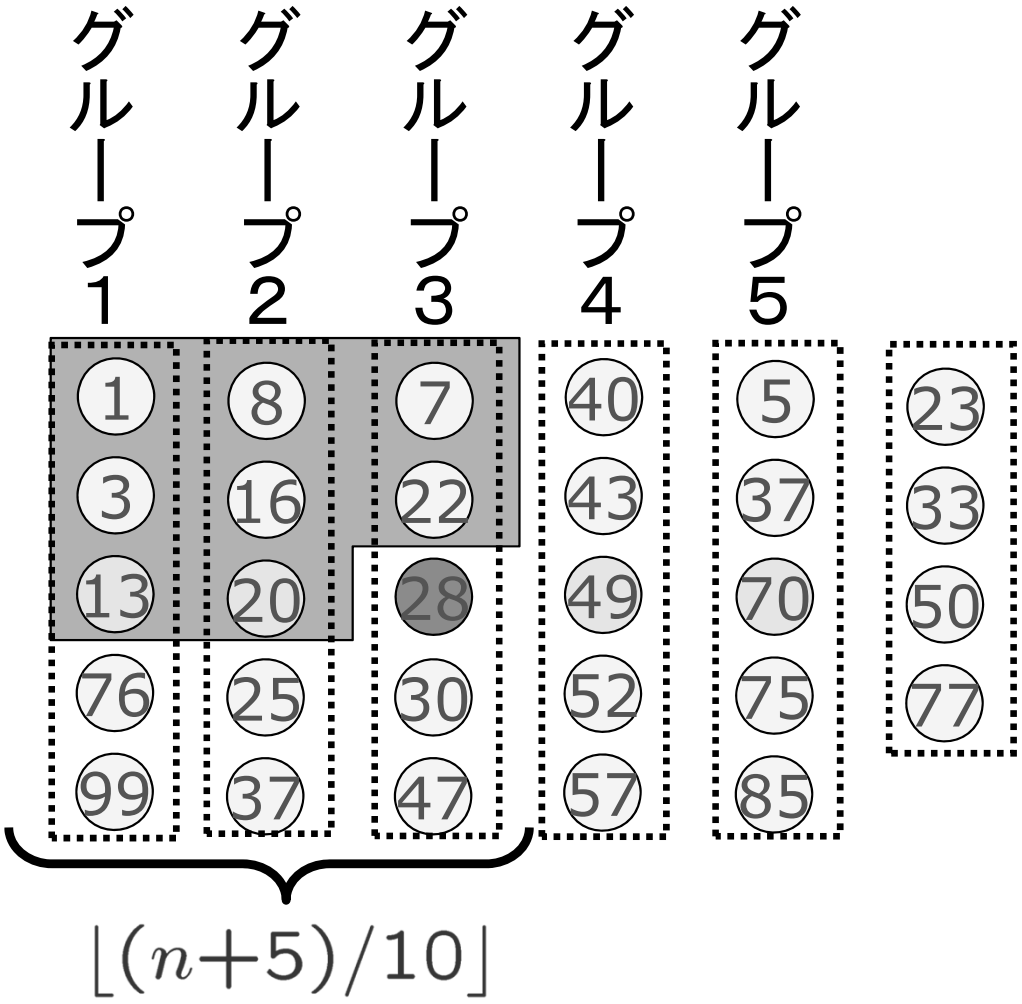
➔各グループには a未満の要素が3つ以上存在

➔a 未満の要素数

$\geqq 3 \times \lfloor (n+5)/10 \rfloor - 1$

n≧50のとき，この値はn/4以上

n≧50,かつ見つけたい要素が軸要素以上➔
次の反復での要素数≦3/4n



各反復での時間計算量の解析

軸要素 a は，各グループの中央値の中の中央値，$\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ 番目に大きい要素

「グループの中央値＞a」となるグループ数は $\lfloor n/10 \rfloor - 1$

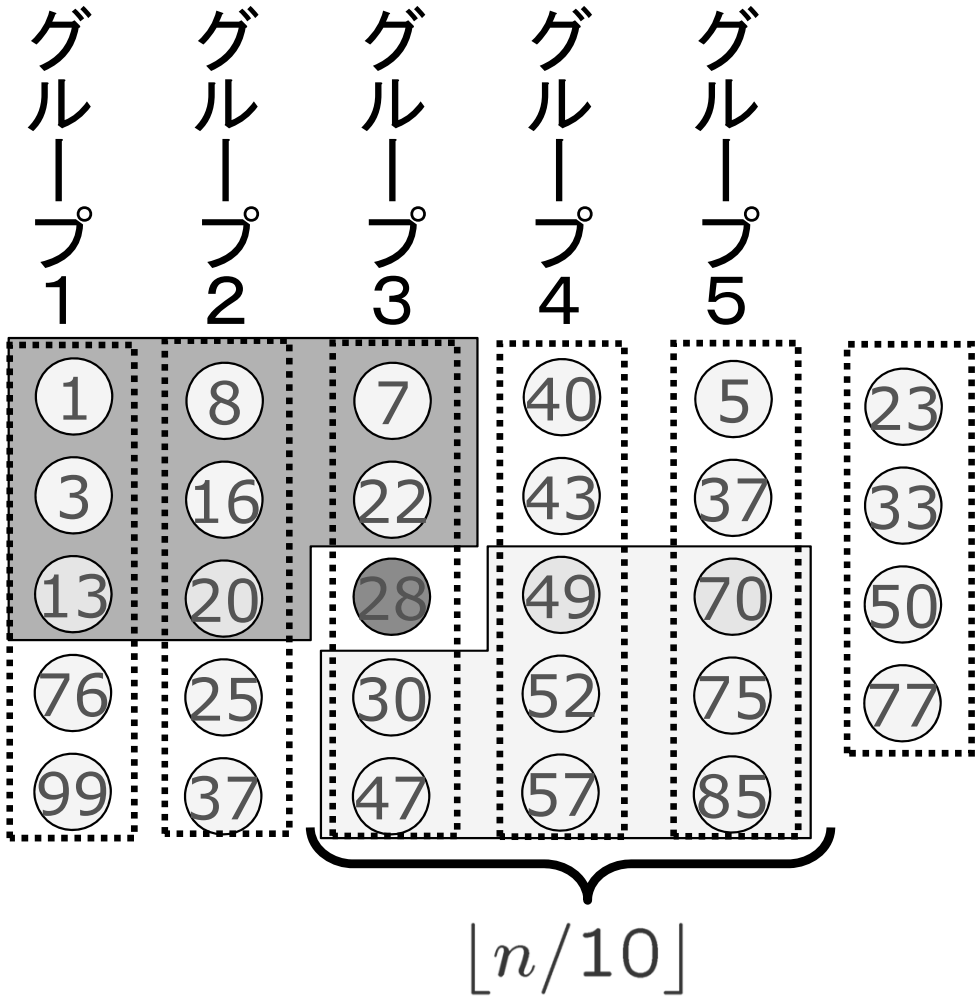
➔各グループには aより大きい要素が3つ以上存在

➔a より大きい要素の数

$\geqq 3 \times \lfloor n/10 \rfloor - 1$

n≧80のとき，この値はn/4以上

n≧80,かつ見つけたい要素が軸要素未満➔
次の反復での要素数≦3/4n



SELECTの時間計算量の解析

- アルゴリズムSELECTでは,
  - n≧80のとき，一回の反復で要素数が n から 3/4n 以下に減少する
  - n<80のとき，定数時間で第p要素を選択することが可能（要素数nが定数以下なので）
- 反復回数を k とおく

➔$(3/4)^{k-1} \geqq 80$, $(3/4)^{k} < 80$を満たす

➔合計の時間計算量は

$c\{n + 3/4n + (3/4)^2 n + \ldots + (3/4)^{k-2} n + 80\}$

$\leqq c \times 4n + 80$ $c = O(n)$