アルゴリズムとデータ構造

第10回　整列のアルゴリズム（2）

注意：
分かり易いように動作説明を
何枚かのスライドに分けて
少しずつ書いています
（枚数にビックリしないでね）

小技：
同じような絵が描いてある
ページは、PCのキー操作で
次前次前次前と往復すると
変化した場所が分かりやすい

今日の内容

- 今日の内容
  - 整列アルゴリズムの種類と特徴（おさらい）
  - 最悪時計算量 $O(n \log n)$ 時間の整列アルゴリズム
    - マージソート、ヒープソート
  - 比較に基づくソートの漸近的下界
  - 整数データに対する $O(n)$ 時間の整列アルゴリズム
    - バケットソート、基数ソート

整列アルゴリズムの種類と特徴

アルゴリズム	最悪時間計算量の漸近的上界	コメント
選択ソート (selection sort) 挿入ソート (insertion sort) バブルソート (bubble sort)	$O(n^2)$	直感的に理解しやすい
シェルソート (shell sort)	$O(n(\log n)^2)$	実用性は高い．平均時間計算量で$O(n \log n)$であるかは未解決
クイックソート (quick sort)	$O(n^2)$	平均時間計算量は$O(n \log n)$ 実用上最も高速．分割統治法
マージソート (merge sort) ヒープソート (heap sort)	$O(n \log n)$	最悪時間計算量の漸近的上界が最小．マージソートは分割統治法
バケットソート (bucket sort) 基数ソート (radix sort)	$O(n)$ 注)	高速だが，ある範囲に限定された整数に対してのみ適用可能

注）バケット数と桁数を定数とみた場合

マージソート（merge sort、併合ソート）

配列を2つに分割し、それぞれを整列してからマージする
分割統治法による整列アルゴリズム

msort(A, i, j): A[i],A[i+1],…,A[j]を整列
step 1: 要素数(j-i+1)が1なら何もしないでリターン
step 2: h ← (i+j)/2
Step 3: msort(A, i, h) と msort(A, h+1, j) を実行
Step 4: 2つのソート済みの配列
A[i], A[i+1], …, A[h] と
A[h+1], A[h+2], …, A[j] をマージして
A[i], A[i+1], …, A[j]に格納

マージ（併合）:
ソート済みの2つの列を合成して、1つのソート済みの列を作成する処理

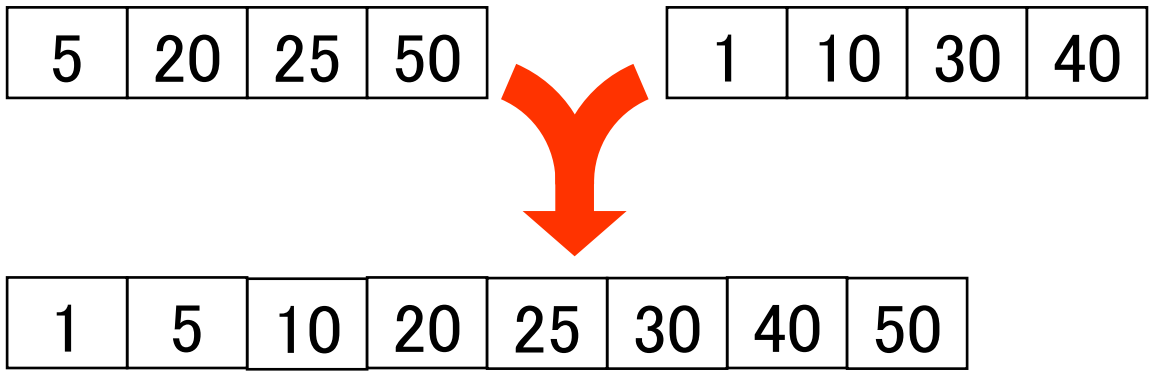
最悪/最良/平均時間計算量は $O(n \log n)$

マージ（併合）

- ソートされた2つの列

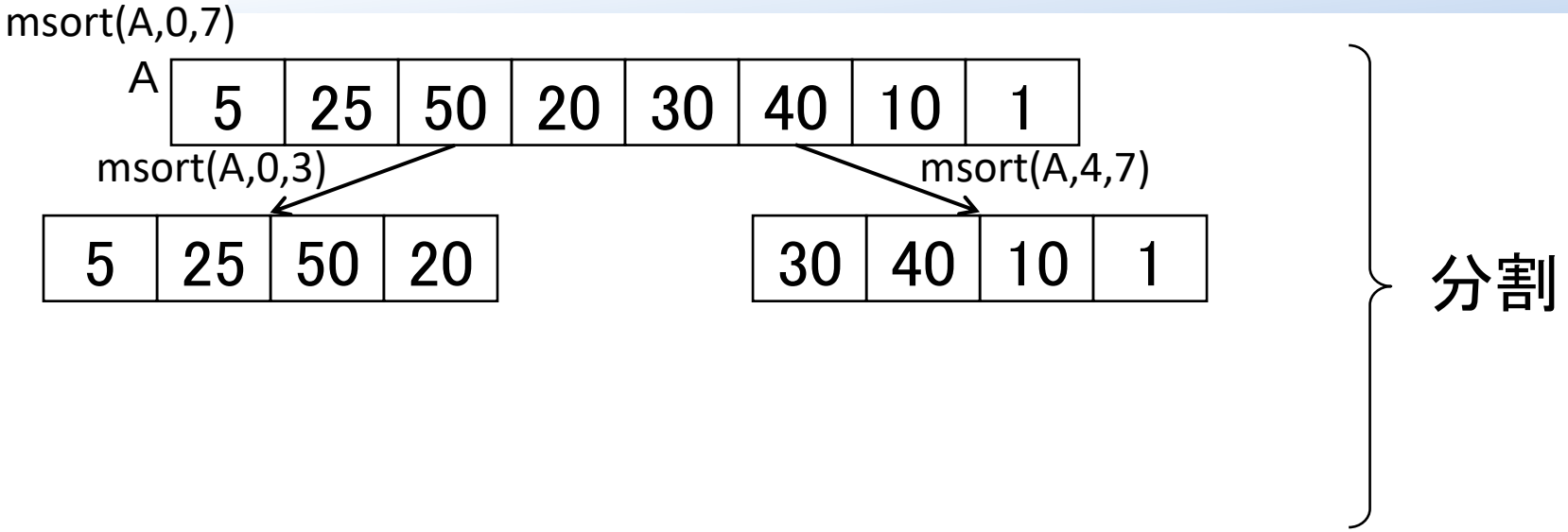
  → 1つにマージ（併合）するのは簡単

  （各列の先頭を見比べながら、小さい方を取っていく）

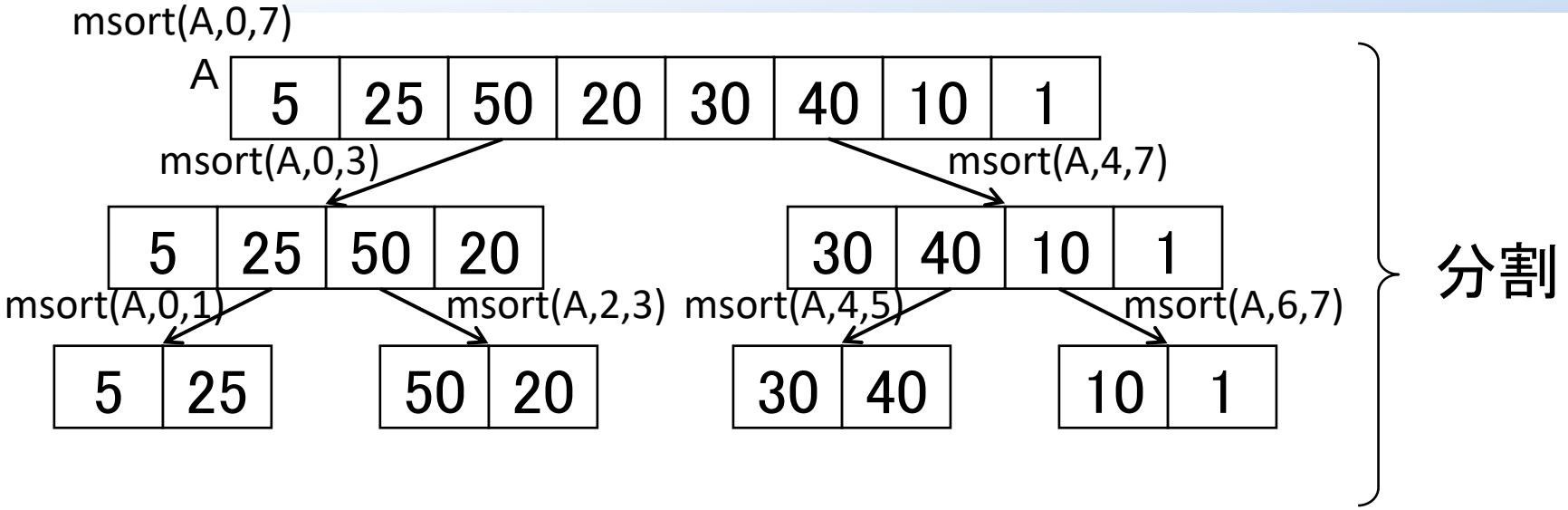


- 1回のマージの実行時間は、要素数に比例

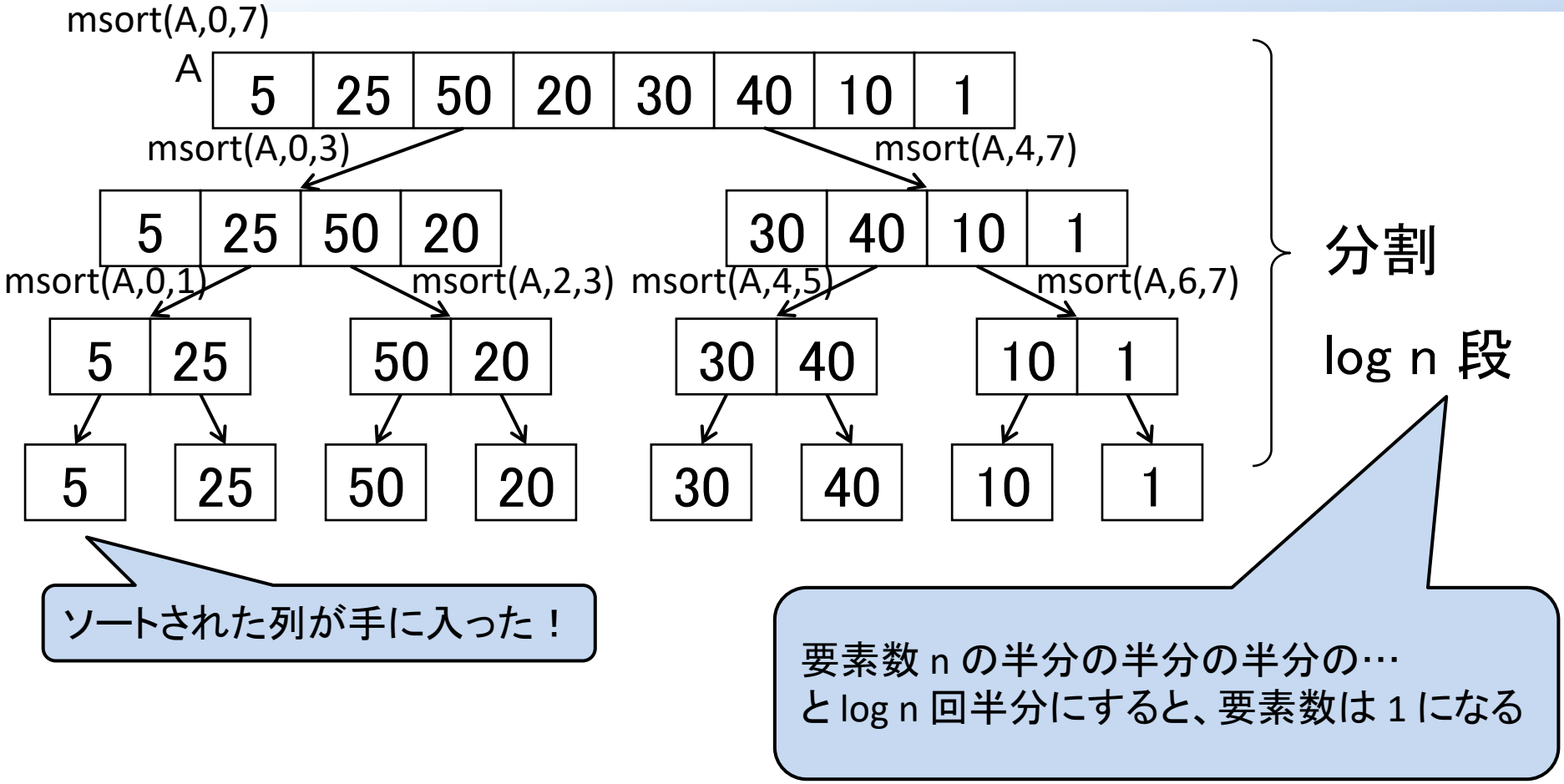
マージソートの動作例


msort(A,0,7)
A
5
25
50
20
30
40
10
1
msort(A,0,3)
msort(A,4,7)
5
25
50
20
30
40
10
1
分割

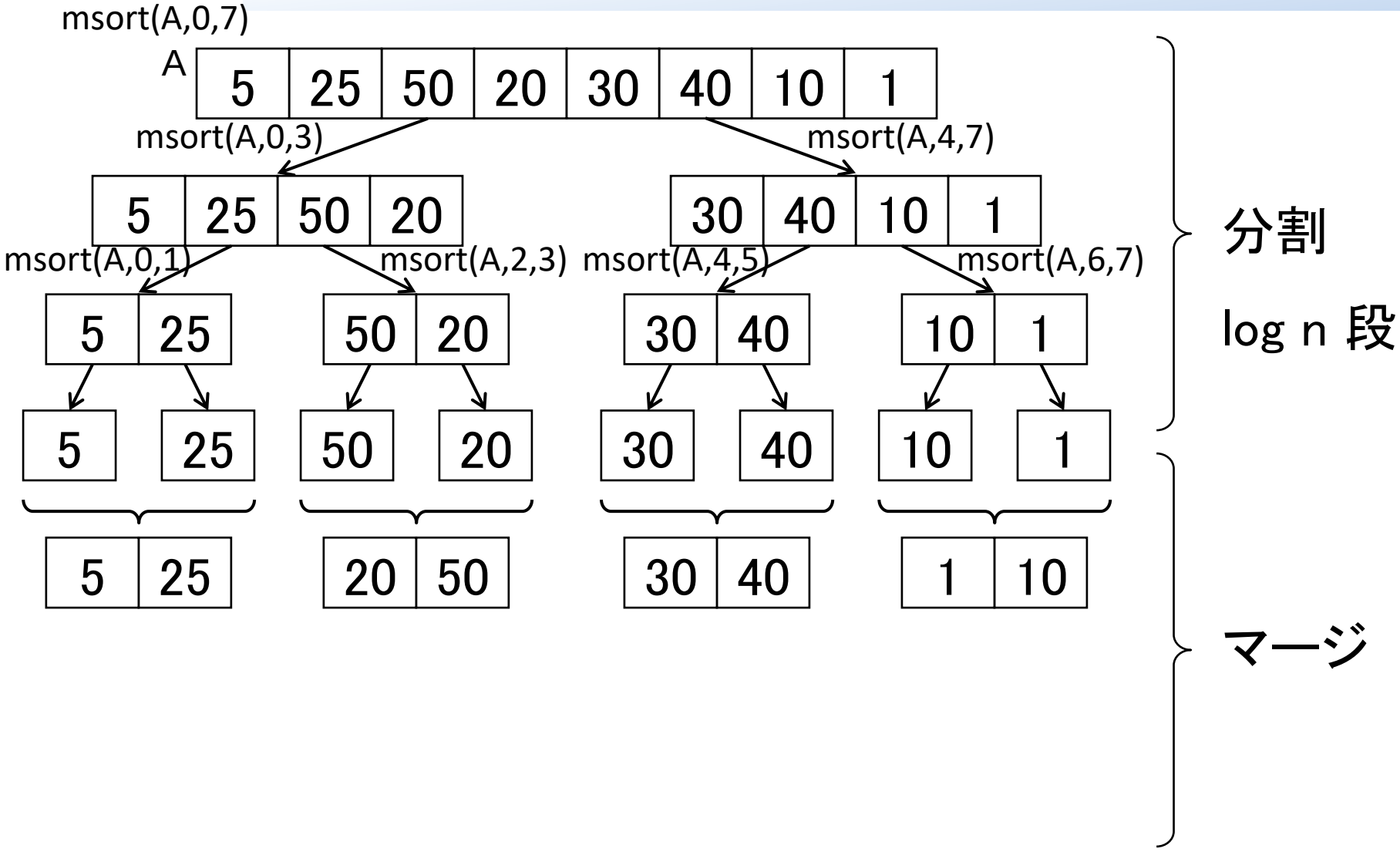
マージソートの動作例


msort(A,0,7)
A
5 25 50 20 30 40 10 1
msort(A,0,3)
msort(A,4,7)
5 25 50 20
30 40 10 1
msort(A,0,1)
msort(A,2,3)
msort(A,4,5)
msort(A,6,7)
5 25
50 20
30 40
10 1
分割

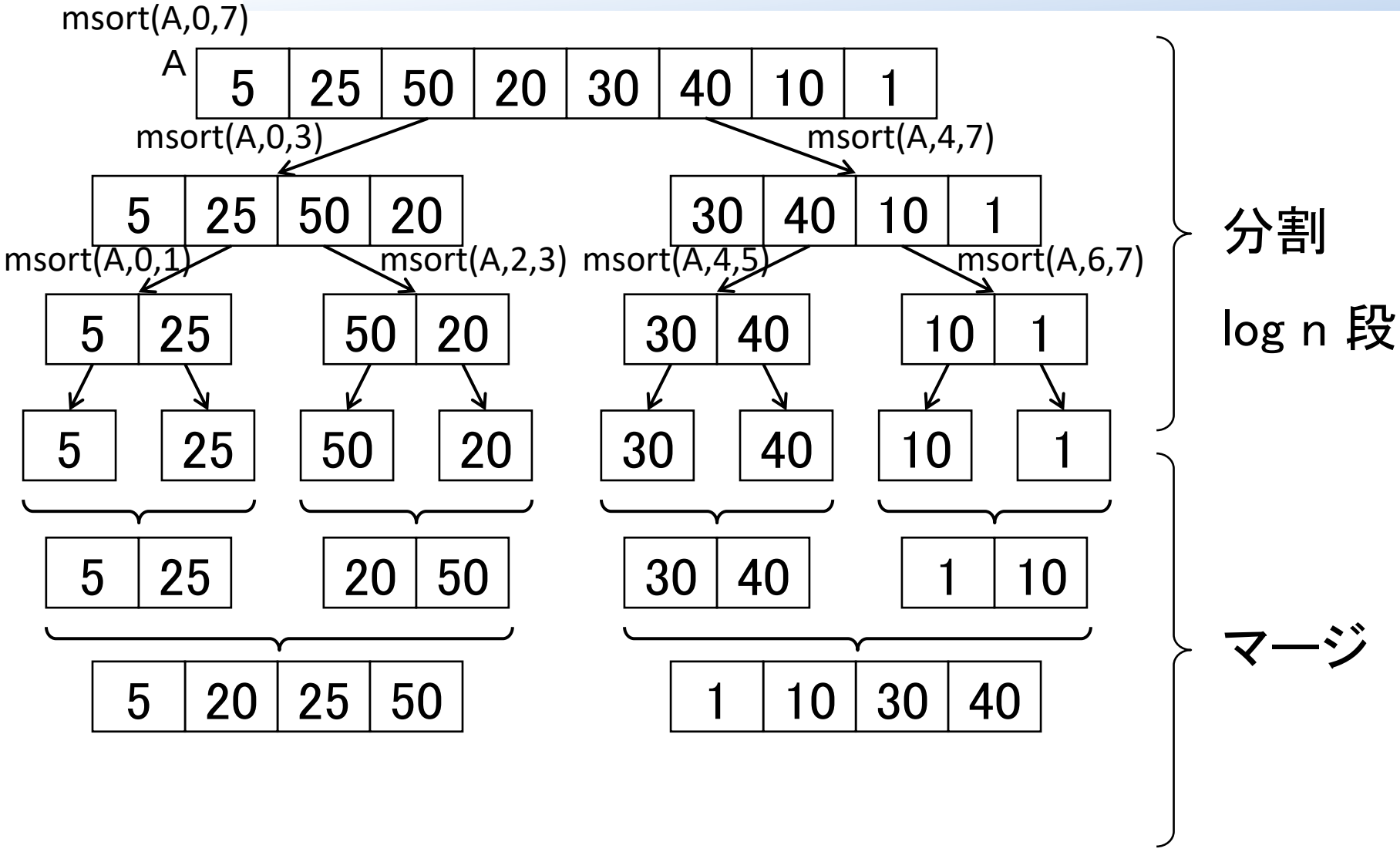
マージソートの動作例


msort(A,0,7)
A
5 25 50 20 30 40 10 1
msort(A,0,3)
msort(A,4,7)
5 25 50 20
30 40 10 1
msort(A,0,1)
msort(A,2,3)
msort(A,4,5)
msort(A,6,7)
5 25
50 20
30 40
10 1
5
25
50
20
30
40
10
1
分割
log n 段
ソートされた列が手に入った！
要素数 n の半分の半分の半分の…
と log n 回半分にすると、要素数は 1 になる

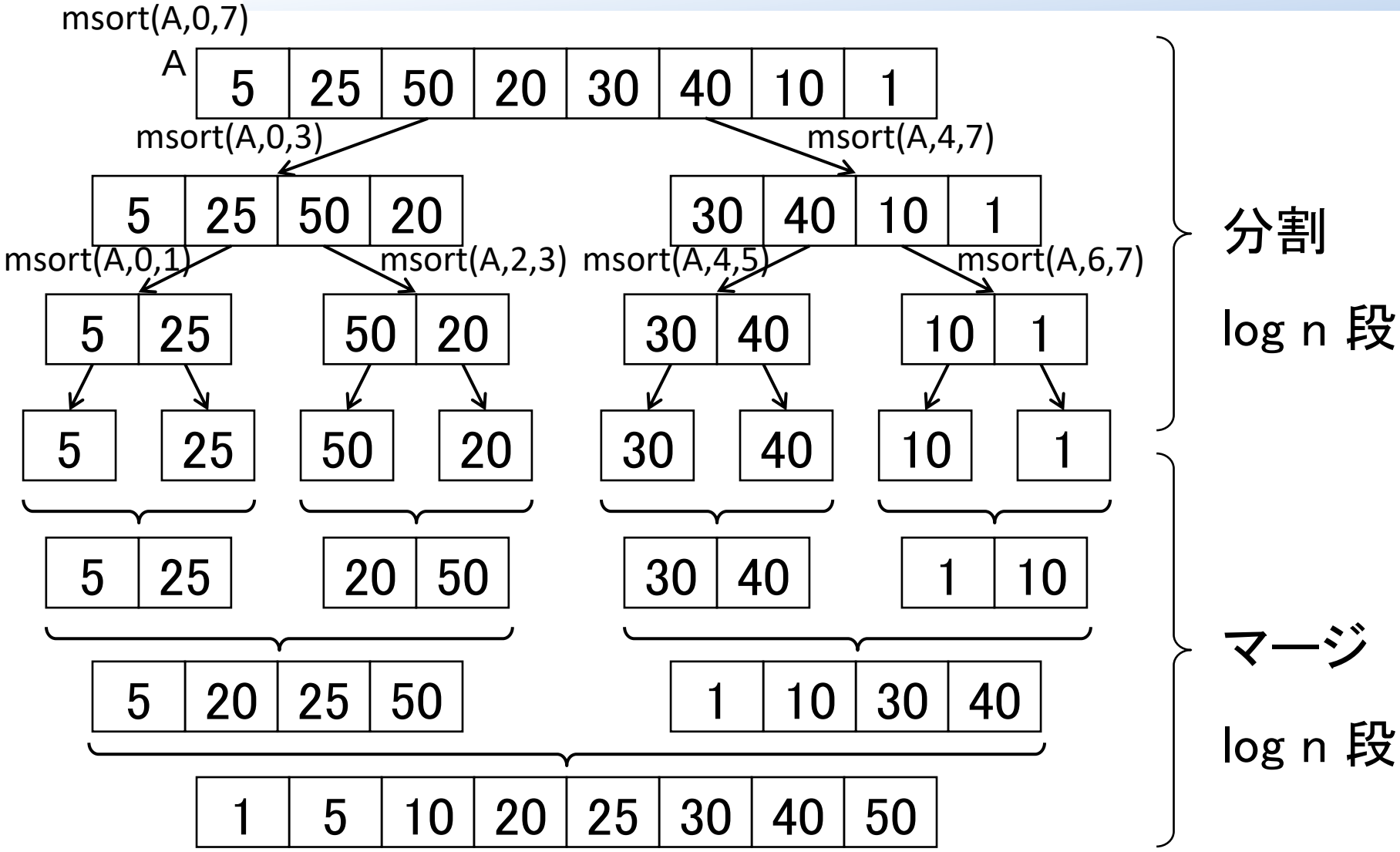
マージソートの動作例


msort(A,0,7)
A
5 25 50 20 30 40 10 1
msort(A,0,3)
msort(A,4,7)
5 25 50 20
30 40 10 1
msort(A,0,1)
msort(A,2,3)
msort(A,4,5)
msort(A,6,7)
5 25
50 20
30 40
10 1
5
25
50
20
30
40
10
1
5 25
20 50
30 40
1 10
分割
log n 段
マージ

マージソートの動作例


msort(A,0,7)
A
5 25 50 20 30 40 10 1
msort(A,0,3)
msort(A,4,7)
5 25 50 20
30 40 10 1
msort(A,0,1)
msort(A,2,3)
msort(A,4,5)
msort(A,6,7)
5 25
50 20
30 40
10 1
5
25
50
20
30
40
10
1
5 25
20 50
30 40
1 10
5 20 25 50
1 10 30 40
分割
log n 段
マージ

マージソートの動作例



1 段に O(n) 時間 → 全部で O(n log n) 時間

マージの詳細

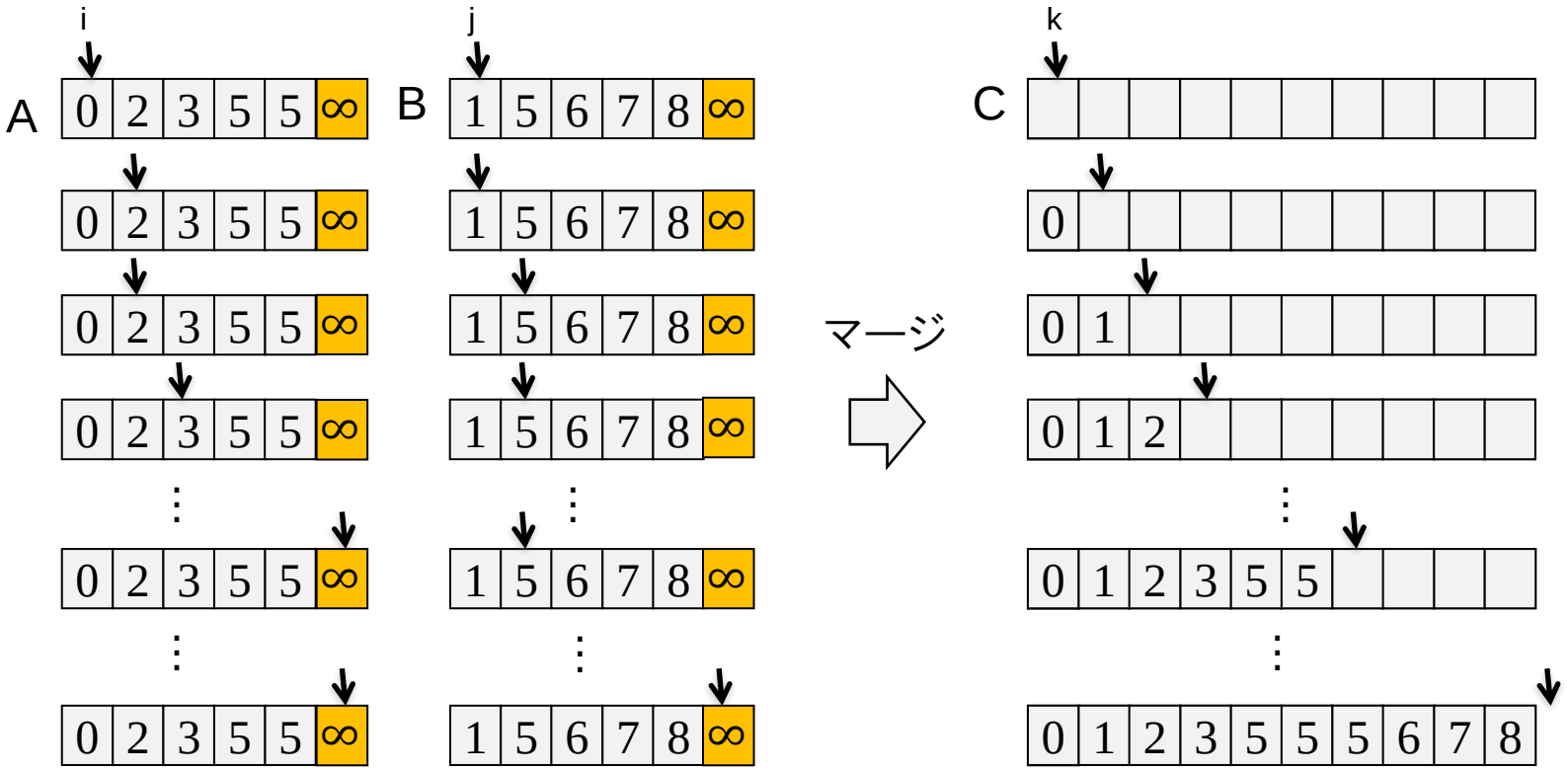
「番兵」と呼ばれるテクニック
（これが無いと、本質でない所で処理の記述が面倒になる）

ソート済みの2つの配列 A[0], A[1], …, A[n-1] と B[0], B[1], …, B[m-1] をマージして
ソート済みの配列 C[0], C[1], …, C[n+m-1] を作成する（A[n]＝B[m]＝∞ と仮定）

step 1: i, j, k ← 0

step 2: A[i] ≦ B[j]であれば、C[k] ← A[i], i ← i+1 そうでなければ C[k] ← B[j], j ← j+1

step 3: i＝n かつ j＝m ならば停止、そうでなければ k ← k+1 として step 2 へ



マージソートの最悪時間計算量：$O(n \log n)$

p. 11 よりきちんとした証明

$T(n)$ を n 個の要素をマージソートするときの計算時間とする.
まず, $n = 2^k$ のとき, 成り立つことを示す. このとき適当な定数 c を用いて,

$$T(n) \leq 2T(n/2) + cn \ .$$

cn はマージにかかる時間

よって,

$$\begin{aligned} T(n) &\leq 2(2T(n/2^2) + c(n/2)) + cn \\ &= 2^2 T(n/2^2) + 2cn \\ &\cdots \\ &\leq 2^k T\left(n/2^k\right) + kcn = nT(1) + cn \log n \ . \end{aligned}$$

$T(1)$は定数！

したがって, $T(n) = O(n \log n)$ である.

$n \neq 2^k$ のとき, $2^{k-1} < n < 2^k$ を満たす k が存在する. このとき $n' = 2^k$ とすれば, msort の深さ i の再帰呼出しに渡る要素数は $n'/2^i$ を超えない.
つまり, n 個の要素に対して msort を実行する場合は, どの再帰呼出しにおいても n' 個の要素に対して msort を実行する場合より要素数は多くならない. よって,

$$T(n) \leqq T(n') = O(n' \log n') = O(2n \log 2n) = O(n \log n)$$

$2^k = 2 \cdot 2^{k-1} < 2n$

休憩

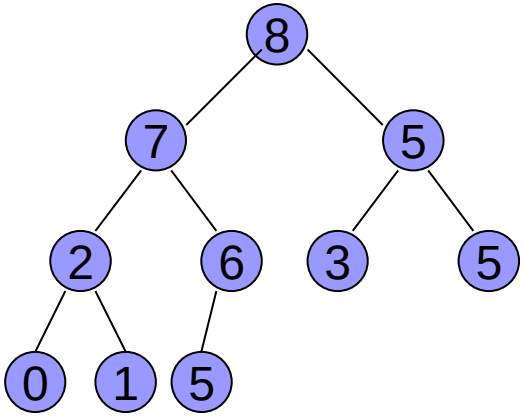
- ここで、少し休憩しましょう。
- 深呼吸したり、肩の力を抜いてから、
  次のビデオに進んでください。

ヒープ（heap）

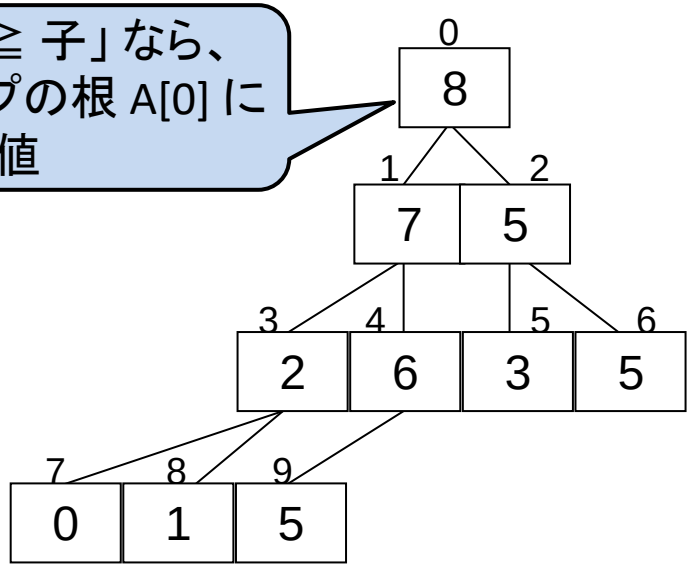
木の絵 ⇔ 配列
素直に対応させられる

以前の復習

- 任意の節点 u に対し、「u の親の要素 ≦ u の要素」が成立
- 今回の「ヒープソート」では、逆に「親 ≧ 子」とする



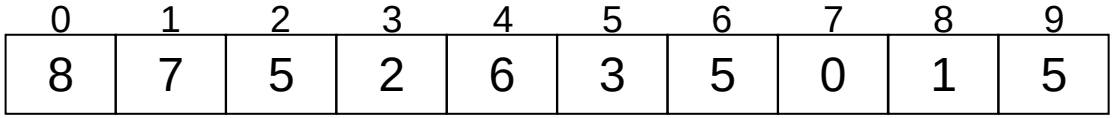
「親 ≧ 子」なら、
ヒープの根 A[0] に
最大値



- ヒープは、配列で表せる

A[*i*] の

- 親　　A[$\lfloor (i-1)/2 \rfloor$]
- 左の子　A[$2i + 1$]
- 右の子　A[$2i + 2$]



ヒープソート（heap sort）の 概要

重要

- 「親 ≧ 子」ヒープ

まずは、ブラックボックスと思って、概要を学習

  - **最大値**は何？
    - A[0] を見るだけ
    - 時間計算量 O(1)
  - 最大値を削除したい
    - 残りの要素が「親 ≧ 子」を満たすように、**「ヒープの維持」**をする必要がある
    - 木（ヒープ）の深さに比例する … 時間計算量 O(log n)

「ヒープの維持」が分かれば、ヒープソートはマスターできます（詳細は、次のページ）

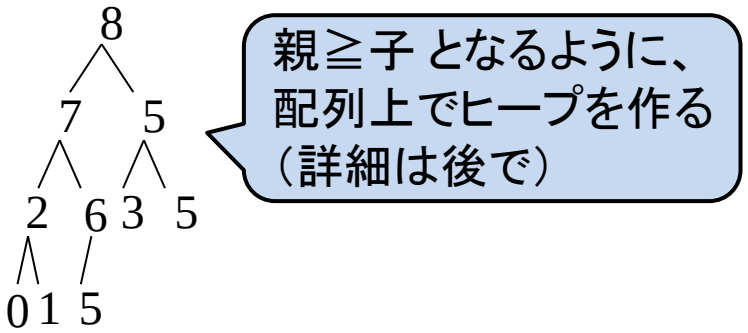
n は要素数

- ヒープソート
  - 「**最大値**を取り出して、**ヒープの維持**」を n 回繰り返すと、**値の大きい順**に取り出せる … 時間計算量 O(n log n)

ヒープソート（heap sort）の 概要

重要

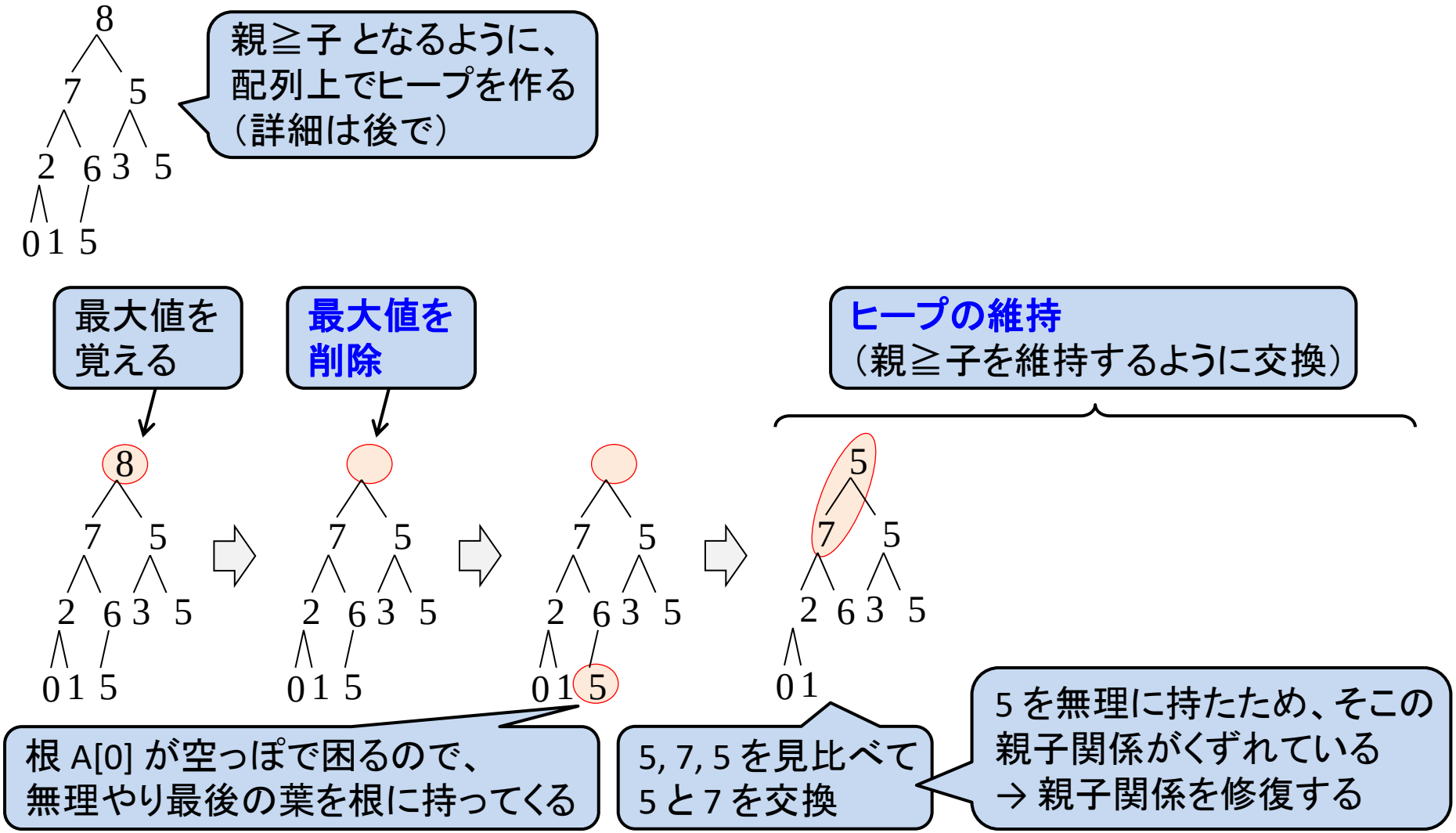
- 「親 ≧ 子」ヒープを作り、最大値を1つずつ取り出して並べる



ヒープソート（heap sort）の 概要

重要

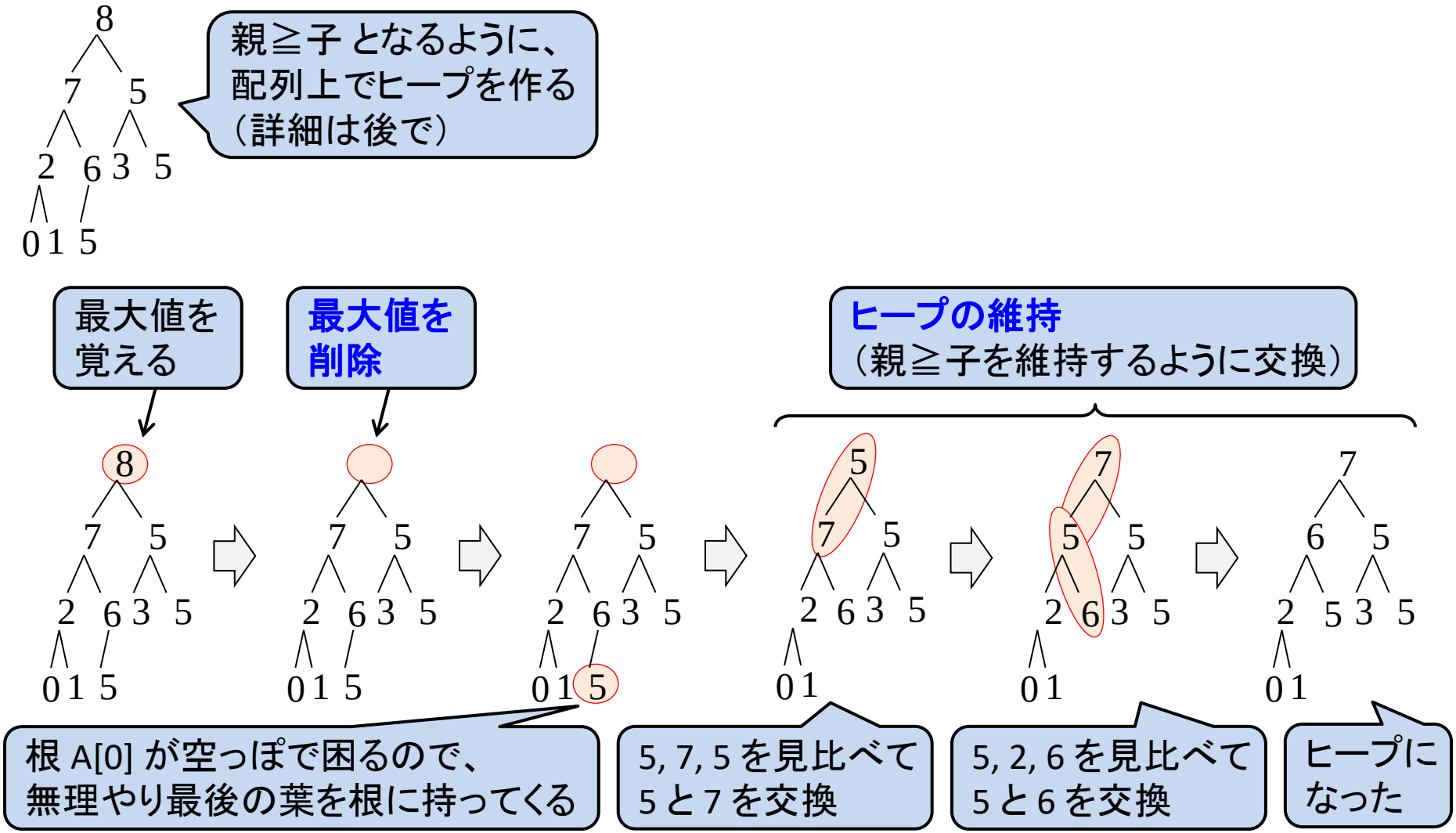
- 「親 ≧ 子」ヒープを作り、最大値を1つずつ取り出して並べる



ヒープソート（heap sort）の 概要

重要

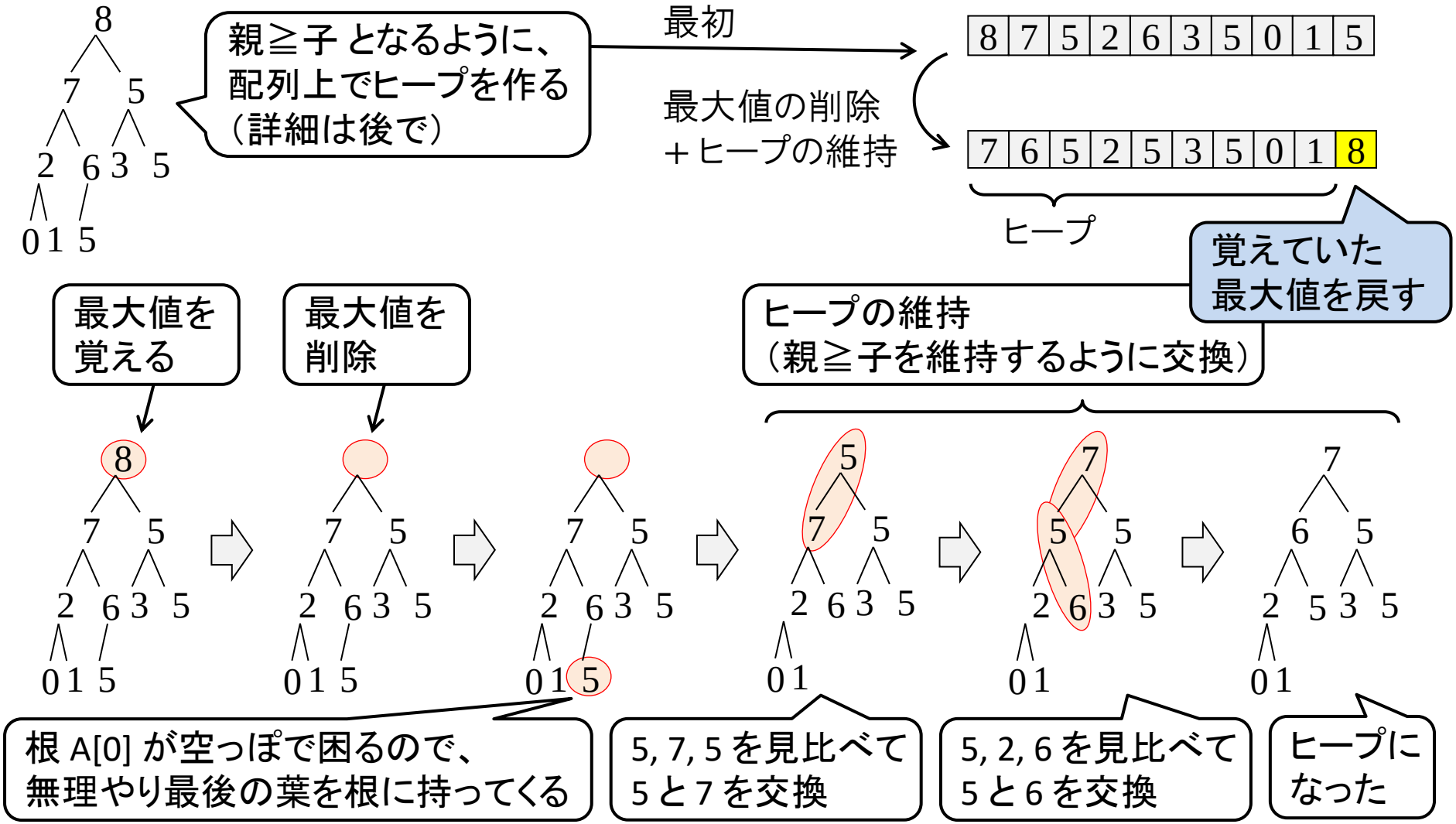
- 「親 ≧ 子」ヒープを作り、最大値を1つずつ取り出して並べる



ヒープソート（heap sort）の 概要

配列の動きを追うと

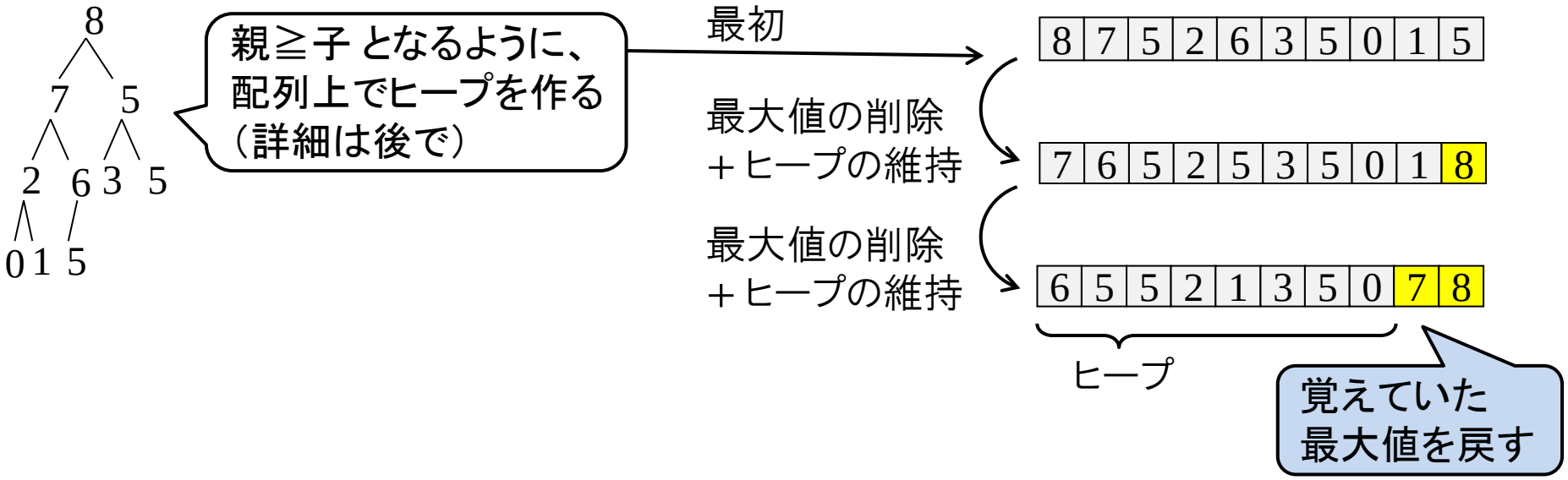
- 「親 ≧ 子」ヒープを作り、最大値を1つずつ取り出して並べる



ヒープソート（heap sort）の 概要

配列の動きを追うと

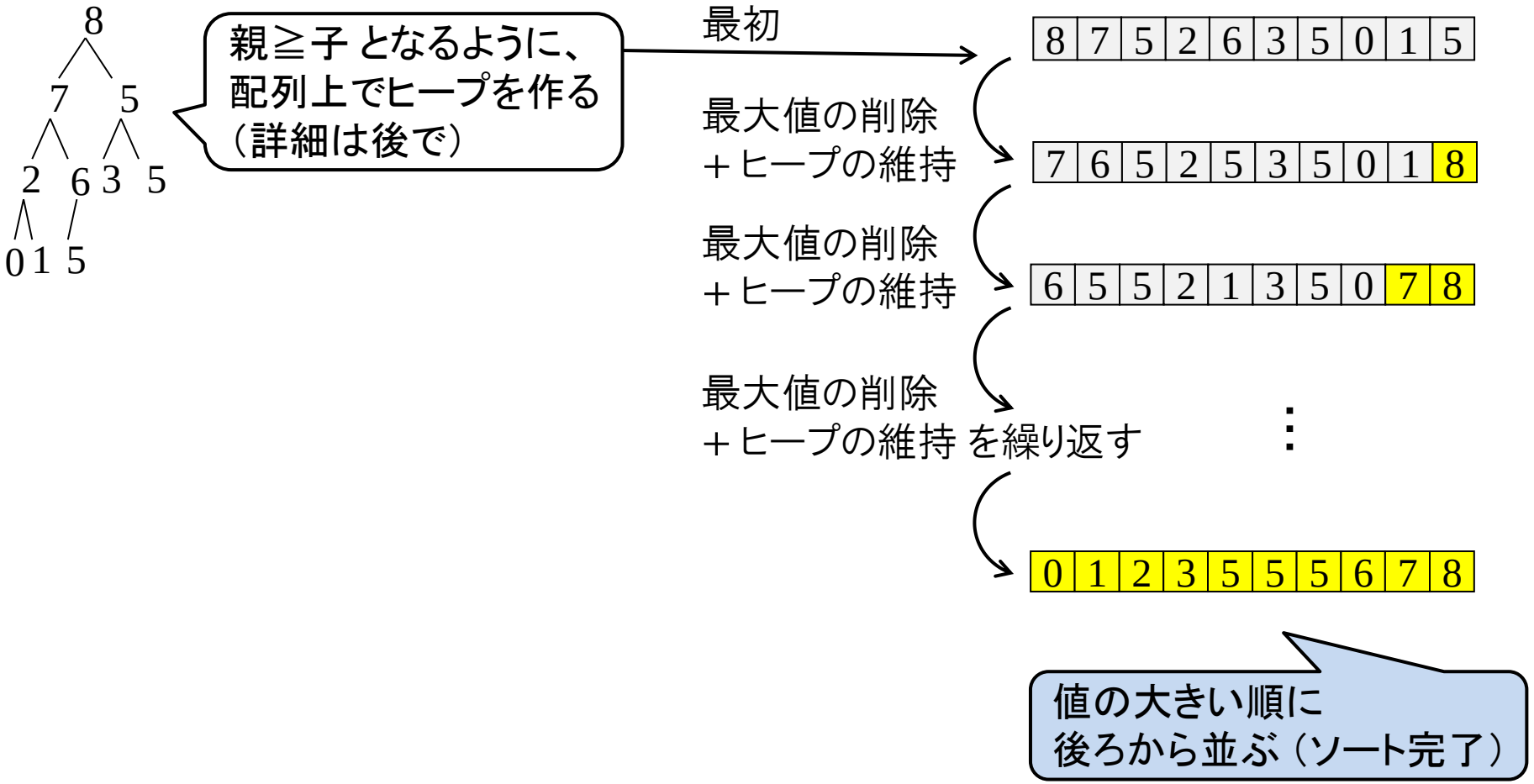
- 「親 ≧ 子」ヒープを作り、最大値を1つずつ取り出して並べる



ヒープソート（heap sort）の 概要

配列の動きを追うと

- 「親 ≧ 子」ヒープを作り、最大値を1つずつ取り出して並べる



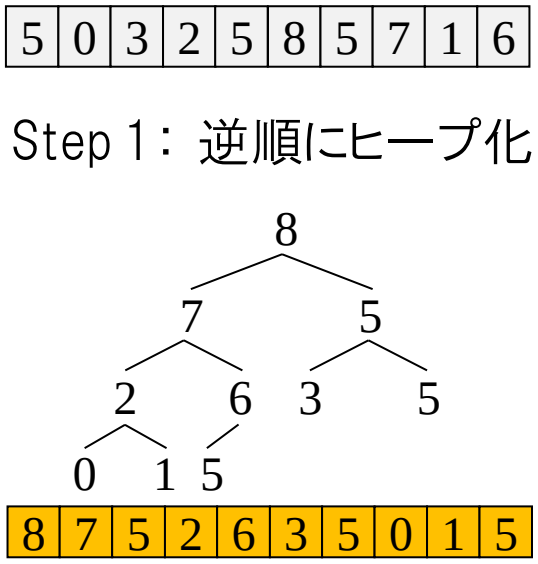
ヒープソート（heap sort）

図解は、以前のページのを簡略化しただけ

「親 ≧ 子」ヒープを作り、最大値を1つずつ取り出して並べる

heapsort(A,n):
step 1: Aを逆順にヒープ化
step 2: i ← n-1
step 3: temp ← A[0]
step 4: DELETEMAX(A,i)を実行
step 5: A[i] ← temp
step 6: i ← i-1
step 7: i < 1 ならば停止。そうでなければ step 3 へ

親≧子 となるように配列上でヒープを作る（詳細は後で）



ヒープソート（heap sort）

図解は、以前のページのを簡略化しただけ

「親 ≧ 子」ヒープを作り、最大値を1つずつ取り出して並べる

heapsort(A,n):
step 1: Aを逆順にヒープ化
step 2: i ← n-1
step 3: temp ← A[0]
step 4: DELETEMAX(A,i)を実行
step 5: A[i] ← temp
step 6: i ← i-1
step 7: i < 1 ならば停止。そうでなければ step 3 へ

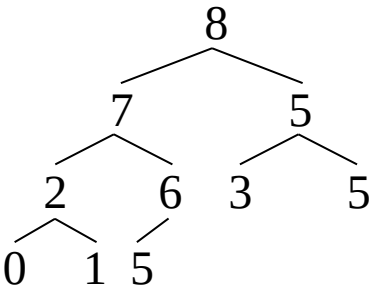
親≧子 となるように配列上でヒープを作る（詳細は後で）

A[0] がヒープの根（最大値）

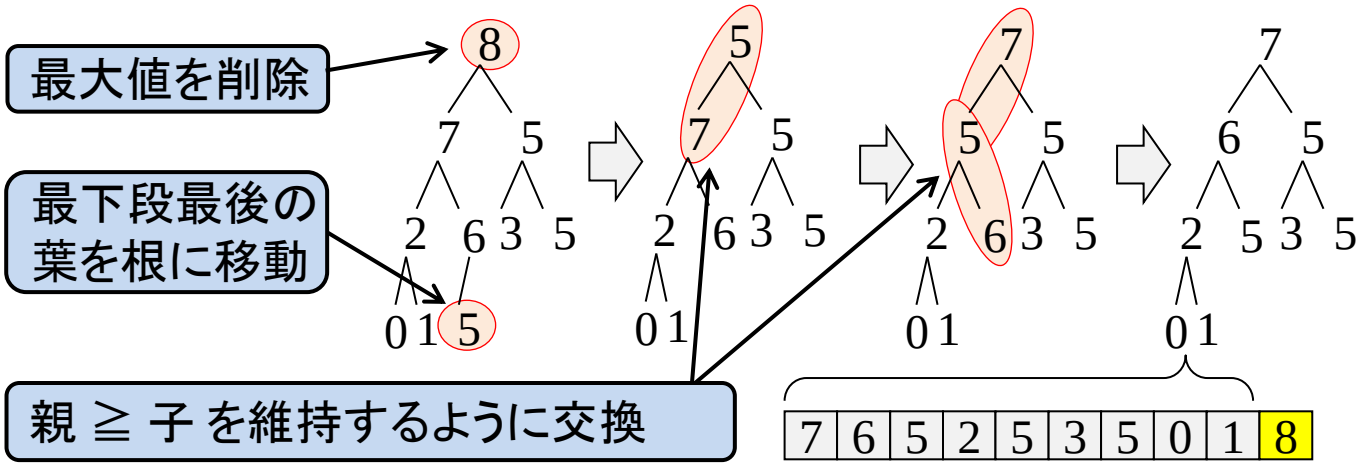
最大値を削除+ヒープの維持（詳細は後で）

5	0	3	2	5	8	5	7	1	6

Step 1： 逆順にヒープ化



8	7	5	2	6	3	5	0	1	5



7	6	5	2	5	3	5	0	1	8

ヒープソート（heap sort）

図解は、以前のページのを簡略化しただけ

「親 ≧ 子」ヒープを作り、最大値を1つずつ取り出して並べる

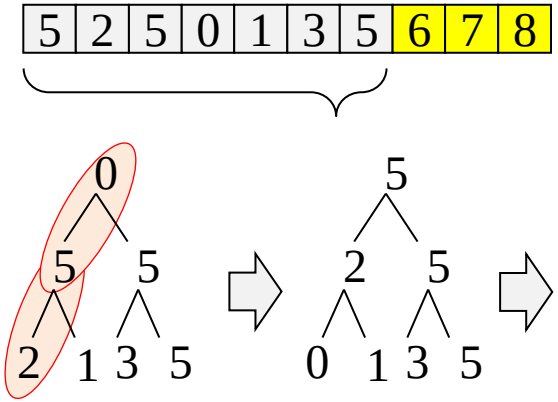
heapsort(A,n):
step 1: Aを逆順にヒープ化
step 2: i ← n-1
step 3: temp ← A[0]
step 4: DELETEMAX(A,i)を実行
step 5: A[i] ← temp
step 6: i ← i-1
step 7: i < 1 ならば停止。そうでなければ step 3 へ

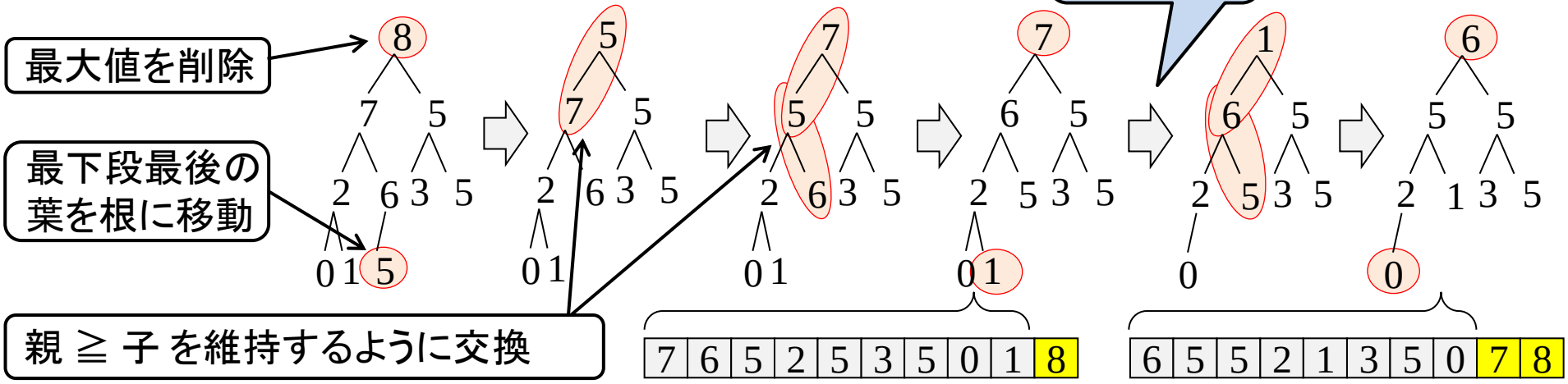
親≧子 となるように配列上でヒープを作る（詳細は後で）

A[0] がヒープの根（最大値）

最大値を削除+ヒープの維持（詳細は後で）

Step 3〜7を繰り返す





ヒープの維持 HEAPIFY(A,i,n)

節点 i に関してヒープの維持

ヒープの大きさは n

左部分木　右部分木

要素数が n の配列 A において、A[2i+1] と A[2i+2] をそれぞれ根とする
2つのヒープを、A[i]を根とするヒープに統合する

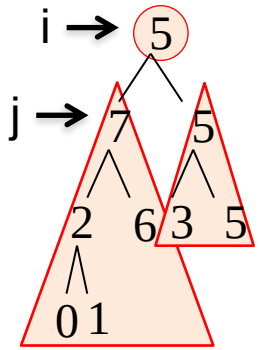
アイデア

A[ ] の値が大きい方の子を j

Step 1: $j \leftarrow \arg\max \{A[k]: k \in \{2i+1, 2i+2\}, k < n\}$

Step 2: $A[i] \geq A[j]$ ならば停止.
そうでなければ A[i] と A[j] の中身の交換（これを子で繰り返す）

HEAPIFY(A,0,9) 実行例



ヒープの維持 HEAPIFY(A,i,n)

節点 i に関してヒープの維持

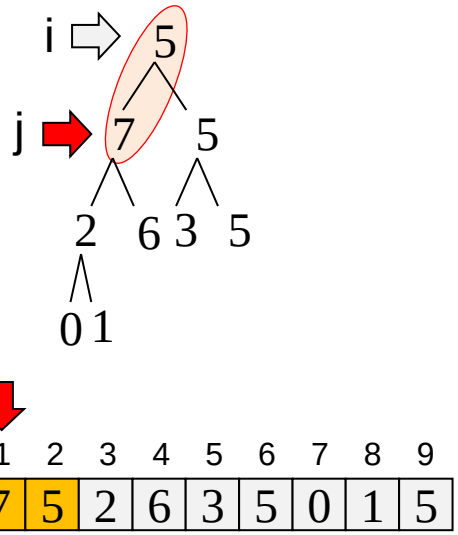
ヒープの大きさは n

左部分木　右部分木

要素数が n の配列 A において、A[2i+1] と A[2i+2] をそれぞれ根とする
2つのヒープを、A[i]を根とするヒープに統合する

Step 1: 2i+1 > n-1 ならば停止.　配列の範囲外
　　そうでないなら j ← arg max {A[k]: k∈{2i+1, 2i+2}, k < n}　A[ ] の値が大きい方の子を j
Step 2: A[i] ≥ A[j] ならば停止.
　　そうでなければ A[i] と A[j] の中身の交換、i ← j として step 1 へ

HEAPIFY(A,0,9) 実行例



0	1	2	3	4	5	6	7	8	9
5	7	5	2	6	3	5	0	1	5

・もともと A は A[9] までの配列だった
・今回は n = 9 (ヒープの大きさ 9) で呼ばれたので、
　それと関係なく A[0] ～ A[n-1] をヒープとして扱う

ヒープの維持 HEAPIFY(A,i,n)

節点 i に関してヒープの維持

ヒープの大きさは n

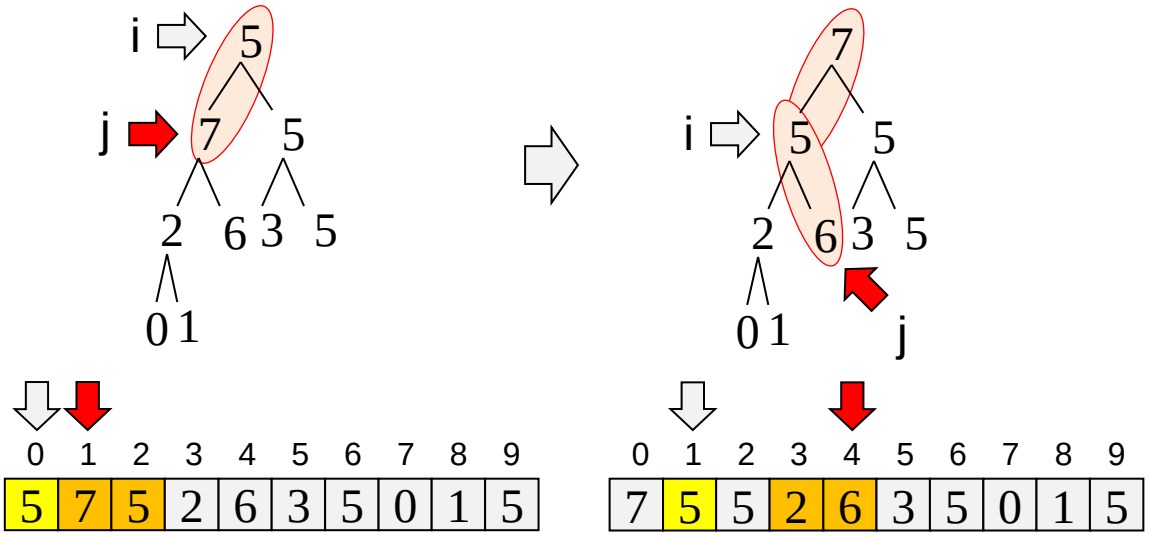
左部分木　右部分木

要素数が n の配列 A において、A[2i+1] と A[2i+2] をそれぞれ根とする
2つのヒープを、A[i]を根とするヒープに統合する

Step 1: 2i+1 > n-1 ならば停止.　配列の範囲外　A[ ] の値が大きい方の子を j
　　そうでないなら j ← arg max {A[k]: k∈{2i+1, 2i+2}, k < n}

Step 2: A[i] ≥ A[j] ならば停止.
　　そうでなければ A[i] と A[j] の中身の交換、i ← j として step 1 へ

HEAPIFY(A,0,9) 実行例



ヒープの維持 HEAPIFY(A,i,n)

節点 i に関してヒープの維持

ヒープの大きさは n

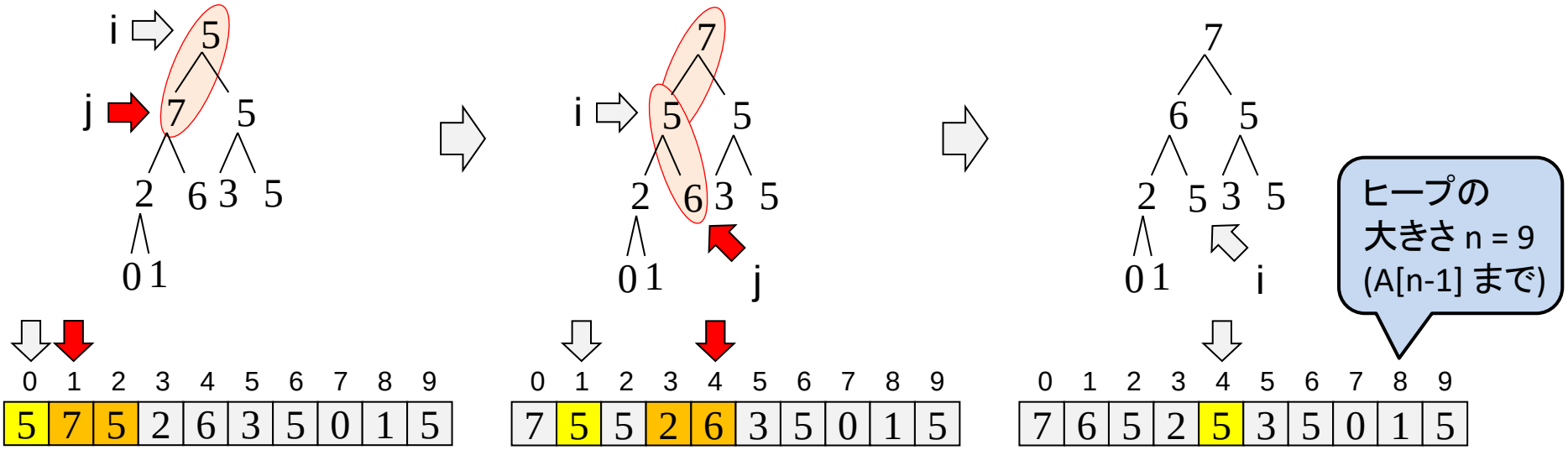
左部分木　右部分木

要素数が n の配列 A において、A[2i+1] と A[2i+2] をそれぞれ根とする
2つのヒープを、A[i]を根とするヒープに統合する

Step 1: 2i+1 > n-1 ならば停止.　配列の範囲外　A[ ] の値が大きい方の子を j
　　そうでないなら j ← arg max {A[k]: k∈{2i+1, 2i+2}, k < n}

Step 2: A[i] ≥ A[j] ならば停止.
　　そうでなければ A[i] と A[j] の中身の交換、i ← j として step 1 へ

HEAPIFY(A,0,9) 実行例



サブルーチン DELETEMAX(A,tail)

ヒープは
最初 A[0] ～ A[tail]
に入っている

逆順ヒープの最大値を削除（A[tail] の要素を根に）

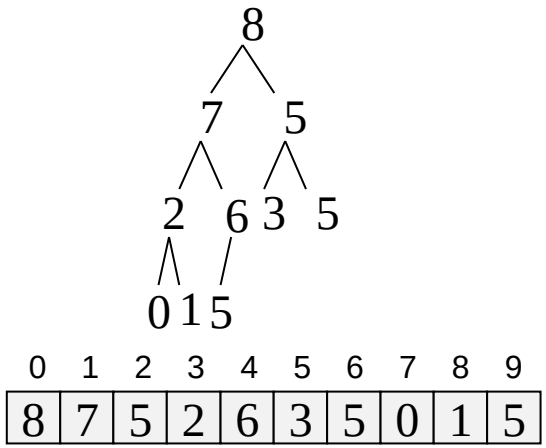
+ヒープの維持

step 1: A[0] ← A[tail]

step 2: HEAPIFY(A,0,tail) を実行

ヒープは A[0] ～ A[tail-1] に
入っている（Aの大きさは tail）

heapsort(A,10) を実行すると、
最初の DELETEMAX( ) は、DELETEMAX(A, 9) で実行



サブルーチン DELETEMAX(A,tail)

ヒープは
最初 A[0] ～ A[tail]
に入っている

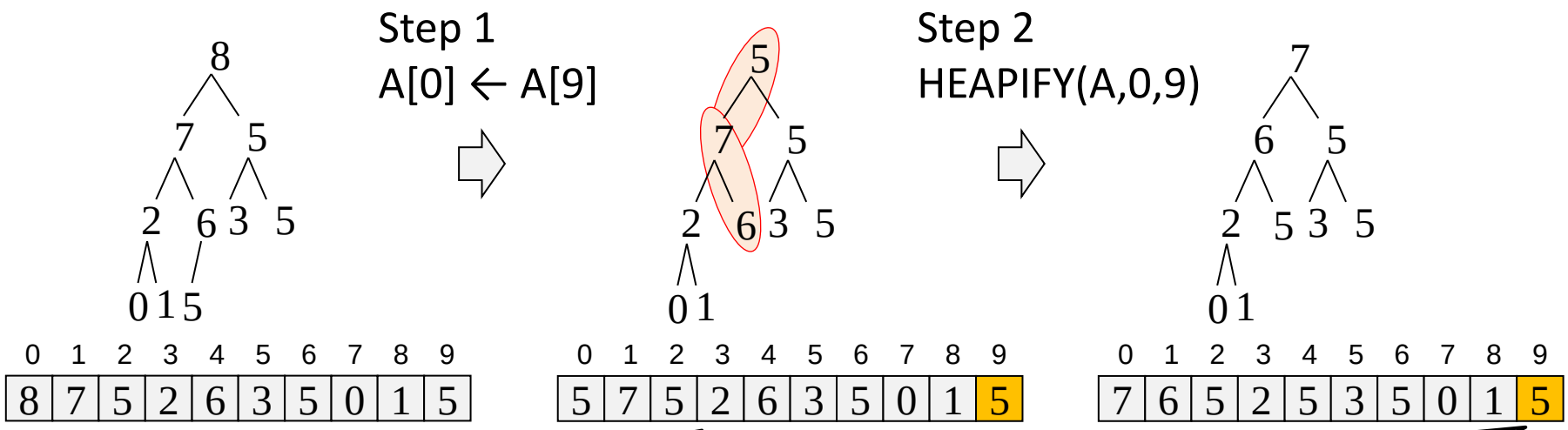
逆順ヒープの最大値を削除（A[tail] の要素を根に）
+ヒープの維持

step 1: A[0] ← A[tail]

step 2: HEAPIFY(A,0,tail) を実行

ヒープは A[0] ～ A[tail-1] に
入っている（Aの大きさは tail）

heapsort(A,10) を実行すると、
最初の DELETEMAX( ) は、DELETEMAX(A, 9) で実行



ヒープは A[0] ～ A[8]

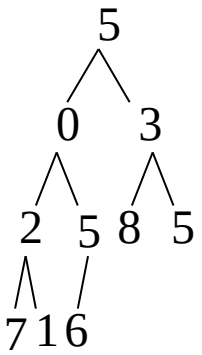
後で、A[9] ← temp（もとの最大値）を実行する

ヒープ構築の概要

- 節点 i の左部分木がヒープ、右部分木がヒープ、

⇩

- HEAPIFY(A, i, n) を実行すると、
  節点 i と、その左部分木のヒープ、右部分木のヒープが
  1 つのヒープになる（既に学習済み）



5	0	3	2	5	8	5	7	1	6

- 葉は、大きさ 1 のヒープ
- 葉以外の節点は、葉の側から根の側へ、ヒープにしていけばよい

i = ⌊(n-2)/2⌋, ⌊(n-2)/2⌋–1, …, 1, 0 の節点 i

HEAPIFY(A, i, n)

最後の葉 A[n – 1] の親

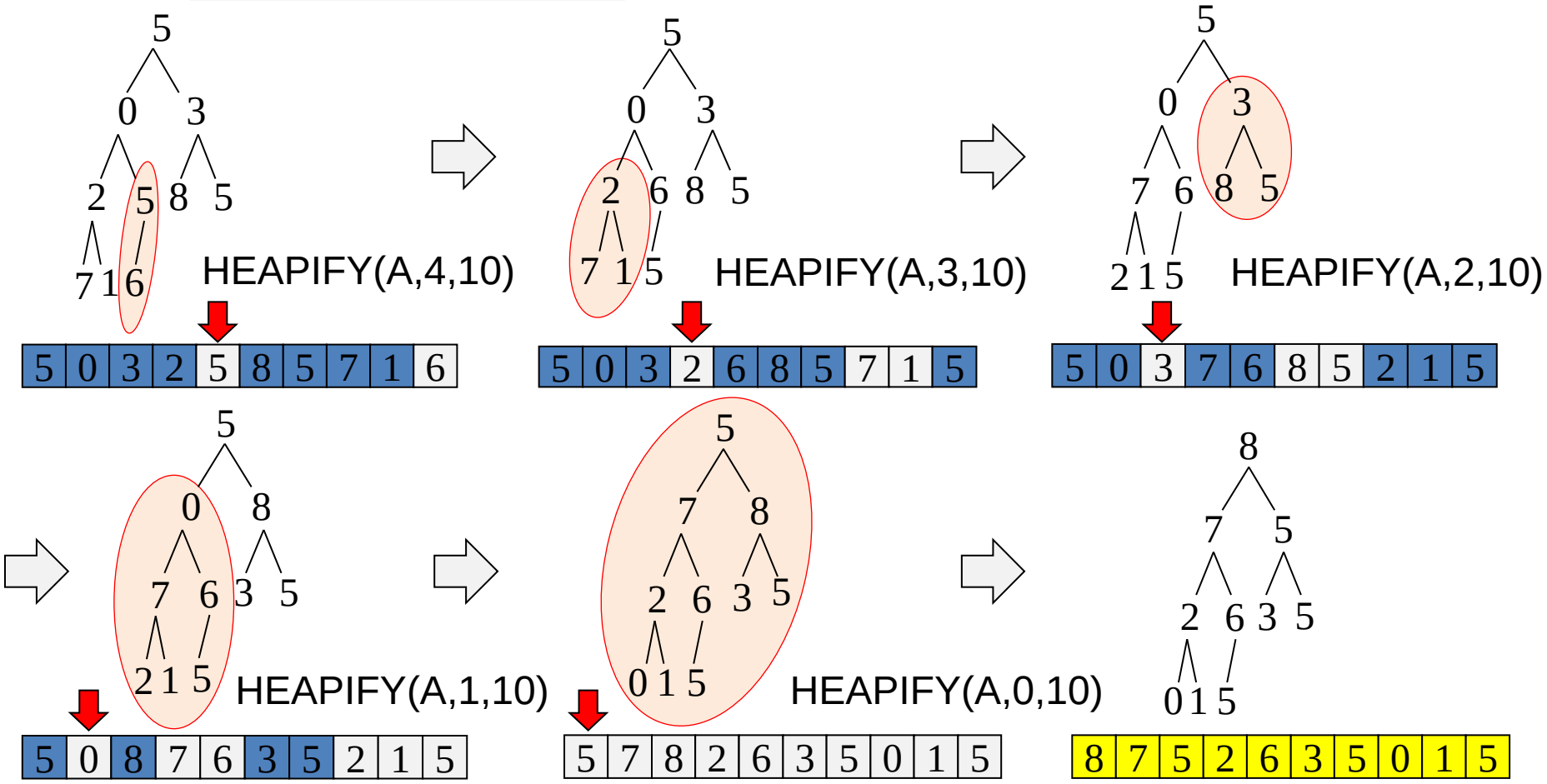
根

ヒープ構築アルゴリズム

Step 1. For i = ⌊(n-2)/2⌋, …, 0 HEAPIFY(A, i, n)

最後の葉 A[n – 1] の親

根



最悪時間計算量 $O(n \log n)$ である証明

発展

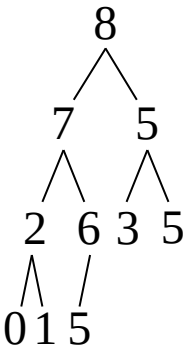
【1】ヒープ構築にかかる時間計算量が$O(n)$であることを示す．

n個の要素からなるヒープの木の**高さをh**とする．高さ i の部分木は高々 2^{h-i} 個ある．

高さ i の木でのHEAPIFYは高々 i 回の置き換えが起こる．計算時間は

$$O\left(2^{h-1}\cdot 1 + 2^{h-2}\cdot 2 + \cdots + 2\cdot(h-1) + 1\cdot h\right)$$

となる．$2^h \leq n$ なので，



$$2^{h-1}\cdot 1 + 2^{h-2}\cdot 2 + \cdots + 2^{h-(h-1)}\cdot(h-1) + 1\cdot h$$

$$\leq \left(\frac{n}{2}\right)\cdot 1 + \left(\frac{n}{2^2}\right)\cdot 2 + \cdots + \left(\frac{n}{2^{h-1}}\right)\cdot(h-1) + \frac{n}{2^h}\cdot h \leq \left(\frac{n}{2}\right)\sum_{i=1}^{h}\left(\frac{i}{2^{i-1}}\right) \leq 2n$$

※ $S_h = \sum_{i=1}^{h}\left(\frac{i}{2^{i-1}}\right)$は，$2S_h - S_h = 2 + \sum_{i=1}^{h-1}\left(\frac{1}{2^{i-1}}\right) - \frac{h}{2^{h-1}}$ より，4以下に抑えられる

よって，**ヒープ構築**の時間計算量は **$O(n)$**．

【2】**DELETEMAX** を $n-1$ 回実行するのにかかる時間計算量は

$$O\left(\sum_{i=1}^{n-1}\log i\right) \leq \mathbf{O(n\ \log n)}.$$

よって，**全体**の時間計算量は **$O(n \log n)$** となる．

休憩

- ここで、少し休憩しましょう。
- 深呼吸したり、肩の力を抜いてから、
  次のビデオに進んでください。

発展

比較に基づくソートの漸近的下界

比較に基づくなら、どんなアルゴリズムでも、この下界

2要素の比較に基づくn要素のソートの
最悪(平均)時間計算量の漸近的下界は
$\Omega(n \log n)$

比較に基づくソートの漸近的下界

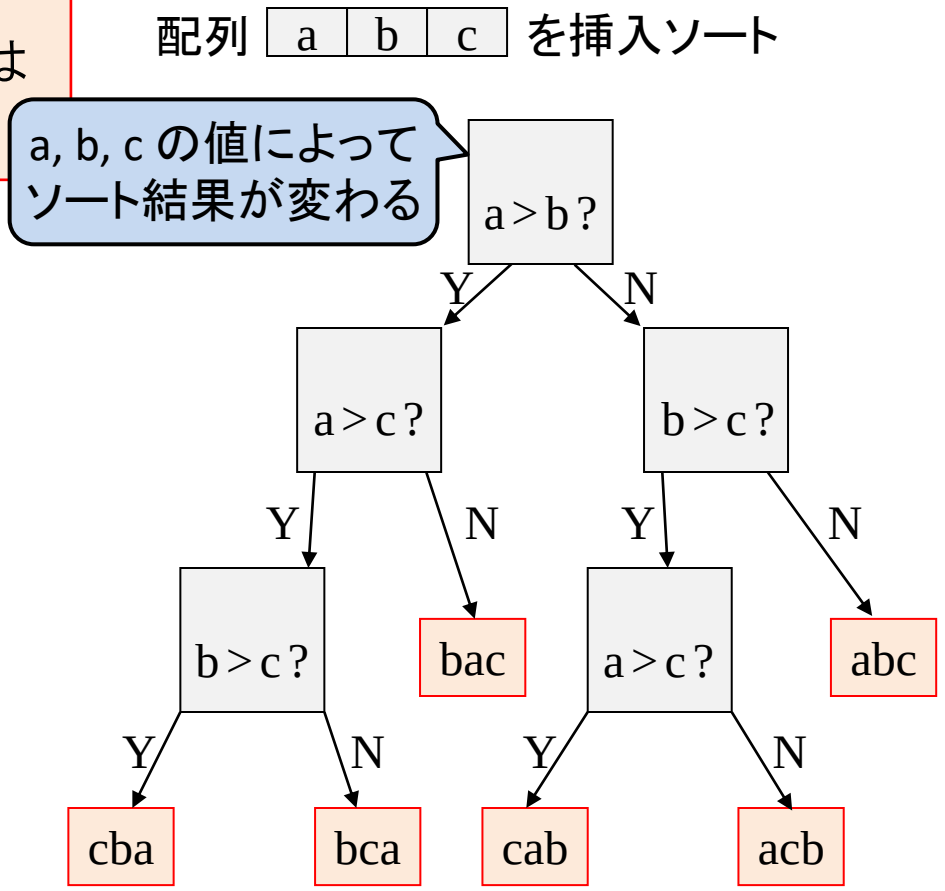
比較に基づくなら、どんなアルゴリズムでも、この下界

2要素の比較に基づくn要素のソートの
最悪(平均)時間計算量の漸近的下界は
$\Omega(n \log n)$

簡単のため，すべての値は異なると仮定．
整列アルゴリズムは，**比較命令の結果により次の状態が決まり，状態ごとに次の処理が決まる**と考えることができる．

⇩

計算の過程は状態を節点とする二分木の根の状態から出発し，その節点の処理である比較命令の結果により2つの子のどちらかに進み，最後には葉の節点まで到達するパス(path)で表現できる．
(**木の高さが，計算時間に対応する**．)



異なる順列は同じ処理では整列できないので，必ず異なる葉に到達する．

比較に基づくソートの漸近的下界（続き）

発展

到達した葉の深さは比較命令の回数であるので，少なくとも葉の深さの定数倍の計算時間が必要である．

したがって，最悪の場合，木の高さの定数倍の時間が必要である．

木の高さを h とすれば，木の葉の数は高々 2^h 個であるので

$$2^h \geq n!$$

が成り立たなければならない．

n要素の異なる順列の個数は $n!$ 個

スターリングの近似公式（Stirling's formula）により，$\log n!$ は十分大きな n に対して $n \log n - n$ で近似できるので，適当な定数 c を用いて

$$\boldsymbol{h \geq \log n! \geq c\, n \log n}$$

が十分大きな n に対して成り立つ．

$$\lim_{n\to\infty} \frac{\ln n!}{n \ln n - n} \to 1$$

したがって $\boldsymbol{\Omega(n \log n)}$ は，すべての比較に基づく整列アルゴリズムの最悪時間計算量の**漸近的下界**といえる．

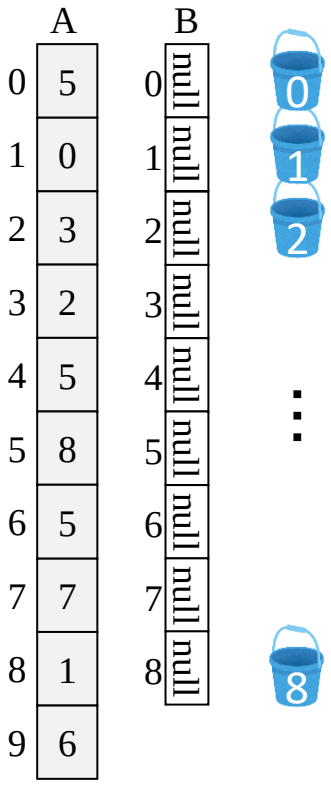
（値を比較しながら整列する場合，少なくとも$cn \log n$時間はかかるということ）

バケットソート（bucket sort）

- 0 以上 m-1 以下の整数の列を整列するのに用いることができる
- 値ごとに外部ハッシュ（バケット）に格納してから大きい順に取り出す

bucketsort(n,A):
A[0],A[1],…,A[n-1]: 0以上m-1以下の整数からなる配列
B[0],B[1],…,B[m-1]: バケット（連結リストのinitポインタ）

step 1: i=0,1,…,n-1の順でB[A[i]]の指すリストの先頭にA[i]を挿入する
step 2: i=m-1,m-2,…,1の順でB[i]の指すリストを，j=n-1,n-2,…,0の順番でA[j]に格納する．ただし，リストは先頭から処理するものとする



最悪/最良/平均時間計算量：$\Theta(m+n)$

m: バケット数, n: 要素数

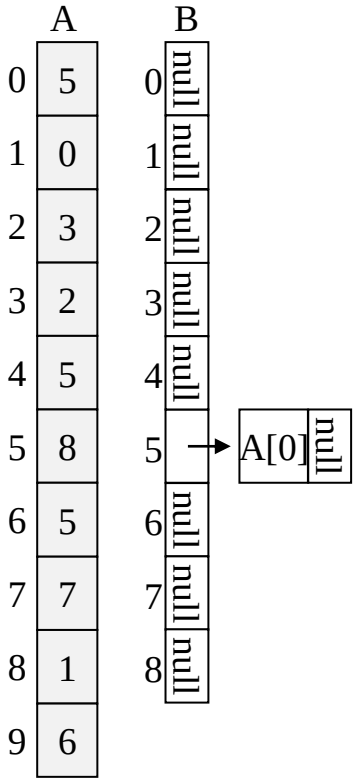
バケットソート（bucket sort）

まず、値ごとにバケツに入れる

- 0 以上 m-1 以下の整数の列を整列するのに用いることができる
- 値ごとに外部ハッシュ（バケット）に格納してから大きい順に取り出す

bucketsort(n,A):
A[0],A[1],…,A[n-1]: 0以上m-1以下の整数からなる配列
B[0],B[1],…,B[m-1]: バケット（連結リストのinitポインタ）

step 1: i=0,1,…,n-1の順でB[A[i]]の指すリストの先頭にA[i]を挿入する
step 2: i=m-1,m-2,…,1の順でB[i]の指すリストを，j=n-1,n-2,…,0の順番でA[j]に格納する．ただし，リストは先頭から処理するものとする



最悪/最良/平均時間計算量：$\Theta(m+n)$

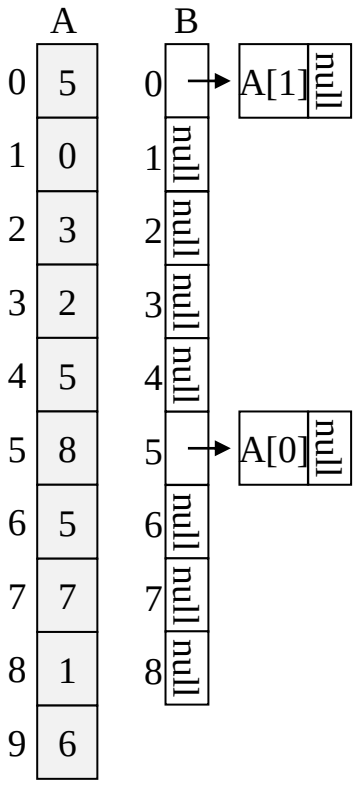
m: バケット数, n: 要素数

バケットソート（bucket sort）

- 0 以上 m-1 以下の整数の列を整列するのに用いることができる
- 値ごとに外部ハッシュ（バケット）に格納してから大きい順に取り出す

bucketsort(n,A):
A[0],A[1],…,A[n-1]: 0以上m-1以下の整数からなる配列
B[0],B[1],…,B[m-1]: バケット（連結リストのinitポインタ）

step 1: i=0,1,…,n-1の順でB[A[i]]の指すリストの先頭にA[i]を挿入する
step 2: i=m-1,m-2,…,1の順でB[i]の指すリストを，j=n-1,n-2,…,0の順番でA[j]に格納する．ただし，リストは先頭から処理するものとする



最悪/最良/平均時間計算量：$\Theta(m+n)$

m: バケット数, n: 要素数

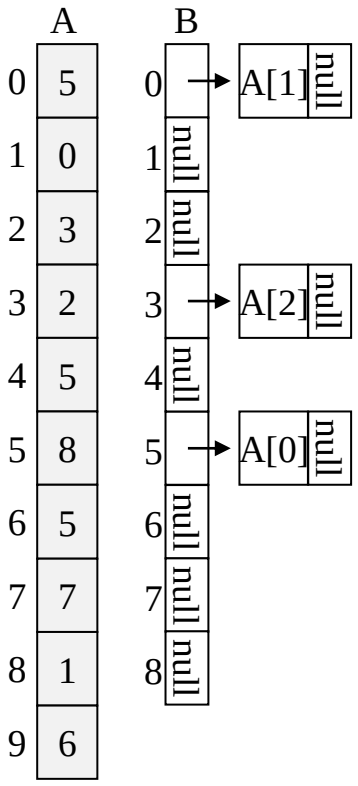
バケットソート（bucket sort）

まず、値ごとにバケツに入れる

- 0 以上 m-1 以下の整数の列を整列するのに用いることができる
- 値ごとに外部ハッシュ（バケット）に格納してから大きい順に取り出す

bucketsort(n,A):
A[0],A[1],…,A[n-1]: 0以上m-1以下の整数からなる配列
B[0],B[1],…,B[m-1]: バケット（連結リストのinitポインタ）

step 1: i=0,1,…,n-1の順でB[A[i]]の指すリストの先頭にA[i]を挿入する
step 2: i=m-1,m-2,…,1の順でB[i]の指すリストを，j=n-1,n-2,…,0の順番でA[j]に格納する．ただし，リストは先頭から処理するものとする



最悪/最良/平均時間計算量：$\Theta(m+n)$

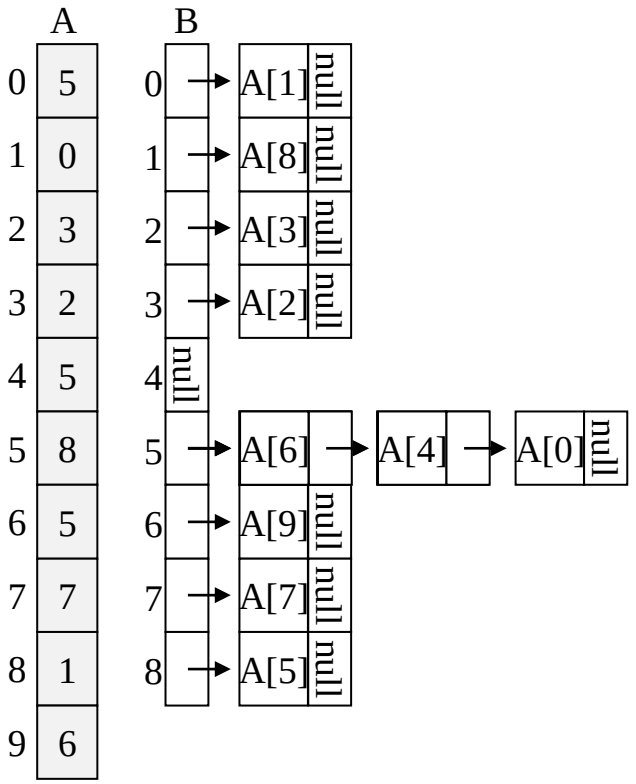
m: バケット数, n: 要素数

バケットソート（bucket sort）

- 0 以上 m-1 以下の整数の列を整列するのに用いることができる
- 値ごとに外部ハッシュ（バケット）に格納してから大きい順に取り出す

bucketsort(n,A):
A[0],A[1],…,A[n-1]: 0以上m-1以下の整数からなる配列
B[0],B[1],…,B[m-1]: バケット（連結リストのinitポインタ）

step 1: i=0,1,…,n-1の順でB[A[i]]の指すリストの先頭にA[i]を挿入する
step 2: i=m-1,m-2,…,1の順でB[i]の指すリストを，j=n-1,n-2,…,0の順番でA[j]に格納する．ただし，リストは先頭から処理するものとする



最悪/最良/平均時間計算量：$\Theta(m+n)$

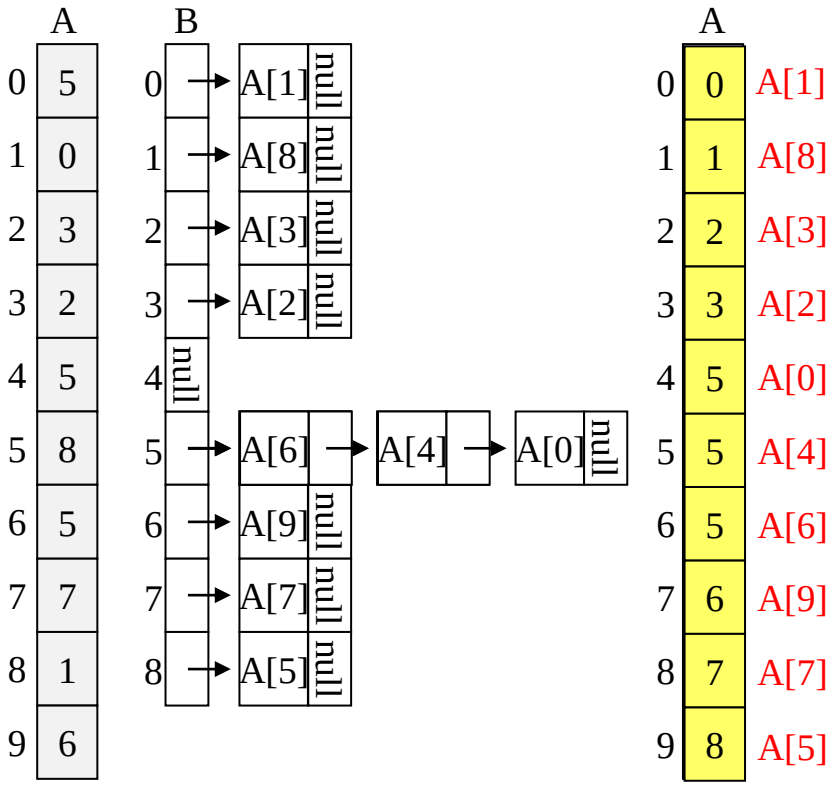
m: バケット数, n: 要素数

バケットソート（bucket sort）

- 0 以上 m-1 以下の整数の列を整列するのに用いることができる
- 値ごとに外部ハッシュ（バケット）に格納してから大きい順に取り出す

bucketsort(n,A):
A[0],A[1],…,A[n-1]: 0以上m-1以下の整数からなる配列
B[0],B[1],…,B[m-1]: バケット（連結リストのinitポインタ）

step 1: i=0,1,…,n-1の順でB[A[i]]の指すリストの先頭にA[i]を挿入する
step 2: i=m-1,m-2,…,1の順でB[i]の指すリストを，j=n-1,n-2,…,0の順番でA[j]に格納する．ただし，リストは先頭から処理するものとする



最悪/最良/平均時間計算量：$\Theta(m+n)$

m: バケット数, n: 要素数

基数ソート（radix sort）

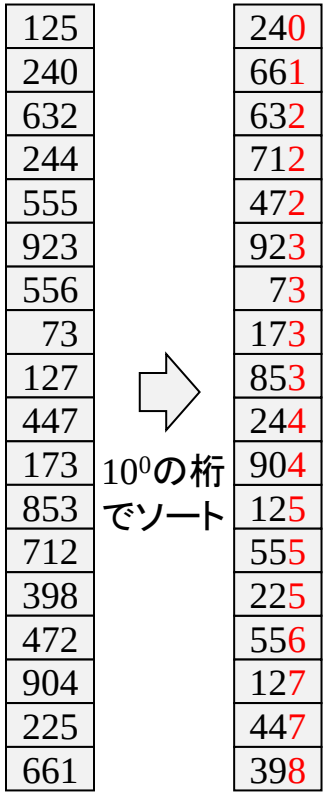
- 0 以上 m^K-1 以下の整数からなる配列に対して用いられる
- 小さい桁から順に桁ごとにバケットソートを適用してソートする

radixsort(n,A):
A[0],A[1],…,A[n-1]: 0以上m^K-1以下の整数からなる配列
B[0],B[1],…,B[m-1]: バケット（連結リストのinitポインタ）

step 1: j=0,1,…,K-1の順に、A[i]のm^j 桁目を使ってバケットソート

※ m^j 桁目が同じ値なら、元の出現順を保つこと

この性質を持つソートは「安定ソート」と呼ばれる



最悪/最良/平均時間計算量：$\Theta(K(m+n))$

m: バケット数, n: 要素数, K: 桁数

基数ソート（radix sort）

・10^1 の桁の小さい順に並ぶ
・10^1 の桁が同じ場合には、
10^0 の桁でソートした順になる

- 0 以上 m^K-1 以下の整数からなる配列に対して用いられる
- 小さい桁から順に桁ごとにバケットソートを適用してソートする

radixsort(n,A):
A[0],A[1],…,A[n-1]: 0以上m^K-1以下の整数からなる配列
B[0],B[1],…,B[m-1]: バケット（連結リストのinitポインタ）

step 1: j=0,1,…,K-1の順に、A[i]のm^j 桁目を使ってバケットソート

※ m^j 桁目が同じ値なら、元の出現順を保つこと

この性質を持つソートは「安定ソート」と呼ばれる

	10^0の桁でソート		10^1の桁でソート	
125		240		904
240		661		712
632		632		923
244		712		125
555		472		225
923		923		127
556		73		632
73		173		240
127		853		244
447		244		447
173		904		853
853		125		555
712		555		556
398		225		661
472		556		472
904		127		73
225		447		173
661		398		398

最悪/最良/平均時間計算量：$\Theta(K(m+n))$

m: バケット数, n: 要素数, K: 桁数

基数ソート（radix sort）

・10^2 の桁の小さい順に並ぶ
・10^2 の桁が同じ場合には、
10^1, 10^0 の桁でソートした順になる

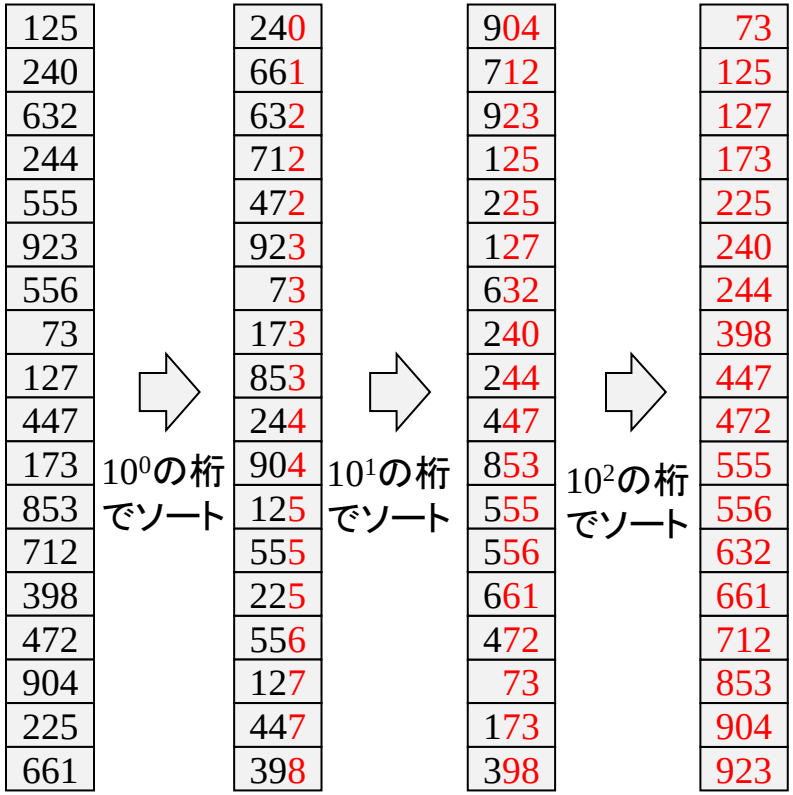
- 0 以上 m^K-1 以下の整数からなる配列に対して用いられる
- 小さい桁から順に桁ごとにバケットソートを適用してソートする

radixsort(n,A):
A[0],A[1],…,A[n-1]: 0以上m^K-1以下の整数からなる配列
B[0],B[1],…,B[m-1]: バケット（連結リストのinitポインタ）

step 1: j=0,1,…,K-1の順に、A[i]のm^j 桁目を使ってバケットソート

※ m^j 桁目が同じ値なら、元の出現順を保つこと

この性質を持つソートは「安定ソート」と呼ばれる



最悪/最良/平均時間計算量：$\Theta(K(m+n))$

m: バケット数, n: 要素数, K: 桁数

今日のまとめ

- 整列アルゴリズムの種類と特徴（おさらい）

- 最悪時計算量 $O(n \log n)$ 時間の整列アルゴリズム
  - **マージソート**：配列を2つに分けてそれぞれソートし、それらをマージする
  - **ヒープソート**：逆順のヒープを構築してから最大値を1つずつ取り出す
- 比較に基づくソートの漸近的下界は $\boldsymbol{\Omega(n \log n)}$
- 整数データに対する $O(n)$ 時間の整列アルゴリズム
  - **バケットソート**：値ごとにバケットに格納 + 小さい順に取り出す
  - **基数ソート**：下の桁から桁ごとにバケットソートする

付録：Introspective sort※（introsort, 内省的ソート）

- クイックソートの分割の深さが $\log n$ 段になったらヒープソートに切り替えることでクイックソートの弱点を補完する手法

（さらに，分割の範囲が小さくなった時に挿入ソートに切り替えるバリエーションもある）

```
main(n,A):
    maxdepth = 2×⌊log(length(A))⌋
    introsort(A, maxdepth)

introsort(A, maxdepth):
    n ← length(A)
    p ← partition(A)  // pは軸要素の位置
    if n ≤ 1: //要素1個以下なら何もしない
        return
    else if p > maxdepth:
        heapsort(A)
    else:
        introsort(A[0:p], maxdepth - 1)
        introsort(A[p+1:n], maxdepth - 1)
```

Wikipedia(https://en.wikipedia.org/wiki/Introsort)
の疑似コードをもとに改編

最悪時間計算量$O(n \log n)$
平均時間計算量$O(n \log n)$

※ David Musser, “Introspective Sorting and Selection Algorithms”, *Software: Practice and Experience*, Wiley, 27 (8), pp.983–993, 1997.

付録：Tim Peters のソート手法※（TimSort）

- 挿入ソートを使いながら，配列をソート済みの小片（runと呼ばれる長さが32～64の列）に分解し，その後にマージソートを行う
- 様々なヒューリスティックを取り入れたハイブリッドソートの代表格で、PythonやJavaなどの標準ソートとして実装されている

Timsort(n,A):
step 1: Aの先頭から単調増加している接頭辞を求める．その接頭辞が十分に長ければ（[32,64]の範囲にあれば）それを一つのrunとする．短い場合は挿入ソートを使って拡張する
step 2: 隣り合うrunどうしをトーナメント的にマージする

単調増加する接頭辞が長すぎるときは区切る

最悪時間計算量$O(n \log n)$
最良時間計算量$O(n)$
平均時間計算量$O(n \log n)$

2分探索する2分挿入ソートを使う

runの個数は，なるべく2の累乗にする

安定ソート

※Tim Peters, “[Python-Dev] Sorting”, Python Developers Mailinglist. Retrieved 24 February 2011.
Peter McIlroy, “Optimistic sorting and information theoretic complexity”, In proc. of the fourth annual ACM-SIAM symposium on Discrete algorithms (SODA’93), pp 467-474, January1993.

付録：図書館ソート※（library sort）

- 配列の要素間に隙間を設けて $(1+\varepsilon)n$ の長さにし、挿入ソートの挿入操作を高速化する
- 挿入操作は二分探索で行う

librarysort(n,A):
step 1: $(1+\varepsilon)n$分の配列Sを用意する
step 2: $i \leftarrow 1$ から $\lfloor \log n + 1 \rfloor$ まで次のstep 3を繰り返す
step 3: $j \leftarrow 2^i$ から 2^{i+1} まで次の操作を繰り返す
　　配列A の最初の2^{i-1}個の要素に二分探索を行い次の要素の挿入位置posを見つける．S[pos]にA[j]の要素を挿入する．また，適度に隙間が空くように配列Sを調整する
step 4: 隙間を詰めて整列済み配列Aとして整える

最悪時間計算量$O(n^2)$
平均時間計算量$O(n \log n)$

ただし，高い確率で$O(n \log n)$で動作

※ Michael A. Bender, Martín Farach-Colton, and Miguel Mosteiro, “Insertion Sort is O(n log n)”, *Theory of Computing Systems*, 39 (3), pp. 391–397, 2006.