講義「アルゴリズムとデータ構造」

第１１回　整列のアルゴリズム（２）

大学院情報科学研究院 情報理工学部門
情報知識ネットワーク研究室
喜田拓也

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について

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)

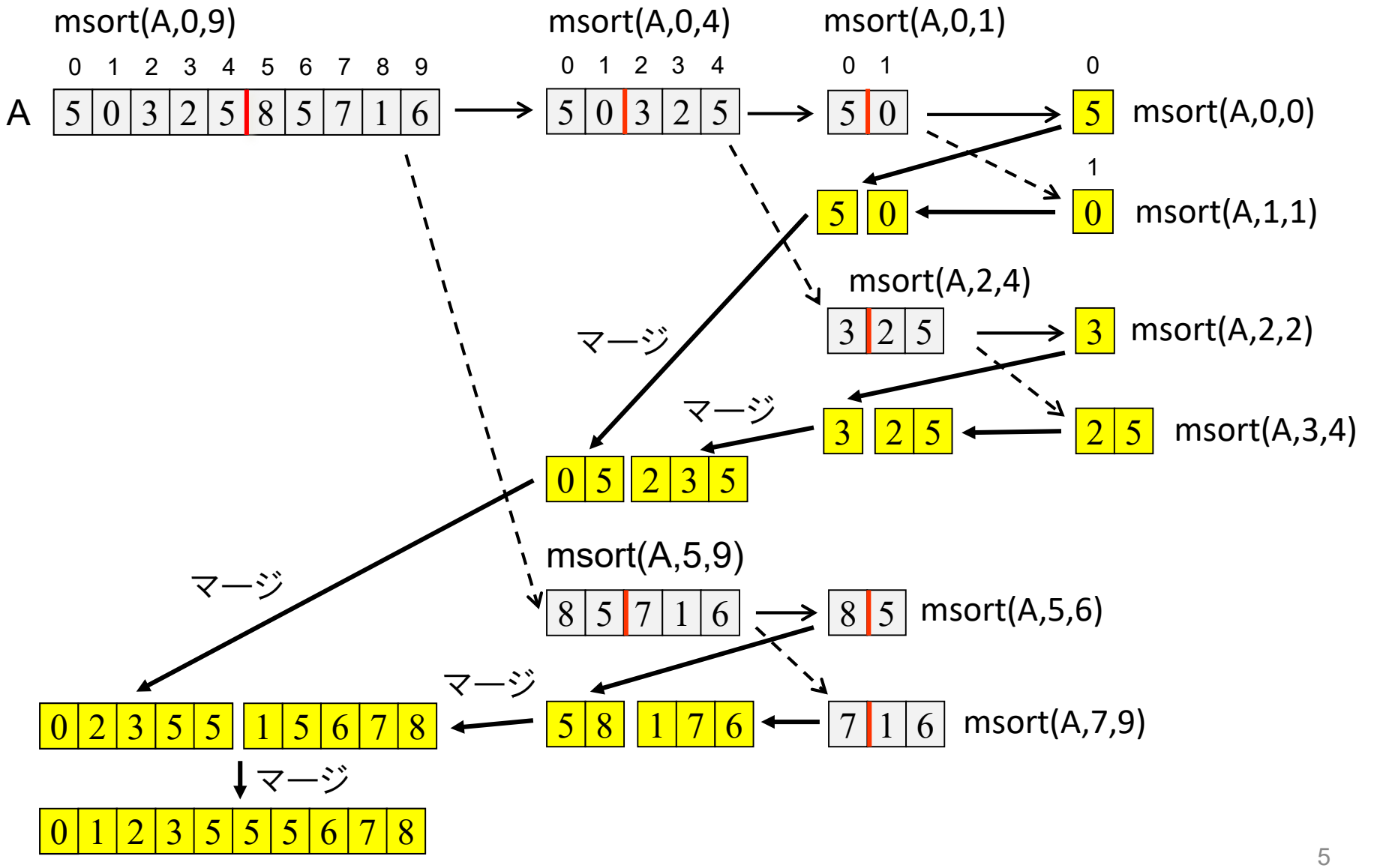
配列を2つに分け，それぞれを整列してからマージする
分割統治法による整列アルゴリズム

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)とは，ソート済みの2つのリストを合成して1つのソート済みリストを作成する処理

最悪/最良/平均時間計算量は$O(n \log n)$

マージソートの動作例


msort(A,0,9)
0 1 2 3 4 5 6 7 8 9
A
5 0 3 2 5 8 5 7 1 6
msort(A,0,4)
0 1 2 3 4
5 0 3 2 5
msort(A,0,1)
0 1
5 0
0
5
msort(A,0,0)
1
5 0
0
msort(A,1,1)
msort(A,2,4)
3 2 5
3
msort(A,2,2)
マージ
3 2 5
2 5
msort(A,3,4)
マージ
0 5 2 3 5
msort(A,5,9)
8 5 7 1 6
8 5
msort(A,5,6)
マージ
0 2 3 5 5
1 5 6 7 8
5 8
1 7 6
7 1 6
msort(A,7,9)
マージ
マージ
0 1 2 3 5 5 5 6 7 8

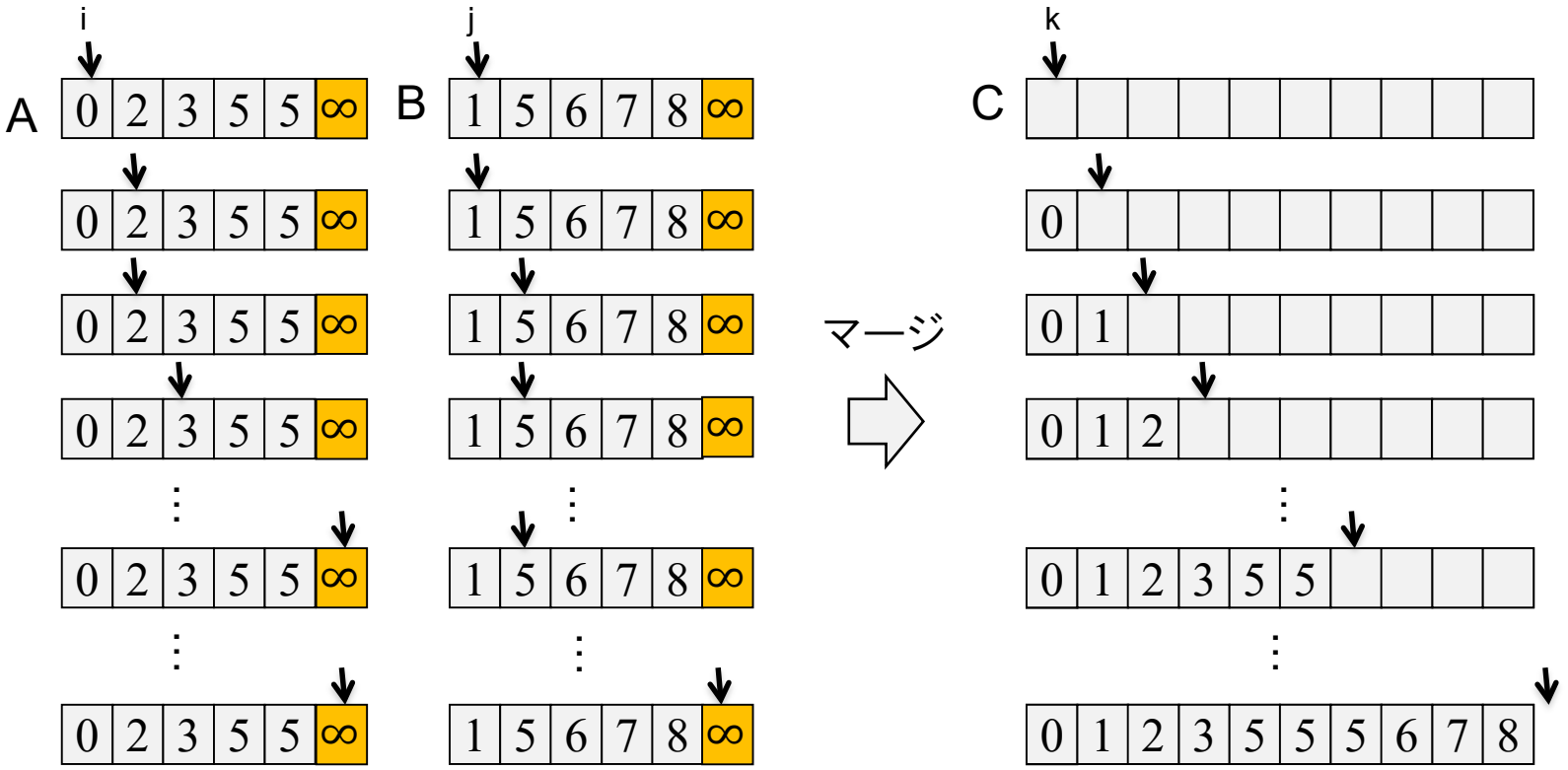
マージ処理

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）←番兵！

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



最悪時間計算量が$O(n \log n)$である証明

$T(n)$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,

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

$T(1)$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,

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

が成り立つ.

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

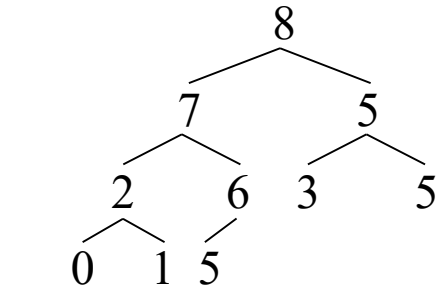
ヒープソート（heap sort）

逆順（大きい順）にヒープを構成し，最大値を1つずつ取り出しながら並べる

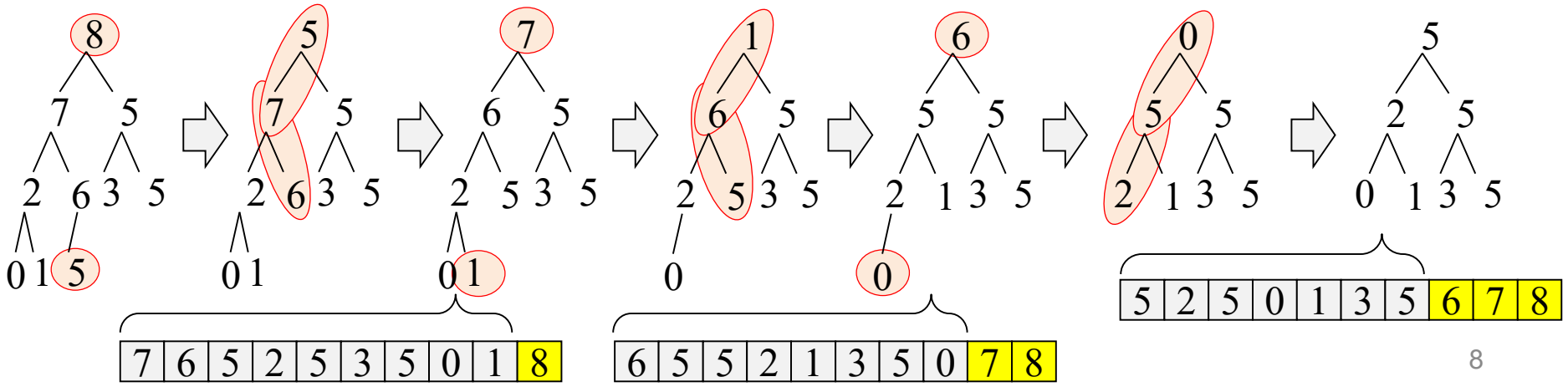
heapsort(A,n):
step 1: Aを逆順にヒープ化
step 2: i←n-1
step 3: temp←A[0]
step 4: DELETEMAX(A,i)を実行
step 5: A[i]←temp
step 6: i←i-1
step 7: i<2ならば停止．そうでなければstep 3へ

5	0	3	2	5	8	5	7	1	6

逆順にヒープ化する



8	7	5	2	6	3	5	0	1	5



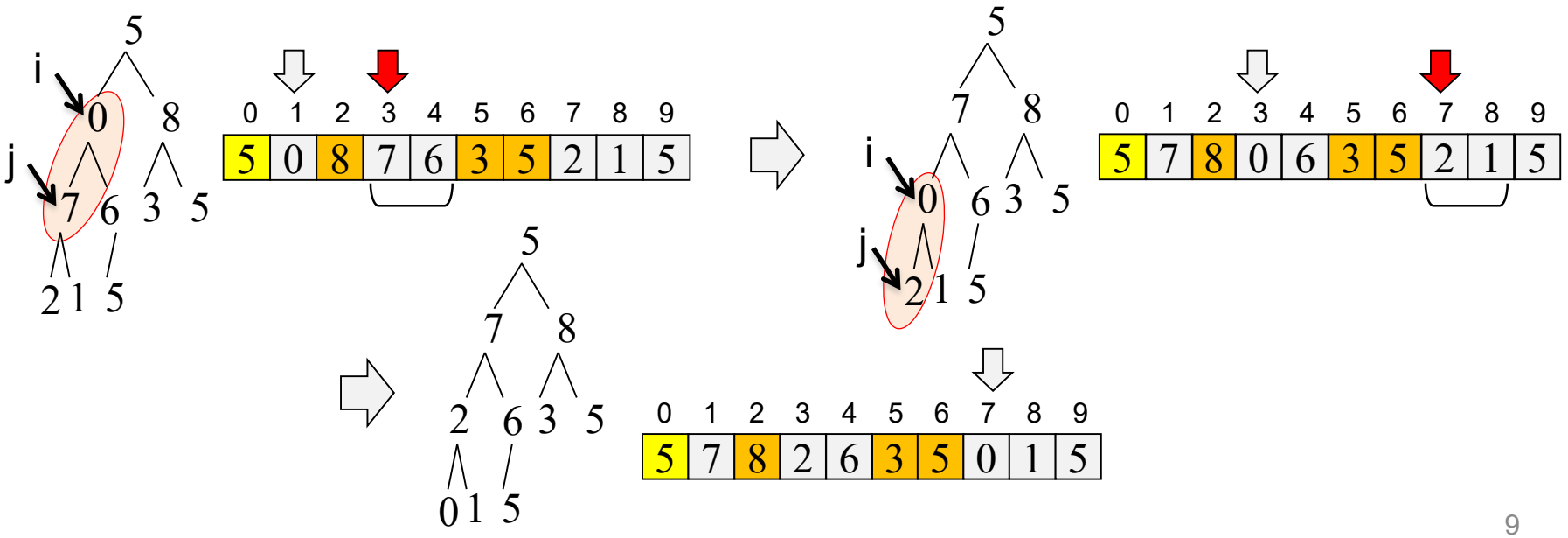
サブルーチンHEAPIFY(A,i,n)

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つの逆順のヒープを，A[i]を根とする逆順のヒープに統合する

step 1: 2i+1>n-1ならば停止．
配列の範囲外
大きいほうの子をj
そうでないなら j←arg max {A[k]: k∈{2i+1,2i+2},k<n} を実行

step 2: A[i]≥A[j]ならば停止．
そうでなければA[i]とA[j]の中身の入れ替え，i←jとしてstep 1へ

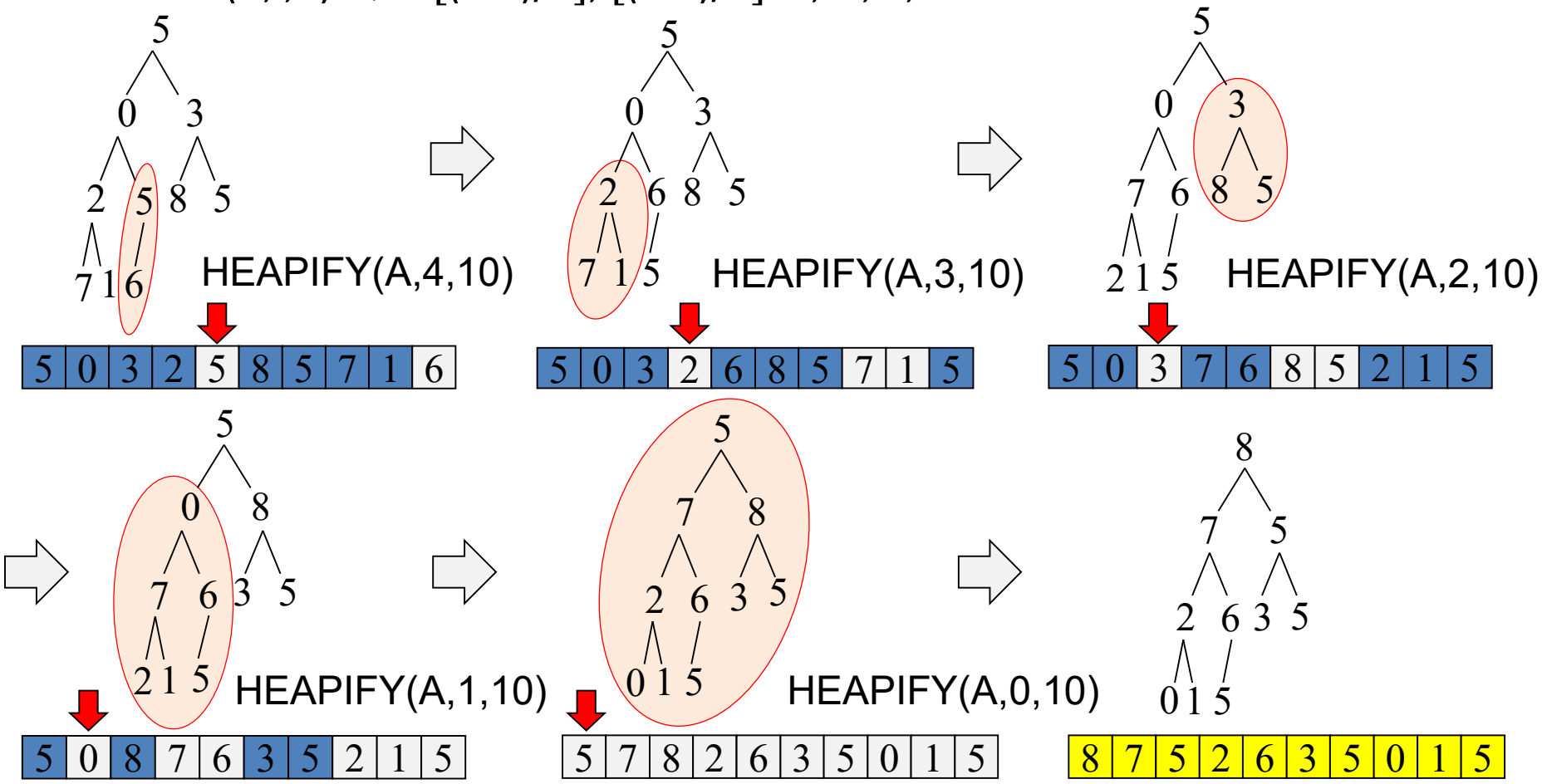
HEAPIFY(A,1,10)実行例



ヒープ構築アルゴリズム

葉ノード

A[⌊(n-2)/2⌋+1], A[⌊(n-2)/2⌋+2], …, A[n-1] を各々1つの節点からなるヒープとみなす
A[2i+1]とA[2i+2]を根とする2つのヒープを，A[i]を根とするヒープに統合する操作
HEAPIFY(A,i,n)を，i=⌊(n-2)/2⌋, ⌊(n-2)/2⌋−1, …, 1, 0 に対して行う

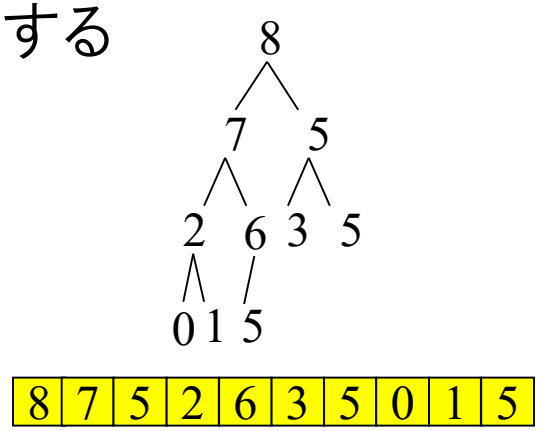


サブルーチンDELETEMAX(A,tail)

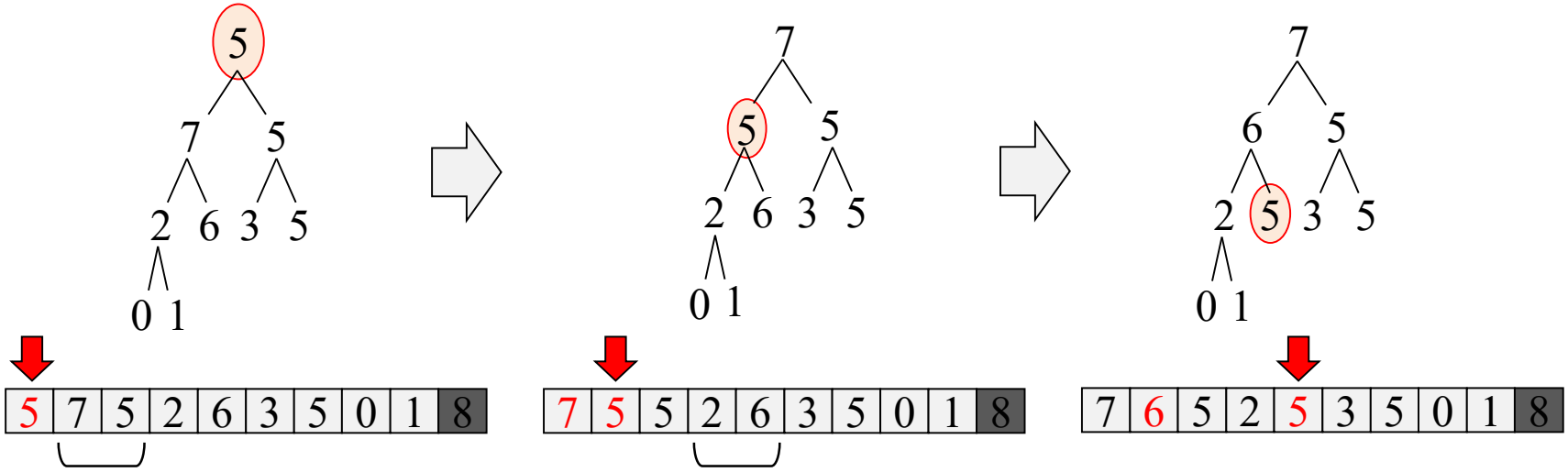
逆順ヒープの最大値（tail番目の要素）を削除する

step 1: A[0]←A[tail]

step 2: HEAPIFY(A,0,tail-1)を実行



A[9]=8を削除したのちの，step 2のHEAPIFY(A,0,9)実行例



最悪時間計算量が$O(n \log n)$である証明

まず, ヒープ化するの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の部分木の数は高々2^{h-i}であり, そのような木でのHEAPIFYはi回の置き換えが起こるので, 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

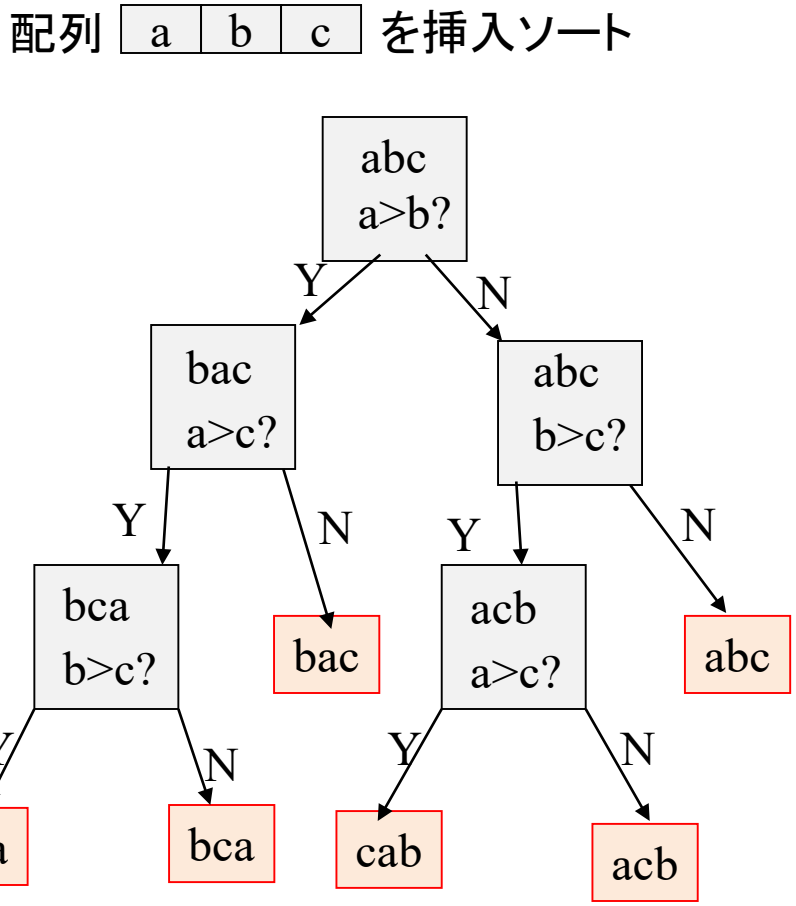
よって, ヒープ化するのにかかる時間計算量は$O(n)$.
次にDELETEMAXを$n-2$回実行するのにかかる時間計算量は

$$O\left(\sum_{i=2}^{n-1} \log i\right) \leq O(n \log n).$$

よって全体の時間計算量は $O(n \log n)$ となる.

比較に基づくソートの漸近的下界

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，全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．

したがって，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．

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）

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

$$h \geq \log n! \geq cn \log n$$

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 $\Omega(n \log n)$ 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以下の整数からなる配列に対して用いることができる

値毎に外部ハッシュ（バケット）に格納してから大きい順に取り出す

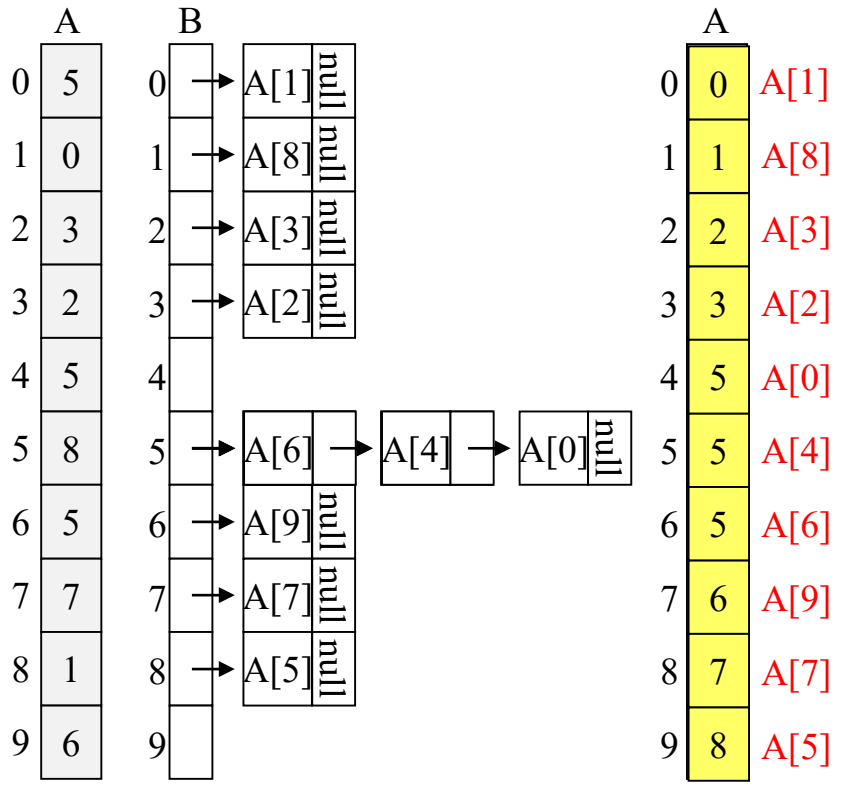
bucketsort(n,A):

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

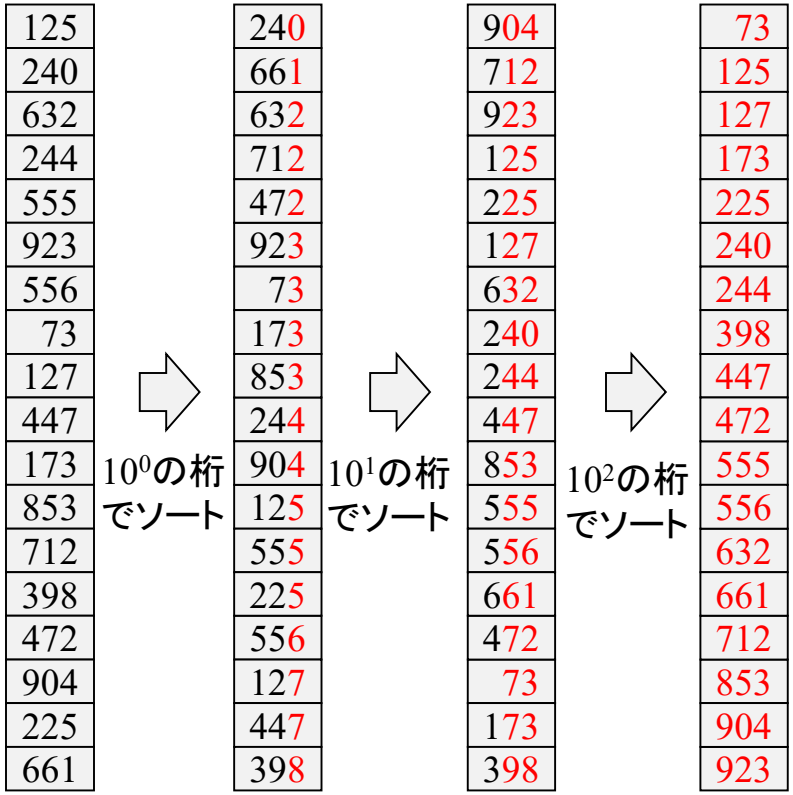
radixsort(n,A):

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: k=0,1,…,K-1の順にA[i]のm^kの桁に関して同じ値の出現順を保つバケットソートを行う

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：配列を二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$\Omega(n \log n)$

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：値ごとに連結リストに格納してから小さい順に取り出す
- 基数ソート：桁ごとにバケットソートする

付録：内省ソート※（introsort）

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.「内省ソート」という訳語はたぶん喜田オリジナル（中国語訳は内省排序）

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.