アルゴリズムとデータ構造

整列 (Sort)

情報学研究科 知能情報学専攻
音声メディア分野
吉井 和佳
yoshii@kuis.kyoto-u.ac.jp

ソートとは

- ある配列データを昇順 (あるいは降順) に並び替えること
  - アルファベットであれば文字コード (整数) の順番に並び替える
  - 他の種類のデータでもデータ間の大小比較さえ定義できればOK

5　3　8　1　6　21　11

ソートアルゴリズム

1　3　5　6　8　11　21

a l g o r i t h m

JIS/ECU/UTF-8の文字コード
61 6C 67 6F 72 69 74 68 6D
(16進数)

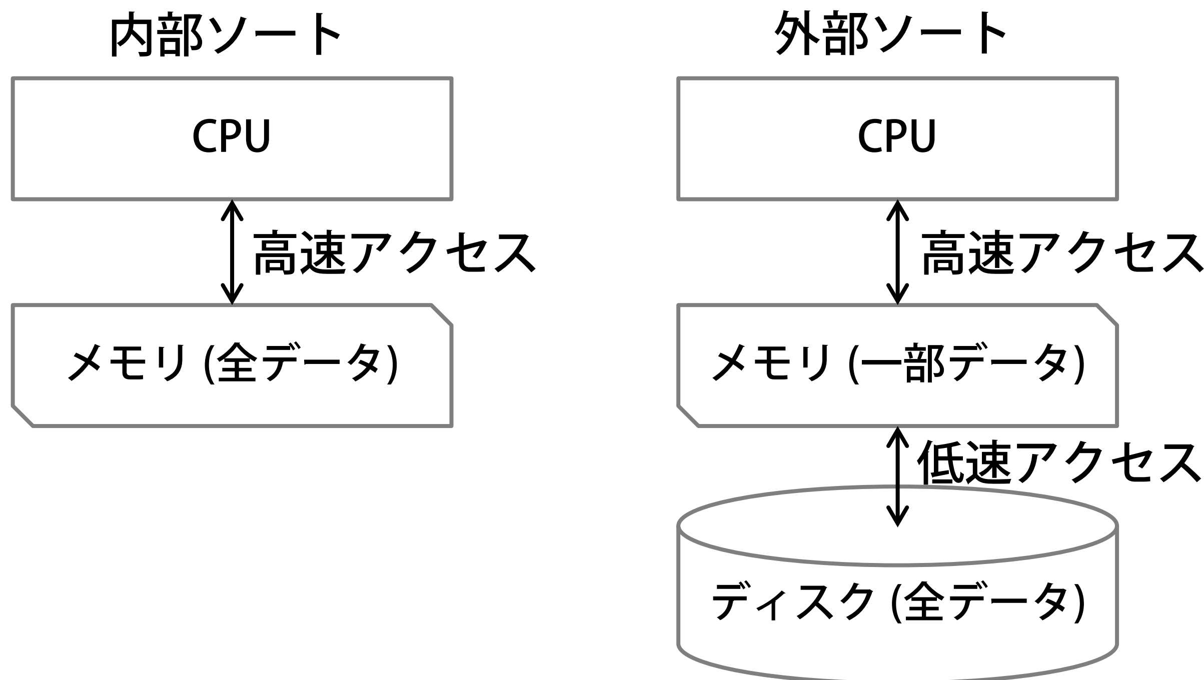
ソートアルゴリズム

a g h i l m o r t

アルゴリズムによって
得意/不得意なデータや
時間/空間計算量が異なる！

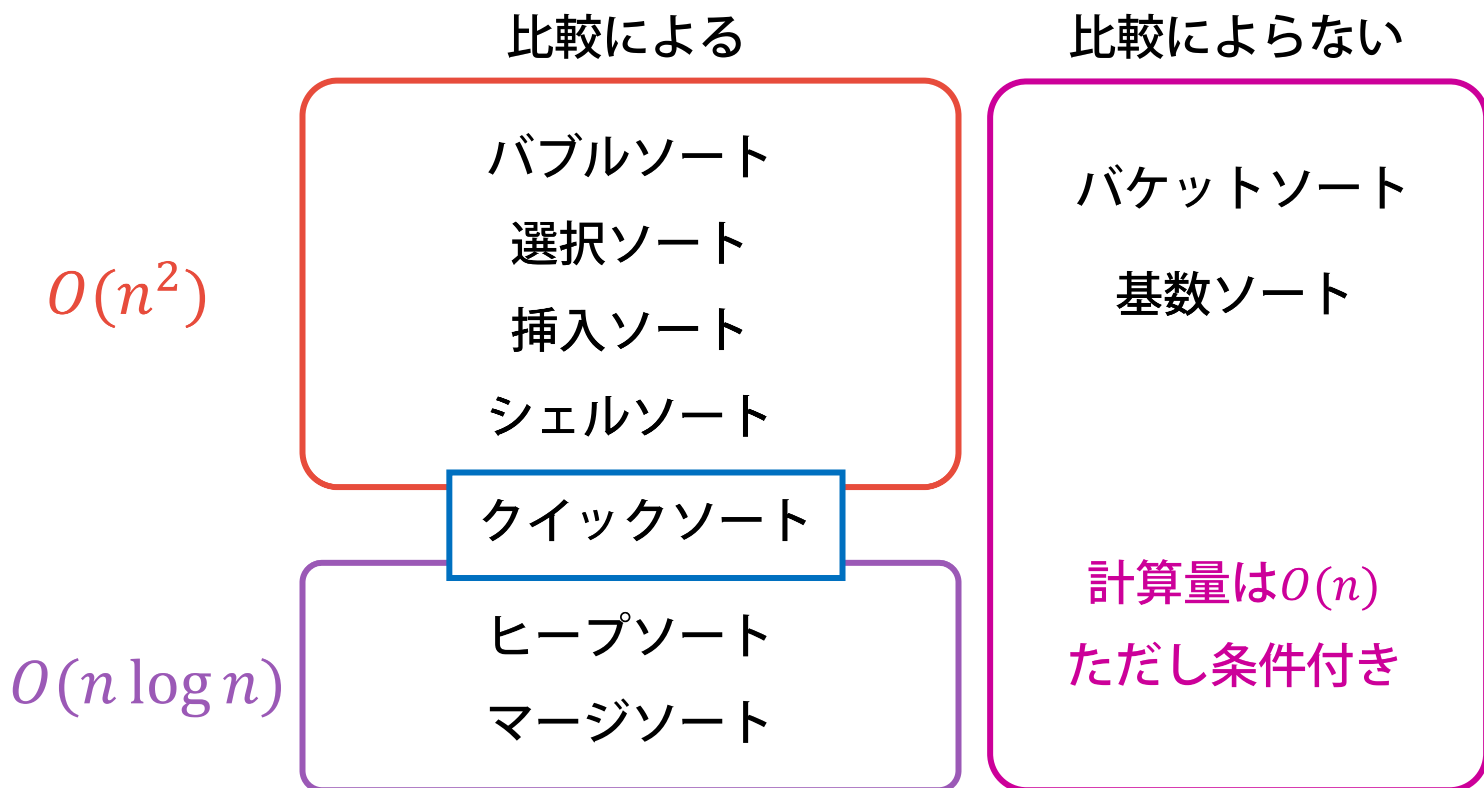
内部ソートと外部ソート

- 全データをメモリ上に保持できるかが設計思想に影響
  - 内部ソート：メモリ上に全データが存在 → こちらを解説
  - 外部ソート：メモリ上には一部のデータが存在



ソートアルゴリズムの種類

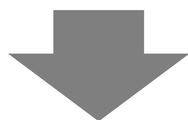
- 計算量が異なる様々なアルゴリズムが存在・適材適所
  - 多くの場合クイックソートが最速 (多くの処理系で組み込み実装)



入出力の仕様

- 入力データは配列 (vector) で与えられると仮定
  - 実際には連想配列 (hash/map) のこともある

A[0]	A[1]	A[2]					A[n-1]
5	3	8	1	4	13	9	2

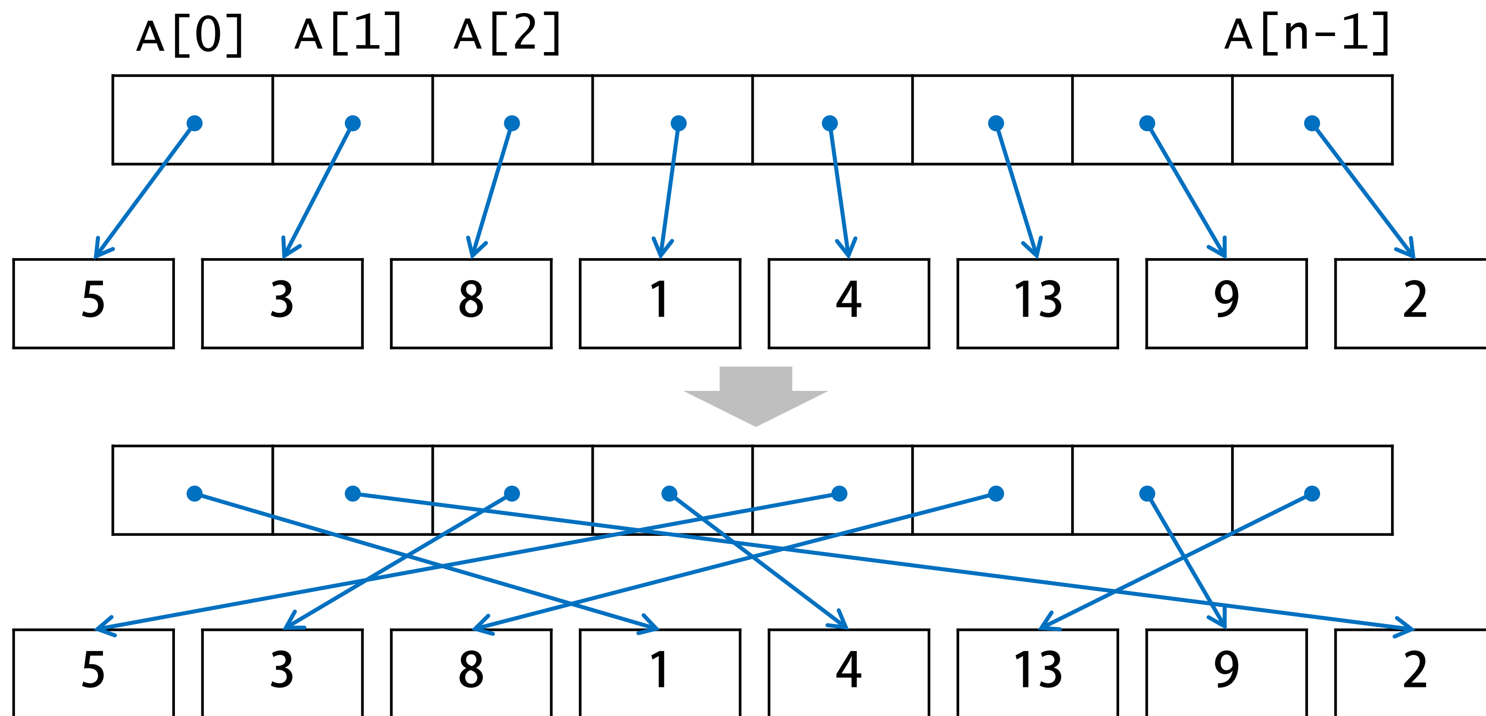


計算途中ではできる限りこれ以外のメモリ領域を使用しない方が望ましい (空間計算量の削減)

A[0]	A[1]	A[2]					A[n-1]
1	2	3	4	5	8	9	13

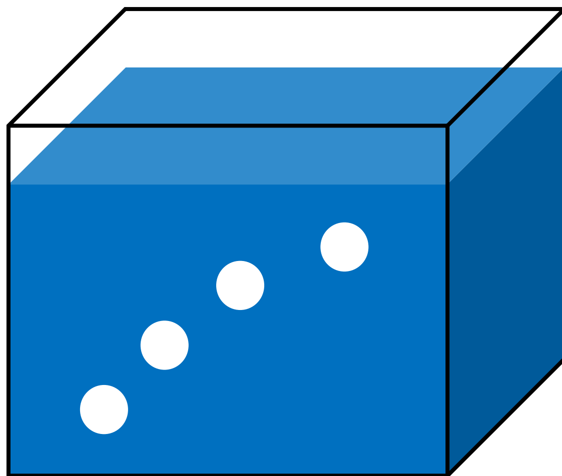
大きいデータの取り扱い方

- 大きいデータはポインタを通して取り扱う
  - データそのものをスワップするのは計算負荷が高い
  - データを格納している”アドレス”のみを操作する



バブルソート

- アルゴリズムの方針
  - 隣同士比べて小さいほうを前方にする交換操作を行う
  - 先頭の方はソートされている状態にしておく
  - これらを繰り返して全体をソートする



バブルソートの由来
泡が浮かびあがってくるかのごとく
小さいデータが徐々に集まってくる

バブルソートの動き

- となり合う要素を順番に比較して入れ替え
  - 小さい要素を先頭に浮かび上がらせていく

先頭のみソートOK

5	5	5	5	5	5	5	1
3	3	3	3	3	3	1	5
8	8	8	8	8	1	3	3
1	1	1	1	1	8	8	8
4	4	4	2	2	2	2	2
13	13	2	4	4	4	4	4
9	2	13	13	13	13	13	13
2	9	9	9	9	9	9	9

交換

交換

交換

非交換

交換

交換

交換

バブルソートの動き

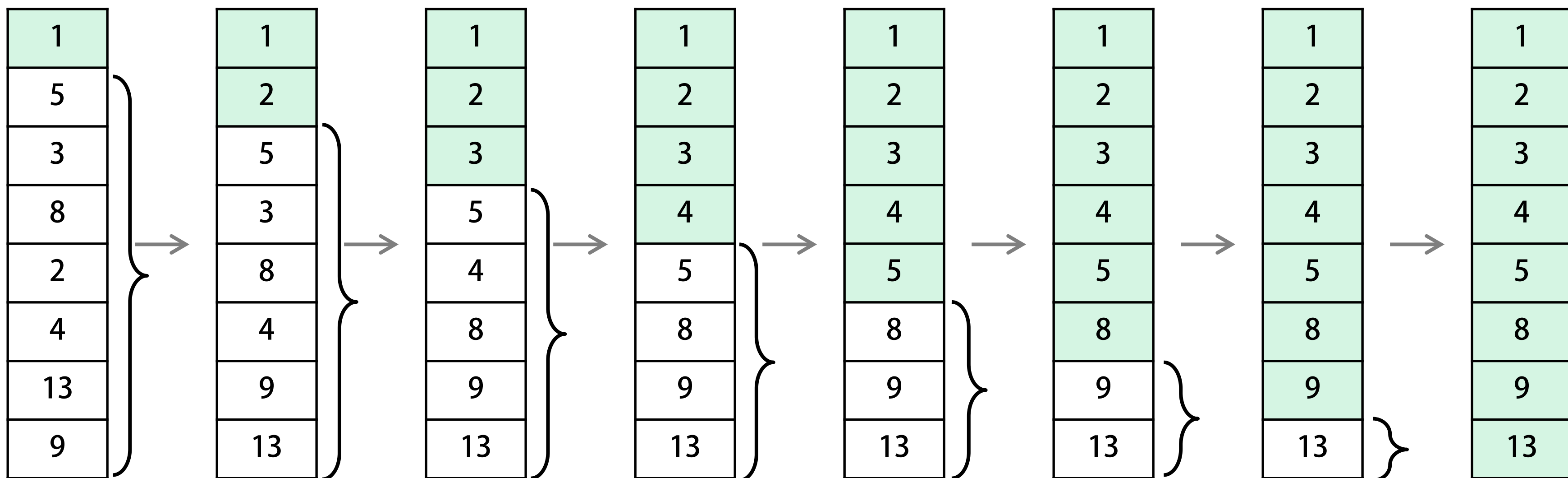
- 全体がソート完了になるまで交換操作を繰り返す
  - 先頭の方から徐々にソート完了領域が拡大していく

	→		→		→		→		→		→		→	
1		1		1		1		1		1		1		1
5		2		2		2		2		2		2		2
3		5		3		3		3		3		3		3
8		3		5		4		4		4		4		4
2		8		4		5		5		5		5		5
4		4		8		8		8		8		8		8
13		9		9		9		9		9		9		9
9		13		13		13		13		13		13		13

バブルソートの計算量

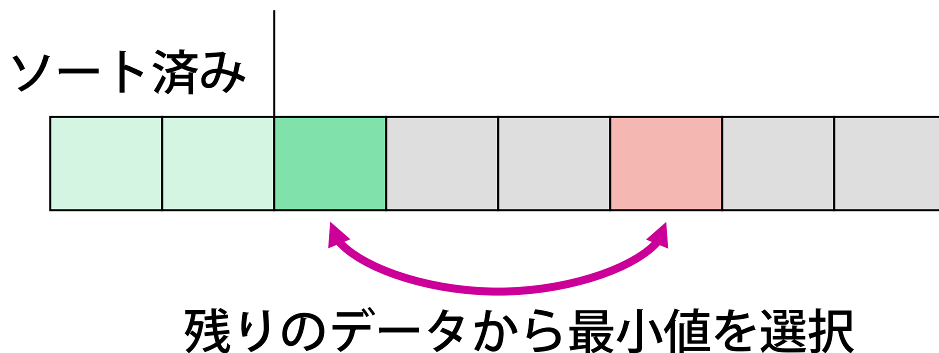
- 時間計算量$O(n^2)$：実装は簡単だが遅くて非実用的
  - 1回目のループ：$n-1$回の比較と交換
  - 2回目のループ：$n-2$回の比較と交換
  - ・・・
  - $n-1$回目のループ：1回の比較と交換

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$



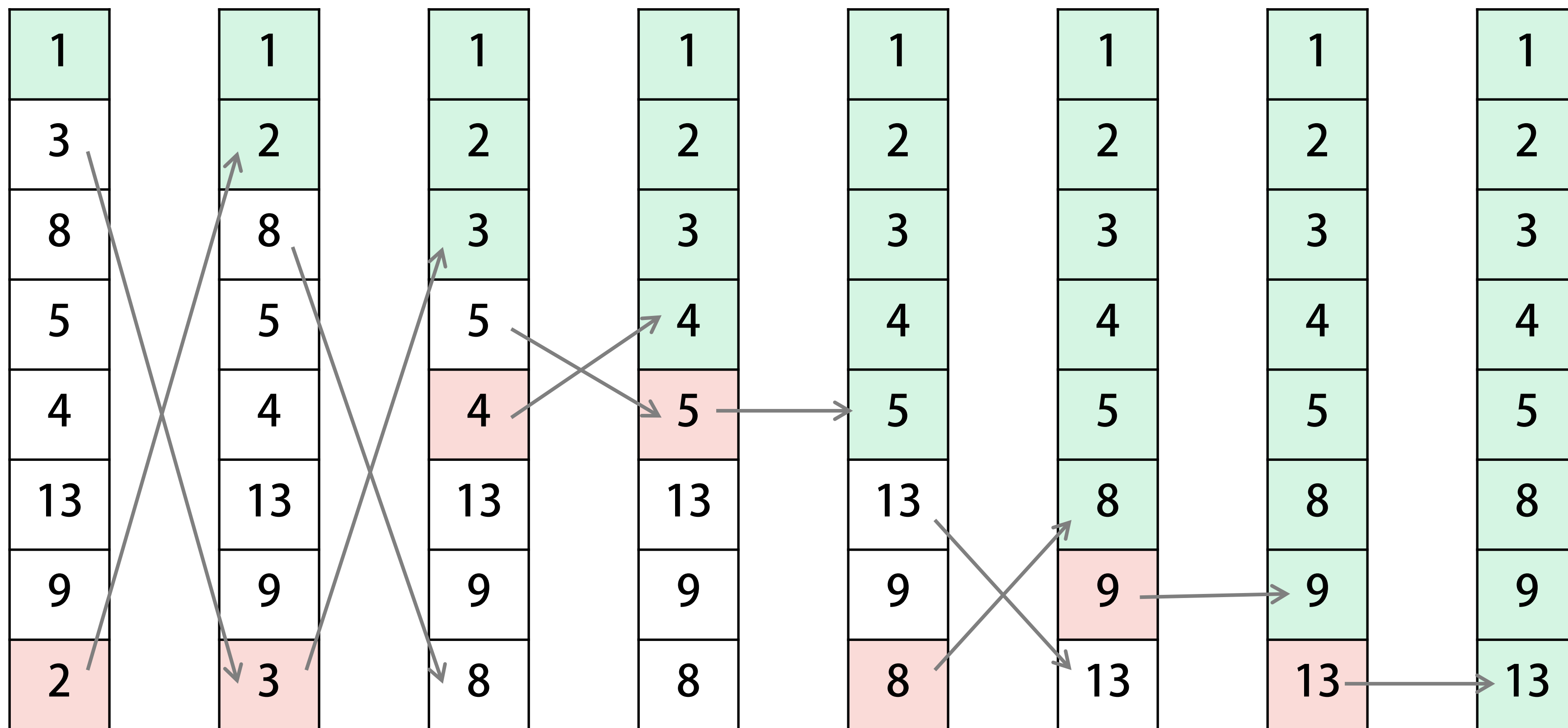
選択ソート

- アルゴリズムの方針
  - 先頭から順にその位置に入るデータを決める
    - 最小値を求める方法で選択する
  - その位置のデータと選択されたデータを交換する
  - これらを繰り返して全体をソートする



選択ソートの動き

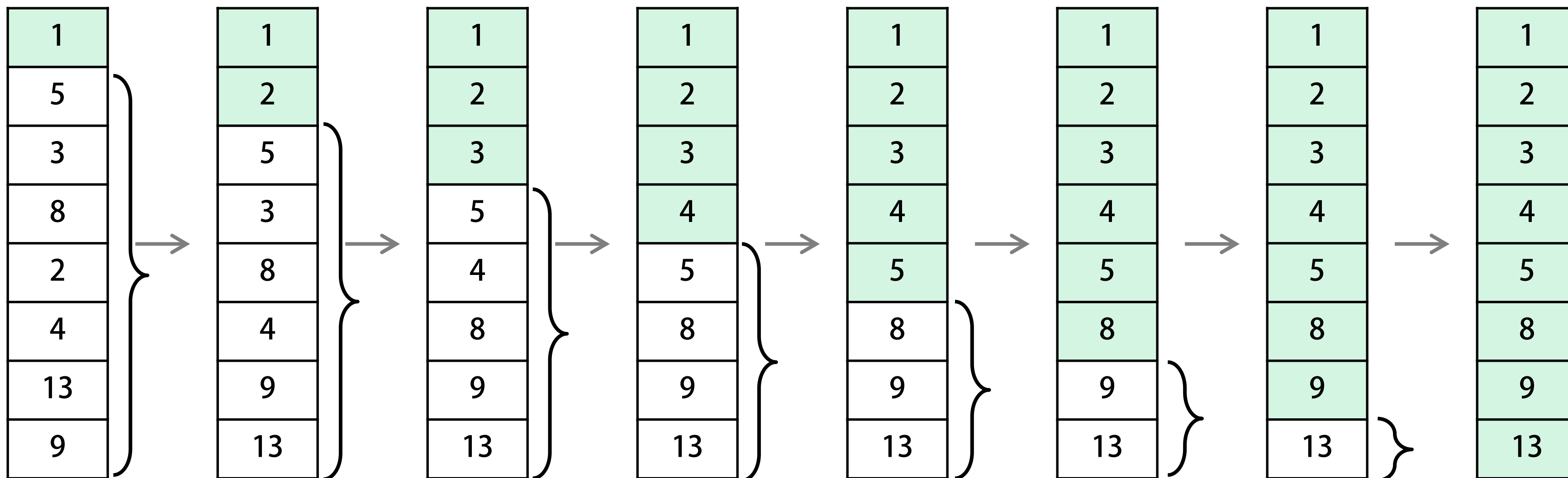
- 全体がソート完了になるまで**最小値探索**を繰り返す
  - 先頭の方から徐々にソート完了領域が拡大していく



選択ソートの計算量

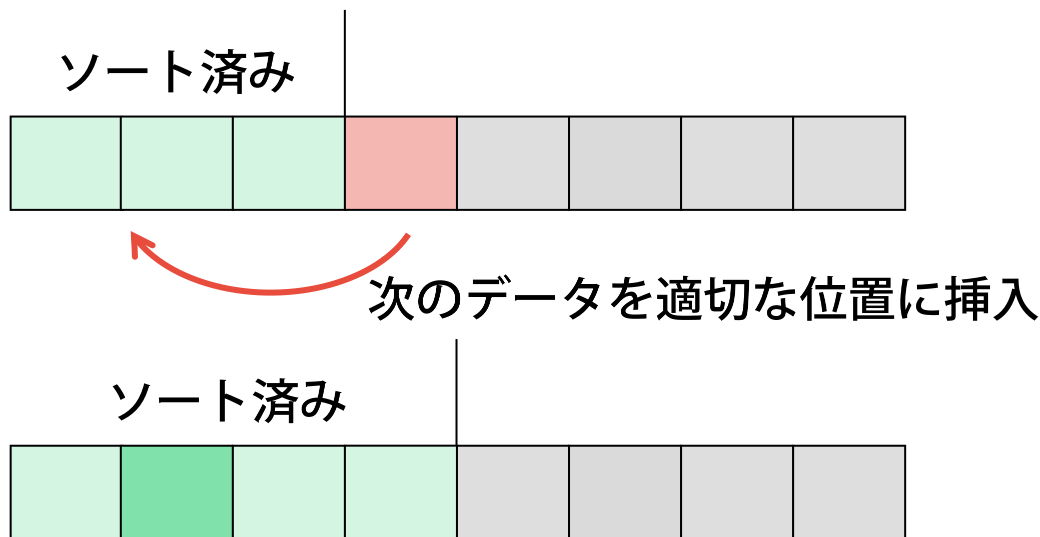
- 時間計算量$O(n^2)$：実装は簡単だが遅くて非実用的
  - 1回目の最小値探索：$n-1$回の比較
  - 2回目の最小値探索：$n-2$回の比較
  - ・・・
  - $n-1$回目の最小値探索：1回の比較と交換

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$



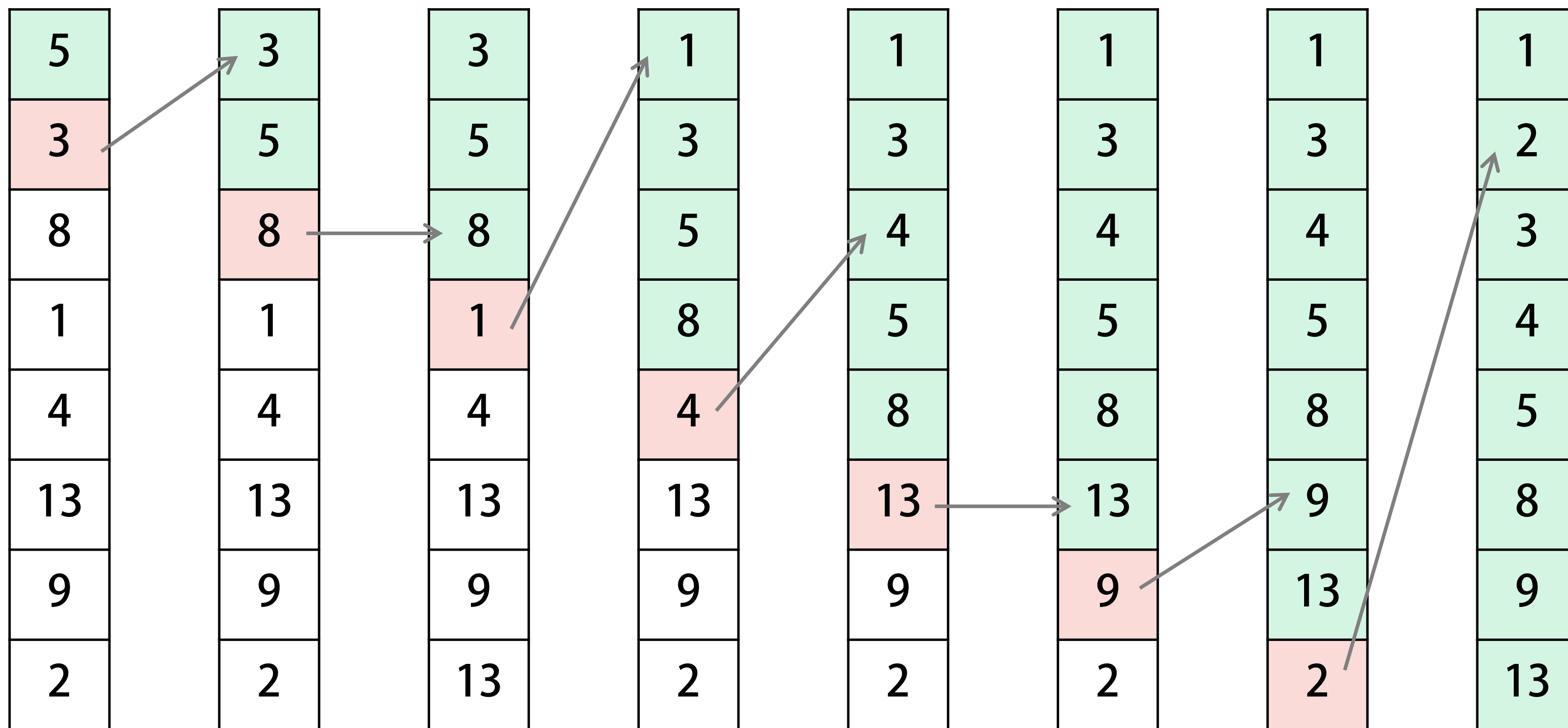
挿入ソート

- アルゴリズムの方針
  - 先頭の方はソート済みの状態にしておく
  - 未ソートのデータをソート済みの列に挿入することでソート済みの列を１つ長くする
  - これらを繰り返して全体をソートする



挿入ソートの動き

- 全体がソート完了になるまで挿入操作を繰り返す
  - 先頭の方から徐々にソートされていく (途中結果≠最終結果に注意！)



挿入ソートの計算量

- 最悪計算量$O(n^2)$
  - 1回目の探索：$n-1$回の比較
  - 2回目の探索：$n-2$回の比較
  - $n-1$回目の探索：1回の比較と交換

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

- 平均計算量$O(n^2)$
  - 1回目の探索：平均して$\frac{n-1}{2}$回の比較
  - 2回目の探索：平均して$\frac{n-2}{2}$回の比較
  - $n-1$回目の探索：平均して$\frac{1}{2}$回の比較と交換

$$\sum_{i=1}^{n-1} \frac{i}{2} = \frac{n(n-1)}{4}$$

挿入ソートは選択ソートに比べて平均的に半分の計算量

データがほぼ正しい順序に並んでいる場合は計算量は$O(n)$に近くなる
→ 他のソートアルゴリズムとの組み合わせが有効

シェルソート

- アルゴリズムの方針
  - 挿入ソートはデータがある程度整列されていれば高速
  - データを分割してあらかじめある程度整列しておきたい
    - 適当な間隔を開けて取り出したデータ列をあらかじめソート
    - それらをまとめてさらに挿入ソート

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

↓ $h=4$グループでそれぞれソート

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

↓ $h=2$グループでそれぞれソート

↓ 全体をソート

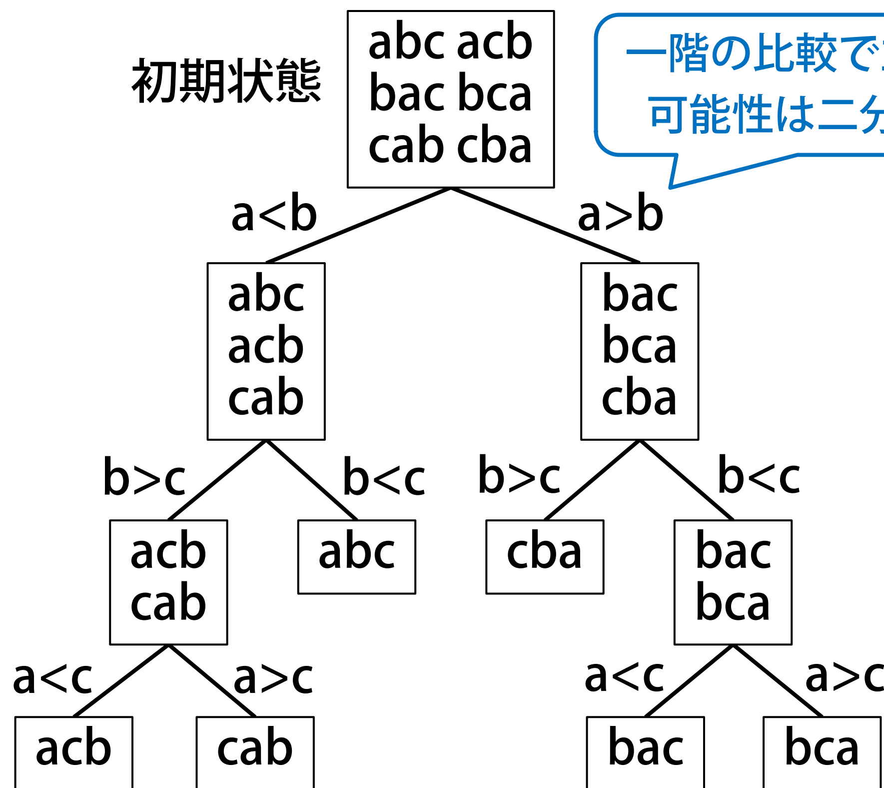
1	2	3	4	5	6	7	8

シェルソートの計算量

- 厳密な計算量の評価は非常に困難
  - グループの分割数 h をどう決めるかが重要
    - $h = 1, 3, 7, 15, \cdots (h_n = 2^n - 1)$を満たす整数列を用いて hを大きい方から適用すると平均時間計算量$O(n^{1.5})$
    - $h = 1, 4, 13, 40, 121, \cdots$ $(h_{n+1} = 3h_n + 1)$を満たす整数列を用いて hを大きい方から適用すると平均時間計算量$O(n^{1.25})$
      - $n = 10000$程度であれば$O(n \log n)$と大差ないので十分に高速
- 前処理なしで挿入ソートを適用するより高速
  - データをh個に分割するなら計算量は$O\left(\frac{n^2}{h^2} \times h\right) = O\left(\frac{n^2}{h}\right)$
  - 例：$h = 5, 2, 1$であれば$\frac{1}{5}\frac{n^2}{2} + \frac{1}{2}\frac{n^2}{2} + 3n = \frac{7}{20}n^2 + 3n < \frac{n^2}{2}$

計算量に関する理論的考察

- ソートの最悪計算量の限界は$O(n \log n)$
  - 決定木を下って並びを確定させるために必要な計算量



最悪の場合を速くするには
可能性が半分になるのが最善

k回繰り返すと可能性は $\frac{n!}{2^k}$ 通りに絞られる

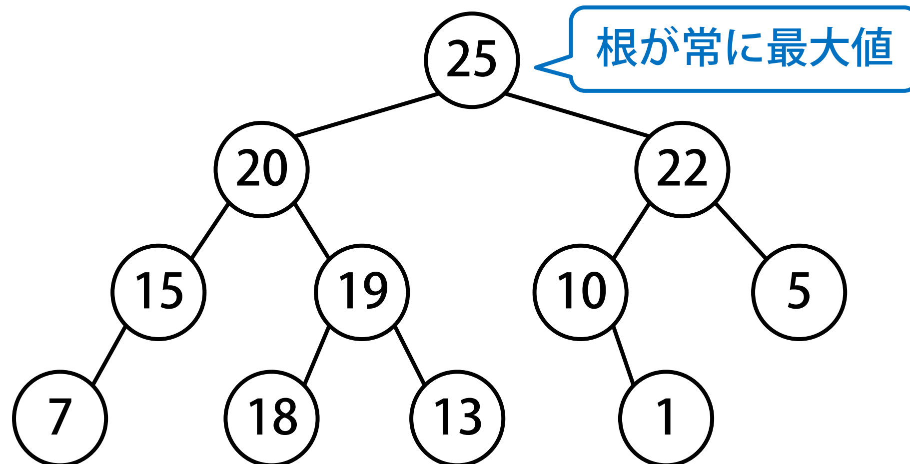
これが1以下になるには $2^k \geq n!$

$$k \geq \log_2 n! \approx \log_2(\sqrt{2\pi n} n^n e^{-n})$$

n個のデータの整列には
およそ$n \log_2 n$回の比較が必要

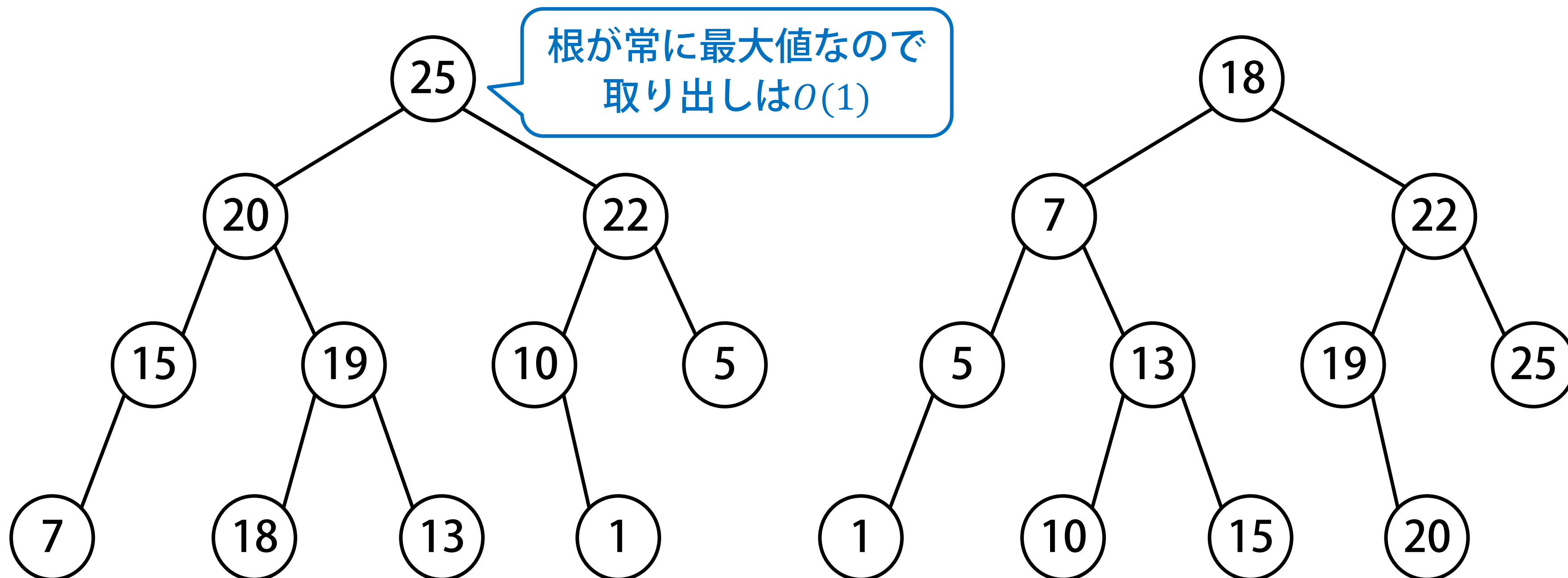
ヒープソート

- アルゴリズムの方針
  - 選択ソートの最大値探索部分を改良
    - 最大値探索に$O(n)$必要 → 全体で$O(n^2)$
    - 最大値探索を$O(\log n)$に改良できれば・・・全体で$O(n \log n)$
  - 部分順序付き木を利用
    - 平衡二分探索木は最大値探索のためにはオーバースペック



ヒープ

- いくつかの制約を持つ部分順序付き木
  - 根に最大値が配置
  - 各頂点の値がその左右いずれの子よりも大きい

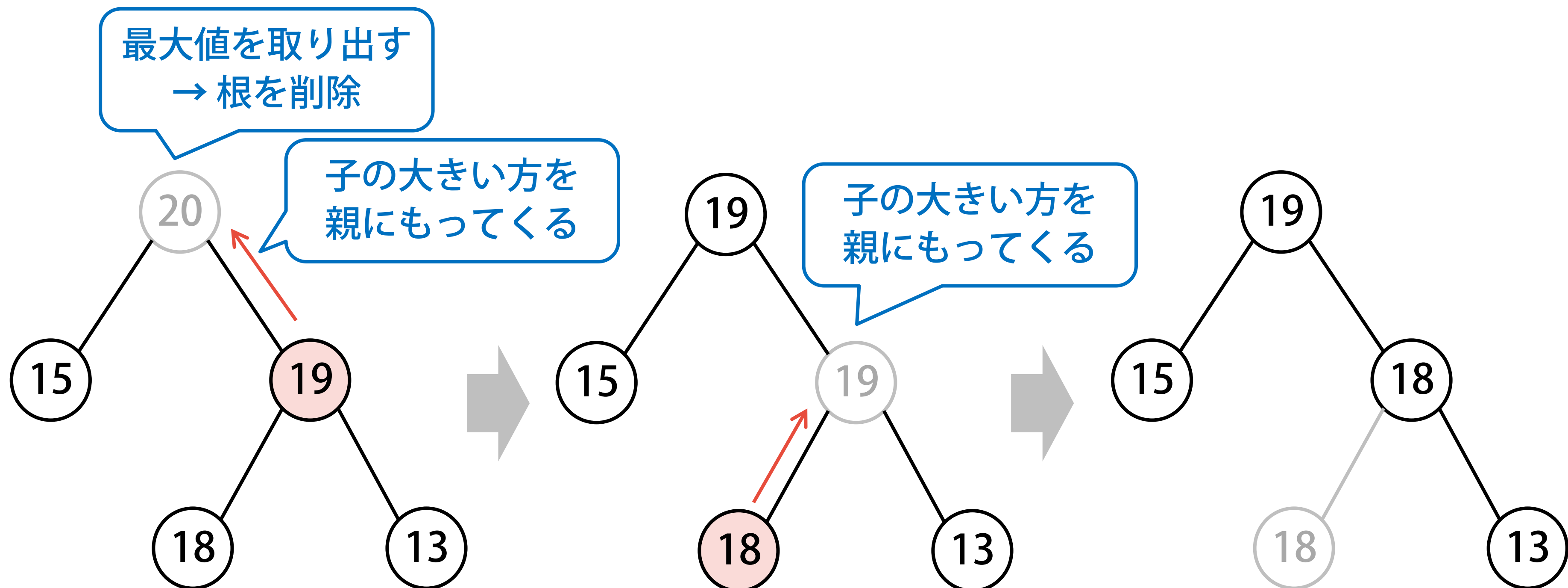


部分順序付き木

二分探索木

最大値取得とヒープの再構成

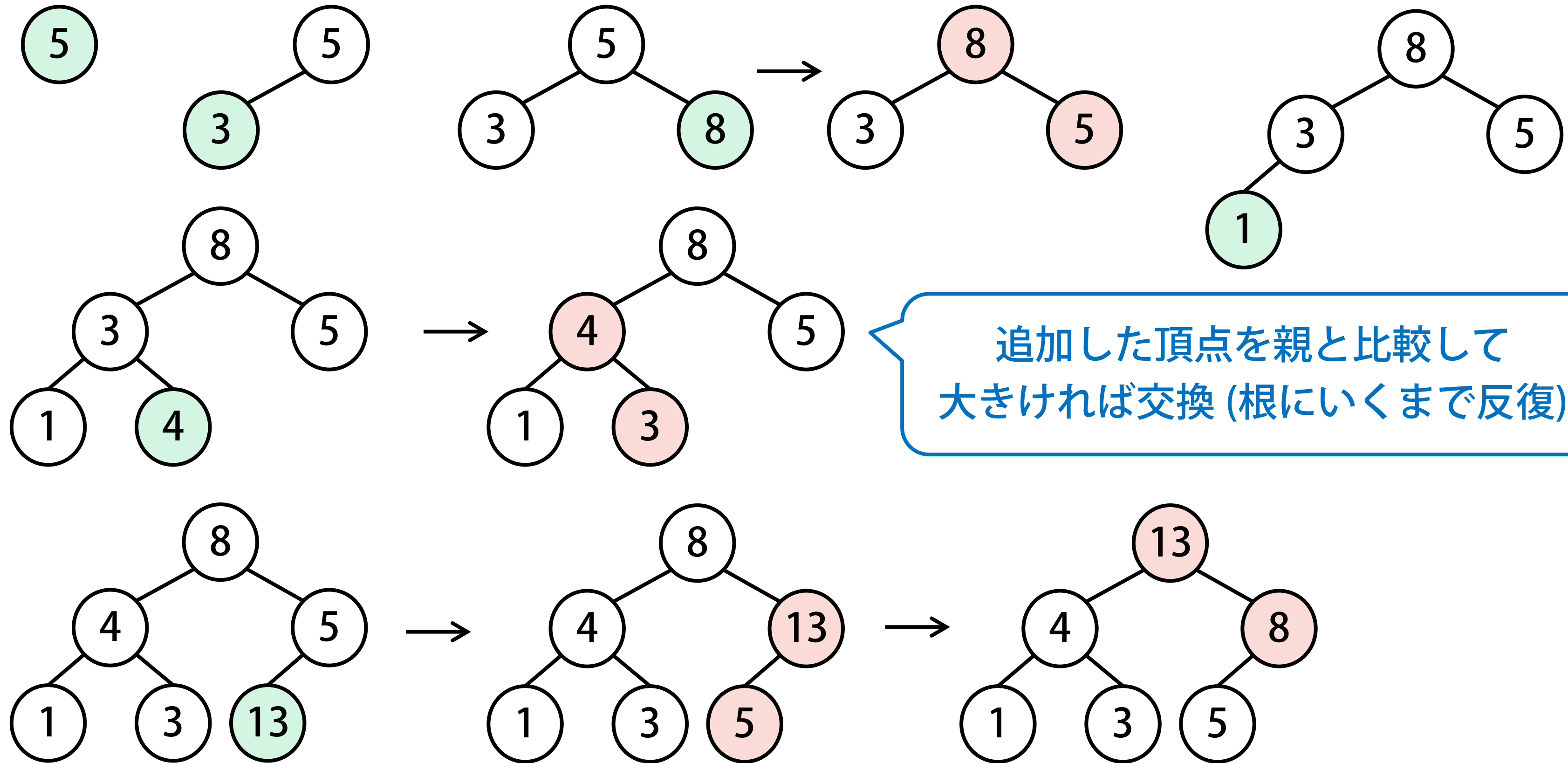
- 再帰的に左右の子の大きい方を親に移す
  - 最大値を取り出すたびに行う → 全データを取り出すまで行う
  - 木の高さが$O(\log n)$ → 再構成の計算量は$O(\log n)$



ヒープの構築法1

- データを順番にヒープに挿入

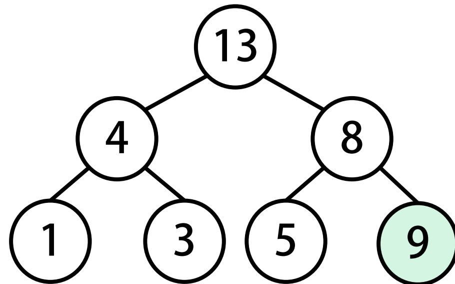
5	3	8	1	4	13	9	2

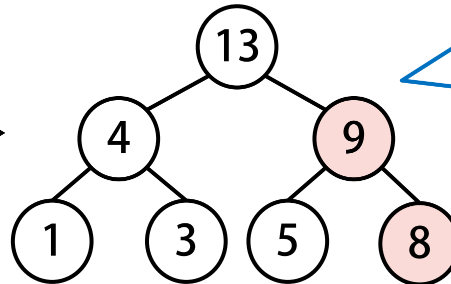


ヒープの構築法1

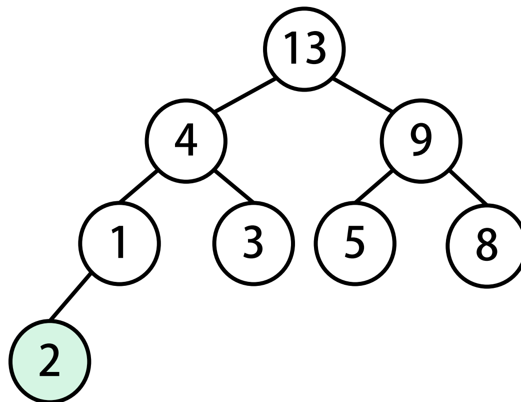
- データを順番にヒープに挿入

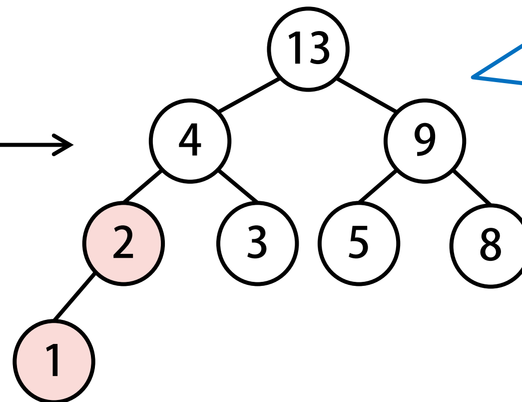
5	3	8	1	4	13	9	2





追加した頂点を親と比較して大きければ交換 (根まで反復)



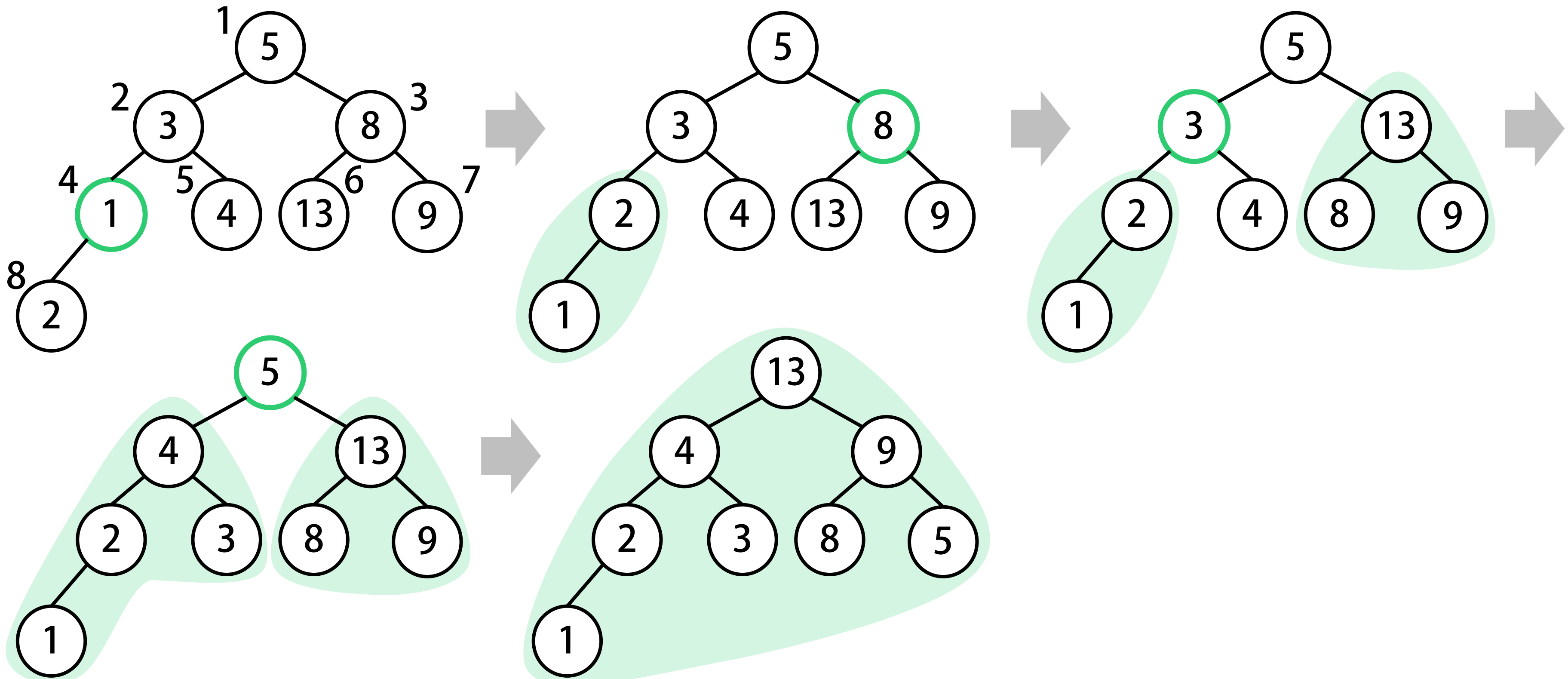


最終的に全データを保持するヒープが得られる

ヒープの構築法2

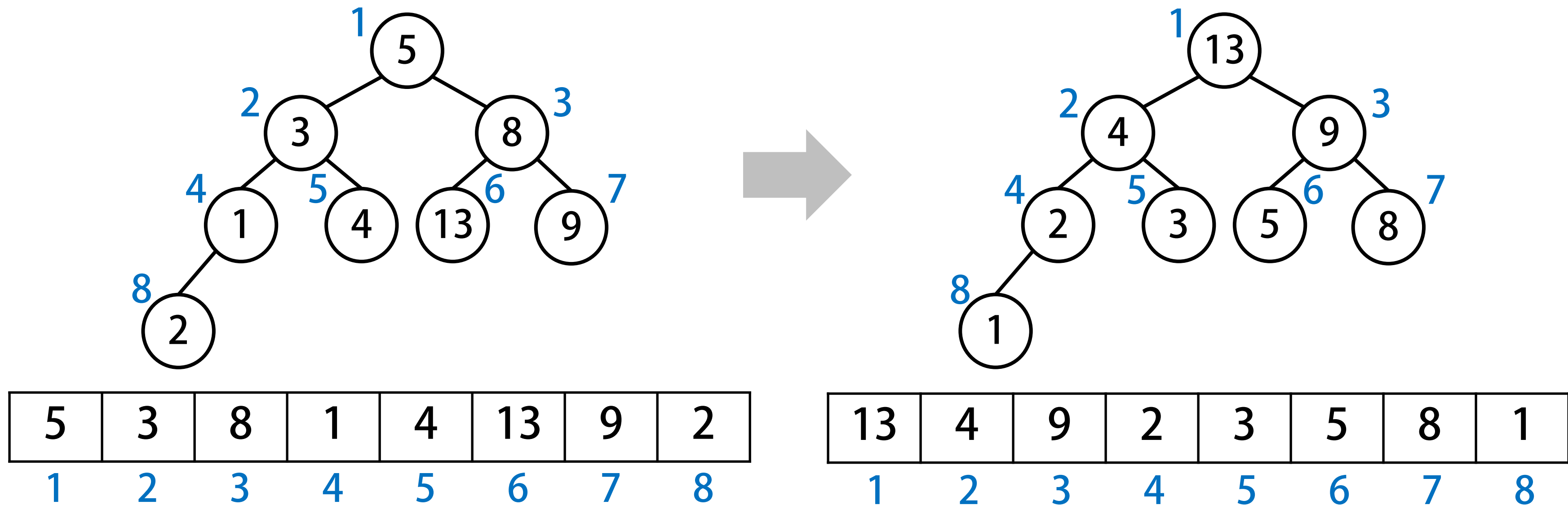
- 配列データを配置 → 下から部分木をヒープに再構成

5	3	8	1	4	13	9	2



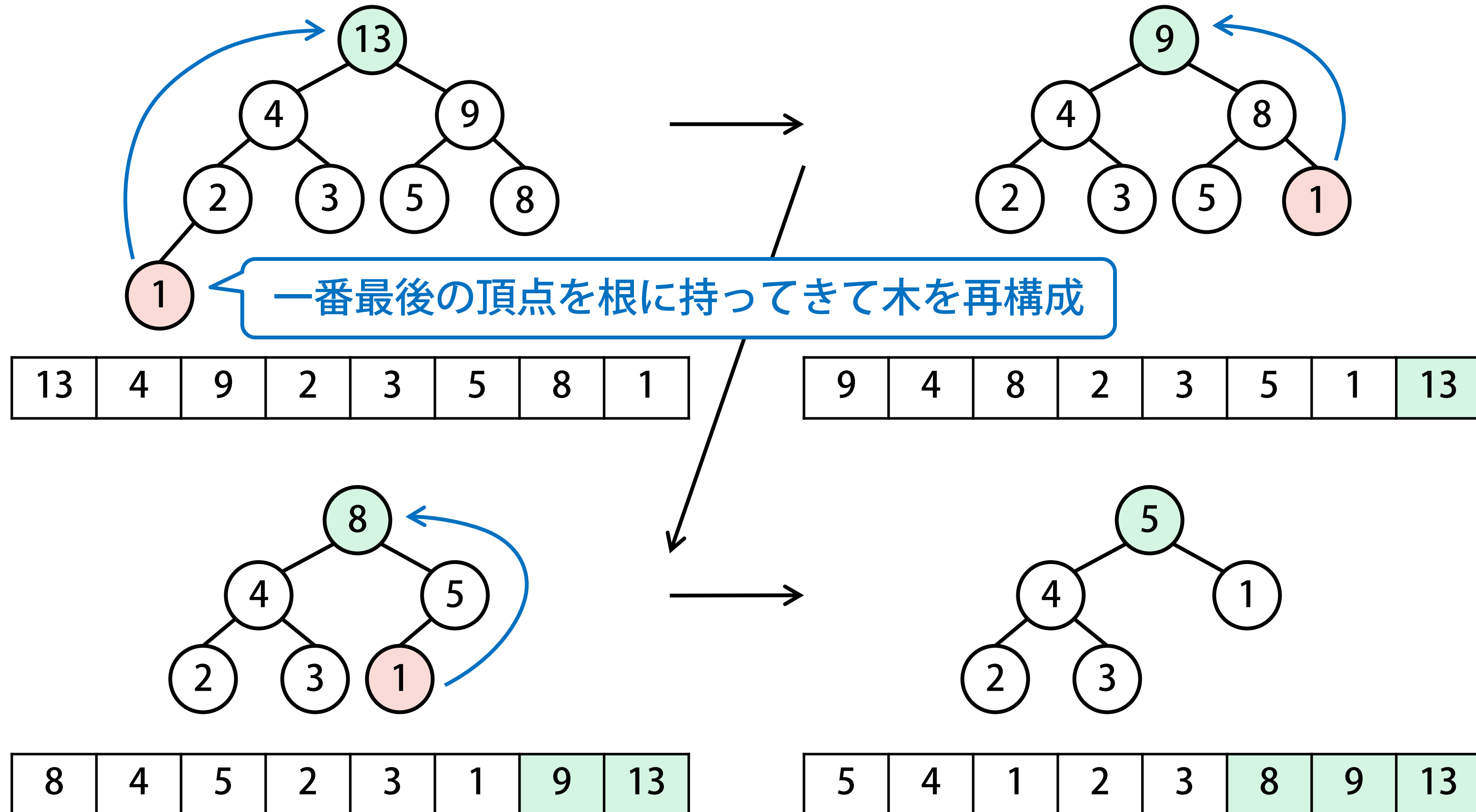
配列としての実装

- 順序付き木の配列上の表現
  - リストやポインタを用いなくても配列を使って木を表現可能
  - 木の操作 = 配列内の要素の入れ替え
    - ヒープからデータを取り出すたびに一番最後に詰める



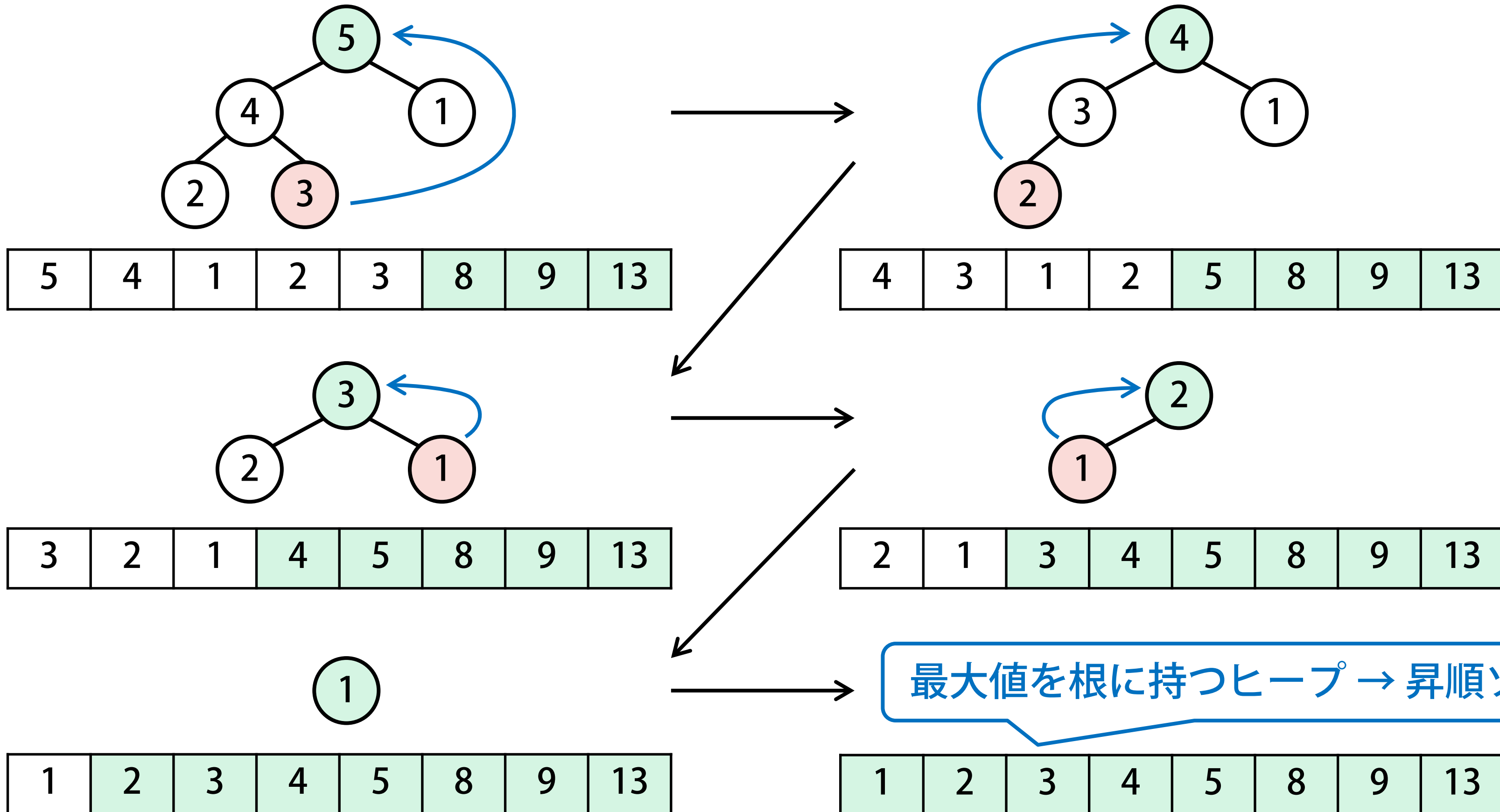
ヒープソートの動き

- 根を取り出し → 最後の要素を根にもって木を再構成



ヒープソートの動き

- 根を取り出し → 最後の要素を根にもって木を再構成



ヒープソートの計算量

- ヒープ構築ステップ
  - 構築法1：$O(n \log n)$
    - 1つのデータを挿入するのに$O(\log n)$
    - n個のデータを挿入するのに$O(n \log n)$
  - 構築法2：$O(n)$
    - 調べる頂点数：$0 \times \frac{n}{2} + 1 \times \frac{n}{4} + 2 \times \frac{n}{8} + \cdots + (\log n - 1) \times 1$
    - $1 \times 2^{k-2} + 2 \times 2^{k-3} + \cdots + (k-2) \times 2 + (k-1) \times 1 = 2^k - (k+1)$

$k = \lceil \log n \rceil$

$k = O(\log n)$なので$O(n)$

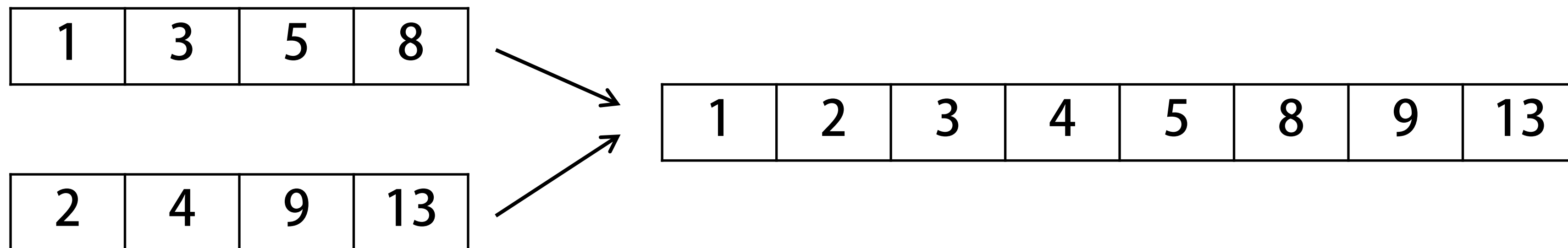
- 最大値取得ステップ
  - 1つのデータを削除するのに$O(\log n)$
  - n個のデータを削除するのに$O(n \log n)$

ヒープソート全体で
平均計算量
最悪計算量ともに
$O(n \log n)$

補助の記憶場所は$O(1)$ だけでOK

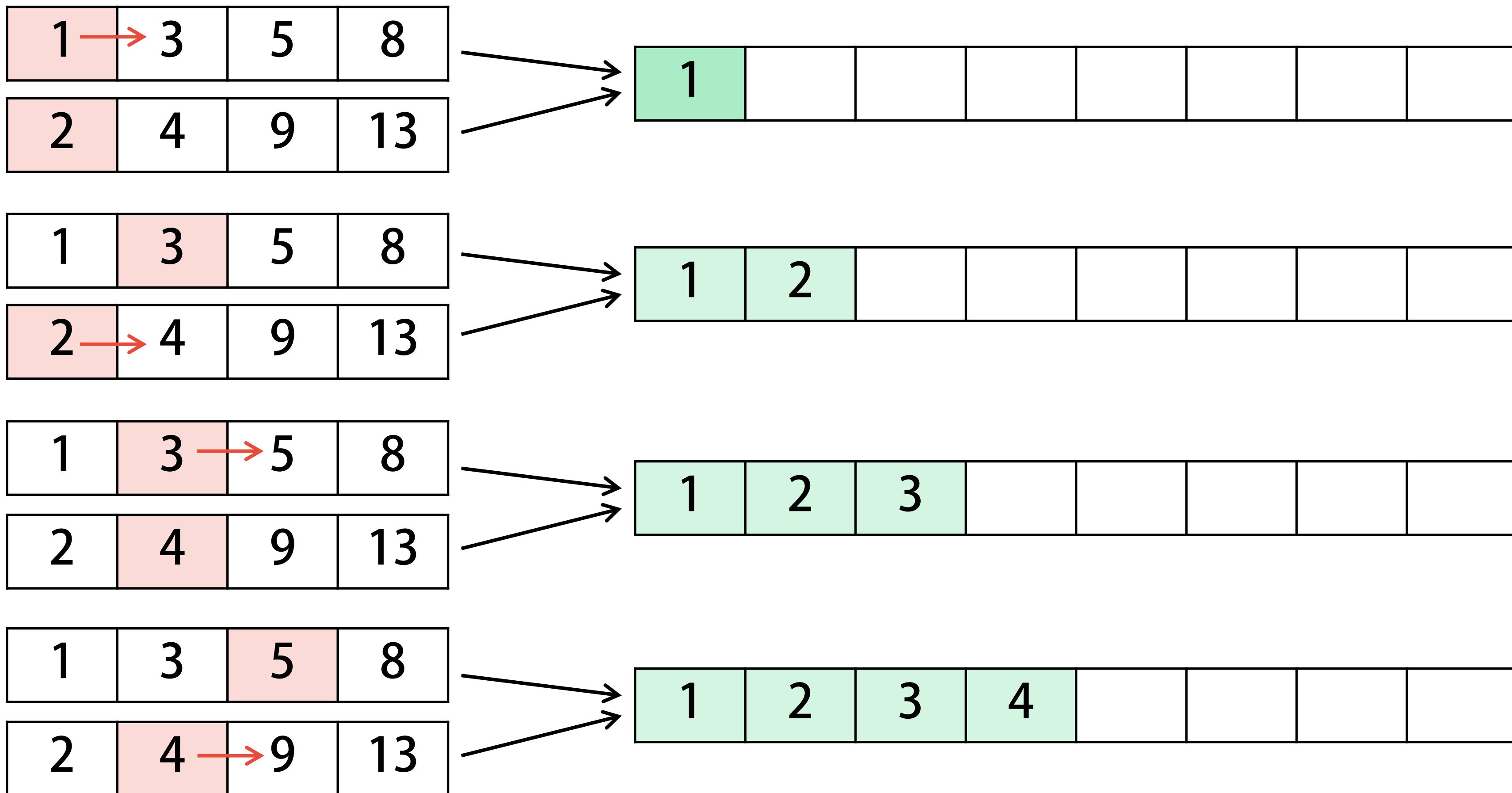
マージソート

- アルゴリズムの方針
  - 分割統治法：問題を分割して結果をあとで組み合わせる
    - ソート済みの配列が2つあればそれらを組み合わせて大きいソートの列をつくるのは高速 (マージ操作)
    - 再帰的に分割しながらソート



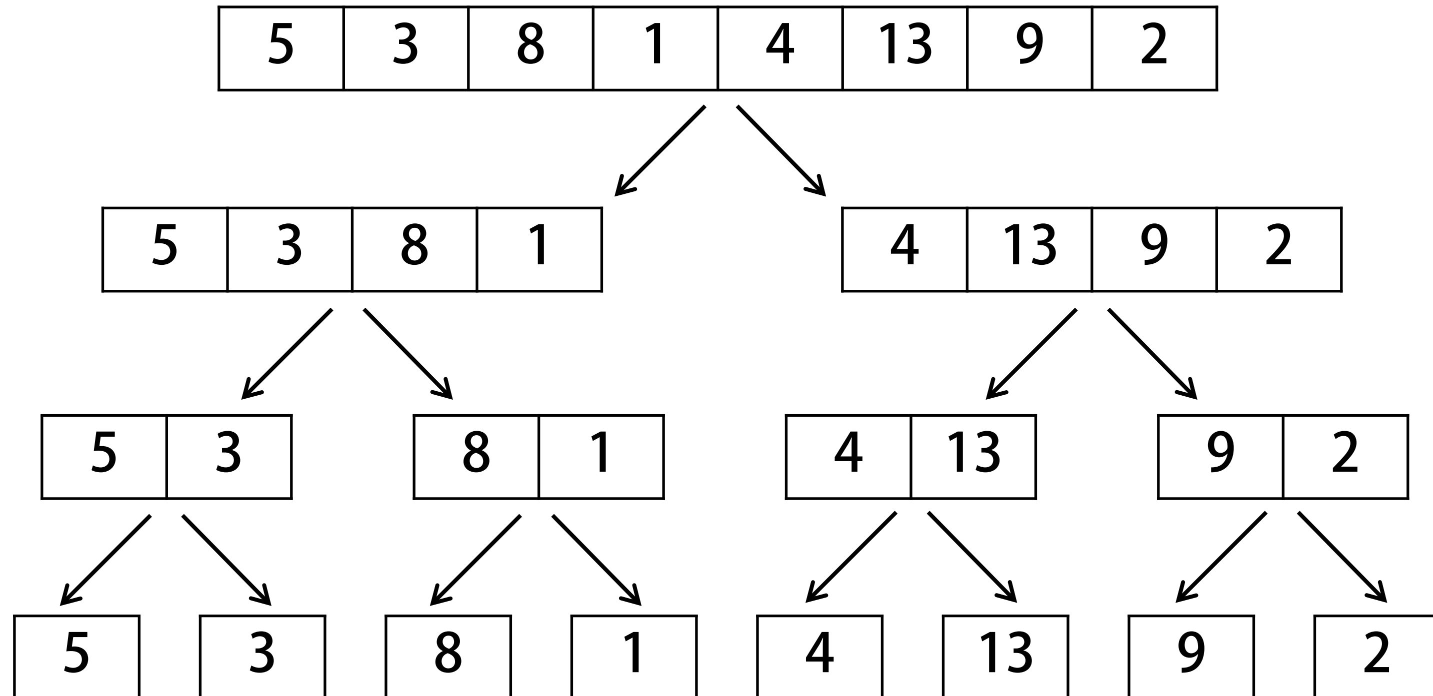
マージの動き

- 各配列のインデクスを1つずつ進めながらマージ



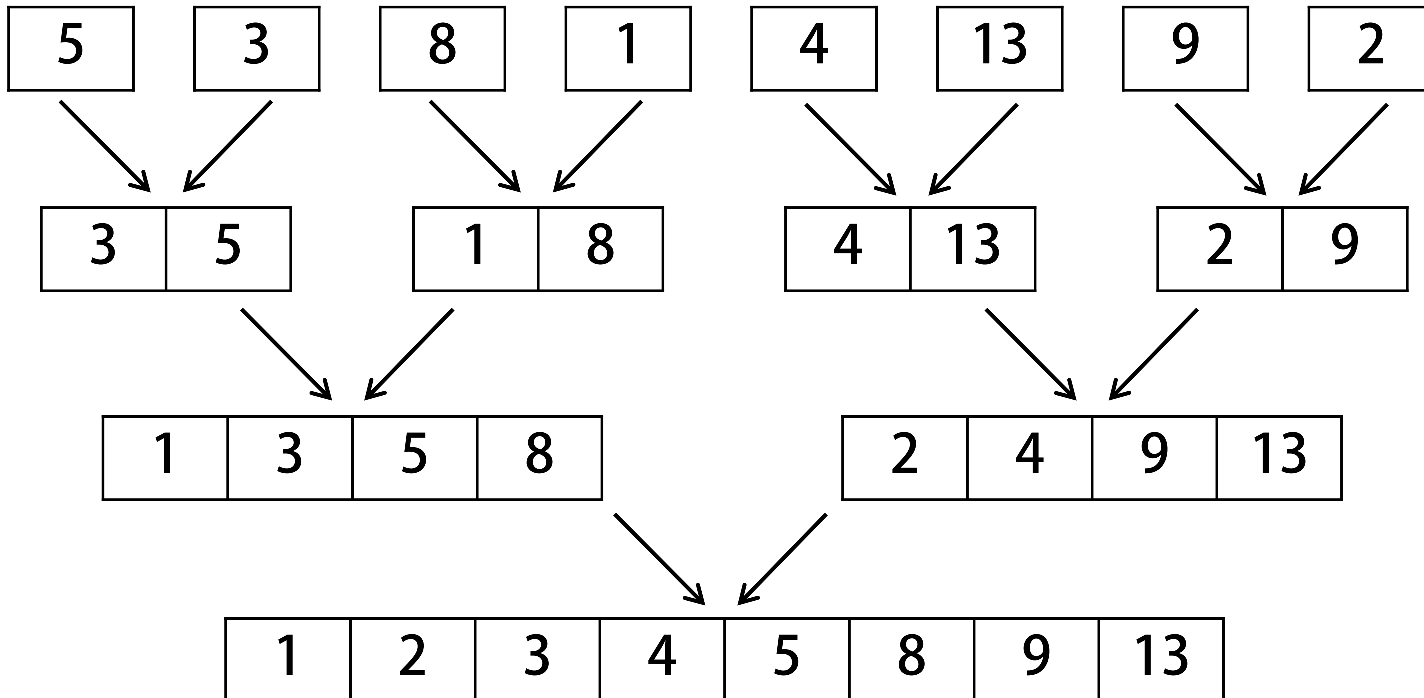
マージソート動き：分割

- 再帰的にデータを半分ずつに分割していく
  - 単純マージソート：データが1つになるまで分割
  - 自然マージソート：データが上昇列になった時点で分割を停止



マージソート動き：統治

- 再帰的にデータをマージしていく
  - 各階層でn個のデータを比較するので計算量は$O(n)$
  - 階層の深さは$O(\log n)$



マージソートの計算量

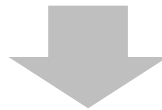
- 平均計算量・最悪計算量ともに$O(n \log n)$

データ数が $n = 2^k$ であるときを仮定

データ数 n に対するマージソートの計算量：$T(n)$

$$T(1) = c_1 \qquad T(n) \le T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + O(n) = 2T\left(\frac{n}{2}\right) + c_2 n$$

マージの計算量

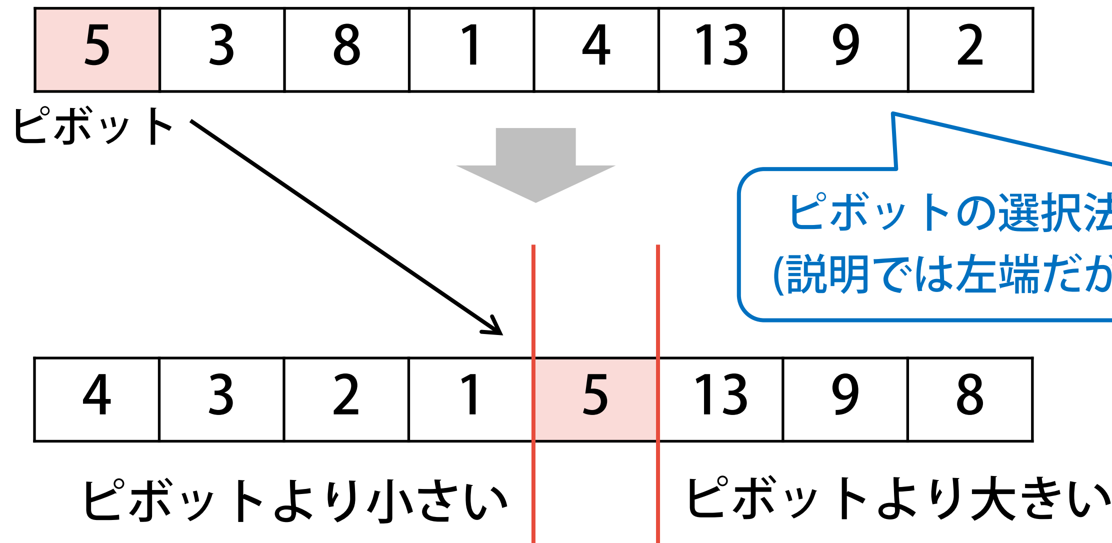


$$T(n) = O(n \log n)$$

同様に平均計算量・最悪計算量$O(n \log n)$をもつヒープソートより少し高速
ただし、元のデータ領域と同じ大きさの補助領域が必要
実際のソートに利用されることは少ない

クイックソート

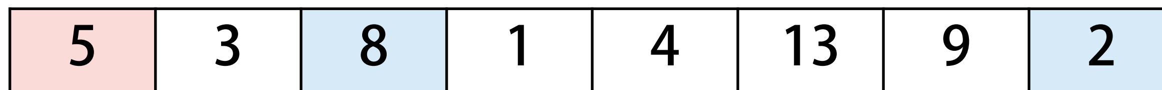
- アルゴリズムの方針
  - 分割統治法：問題を分割して結果をあとで組み合わせる
    - 前半は特定要素 (ピボット) より小さく、後半は大きくする
    - ピボットで分割したデータは再帰的にソート



ピボットの選択法にはバリエーションが存在
(説明では左端だが実際は中央を選ぶ方がよい)

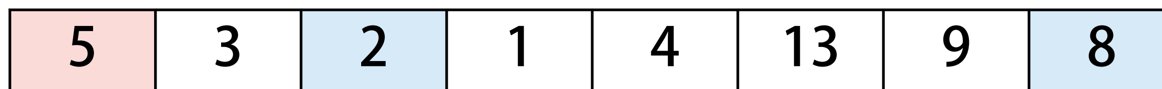
クイックソートの動き：分割

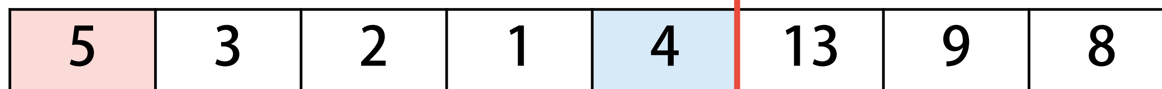
- 両端から順に比較して必要ならば場所を交換
  - 比較位置が交差すればピボットと交換して終了



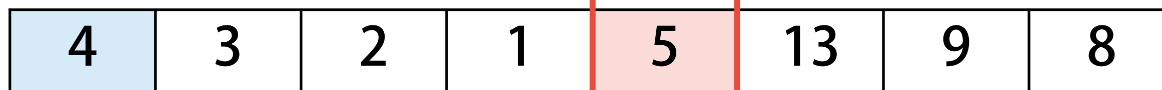
左端からピボットより大きい値を探す　右端からピボットより小さい値を探す

両者を交換



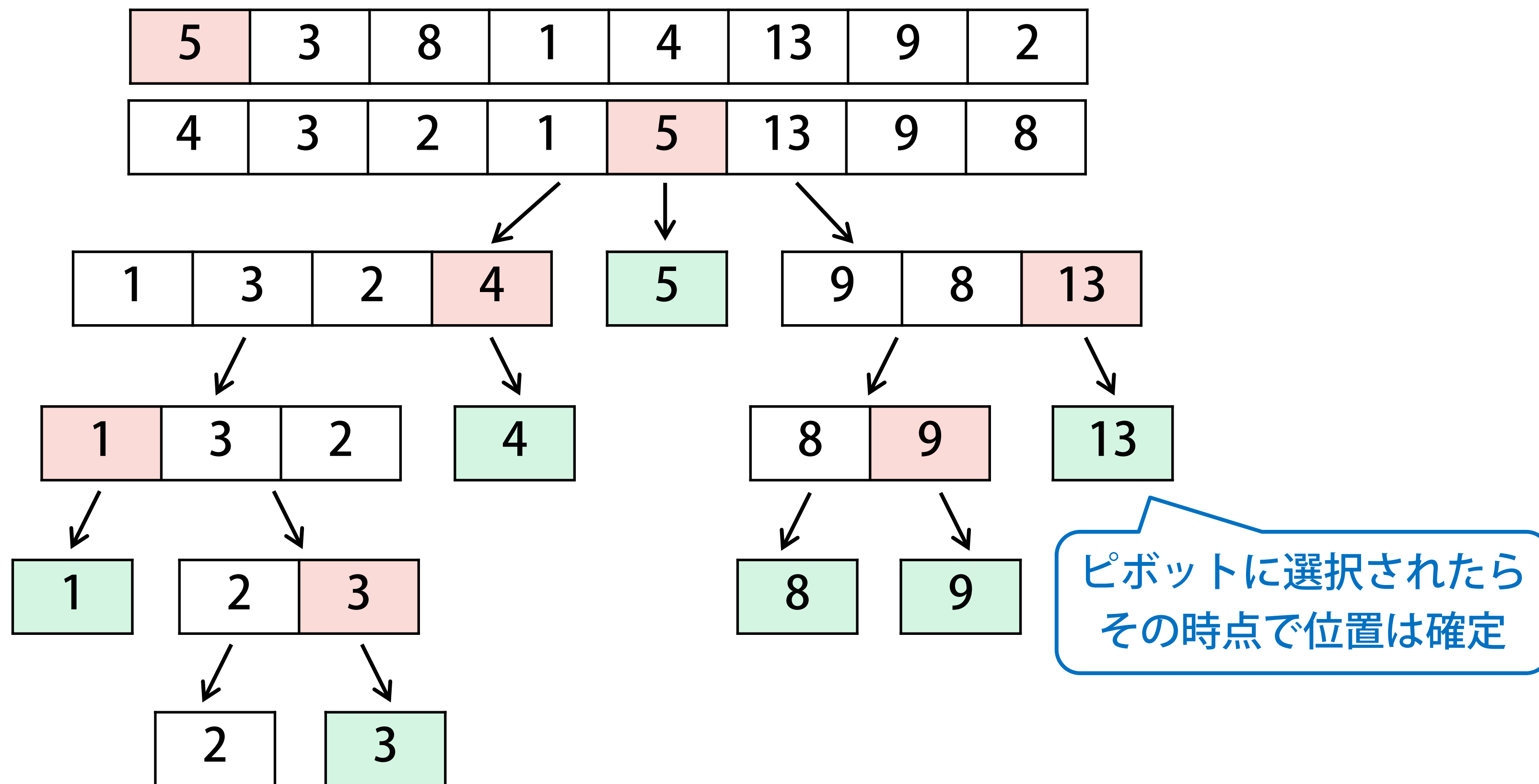


ピボットと境界の左要素を交換



クイックソートの動き：分割

- 再帰的にデータを分割していく
  - 最善ケース：半分ずつに分割　最悪ケース：1個とその他全部に分割



クイックソートの計算量

- 最悪計算量：$O(n^2)$
  - 比較回数$Q_n = (n-1) + (n-2) + \cdots + 1 = \frac{1}{2}n(n-1)$
- 平均計算量：$O(n \log n)$

経験的には最速のソートアルゴリズム

  - 比較回数$Q_n = n - 1 + Q_a + Q_b$　ただし　$a + b = n - 1$
  - 分割$(a, b) = (0, n-1), (1, n-2), \cdots, (n-1, 0)$が等確率で生起と仮定
  - $Q_n = n - 1 + \frac{1}{n}\left((Q_0 + Q_{n-1}) + (Q_1 + Q_{n-2}) + \cdots + (Q_{n-1} + Q_0)\right)$

$$= n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} Q_k$$

ただし $Q_1 = 0$ かつ $Q_0 = 0$

$$Q_n = 2(n+1)(H_{n+1} - 2) + 2$$
$$\approx 2(n+1)(\log_e(n+1) - 2) + 2$$
$$\approx 2n \log_e n \approx 1.39 n \log_2 n$$

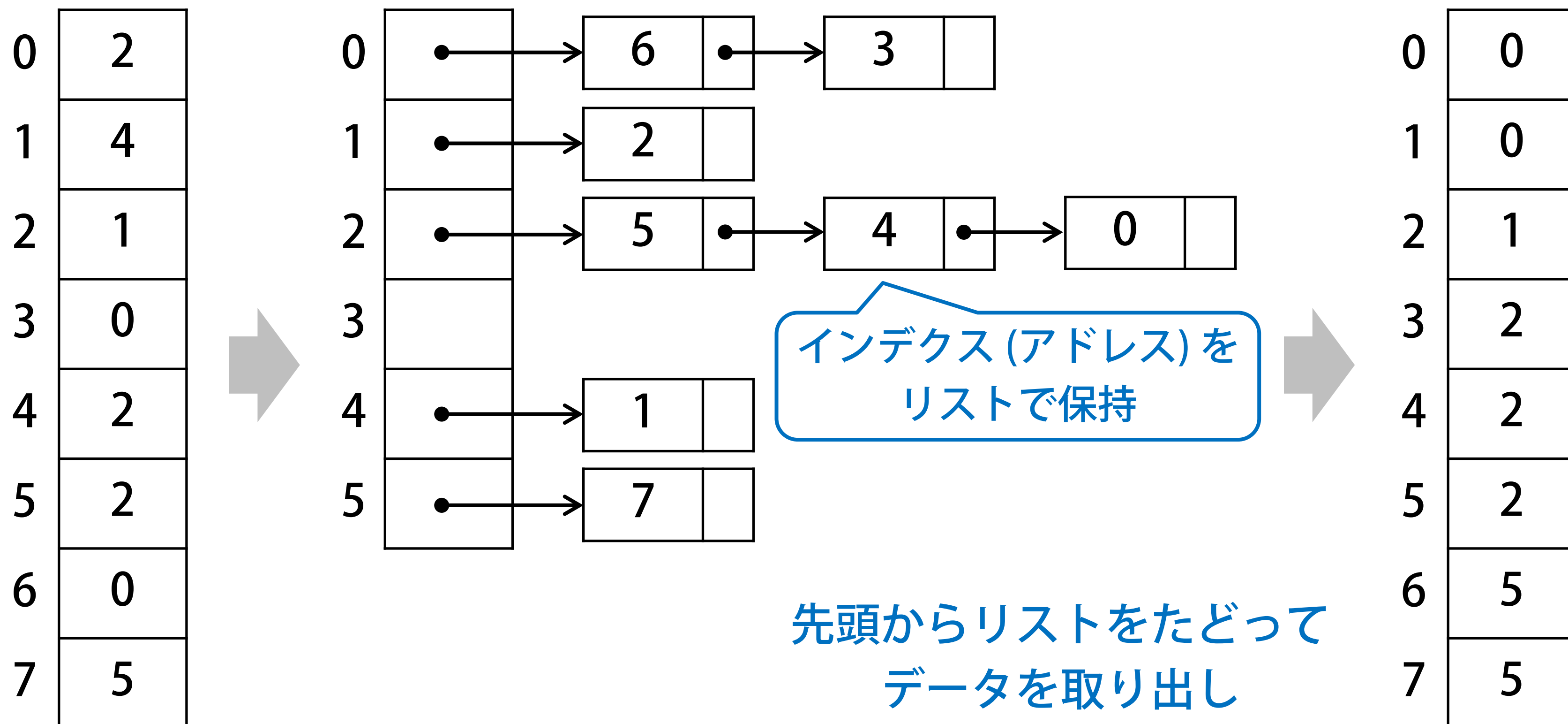
安全クイックソート：
最悪でも$O(n \log n)$に抑えるために
再帰が一定段数を越えたら
ヒープソートに切り替え

比較によらないソート

- バケットソート (ビンソート)
  - データに上限・下限が存在する場合に適用可能
    - 例：ある範囲内の整数
  - データの種類が定数種類しかない場合にも適用可能
    - ハッシュ関数で整数に変えてから用いてもOK
- 基数ソート
  - 大きい桁の数に対して桁毎にバケットソート
  - 下位の桁から順番

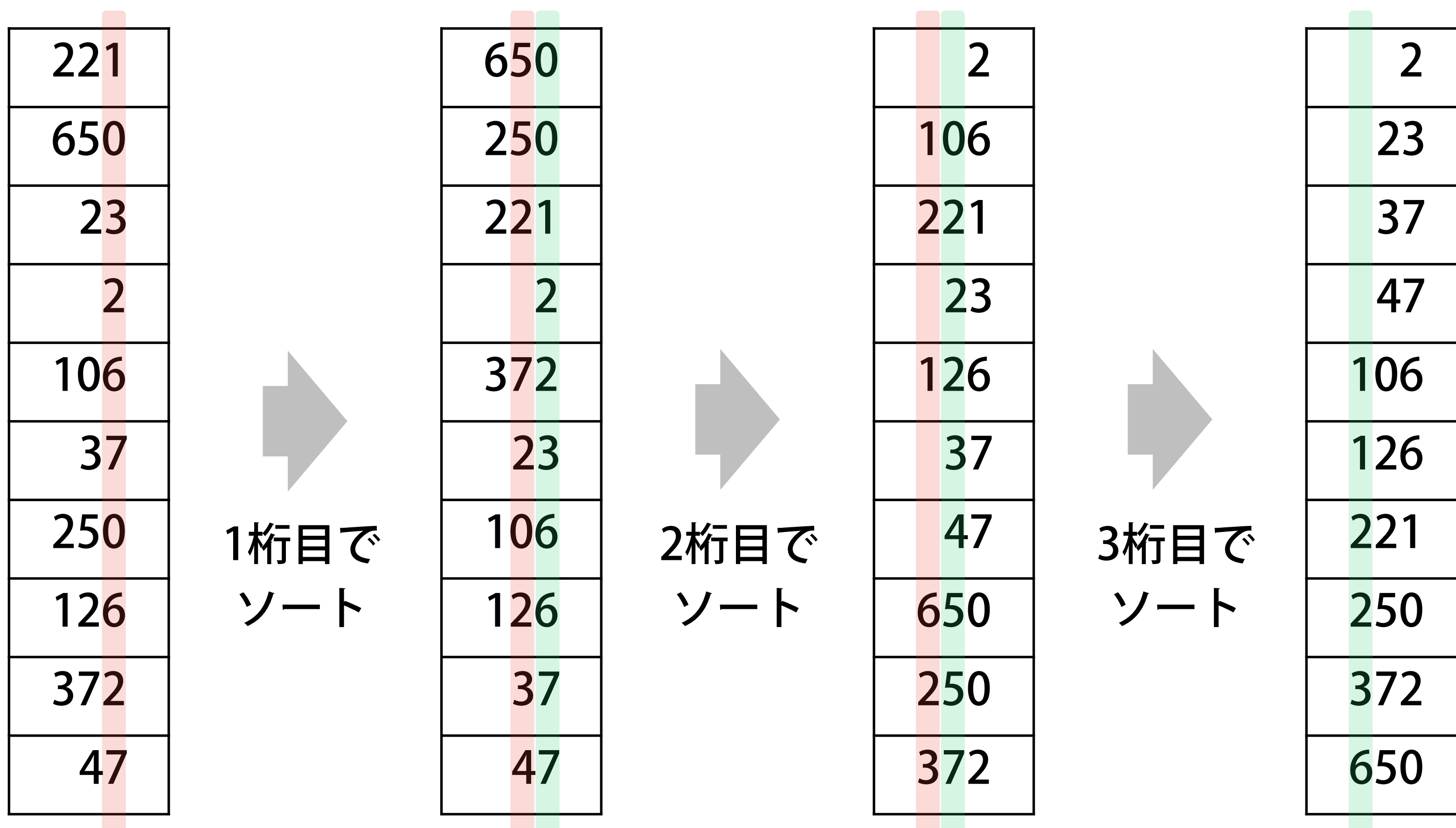
バケットソートの動き

- データの種類数分の"バケツ"を準備してデータを格納
  - n個のデータをm個のバケツを使ったソートの計算量：$O(m+n)$



基数ソートの動き

- 下位の桁から順番にバケットソート



基数ソートの計算量

- 最悪計算量：$O\left(k(m+n)\right)$
  - m種類のn個のデータをバケットソートの計算量：$O(m+n)$
  - k桁をバケットソートするための計算量：$O(k(m+n))$
  - N個のデータを区別するには $k=\log_m N$ 桁必要
  - $n=N$のときの時間計算量：$O(m\log N+N\log N)$

データ範囲 m や桁数 k に注意が必要