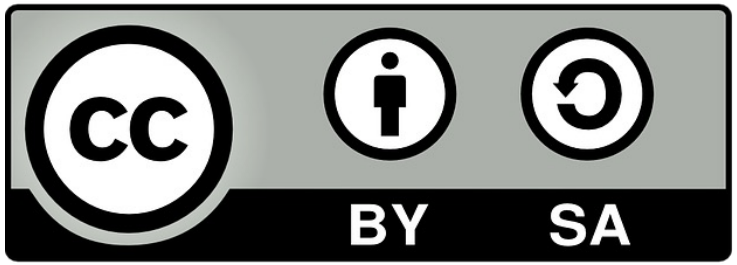
# Algorithms (2024 Summer)
# #5：整列（ソート）

矢谷 浩司
東京大学工学部電子情報工学科



# 授業アンケート1回目実施中！

ぜひ皆さんの声をお聞かせください．slackにてURLを流しております．無記名のアンケートです．

始まって1ヶ月くらいたった所ですが，みなさんの感じるところをお教えください．聴講のみの方もぜひよろしくお願いいたします．

次回は6月中旬くらいを予定しております．

# ソートの典型問題

ランダムな整数が格納されている長さNの配列を昇順・降順に並べ替える.

[3, 5, 2, 1, 6, 4] -> [1, 2, 3, 4, 5, 6]

（今日の説明では上のように昇順にソートすることを前提としますが，降順でも考え方は同じです．）

# なぜソート？

人間にとってわかりやすい順序（スプレッドシートの並べ順など）．

探索の時にソートされていることが有利に働く．

# 実際問題は．．．

ソートに関するライブラリ，関数はどの言語でも大概充実しているので，それを使うのがよい．

非常に効率的に動くように実装されているので，自前の関数でやるメリットはほぼない．．．😅

使えるメモリ量も十分なことが多いので，ソートのやり方による領域計算量の影響が小さくなった．

とはいえ，その中身を理解することはとても重要．

# 内部 vs. 外部 != 原地排序（只占用常数空间, 无需额外数据结构）、非原地排序

ソート実行時に一時的に必要な記憶領域が元々のデータ量かそれ以上，つまり$O(n)$以上の場合，外部ソートと呼ぶ．
（外部の記憶領域が必要になるようなイメージ）

内存存不下

一方，$O(1)$や$O(\log n)$程度の一時的な記憶領域でよいものは内部ソートと呼ぶ．（メモリ上で対応できる，というイメージ）

ただし今は記憶容量が十分にあると仮定できることが多く，あまり重要視されない（と思う）．

# 安定 vs. 安定でない

**稳定，不稳定排序**

同一の要素が複数ある場合，最初の並び順がソート完了後も保持されている場合，「安定」という．

安定でないソートアルゴリズムでもソートはちゃんとされるが，同一要素間での並びは変わってしまう場合あり．

要素を飛び越えて入れ替えるようなアルゴリズムの場合，安定でなくなる．

# 安定 vs. 安定でない

例）[3, 1, 4, 2, 5, 3]

安定なソート：必ず[1, 2, 3, 3, 4, 5]
安定でないソート：[1, 2, 3, 3, 4, 5]になることがある

要素の出てくる順番が重要になるかどうか.

# 安定 vs. 安定でない

複数の値でのソート（例えば，IDでソートしてさらに成績順にソート）では，2つ目で使うソートが安定でないとうまく行かない．

安定でないソートでも，与えられた配列$[a_0, a_1, \ldots, a_{n-1}]$に対して，$[(a_0, 0), (a_1, 1), \ldots, (a_{n-1}, n-1)]$というインデックスをペアにしたデータを考えて，ソートをする際にインデックスに関しても整列関係を崩さないようにすればよい．

只对于值相等的元素，要维持index顺序 ==>改进为稳定排序

安定性も昨今ではあまり重要視されない（と思う）．

# ボゴソート

bogus + sort = bogosort

完全運任せソート

与えられた列をランダムにシャッフルし，たまたま完全にソートされている状態になるまで繰り返す．

# ボゴソートの計算量

シャッフル自体の操作は$O(n)$.

ソートされているかどうかをチェックにかかる比較回数の期待値は$n$が十分大きければ$e-1$に収束する.

列をシャッフルした後，たまたまソートされた状態になっているという確率は$1/n!$なので，その場合に遭遇するために必要な試行回数の期待値は$n!$.

Gruber, H., Holzer, M., & Ruepp, O. (2007, June). Sorting the slow way: an analysis of perversely awful randomized sorting algorithms. In *International Conference on Fun with Algorithms,* 183-197.

# ボゴソートの計算量

よって，全体では$O(n!\,n)$という素晴らしいほどに
非効率なアルゴリズム. 😇

かつ不安定.

而且

Gruber, H., Holzer, M., & Ruepp, O. (2007, June). Sorting the slow way: an analysis of perversely awful randomized sorting algorithms. In *International Conference on Fun with Algorithms,* 183-197.

# ボゴソートを理解しておく理由

非効率なのはすぐにわかる.

だけど「どのくらい」非効率かをしっかりと言えることが重要.

きちんと計算量を見積もることが出来るか，のよいエクササイズ. ☺️

估计

# 挿入ソート（Insertion sort）

与えられた配列の内，頭からi番目まではソートされているとする．

その状況下でi+1番目の要素を入れる場所を順番にチェックして探し，その場所に挿入する．

上記を要素の先頭から末尾まで順に行う．

# 挿入ソート（Insertion sort）

i+1番目の要素を一旦別の場所に退避させる.

i番目と比較し，退避させた値がより小さければi+1番目をi番目と置き換える.

次に，i-1番目と比較し，退避させた値がより小さければi番目をi-1番目と置き換える.

これを退避させた値のほうが大きい場所まで繰り返し，止まったところで退避させた値を戻す.

# 挿入ソート例

初期状態：[3, 5, 2, 1, 6, 4]

# 挿入ソート例

初期状態：[3, 5, 2, 1, 6, 4]

#1：[3, 5, 2, 1, 6, 4] そのまま

# 挿入ソート例

初期状態：[3, 5, 2, 1, 6, 4]
#1：[3, 5, 2, 1, 6, 4] そのまま
#2：[3, 5, 2, 1, 6, 4]

# 挿入ソート例

初期状態：[3, 5, 2, 1, 6, 4]

#1：[3, 5, 2, 1, 6, 4] そのまま

#2：[3, 5, 2, 1, 6, 4]

-> 「2」を別の領域に退避

-> [3, 5, 2, 1, 6, 4]，5は2より大きい -> [3, 5, 5, 1, 6, 4]

-> [3, 5, 5, 1, 6, 4]，3は2より大きい -> [3, 3, 5, 1, 6, 4]

-> 先頭まで行ったので，退避した2を持ってくる

-> [2, 3, 5, 1, 6, 4]

# 挿入ソート例

初期状態：[3, 5, 2, 1, 6, 4]

#1：[3, 5, 2, 1, 6, 4] そのまま

#2：[3, 5, 2, 1, 6, 4] -> … -> [2, 3, 5, 1, 6, 4]

#3：[2, 3, 5, 1, 6, 4] -> … -> [1, 2, 3, 5, 6, 4]

# 挿入ソート例

初期状態：[3, 5, 2, 1, 6, 4]
#1：[3, 5, 2, 1, 6, 4] そのまま
#2：[3, 5, 2, 1, 6, 4] -> … -> [2, 3, 5, 1, 6, 4]
#3：[2, 3, 5, 1, 6, 4] -> … -> [1, 2, 3, 5, 6, 4]
#4：[1, 2, 3, 5, 6, 4] そのまま

# 挿入ソート例

初期状態：[3, 5, 2, 1, 6, 4]
#1：[3, 5, 2, 1, 6, 4] そのまま
#2：[3, 5, 2, 1, 6, 4] -> … -> [2, 3, 5, 1, 6, 4]
#3：[2, 3, 5, 1, 6, 4] -> … -> [1, 2, 3, 5, 6, 4]
#4：[1, 2, 3, 5, 6, 4] そのまま
#5：[1, 2, 3, 5, 6, 4] -> … -> [1, 2, 3, 4, 5, 6]

# 挿入ソート実装例

```
def insertionsort(seq):
    for i in range(1, len(seq)):
        j = i - 1
        tmp = seq[i]
        while seq[j] > tmp and j > -1:
            seq[j+1] = seq[j]
            j -= 1
        seq[j+1] = tmp
    return seq
```

是否到达头部

# 挿入ソートの計算量

1回の平均的な比較・移動回数はそれぞれ$i/2$．それを$n-1$回繰り返す．

最後は$n$番目の要素を，最大$n-1$回比較・移動する．

よって，

$$\sum_{i=1}^{n-1}\left(\frac{i}{2}+\frac{i}{2}\right)=\sum_{i=1}^{n-1} i=\frac{(n-1)(n-2)}{2}\rightarrow O(n^2)$$

最悪のケースはどんな場合？ 逆序

# 二分挿入ソート

挿入する場所を探す際に，二分探索を使うこともできる．

挿入する場所を探すための二分探索の方法は，4回目の講義を参照（特に基本課題4-a）．

# 二分挿入ソートの計算量

i番目の要素を挿入するための必要な操作：

挿入するべき場所を見つける：**$\log i$回の操作.**

その場所に挿入する：平均的には$i/2$の位置にいると期待できるため，$i/2$回の要素の移動の操作.

pythonの場合，ここをforループにできるのでさらに定数倍改善できる.

よって,

$$\sum_{i=1}^{n-1}\left(\log i + \frac{i}{2}\right) = \log((n-1)!) + \frac{(n-1)(n-2)}{4} \to O(n^2)$$

# 挿入ソート

わかりやすい！実装も単純.

追加の記憶領域をほとんど必要としない.

事前にソートされている配列に追加するときには有利.

O(n)

安定的アルゴリズムとして実装できる.

# 二分挿入ソートのボトルネック

二分挿入ソートの着眼点自体は決して悪くない.

挿入するべき場所を見つける部分を$O(\log(n!))$まで落とすことはできている.

その後の挿入するところが引っかかっている.

ここが$O(n^2)$.

要素の挿入をなんとかできれば良さそう？

# ツリーソート

BST

与えられた配列を二分探索木に全て挿入し，さらにその二分探索木を値の最も小さいもの（最も左にあるノード）から順に値を取り出していく．

二分探索木を値の小さい順に辿るのは，左の部分木を辿った後で自ノードに戻り，さらに右の部分木を辿る操作を再帰的に行えば良い．

これを中間順巡回（in-order traversal）という．

# ツリーソート実装例

```
class BinarySearchTree:
    # 二分探索木の実装は前回のスライドを参照.
    # ツリーソートのために2つのメソッドを追加で実装.

    # 中間順巡回を行うメソッド．2つ目の引数はソート
    # されたリストが入る．最初は空リスト.
    def inOrderTraversal(self):
        return self._inOrderRec(self.root, [])
```

# ツリーソート実装例

```
class BinarySearchTree:
    ...
    # 再帰を使って中間順巡回.
    def _inOrderRec(self, node, array):
        if node:
            self._inOrderRec(node.left, array)
            array.append(node.data)
            self._inOrderRec(node.right, array)
        return array
```

# ツリーソート実装例

```
bst = BinarySearchTree()

# 二分探索木を作る
bst.createTree([3, 5, 2, 1, 6, 4])

# 木をたどりながら値を取り出すとソートされている
sorted_array = bst.inOrderTraversal()
```

# ツリーソートの計算量

二分探索木の構築には$O(\log(n!))$.

前回のスライド参照.

$O(n \log n)$として議論されることが多い.

要素の挿入$O(\log n)$が$n$回ある，と考える.

# ツリーソートの計算量

要素$n$の二分探索木を全て辿る処理の回数$T(n)$は，左の部分木と右の部分木を辿る回数と自分自身の処理回数の和になる．もしバランスが取れている木だとすれば，

$$T(n) = 2T(n/2) + 1$$

と表せ，ここから，

$$T(n) = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

ただし，$h$は木の高さで$h = \log n$より，$T(n) = 2n - 1$.

# ツリーソート

したがって二分探索木から取り出すところは$O(n)$.

よって平均的には$O(n \log n)$でソートできると期待できる.

ただし最悪の場合には二分探索木に偏りができ，木の構築に$O(n^2)$かかるため，全体としても$O(n^2)$となる.

# バブルソート

並べたい順に1つずつ「浮き上がらせていく」ソート.

水の中の気泡が浮き上がってくるようなイメージ.

より小さい値をどんどん配列の前に送っていく.

# バブルソート

（以下，配列の要素を順番に後ろから見ていく場合．）

今見ている要素（$n$番目）が1つ前の要素（$n-1$番目）より小さい場合，場所を入れ替える．

同じことを$n-1$番目と$n-2$番目の要素について行い，以降繰り返す．

最後まで行うと，1番目の要素は一番小さい値になる．

# バブルソート例

初期状態：[3, 5, 2, 1, 6, 4]

# バブルソート例

初期状態：[3, 5, 2, 1, 6, 4]
#1：[3, 5, 2, 1, 6, 4] -> [3, 5, 2, 1, 4, 6]

# バブルソート例

初期状態：[3, 5, 2, 1, 6, 4]

#1：[3, 5, 2, 1, 6, 4] -> [3, 5, 2, 1, 4, 6]

#2：[3, 5, 2, 1, 4, 6] そのまま

# バブルソート例

初期状態：[3, 5, 2, 1, 6, 4]
#1：[3, 5, 2, 1, 6, 4] -> [3, 5, 2, 1, 4, 6]
#2：[3, 5, 2, 1, 4, 6] そのまま
#3：[3, 5, 2, 1, 4, 6] -> [3, 5, 1, 2, 4, 6]

# バブルソート例

初期状態：[3, 5, 2, 1, 6, 4]
#1：[3, 5, 2, 1, 6, 4] -> [3, 5, 2, 1, 4, 6]
#2：[3, 5, 2, 1, 4, 6] そのまま
#3：[3, 5, 2, 1, 4, 6] -> [3, 5, 1, 2, 4, 6]
#4：[3, 5, 1, 2, 4, 6] -> [3, 1, 5, 2, 4, 6]

# バブルソート例

初期状態：[3, 5, 2, 1, 6, 4]
#1：[3, 5, 2, 1, 6, 4] -> [3, 5, 2, 1, 4, 6]
#2：[3, 5, 2, 1, 4, 6] そのまま
#3：[3, 5, 2, 1, 4, 6] -> [3, 5, 1, 2, 4, 6]
#4：[3, 5, 1, 2, 4, 6] -> [3, 1, 5, 2, 4, 6]
#5：[3, 1, 5, 2, 4, 6] -> [1, 3, 5, 2, 4, 6]

これで，配列の一番最初の要素はソート済みとなる．
次は[3, 5, 2, 4, 6]を処理．以下同じように続ける．

# バブルソート実装例

```
def bubblesort(seq):
    size = len(seq)
    for i in range(size):
        for j in range(size-1, i, -1):
            if seq[j] < seq[j-1]:
                seq[j], seq[j-1] = seq[j-1], seq[j]
```

# バブルソートの計算量

こちらも先ほどと似たような感じで，$O(n^2)$.

速くはないが，コードがとてもシンプル.

こちらも安定的.

# シェーカーソート（Cocktail shaker sort）

双方向からやるバブルソート.

1つの方向からバブルソートし，最後まで行ったら今度は逆方向に進める.

最後にスワップを行った場所から逆方向のバブルソートを開始する.

# シェーカーソート例

初期状態：[3, 5, 2, 1, 6, 4]

前から後ろにバブルソート.

[3, 5, 2, 1, 6, 4] -> [3, 2, 5, 1, 6, 4] -> [3, 2, 1, 5, 6, 4] -> [3, 2, 1, 5, 4, 6]

一番最後のスワップはindex: 4の場所（4）.

# シェーカーソート例

次は，後ろから前にバブルソート．

[3, 2, 1, 5, 4, 6] -> [3, 2, 1, 4, 5, 6] -> [3, 1, 2, 4, 5, 6] -> [1, 3, 2, 4, 5, 6]

一番最後のスワップはindex: 0の場所（1）．

また逆方向からバブルソート．以降これを繰り返す．

# シェーカーソート実装例

```
def shakersort(seq):
    # ソート済みの左端，右端を保持する変数
    right = len(seq) - 1
    left = 0

    # 最後にスワップが起きた場所を格納する変数
    swapped = 0
```

# シェーカーソート実装例

```
def shakersort(seq):
    ...
    while left < right:
        for i in range(left, right): # 先頭からチェック
            if seq[i+1] < seq[i]:
                seq[i], seq[i+1] = seq[i+1], seq[i]
                swapped = i
        # 最後のスワップの場所でrightを更新
        right = swapped
```

# シェーカーソート実装例

```
def shakersort(seq):
    ...
    while left < right:
        ...
        [次は後ろからチェック]
        [最後のスワップの場所でleftを更新]


        # このwhileループ1回で左右からのチェック
        # を済ませることになる.
```

# バブルソートと何が違う？

もし，1方向動くときに最後に連続してk回スワップがなかった場合，そのk個分の要素はすでにソートされていることになる．

よってその分は次回以降考えなくて良いことになる．
（最後にスワップした場所を覚えておく理由）

この分だけバブルソートよりもちょっと効率が良い．

# シェーカーソートの計算量

最悪計算量はこちらも$O(n^2)$.

ただしバブルソートよりはちょっと速いことが期待できる.

ただし，，，

今までに紹介したものは多少の差はあれど，$O(n^2)$.

まだまだ遅い．．．

# シェルソート

Donald L. Shellさんが1959年に発表.

間隔の離れた要素の組に対して挿入ソートを行う.

この間隔を順次小さくしながら挿入ソートを繰り返す.

# シェルソートでの間隔の設定

間隔$h$の決め方は少し難しいが，倍数関係にあるものは避けたほうがいい．

例えば，$h = 4, 2, 1$のように倍数で順に変化させると，最後まで偶数番目にある要素と奇数番目にある要素が入れ替わることがないため，効率が悪くなる．

# シェルソートでの間隔の設定

よく知られている$h$の選び方として，

$$h_i = 2^i - 1 \quad (h = \cdots, 31, 15, 7, 3, 1)$$

$$h_i = \frac{3^i - 1}{2} \quad \text{or} \quad h_i = 3h_{i-1} + 1 \quad (h = \cdots, 121, 40, 13, 4, 1)$$

などがある．ただし，$h_i \leq n.$

$h_i$があまり$n$に近すぎる値にならないようにする場合もある．

hi<(3^i−1)/2

$h_i = \frac{3^i - 1}{2}$の場合，

hi<n/3

$h_i < \frac{n}{3}$とするなど

# シェルソートの例

以下の例では，間隔を$h_i = 3h_{i-1} + 1$で決めるとする．

配列の長さが10なので，最初の$h$は4．

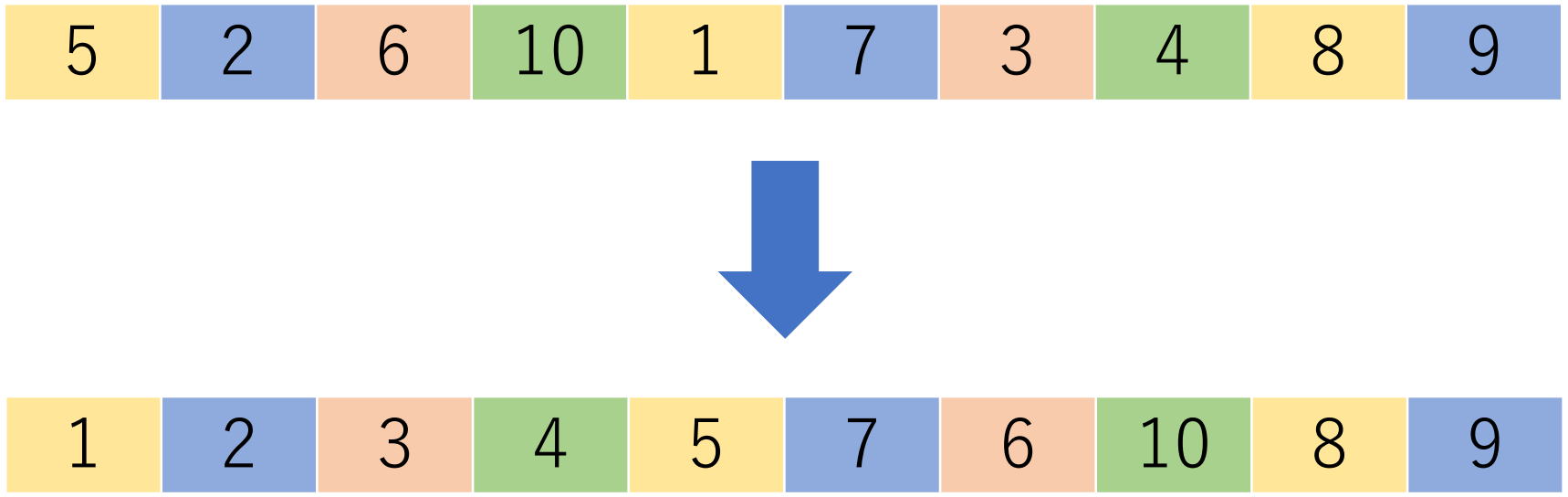
| 5 | 2 | 6 | 10 | 1 | 7 | 3 | 4 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|

# シェルソートの例

以下に示すように，間隔4ごとに要素を挿入ソートでソートをする．

# シェルソートの例

以下に示すように，間隔4ごとに要素を挿入ソートでソートをする．



# シェルソートの例

次に，$h_i = 3h_{i-1} + 1$に従って間隔を狭める．今回の場合は1になるので，単純に挿入ソートをする．

| 1 | 2 | 3 | 4 | 5 | 7 | 6 | 10 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|

# シェルソートの例

次に，$h_i = 3h_{i-1} + 1$に従って間隔を狭める．今回の場合は1になるので，単純に挿入ソートをする．

| 1 | 2 | 3 | 4 | 5 | 7 | 6 | 10 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|

# シェルソートの例

間隔を狭めていくごとに，小さい値はより左側に来ていることが期待できる．

一番最後の挿入ソート（間隔1）の時には，ある程度ソート済みの配列が出来上がっていると期待でき，最初から挿入ソートをするよりも効率が良いことが多い．

| 5 | 2 | 6 | 10 | 1 | 7 | 3 | 4 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|

| 1 | 2 | 3 | 4 | 5 | 7 | 6 | 10 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|

# シェルソートの実装例

```
def shellsort(seq):
    # 一番大きなhを決める.
    h = 1
    while h < len(seq) / 3:
        h = h * 3 + 1
```

# シェルソートの実装例

```
def shellsort(seq):
        …
        while h > 0:
                for i in range(h, len(seq)):    # 間隔hで挿入ソート
                        j = i
                        while j >= h and seq[j-h] > seq[j]:
                                seq[j-h], seq[j] = seq[j], seq[j-h]
                                j -= h
                h //= 3
    return seq
```

# シェルソートの計算量

間隔$h$の時，$n/h$個の要素を挿入ソートでソートするのが，$h$回ある．

$$\rightarrow O\left(h\times\left(\frac{n}{h}\right)^2\right)=O\left(\frac{n^2}{h}\right)$$

$h$が大きければ，$h\sim n$となるので，$O(n)$とみなせる．

一方，$h$が小さくなった時には，ソート済みに近い状態になっており，最初から挿入ソートするよりも時間がかからないと期待できる．

# シェルソートの計算量

$h$に依存するため，正確な計算量の分析は難しい.

$h_i = 2^i - 1$の時には，最悪の場合$O(n^{1.5})$,

$h_i = \frac{3^i - 1}{2}, h_i < \frac{n}{3}$の時には，平均$O(n^{1.25})$，最悪$O(n^{1.5})$,

となることが知られている.

その他いろんな$h$が提案されており，最悪でも$O\left(n^{4/3}\right)$や$O(n(\log n)^2)$になるものなどがある.

# もう少しなんとかならない？

シェルソートは今までの中では早いがそれでも$O(n^k)$程度.

多くの場合，$k$は1.25～1.5の値.

与えられた配列をそのまま扱うのでは無理がある.

じゃあどうする？

処理で工夫する

データ構造で工夫する

# 分割統治法（divide and conquer）

1つをまとめて処理するのは大変．．．

そこで領域をすぐに処理できるレベルまで小分けにして，
そこで処理する．

それを順にくっつけて戻していけば，最終的に達成したい
ゴールにたどり着く．

クイックソートとマージソートが代表例．

# クイックソート

その名の通り，速い！（ごく例外的なケースを除く）

Tony Hoareさんにが1962年に発表.

Hoare, C. A. (1962). Quicksort. *The Computer Journal*, *5*(1), 10-16.

# クイックソートの考え方

配列の中から1つ値を何かしらの基準で選ぶ．
　　選んだ値は枢軸（ピボット，pivot）と呼ばれる．

枢軸より小さいものと大きいものを振り分ける．この時点ではソートされていなくても良い．

振り分けた後，2つのグループに分割し，各グループに対して同じ処理を行う．最終的に要素1つのグループになる．

それらを全部結合すれば終わり！

# クイックソートの実装

「枢軸より小さいものと大きいものを振り分ける．この時点ではソートされていなくても良い．」

↑の実装方法としてはいくつかある．

よくあるのは，枢軸より小さいものと大きいものを格納する新しいリストを作り，要素を振り分けていく．

# クイックソートの実装

今日は新たにリスト（配列）を作らない方法でクイックソートを実装する例を紹介します．

以前にクイックソートを実装したことがある人も，ぜひ今回のやり方でも実装してみてください．

# クイックソートの実装方針

与えられている配列の一番後ろに位置する要素を枢軸とする.

2つのカーソルを準備する.

A：左から順に値を走査するためのカーソル

B：スワップ先の要素の位置を保持するカーソル

# クイックソートの実装方針

左から順に走査をし，以下のルールに従ってスワップを行う．

カーソルAが指す要素が枢軸よりも大きい．

→カーソルAを1つ進めるだけ．

カーソルAが指す要素が枢軸以下である．

→カーソルAとカーソルBの位置にある要素を交換し，それぞれのカーソルを1つ進める．

# クイックソートの実装方針

カーソルAが枢軸の位置まで到達したら，枢軸とカーソルBの位置にある要素を交換し，終了する．

この後，枢軸の左側のグループと右側のグループとで同じ処理を繰り返していく．

# クイックソート例

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

# クイックソート例

| 1 | 7 | 2 | 8 | 6 | 4 | 3 | 5 |
|---|---|---|---|---|---|---|---|
| ↑ | | | | | | | |
| ↑ | | | | | | | |

緑：左から順に値を走査するためのカーソル
橙：スワップ先の要素の位置を保持するカーソル

枢軸は一番後ろにある要素とするので，5.

# クイックソート例

| 1 | 7 | 2 | 8 | 6 | 4 | 3 | 5 |
|---|---|---|---|---|---|---|---|
| ↑ | | | | | | | |
| ↑ | | | | | | | |

1は枢軸の5以下である.

→緑と橙の位置にある要素を交換し，それぞれのカーソルを1つ進める.

（今回の場合は，2つのカーソルが同じ位置にあるので，実質的には要素のスワップはない.）

# クイックソート例

| 1 | 7 | 2 | 8 | 6 | 4 | 3 | 5 |
|---|---|---|---|---|---|---|---|
| | ↑ | | | | | | |
| | ↑ | | | | | | |

7は枢軸の5よりも大きい.

→緑のカーソルを1つ進めるだけ.

# クイックソート例

| 1 | 7 | 2 | 8 | 6 | 4 | 3 | 5 |
|---|---|---|---|---|---|---|---|
| | | ↑ | | | | | |
| | ↑ | | | | | | |

2は枢軸の5以下である.

→緑と橙の位置にある要素を交換し，それぞれのカーソルを1つ進める.

# クイックソート例

| 1 | 2 | 7 | 8 | 6 | 4 | 3 | 5 |
|---|---|---|---|---|---|---|---|
| | | ↑ | | | | | |
| | ↑ | | | | | | |

2は枢軸の5以下である.

→緑と橙の位置にある要素を交換し，それぞれのカーソルを1つ進める.

# クイックソート例

| 1 | 2 | 7 | 8 | 6 | 4 | 3 | 5 |
|---|---|---|---|---|---|---|---|
| | | | ↑ | | | | |
| | | ↑ | | | | | |

8は枢軸の5よりも大きい.

→緑のカーソルを1つ進めるだけ.

# クイックソート例

| 1 | 2 | 7 | 8 | 6 | 4 | 3 | 5 |
|---|---|---|---|---|---|---|---|
| | | | | ↑ | | | |
| | | ↑ | | | | | |

6は枢軸の5よりも大きい.

→緑のカーソルを1つ進めるだけ.

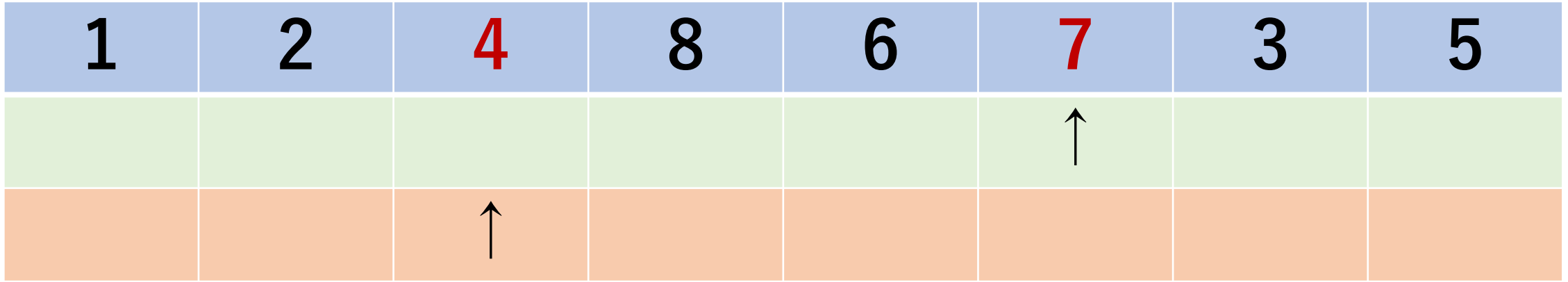
# クイックソート例

| 1 | 2 | 7 | 8 | 6 | 4 | 3 | 5 |
|---|---|---|---|---|---|---|---|
| | | | | | ↑ | | |
| | | ↑ | | | | | |

4は枢軸の5以下である.

→緑と橙の位置にある要素を交換し，それぞれのカーソルを1つ進める.

# クイックソート例




4は枢軸の5以下である．想法：把大于pivot的值攒起来

→緑と橙の位置にある要素を交換し，それぞれのカーソルを1つ進める．

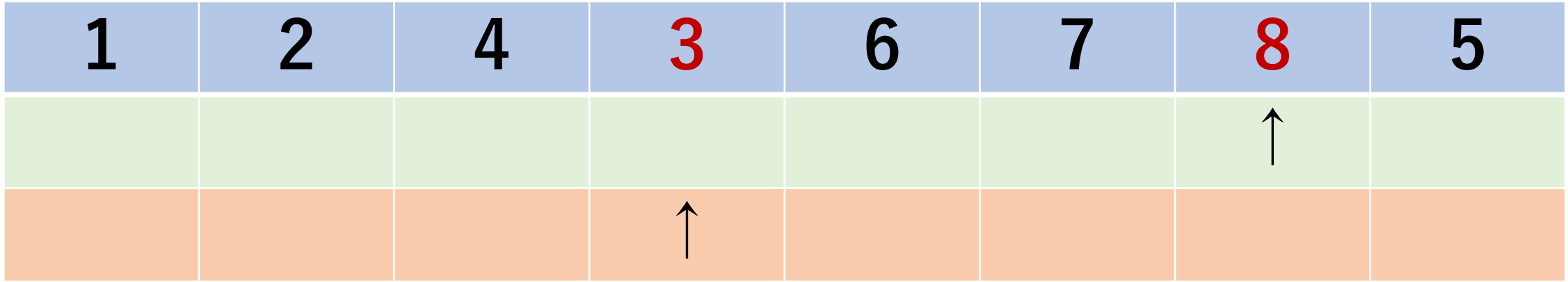
# クイックソート例

| 1 | 2 | 4 | 8 | 6 | 7 | 3 | 5 |
|---|---|---|---|---|---|---|---|
| | | | | | | ↑ | |
| | | | ↑ | | | | |

3は枢軸の5以下である.

→緑と橙の位置にある要素を交換し，それぞれのカーソルを1つ進める.

# クイックソート例




3は枢軸の5以下である.

→緑と橙の位置にある要素を交換し，それぞれのカーソルを1つ進める.

# クイックソート例

| 1 | 2 | 4 | 3 | 6 | 7 | 8 | 5 |
|---|---|---|---|---|---|---|---|
| | | | | | | | ↑ |
| | | | | ↑ | | | |

枢軸まで辿り着いた.

→緑と橙の位置にある要素を交換し，終了.

# クイックソート例

| 1 | 2 | 4 | 3 | 5 | 7 | 8 | 6 |
|---|---|---|---|---|---|---|---|
| | | | | | | | ↑ |
| | | | | ↑ | | | |

枢軸まで辿り着いた.

→緑と橙の位置にある要素を交換し，終了.

# クイックソート例

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

1回目の交換操作の結果は上のようになる.

# クイックソート例

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

続いて，枢軸の左側，右側について同じように処理を行っていく．

（この実装には何を使えば良い？）

# クイックソートの実装例

```
# seqを直接編集し，ソートする関数
# seq: ソートを施すリスト
# left: 左端index，　right: 右端index
def qsort(seq, left, right):
```

［再帰の終了条件は？］

**left >= right**

# クイックソートの実装例

```
def qsort(seq, left, right):
    ...
    pivot = seq[right]      # もっと右にある要素を枢軸にする
    j = left                # スワップ先のカーソル（カーソルB）
```

递归出口

左闭右闭区间

# クイックソートの実装例

```
def qsort(seq, left, right):
        …
        # iはカーソルA
        for i in range(left, right):
                if seq[i] <= pivot:
```

A实际上每次必前进一格

[カーソルAとカーソルBの位置にある要素を
交換し，それぞれのカーソルを1つ進める．]

# クイックソートの実装例

```
def qsort(seq, left, right):
        …
        for i in range(left, right):
                …
                # 枢軸の移動
<==
缩进不对 seq[right], seq[j] = seq[j], seq[right]
```

# クイックソートの実装例

```
def qsort(seq, left, right):
    ...
    for i in range(left, right):
        ...
        seq[right], seq[j] = seq[j], seq[right]
    [左側グループを再帰で処理]
    [右側グループを再帰で処理]
    return seq
```

<== 缩进不对

# クイックソートの実行例

seq = [3, 8, 14, 12, 90, 1, 4, 29, 43 , 2, 10, 6, 37, 78, 50, 18]
qsort(seq, 0, len(seq)-1)
-----実行結果-----
[3, 8, 14, 12, 1, 4, 2, 10, 6, 18, 29, 43, 37, 78, 50, 90]
[3, 1, 4, 2, 6, 14, 12, 10, 8, 18, 29, 43, 37, 78, 50, 90]
[1, 2, 4, 3, 6, 14, 12, 10, 8, 18, 29, 43, 37, 78, 50, 90]
[1, 2, 3, 4, 6, 14, 12, 10, 8, 18, 29, 43, 37, 78, 50, 90]
…

# クイックソートの実行例

seq = [3, 8, 14, 12, 90, 1, 4, 29, 43 , 2, 10, 6, 37, 78, 50, 18]
qsort(seq, 0, len(seq)-1)
-----実行結果-----
[3, 8, 14, 12, 1, 4, 2, 10, 6, 18, 29, 43, 37, 78, 50, 90]
[3, 1, 4, 2, 6, 14, 12, 10, 8, 18, 29, 43, 37, 78, 50, 90]
[1, 2, 4, 3, 6, 14, 12, 10, 8, 18, 29, 43, 37, 78, 50, 90]
[1, 2, 3, 4, 6, 14, 12, 10, 8, 18, 29, 43, 37, 78, 50, 90]
…
[1, 2, 3, 4, 6, 8, 10, 12, 14, 18, 29, 37, 43, 50, 78, 90]

# クイックソートの実行例

seq = [3, 8, 14, 12, 90, 1, 4, 29, 43 , 2, 10, 6, 37, 78, 50, 18]

qsort(seq, 0, len(seq)-1)

-----実行結果-----

[3, 8, 14, 12, 1, 4, 2, 10, 6, 18, 29, 43, 37, 78, 50, 90]

[3, 1, 4, 2, 6, 14, 12, 10, 8, 18, 29, 43, 37, 78, 50, 90]

[1, 2, 4, 3, 6, 14, 12, 10, 8, 18, 29, 43, 37, 78, 50, 90]

[1, 2, 3, 4, 6, 14, 12, 10, 8, 18, 29, 43, 37, 78, 50, 90]

…

[1, 2, 3, 4, 6, 8, 10, 12, 14, 18, 29, 37, 43, 50, 78, 90]

# クイックソートの実行例

seq = [3, 8, 14, 12, 90, 1, 4, 29, 43 , 2, 10, 6, 37, 78, 50, 18]

qsort(seq, 0, len(seq)-1)

-----実行結果-----

[3, 8, 14, 12, 1, 4, 2, 10, 6, 18, 29, 43, 37, 78, 50, 90]

[3, 1, 4, 2, 6, 14, 12, 10, 8, 18, 29, 43, 37, 78, 50, 90]

[1, 2, 4, 3, 6, 14, 12, 10, 8, 18, 29, 43, 37, 78, 50, 90]

[1, 2, 3, 4, 6, 14, 12, 10, 8, 18, 29, 43, 37, 78, 50, 90]

…

[1, 2, 3, 4, 6, 8, 10, 12, 14, 18, 29, 37, 43, 50, 78, 90]

# クイックソートの計算量

$n$個の配列（$n \geq 2$）のソートにかかる比較の総回数を$Q_n$とする.

入れ替えの回数は多くても比較の回数の定数倍なので,
今回の計算量の議論では比較の回数のみを考える.

$n$個の配列に対して，すべての要素を枢軸の値に応じて左右に振り分けるのに必要な比較の回数は，最低でも$n-1$回.

https://cs.pwr.edu.pl/cichon/Math/QSortAvg.pdf

# クイックソートの計算量：最良な場合

すごくラッキーで，毎回半分に分割できる．よって，

$$Q_n = (n-1) + 2Q_{n/2}$$

議論を簡単にするため，$n = 2^m$とすると，

$$Q_{2^m} = (2^m - 1) + 2Q_{2^{m-1}}$$

$$\frac{Q_{2^m}}{2^m} = \left(1 - \frac{1}{2^m}\right) + \frac{Q_{2^{m-1}}}{2^{m-1}}$$

# クイックソートの計算量：最良な場合

よって，$\frac{Q_{2^m}}{2^m}$の一般解は，

$$\frac{Q_{2^m}}{2^m} = \sum_{k=1}^{m} \left(1 - \frac{1}{2^k}\right) = m - 1 - \left(1 - \frac{1}{2^m}\right)$$

よって $Q_{2^m} = 2^m(m-2) + 1$となり，$n = 2^m$なので，
$O(Q_n) = O(n \log n)$.

# クイックソートの計算量：最悪な場合

すごくアンラッキーで，毎回すべて片方に偏る．（先の実装ではどんな時に起きる？）

每次pivot都是 最大or最小元素
仅让问题规模–1

この時，

$$Q_n = (n-1) + Q_{n-1}$$

これは，

$$Q_n = \frac{(n-1)n}{2}$$

となり，$O(Q_n) = O(n^2)$となる．

# クイックソートの計算量：一般的な場合

$n$個の配列が$k$個と$n-k-1$個のグループに分割された，
とする．

$k$は0から$n-1$までの値を取り，どれも等確率で起きると
仮定すると，分割後の比較の回数の期待値は，

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

と表せる．

よって，$Q_n = (n-1) + \frac{2}{n}\sum_{k=0}^{n-1}Q_k$

# クイックソートの計算量：一般的な場合

式変形をし，$Q_{n-1}$に関する式も作ると，

$$nQ_n = n(n-1) + 2\sum_{k=0}^{n-1} Q_k$$

$$(n-1)Q_{n-1} = (n-1)(n-2) + 2\sum_{k=0}^{n-2} Q_k$$

上の式から下の式を引いて，

$$nQ_n - (n-1)Q_{n-1} = n(n-1) - (n-1)(n-2) + 2Q_{n-1}$$

# クイックソートの計算量：一般的な場合

整理すると，

$$nQ_n = 2(n-1) + (n+1)Q_{n-1}$$

$n(n+1)$で両辺を割って，

$$\frac{Q_n}{n+1} = \frac{2(n-1)}{n(n+1)} + \frac{Q_{n-1}}{n}$$

この式から，$\frac{Q_n}{n+1}$の一般解を求めることができ，

$$\frac{Q_n}{n+1} = 2\sum_{k=1}^{n} \frac{k-1}{k(k+1)}$$

# クイックソートの計算量：一般的な場合

更に変形すると,

$$Q_n = (n+1)\left(4\sum_{k=1}^{n}\frac{1}{k+1} - 2\sum_{k=1}^{n}\frac{1}{k}\right)$$

さらに,

$$\sum_{k=1}^{n}\frac{1}{k+1} = \frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}+\frac{1}{n+1}$$
$$= \left(1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}\right)-1+\frac{1}{n+1}$$
$$= \sum_{k=1}^{n}\frac{1}{k}-\frac{n}{n+1}$$

と変形していくと,

# クイックソートの計算量：一般的な場合

$$Q_n = 2(n+1)\sum_{k=1}^{n}\frac{1}{k} - 4n$$

とできる.

ここで，$n$が大きければ，$\sum_{k=1}^{n}\frac{1}{k} \approx \log n + \gamma$に近似できることが知られている.

$\gamma$はオイラーの定数で0.57721…

よって，$O(Q_n) = O(n\log n)$.

# クイックソートの計算量

一般的には，$O(n \log n)$．よってかなり効率的！

ただし，最悪の場合には$O(n^2)$になる．

一般的にはここまで悪くなることは頻発しないが，
ある程度再帰が深くなってしまうことはあり得る．

枢軸をランダムに選ぶことで，意地悪いケースでも
うまく対応できることがある．（乱択化）

不稳定

安定的ではない（要素を飛び越えてスワップするため）．

# マージソート

分割統治法による代表的なもう1つのアルゴリズム.

与えられた配列を2分割していき，要素1個の配列まで小さくする.

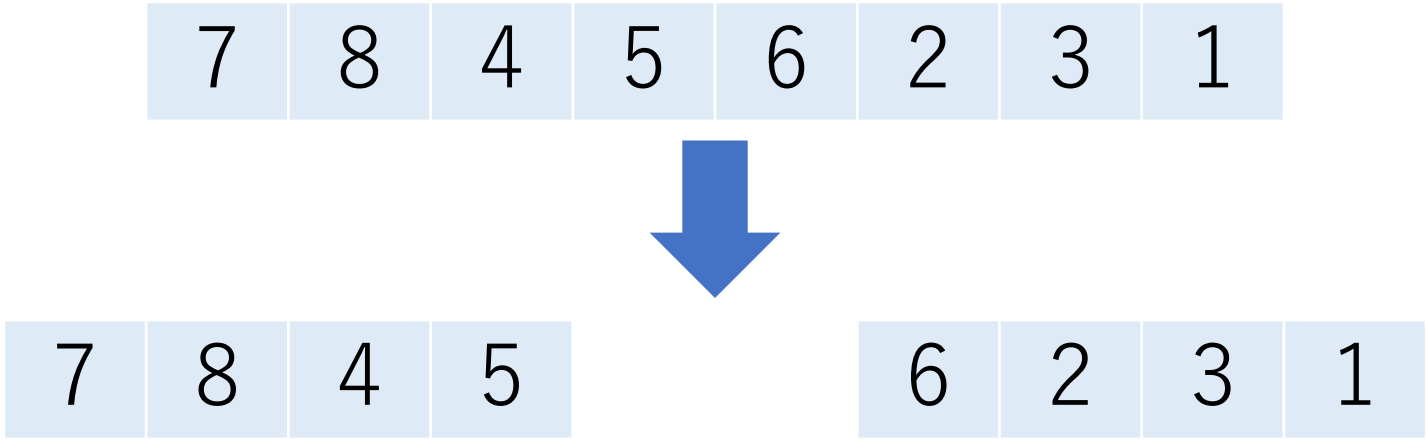
分割したもの同士をソートをしながら結合して，元の配列の大きさに戻す.

フォン・ノイマンによる考案（1945年）とされている.

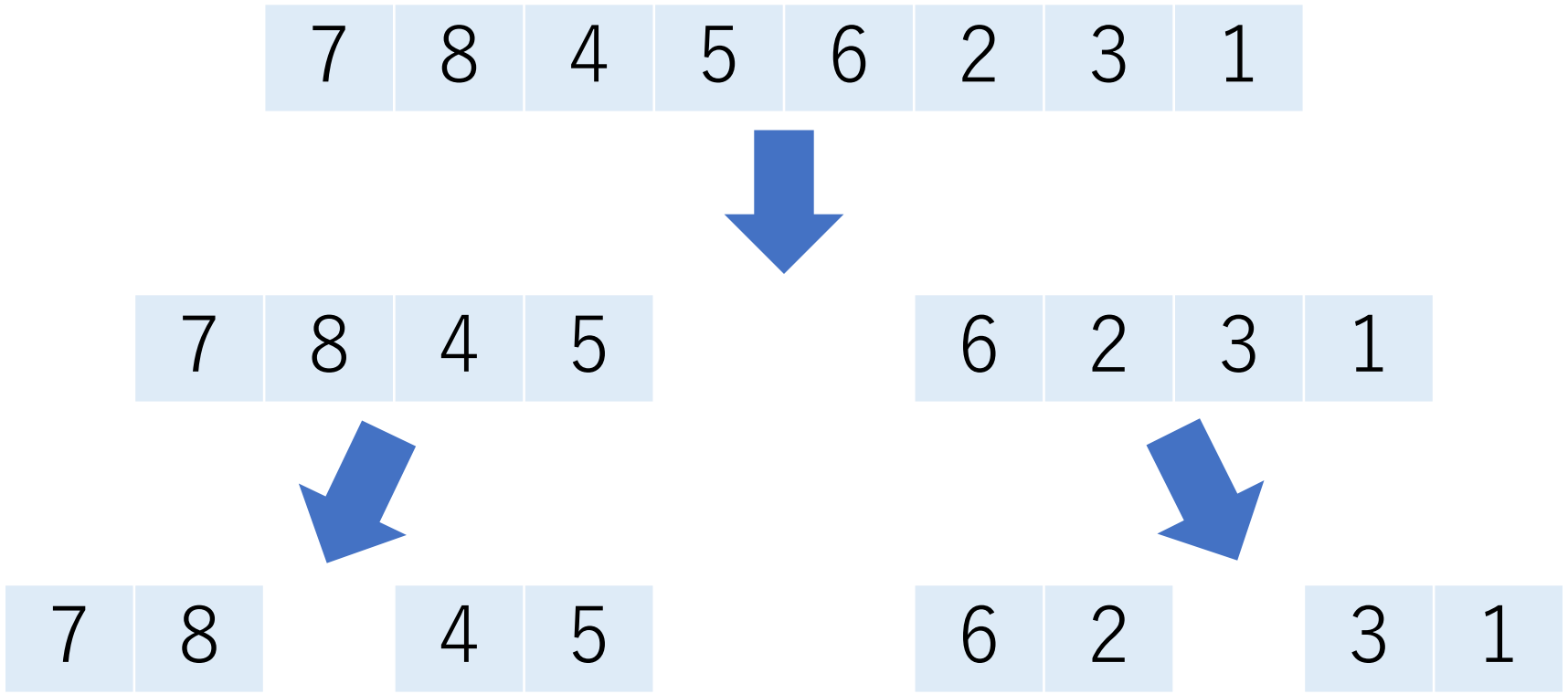
# マージソート例

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

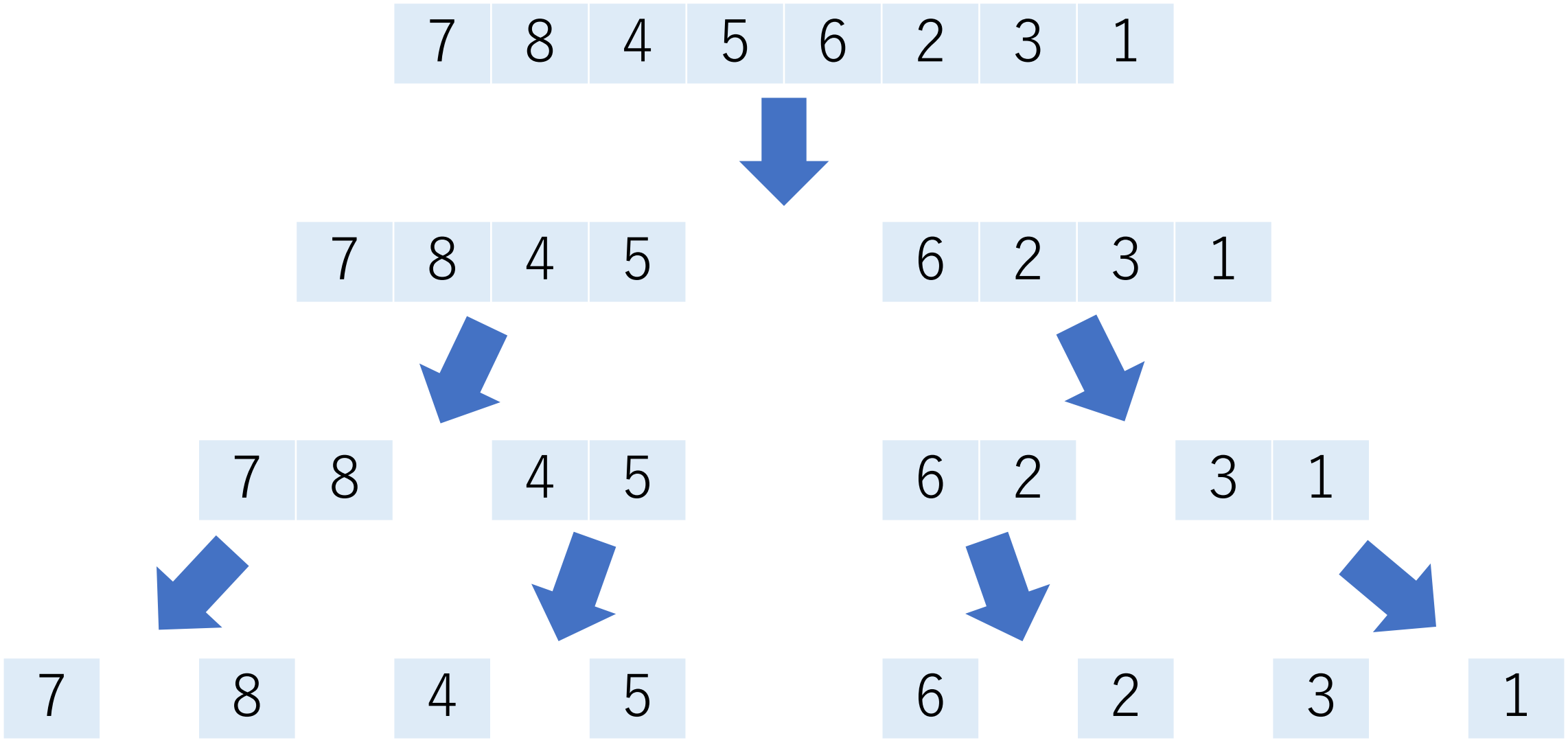
# マージソート例



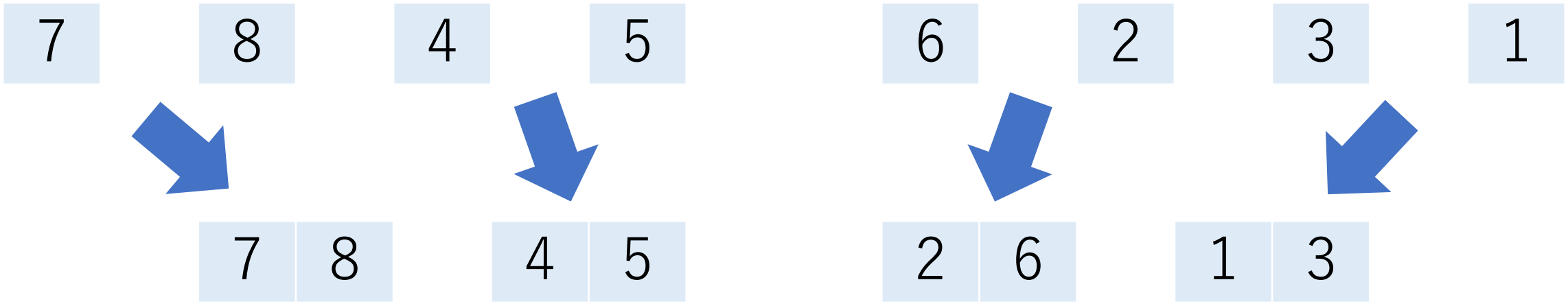
# マージソート例



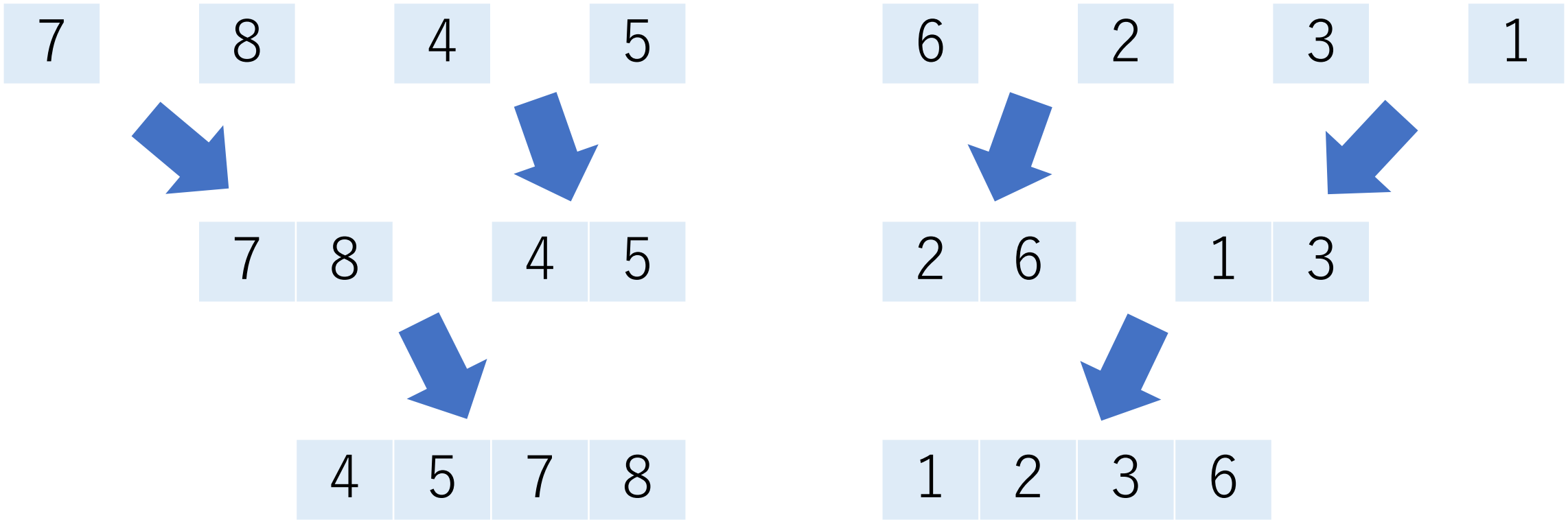
# マージソート例



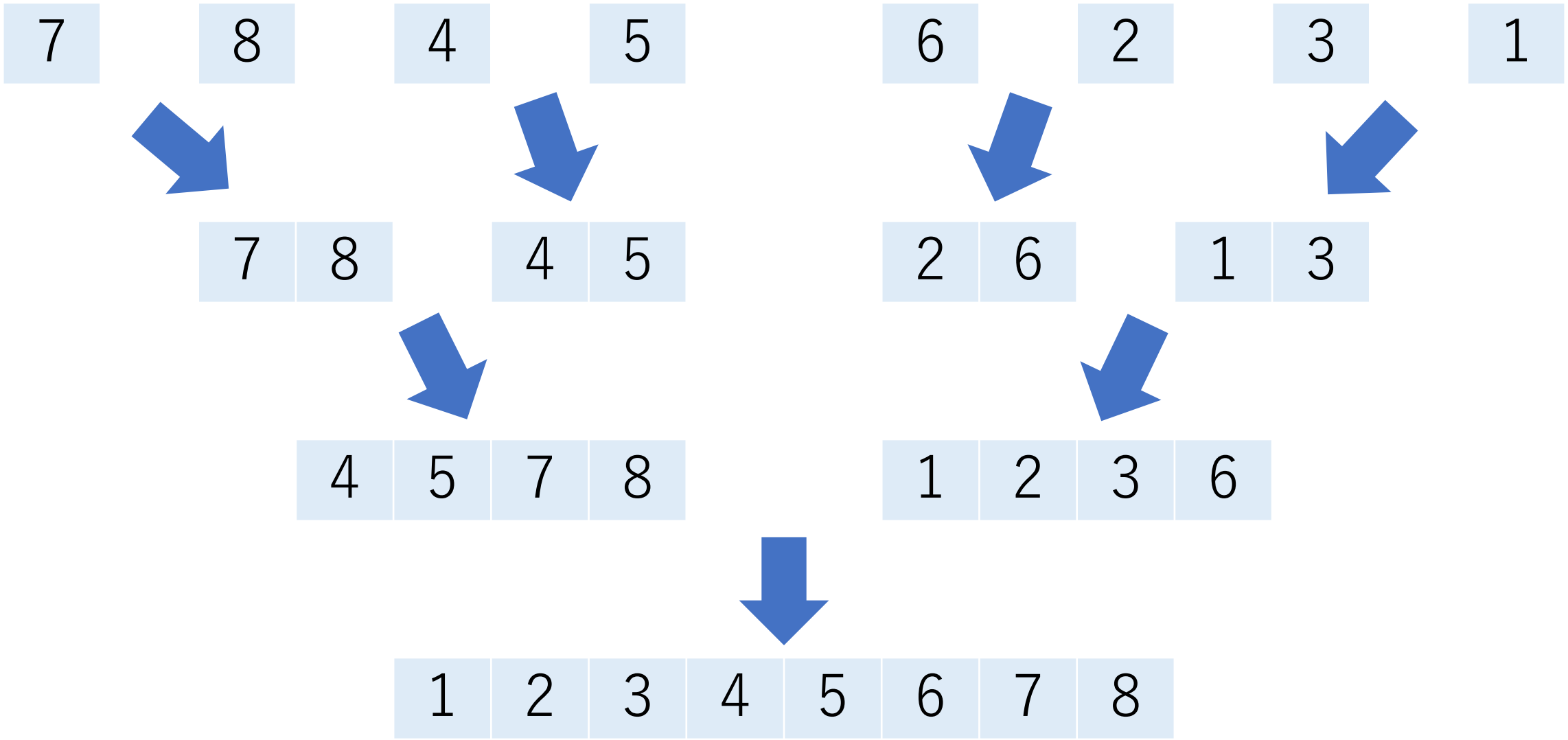
# マージソート例



# マージソート例



# マージソート例



# マージソートの実装例

```
def mergesort(seq):
    if len(seq) <= 1: return seq
```

**递归出口**

```
    # 2つに分割するだけ
    left = mergesort(seq[:len(seq)//2])
    right = mergesort(seq[len(seq)//2:])


    # これらの値が返ってきたときには，left，right
    # 各々でソートができている状態になっている．
```

# マージソートの実装例

```
def mergesort(seq):
        …
        merged = []
        cur_l = cur_r = 0
        # マージ作業．小さい方から順にmergedに入れる．
        while cur_l < len(left) and cur_r < len(right):
                if left[cur_l] <= right[cur_r]:   # 安定性を確保
                        merged.append(left[cur_l])
                        cur_l += 1
                else:
                        merged.append(right[cur_r])
                        cur_r += 1
```

# マージソートの実装例

```
def mergesort(seq):
    ...
    # もし余った要素があればくっつける.
    if cur_l < len(left):
        merged.extend(left[cur_l:])
    elif cur_r < len(right):
        merged.extend(right[cur_r:])

    return merged
```

# マージソートの計算量：最良な場合

$n$個の配列（$n \geq 2$）のソートにかかる比較の総回数を$M_n$とする.

$n$個の配列のソートは，$n/2$個の配列2つをマージすることにより実現される.

最良の場合，マージの際に必要な比較の回数は$n/2$回となる.
どちらかの配列から先に全部の要素を取り，
もう1つの配列を後で後ろにくっつける場合.

# マージソートの計算量：最良な場合

これを式で表すと，

$$M_n = \frac{n}{2} + 2M_{n/2}$$

議論を簡単にするため，$n = 2^m$とすると，

$$M_{2^m} = 2^{m-1} + 2M_{2^{m-1}}$$

式を変形して，

$$\frac{M_{2^m}}{2^m} = \frac{1}{2} + \frac{M_{2^{m-1}}}{2^{m-1}}$$

# マージソートの計算量：最良な場合

よって，$\frac{M_{2^m}}{2^m}$の一般解は，

$$\frac{M_{2^m}}{2^m} = \sum_{k=1}^{m} \frac{1}{2} = \frac{m-1}{2}$$

したがって，

$$M_{2^m} = 2^m \frac{m-1}{2} = 2^{m-1}(m-1)$$

$n = 2^m$であるので，$O(M_n) = O(n \log n)$となる．

# マージソートの計算量：最悪な場合

最悪の場合，マージの際に必要な比較の回数は$n-1$回.
2つの配列から交互に要素を取り出す場合.

よって，$M_n = (n-1) + 2M_{n/2}$となり，先程と同様の議論をすれば，$O(M_n) = O(n \log n)$.

$M_{2^m} = 2^m(m-2) + 1$となるので，最良のケースと比べて，比較の総回数がおおよそ2倍になる.

# マージソートの計算量

最悪でも最良でも計算量は$O(n \log n)$なので，クイックソートよりも運の悪さに依存しない．

マージ操作の部分があるため，クイックソートよりは一般的には少し遅い．

メモリを食いやすいので，大きい配列のときは注意．

# クイックソートとマージソート

クイックソート

粗略地

分割するときに（ざっくり）ソートする

結合するときは何も考えない

マージソート

分割するときは何も考えない

結合するときにソートする

# なんとかしようぜ．（再掲）

与えられた配列をそのまま扱うのでは無理がある．

じゃあどうする？

処理で工夫する

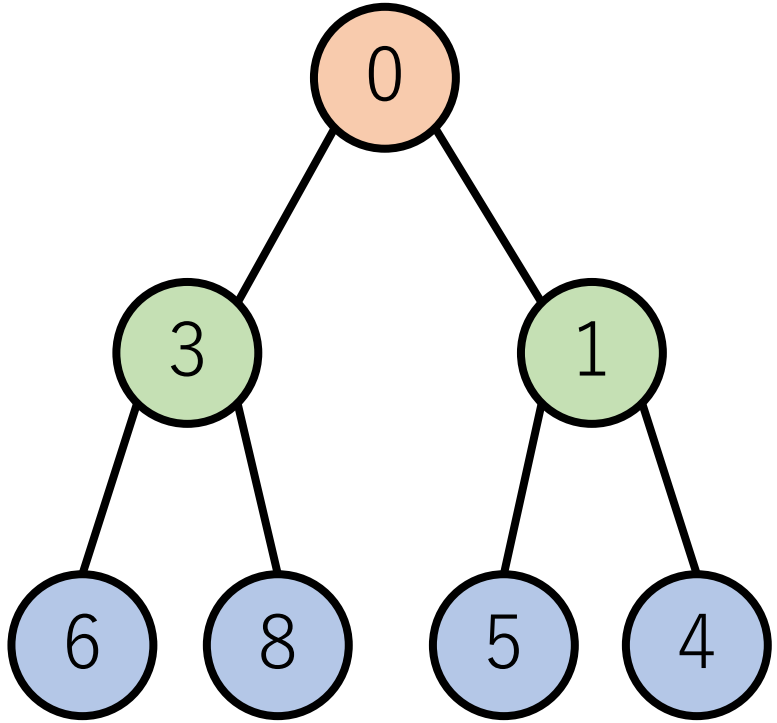
データ構造で工夫する

（ツリーソートはこの1つとも言える．）

# ヒープソート

前に習ったヒープを使おう．ヒープは
根が最大もしくは最小になっている構造.

与えられた配列を全部ヒープに押し
込めた後，1つずつ取り出せば自動的に
ソートされている！




（ヒープに関しては3回目の講義参照）．

# ヒープソートの実装例

```
import heapq


def heapsort(seq):
    heap = []
    while seq:          # ヒープを作る
        heapq.heappush(heap, seq.pop())
    while heap:         # ヒープから取り出す
        seq.append(heapq.heappop(heap))
```

# ヒープソートの計算量

ヒープを作る：

要素追加ごとに $O(\log n)$ の処理が必要.
それが $n$ 回起きる.

ヒープから取り出す：

要素削除ごとに $O(\log n)$ の処理が必要.
それが $n$ 回起きる.

よって, $O(n \log n)$.

# ヒープソートの特徴

メリット：

ヒープを使うので，データの出現パターンにあまり影響されない．最悪の場合でも$O(n \log n)$.

デメリット：

ヒープ処理の分があるため，クイックソートよりは一般的には遅い.

単純な実装では記憶領域$O(n)$が必要（ヒープ分）.

# 組み合わせでさらに改善：イントロソート

当初はクイックソートを使う.

ただし再帰の深さが$\log n$を超えた場合，ヒープソートに切り替える．これにより再帰が深くなりすぎることが防げる.

この工夫で最悪の場合でも$O(n \log n)$を実現.

Musser, D. R. (1997). Introspective sorting and selection algorithms. *Software: Practice and Experience*, *27*(8), 983-993.

# 組み合わせでさらに改善：TimSort

Tim Peterさんによって提案され，PythonやJavaに取り入れられている．

挿入ソートを取り入れてソート済みの長さ32〜64の部分列（run）を作り，マージソートによりrunを結合させる．

runの作成やマージの仕方に様々なヒューリスティクスが組み入れられている．

Tim Peter (2002).
https://mail.python.org/pipermail/python-dev/2002-July/026837.html

# それでもまだ$O(n \log n)$. . .

big O: 上界

比較を用いるソートアルゴリズムでは，どんなに頑張っても，$O(n \log n)$時間かかってしまう最悪の入力ケースが存在することが知られている.

計算量の下界，という.

# 比較に基づいたソートの下界

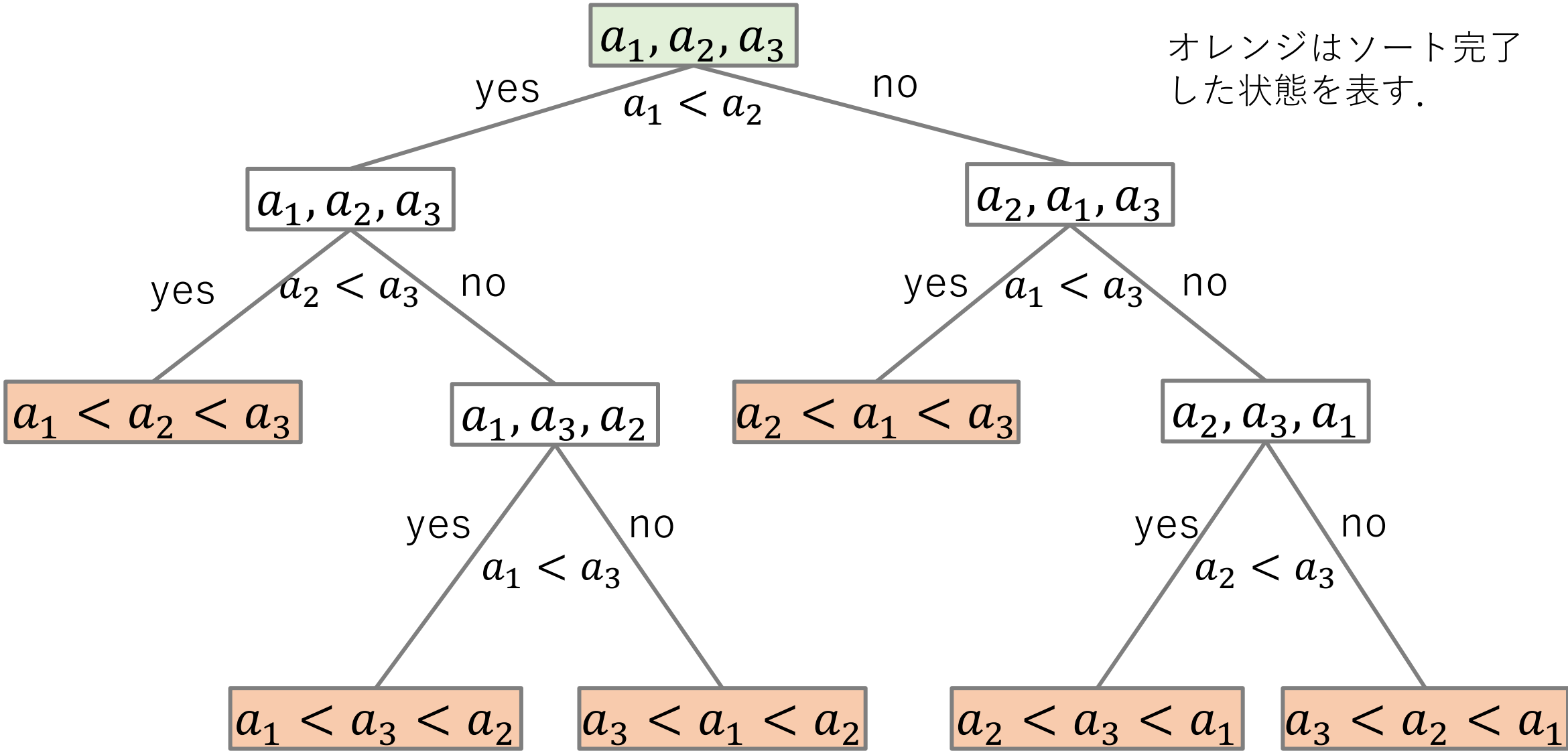
比較・交換を順次行うことで得られるソートの全ての起こりうる手順は決定木で表すことができる.

決策树 決定木：条件式に応じて子ノードへ順次移動すると，葉ノードにおいてたどり着くべき状態が求まる.

さらにその決定木は二分木となる.

比較した結果，入れ替えるか入れ替えないかの2択になるため.

# 3つの値をソートする場合の決定木の例




オレンジはソート完了した状態を表す.

# 比較に基づいたソートの下界

$n$個の要素をソートする場合，取りうる最終状態（先の図のオレンジのノード）の数は，$n!$個あるはず．

これを全部カバーできないと，ちゃんと正しくソートできない場合が存在してしまう．

高さ$h$の二分木の葉ノードの数は高々$2^h$．

従って，全ての取りうる最終状態をカバーするためには，少なくとも$2^h \geq n!$を満たすような$h$でないといけない．

少なくともこの$h$の数だけ比較が必要，ということ．

# 比較に基づいたソートの下界

$2^h \geq n!$の両辺で対数をとって，$h \geq \log(n!)$

スターリングの近似 $\log(n!) \sim n\log n - n$ により，

$$h \geq n\log n - n$$

以上により，比較に基づくソートの場合，$\Omega(n\log n)$となる．

Ω（ラージオメガ）は計算量の下界を表す記号．

# それでもまだ$O(n \log n)$．．．

比較が定義できれば使えるので，今までに紹介したアルゴリズムは広く一般的な場合で使用可能.

例えば，小数点を含む値でもOK.

では，制約をもたせることでもっと速くできない？

# バケットソート（bucket sort）

バケツソート，ビンソートなどとも．

整列したいデータの取りうる値がk種類である前提．

あらかじめk種類の「バケツ」を用意しておく．
与えられた配列をバケツに振り分ける．
振り分け後，整列したい順序でバケツから順番に取り出す．

# バケットソート

例）[3, 2, 1, 1, 3]

1，2，3に対応するバケツを用意.

| 1 | 2 | 3 |
| --- | --- | --- |
| | | |

# バケットソート

例）[3, 2, 1, 1, 3]

先頭の要素から順にバケツに入れていく.

| 1 | 2 | 3 |
|---|---|---|
| | | 3 |

# バケットソート

例）[3, 2, 1, 1, 3]

先頭の要素から順にバケツに入れていく.

| 1 | 2 | 3 |
|---|---|---|
|  | 2 | 3 |

# バケットソート

例）[3, 2, 1, 1, 3]

先頭の要素から順にバケツに入れていく.

| 1 | 2 | 3 |
|---|---|---|
| 1 | 2 | 3 |

# バケットソート

例）[3, 2, 1, 1, 3]

先頭の要素から順にバケツに入れていく．

| 1 | 2 | 3 |
|---|---|---|
| 1, 1 | 2 | 3 |

# バケットソート

例）[3, 2, 1, 1, 3]

先頭の要素から順にバケツに入れていく.

| 1 | 2 | 3 |
|---|---|---|
| 1, 1 | 2 | 3, 3 |

# バケットソート

例）[3, 2, 1, 1, 3]

| 1 | 2 | 3 |
|---|---|---|
| 1, 1 | 2 | 3, 3 |

バケツから所望の順序で取り出す.

[1, 1, 2, 3, 3]

# バケットソート

バケツは可変長リスト（線形リストなど）で実装.

もし不安定でもよければ，各バケツに対応する値の出現回数のみを記録しておき，その情報を基に出力すべき配列を作り出す.

特にこの実装のものを計数ソート（counting sort）とも呼ぶ.

# 計数ソート実装例

```
# 0からmax_valueまでの整数値のみと想定
def countsort(seq, max_value):
        count = [0]*(max_value+1)      # バケツ
        sorted = []        # ソート済み配列

        # 出現回数をカウント
        for i in range(len(seq)):
                count[seq[i]] += 1
```

# 計数ソート実装例

```
def countsort(seq, max_value):
        …
        # 出現回数からソート済み配列を生成
        for i in range(len(count)):
                for j in range(count[i]):
                        sorted.append(i)

        return sorted
```

# バケットソート

先の計数ソートの例では，出てくる要素とバケツに付随する値（キー）が一致しているが，そうでなくてもよい．

例えば，a -> 1，b -> 2などでもよい．

# バケットソートの計算量

配列の長さが$n$，出てくる可能性のある値全ての種類の数（バケツの数）を$k$とすると，

バケツの準備：一般的には$O(k)$.

バケツに入れる：$O(n)$.

バケツから取り出す：$\max(O(n), O(k))$.

よって，$O(n+k)$.

$k$が$O(n)$かそれ以下のオーダーならば，$O(n)$.

# バケットソートの計算量

メリット

速い！$O(n)$. 😊

デメリット

強い制約が存在（整列したいデータの取りうる種類が予めわかっている）.

取りうる値の種類が多い場合，空間計算量的には損することも.

# まとめ

$O(n^2)$のアルゴリズム

稳定，原地

（二分）挿入ソート，バブルソート，シェーカーソート

$O(n^{1.25})$～$O(n^{1.5})$のアルゴリズム

シェルソート

$O(n \log n)$のアルゴリズム

ツリーソート，クイックソート，マージソート，
ヒープソート

$O(n)$のアルゴリズム

バケットソート（使える条件に注意！）

# 実行時間比較例

| [msec] | 50 | 100 | $10^3$ | $10^4$ | $10^5$ | $10^6$ |
|---|---|---|---|---|---|---|
| 挿入ソート | 0.09 | 0.52 | 20.7 | 1833 | — | — |
| 二分挿入ソート | 0.16 | 0.41 | 13.4 | 996 | — | — |
| バブルソート | 0.25 | 1.06 | 41.3 | 3813 | — | — |
| シェーカーソート | 0.20 | 0.87 | 35.2 | 3234 | — | — |
| シェルソート | 0.08 | 0.21 | 3.47 | 27.5 | 427 | 7570 |
| クイックソート | 0.11 | 0.20 | 2.20 | 13.3 | 130 | 1649 |
| マージソート | 0.16 | 0.37 | 3.64 | 21.5 | 235 | 2842 |
| ツリーソート | 0.17 | 0.42 | 4.63 | 31.6 | 365 | 6567 |
| 計数ソート | 0.04 | 0.08 | 0.52 | 3.26 | 17.5 | 199 |

（ランダムな整数の配列で10回試行した平均，実装はスライドで紹介したもの，
二分挿入ソートはforループへの置き換えを行なっていない．）

# まとめ

今日ご紹介したもの以外にもいろんなソートアルゴリズムがあります.

それらがどんなものかを調べ，どうしてその計算量なのか考えてみるといい勉強になります.

現実に実装されているソート関数にどんな工夫がされているかを見るのも面白いと思います.

# コードチャレンジ：基本課題#5-a [1点]

スライドを参考にしながら，シェーカーソートを
自分で実装してください．

sort関数等を使うことや，他のソートアルゴリズムを
利用することは認めません．

# コードチャレンジ：基本課題#5-b [1.5点]

スライドで紹介したクイックソートを，自分で実装してください．スライドで紹介したように，与えられた配列のメモリ領域を直接使う実装をしてください．

左右に振り分けるときに新しくlist等を生成しないやり方を実装してください．

sort関数等を使うことや，他のソートアルゴリズムを利用することは認めません．

# コードチャレンジ：Extra課題#5 [3点]

ソートを利用する問題.

こちらはsort関数等を使ってもらって構いません.