Algorithms (2021 Summer)
#6：整列（ソート）

矢谷 浩司

前回出た質問

「（KMP法において）何文字目というのとindexがいまいちよくわからなかったです．」

→スライドの表現を変更してみました．ぜひもう一度ご確認ください．

前回出た質問

「（KMP法において）スキップテーブルの値は固定ではなく、位置によって変動するという認識で大丈夫ですか？」

→スキップテーブルの値は一度計算したあとは,
パターン文字列が変わらない限りは固定です.
その各々の値はパターン文字列中の文字の出現の仕方に依存します.

前回出た質問

「（KMP法はパターンのインデックスごとのジャンプ方法の情報を持ったスキップテーブルを用いて前方から探索するアルゴリズム、BM法はパターンによる文字ごとのジャンプ方法の情報を持ったスキップテーブルを用いて後方から探索するアルゴリズムだと考えています。そのため実際そこまで大きく違うものとは考えていないのですが、なにが違ってBM法でだけ無限ループに陥るのでしょうか？」

→スキップテーブルだけに依存する方法では，検索対象側のカーソルが右に動いたように見えても，パターン文字列の頭の位置を見ると後戻りしてしまうことがあるためです．

前回出た質問

「（KMP法はパターンのインデックスごとのジャンプ方法の情報を持ったスキップテーブルを用いて前方から探索するアルゴリズム、BM法はパターンによる文字ごとのジャンプ方法の情報を持ったスキップテーブルを用いて後方から探索するアルゴリズムだと考えています。そのため実際そこまで大きく違うものとは考えていないのですが、なにが違ってBM法でだけ無限ループに陥るのでしょうか？」

→BM法全体の実装としてはこれが起きないようになっているので，無限ループに陥るようなことはありません.

（TAの鈴木くんからのコメントも是非参考に．）

前回出た質問

「ハッシュが合っていても文字列が一致しない場合はどのような場合でしょうか」

→剰余を取った際にたまたま同じ値になることがありえる，ということです.

前回出た質問

「どのような文字列探索法を用いるかは、文字列のパターンがどのような性質を持っているかを考慮して決めるのでしょうか？」

→もしその性質ががわかっている場合には，そのようにすることで，より有利に処理をすすめることは可能です.

前回出た質問

「日本語の文字列を探索する場合と、英語の文字列を探索する場合で、実装の難しさに違いはありますか？」

→アルゴリズム自体は紹介したものと同じものが使えますが，日本語特有の問題として，

- 文字の種類が多い（ひらがな，カタカナ，漢字），
- 文字コードを気にしないといけないことがある，

などがあり，そのあたりで実際の実装に苦労することはありえます.

前回出た質問

「KMP法においては、スキップテーブルの要素の値が対応するインデックスより小さくなる照合パターンほど、たくさんスキップできるので効率的に探せるという認識で合っていますか？」

→TAの鈴木さんより.

たしかにスキップテーブルの値が小さい方が照合パターンは大きく進みますが，照合を行うカーソルの位置は結局1つずつしか進まないので計算量のオーダーに違いは出ません.

前回出た質問

「基本課題aのテストケース6-10がうまくいかないのですが、どういった原因が考えられるでしょうか？」

→TAの鈴木さんより.

例えばスキップテーブルの構築に$\Theta(l^2)$かかっている場合はそこでタイムアウトする可能性があります($O(l)$でできます). スキップテーブルの構築が終わった時点でreturnしてみて実行時間を確認してみると良いと思います.

Θ：計算量の上界と下界が一致する時の記号（ラージシータ）

前回出た質問

「基本課題aのテストケース6-10がうまくいかないのですが、どういった原因が考えられるでしょうか？」

→パターン文字同士をずらしてスキップテーブルの値を埋めていく際，マッチしたらそこまでに何文字マッチしてきたかの情報を使って，値を埋めることができます．これによってパターン文字同士をずらしていくのを後戻りせずにできるため，$O(l)$で済みます.

前回出た質問

「基本課題bの大きいケースで2.3sほどオーバーしてしまいます。計算量は$O(l+n)$になっていると思うのですが、どう工夫をしたらはやくなるでしょうか。」

→どこかで剰余を取り切れておらず，非常に大きな値の掛け算が発生してしまっていることがあると，そこが足を引っ張ることになります．あるいは，毎回のハッシュの計算でa^iを1から計算していると遅くなります．

前回出た質問

「KMP法のスキップテーブルですが、より良く改良出来ると思います。照合の失敗した位置では文字の候補が一つ減るわけですからそれを踏まえるとより高速化できる可能性があると感じました。例えばスライドのABABDの例ではインデックス1234に対して0012となるテーブルでしたが、これは0002とできるはずです。インデックス3で間違えた場合、照合する文字列のその部分がBではないということが分かっており、インデックス1に対応させるのは悪手な気がします。」

前回出た質問

「KMP法のスキップテーブルですが、より良く改良出来ると思います。」

→KMP法の良いところはスキップテーブルを$O(l)$で作ることができる点にあります.

上記の改良はこの例では成立し，確かに定数倍の改善にはなりますが，一般的な場合に対して正しいスキップテーブルを$O(l)$で作れるかどうかを確認しないとなんとも言えなさそうではあります.

前回出た質問

BM法のスキップテーブルの作成方法について不正確な記述がありましたので，訂正いたしました．正しくは，

照合パターンの長さをlとして，

- パターンに含まれていない文字に対しては，移動量l.
- パターンに含まれている文字に対して，
  - 末尾にしかその文字が現れない場合は，移動量l.
  - 末尾に最も近い出現位置がi $(0 \leq i < l - 1)$ならば，移動量$l - i - 1$.
    - 末尾に加えて，それ以外の場所でも出現する場合はこちらを採用.

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]

（以下の説明では上の配列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

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以上，つまり$O(n)$以上の場合，外部ソートと呼ぶ．（ディスク等の外部の記憶領域が必要になるようなイメージ）

一方，$O(1)$や$O(\log n)$の一時的な記憶領域が必要なものは内部ソートと呼ぶ．（なんとなく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考えられることが多く，あまり重要視されない．

安定 vs. 安定でない

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

複数の値でソートしたい（例えば，IDでソートして，さらに成績順にソートする）場合には，2つ目で使うソートが安定でないとうまく行かない．

ただし，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．

安定性も昨今ではあまり重要視されない．

超非効率ソート: 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.

超非効率ソート: ボゴソート

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場合に遭遇するため総試行回数の期待値は$n!$.

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.

さらに，シャッフル自体の操作が$O(n)$.

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.

超非効率ソート: ボゴソート

よって，$O(n!\,n)$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．

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，のいい
エクササイズですね． ☺️

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.

そのときに，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
```

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？

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+1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,

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

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.

安定的アルゴリズムとして実装できる.

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]

後ろから前にバブルソート.
[3, 5, 2, 1, 6, 4] -> … -> [1, 3, 5, 2, 4, 6]

一番最後のスワップはindex：1の場所（3）．

シェーカーソート例

次は，前から後ろにバブルソート．
[1, 3, 5, 2, 4, 6] -> [1, 3, 2, 5, 4, 6] -> [1, 3, 2, 4, 5, 6]

一番最後のスワップはindex：3の場所（4）．

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,

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

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

などがある．ただし，$h_i \leq n$.

また，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.

$h_i = \frac{3^i - 1}{2}$の場合，$h_i < \frac{n}{3}$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．

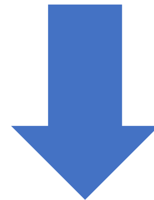
5	2	6	10	1	7	3	4	8	9

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

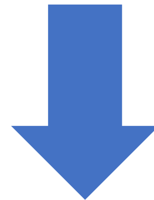
次に，$h_i = 3h_{i-1} + 1$に従って間隔を狭める．今回の場合は1になるので，単純に挿入ソートをする．

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	6	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，ナイーブな挿入ソートよりも時間がかからない．

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，最悪でも$O(n^{1.5})$,

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

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.

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つ値を選ぶ（枢軸とよぶ）．

枢軸の選び方にはいろいろあるが，とりあえずは今与えられている配列の真ん中に位置する要素とする．

クイックソートの考え方

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！

クイックソート例

[7, 8, 4, 5, 6, 2, 3, 1]　　　# 5を枢軸にする

クイックソート例

端からスタート.

[7, 8, 4, 5, 6, 2, 3, 1]

クイックソート例

7は枢軸5よりも大きく，1は枢軸5よりも小さい．

[7, 8, 4, 5, 6, 2, 3, 1]

よって，この2つを入れ替える．

[1, 8, 4, 5, 6, 2, 3, 7]

クイックソート例

[1, 8, 4, 5, 6, 2, 3, 7]

左右のカーソルを1つ進める.

[1, 8, 4, 5, 6, 2, 3, 7]

クイックソート例

8は枢軸5よりも大きく，3は枢軸5よりも小さい．

[1, 8, 4, 5, 6, 2, 3, 7]

よって，この2つを入れ替える．

[1, 3, 4, 5, 6, 2, 8, 7]

クイックソート例

[1, 3, 4, 5, 6, 2, 8, 7]

左右のカーソルを1つ進める.

[1, 3, 4, 5, 6, 2, 8, 7]

クイックソート例

2は枢軸5より小さいのでスワップの候補になる．しかし，
4も枢軸5より小さいので，これはスワップしたくない．

[1, 3, 4, 5, 6, 2, 8, 7]

クイックソート例

そこで左側のみカーソルを1つ進める.

[1, 3, 4, 5, 6, 2, 8, 7]

5は枢軸と同じ（これは枢軸より大きいとみなす）なので,
スワップを行う.

[1, 3, 4, 2, 6, 5, 8, 7]

クイックソート例

左右のカーソルを1つ進めると，2つのカーソルが交差することになる．これで，この段階は終了．

[1, 3, 4, 2, 6, 5, 8, 7]

この時点で，カーソルが交差した場所の左側には枢軸より小さいもの，右側には枢軸より大きいものが集まっている（ただし，ソートは必ずしもされていない）．

クイックソート例

[1, 3, 4, 2]と[6, 5, 8, 7]に分割！

次のステップでは，この分割したものに対してクイックソートを個別に行う．

つまり，再帰を使って実装できる．

これを繰り返していくと最終的には，要素1つだけの配列になり，自動的にソートされたことになる．

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
def qsort(seq (配列), left (左端index), right (右端index)):
    [再帰の終了条件は？]

    pivot = [与えられた範囲の真ん中に位置する要素]
    # カーソルを用意
    cur_l = left
    cur_r = right
```

クイックソートの実装例

```
def qsort(seq, left, right):
    …
    while True:
        [左カーソル（cur_l）をpivotより大きい要素
        まで進める]
        [右カーソル（cur_r）をpivotより小さい要素
        まで進める]
```

クイックソートの実装例

```
def qsort(seq, left, right):
    …
    while True:
        …
        # カーソル交差でループ終了
        if (cur_r <= cur_l): break
```

クイックソートの実装例

```
def qsort(seq, left, right):
    …
    while True:
        …
        # cur_l < cur_rだが，左側にpivotより大きい要素，
        # 右側にpivotより小さい要素がそれぞれ見つかった．
        [cur_rの要素とcur_lの要素を入れ替える]
```

クイックソートの実装例

```
def qsort(seq (配列), left (左端index), right (右端index)):
    …
    while True:
        …
    # このwhileループを抜けるということは，左右の
    # 要素の入れ替えが完了したこと．
```

クイックソートの実装例

```
def qsort(seq (配列), left (左端index), right (右端index)):
    …
    while True:
        …
    # 分割したグループそれぞれに対して，qsortを
    実行したい！ -> 再帰
    [カーソルの交差位置の左側だけでqsort]
    [カーソルの交差位置の右側だけでqsort]
    # どんな引数を与えれば良い？
```

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

一般的には，$O(n \log n)$．よってかなり効率的！

枢軸の選び方によっては$O(n^2)$になる（グルーピングがどちらか一方に完全に偏る場合）が，一般的にはそう頻発しない．

安定的ではない（要素を飛び越えてスワップするため）．

クイックソートの計算量

$O(n \log n)$の前提は，与えられる配列要素の並び方は
$1/n!$で等確率である．

実際のデータにはある種の偏りがあることも．

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．（乱択化）

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

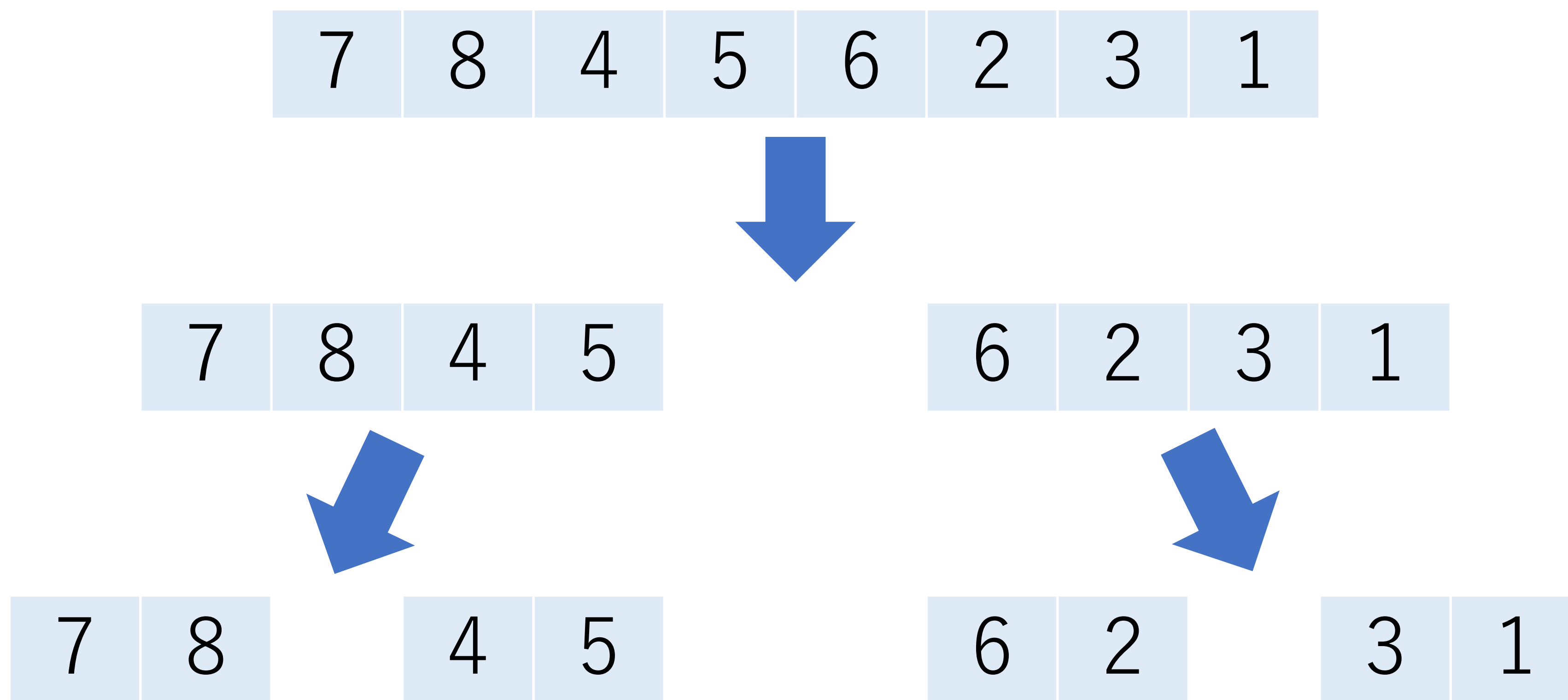
マージソート例

7	8	4	5	6	2	3	1

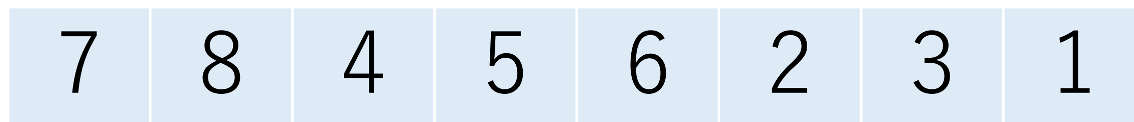
7	8	4	5

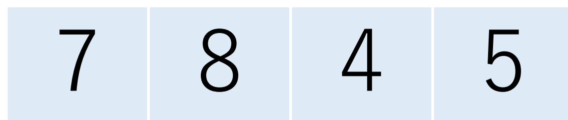
6	2	3	1

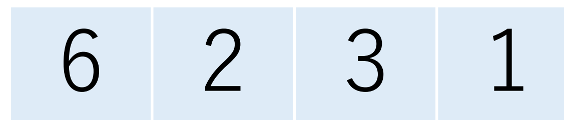
マージソート例

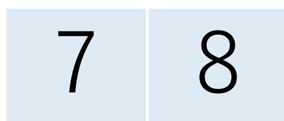


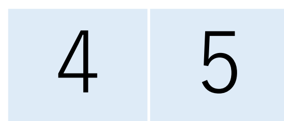
マージソート例

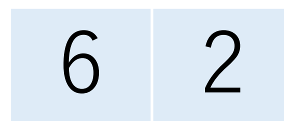


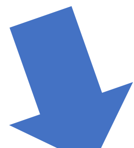


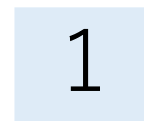




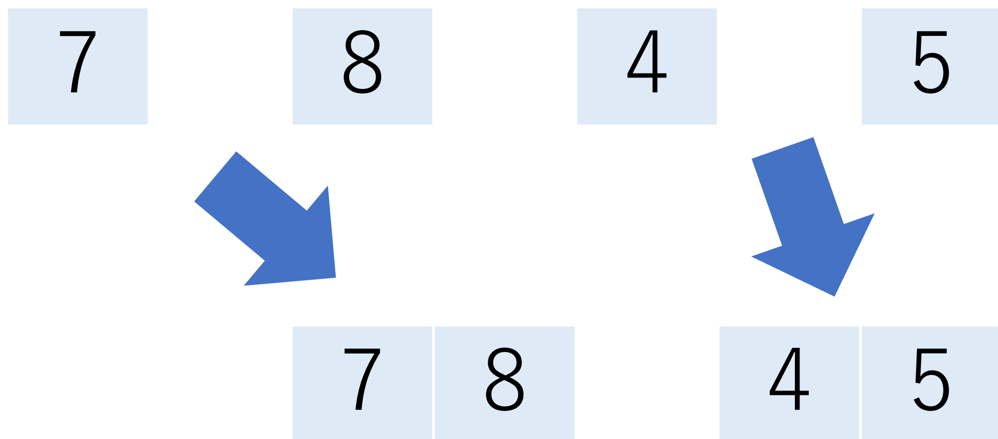


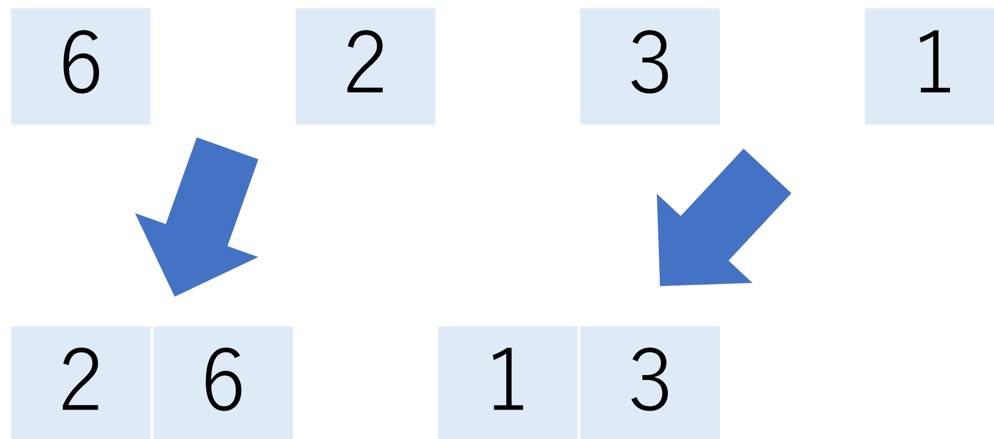




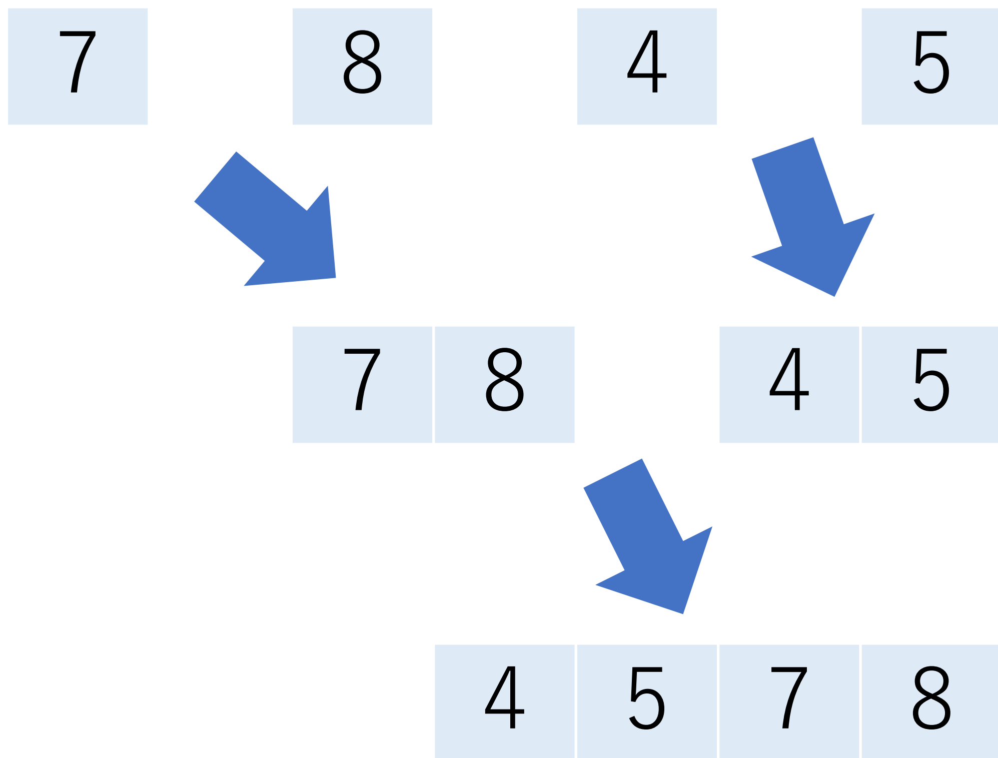


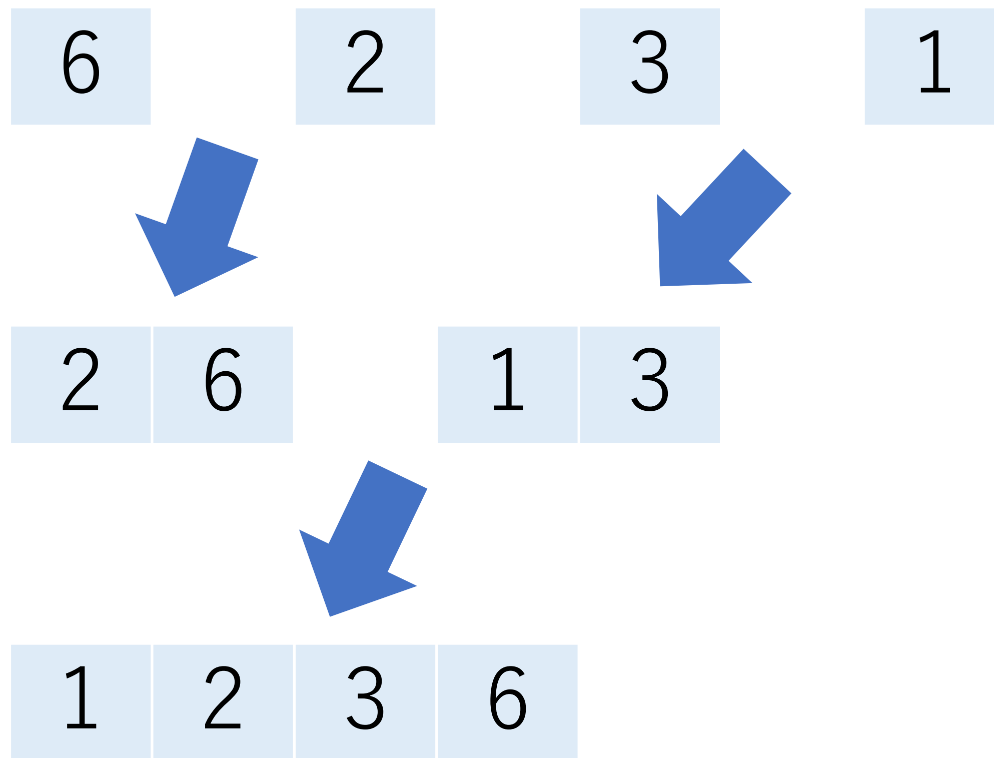
マージソート例



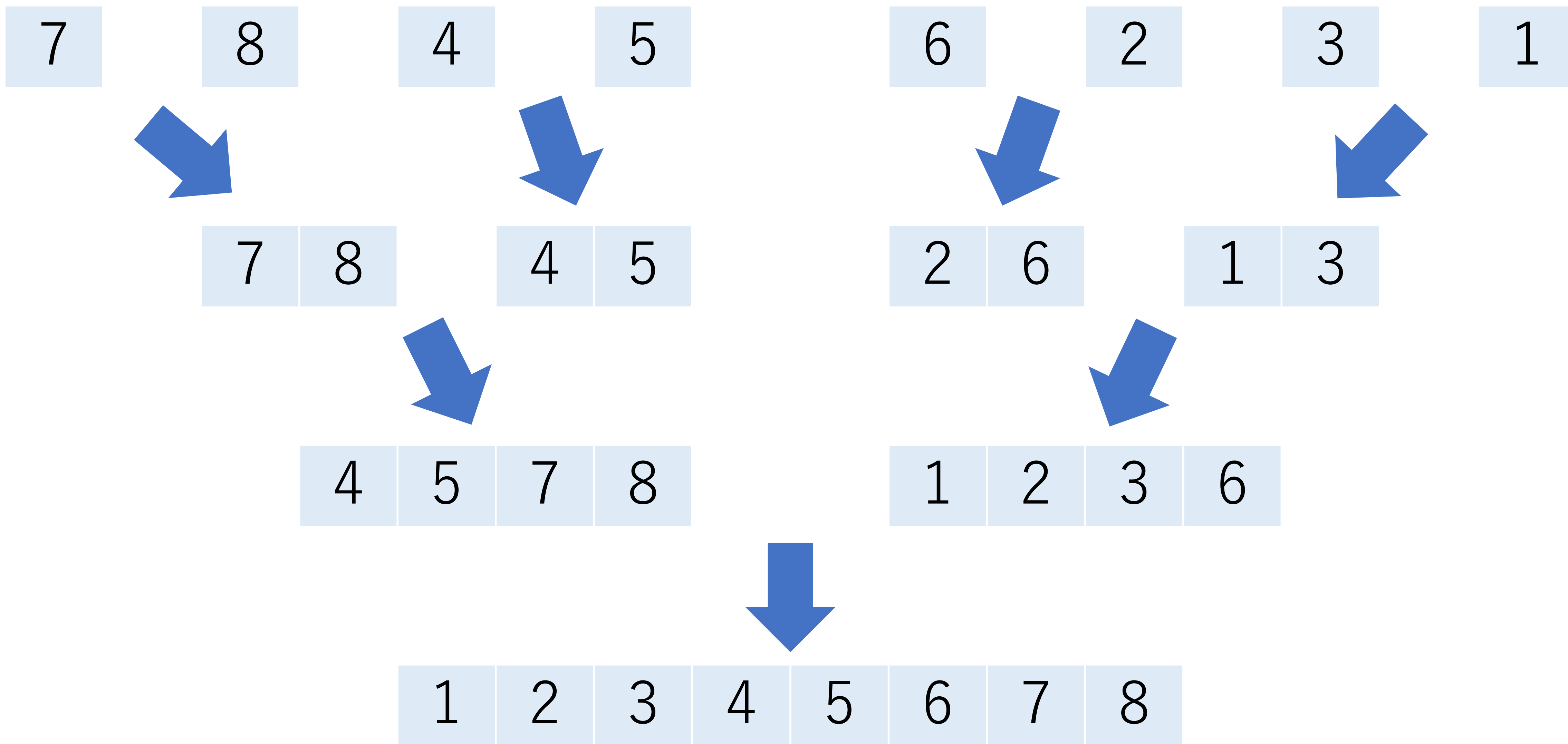


マージソート例





マージソート例



マージソートの実装例

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

    # 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

[長さnの配列のソート]

= 2 * [長さn/2の配列のソート] + [n個の要素のマージ]

大雑把に議論すると，長さが1の配列から元の長さにたどり着くまでには$\log n$段階存在する．各段階におけるマージ操作は全部合わせて$O(n)$.

よって，$O(n \log n)$.

マージソートの計算量

最悪の場合での計算量も$O(n \log n)$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

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

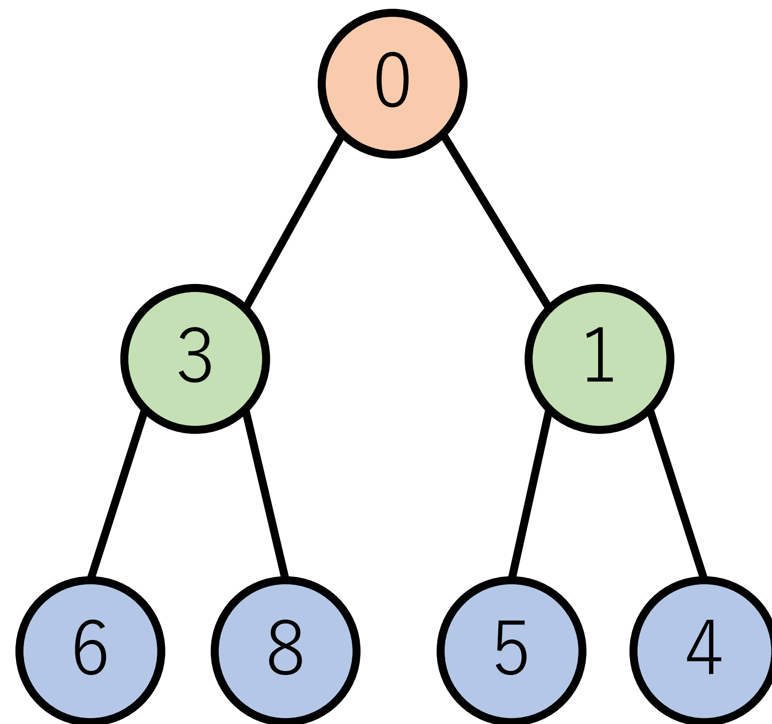
<u>データ構造で工夫する</u>

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.

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$O(n \log n)$．．．

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

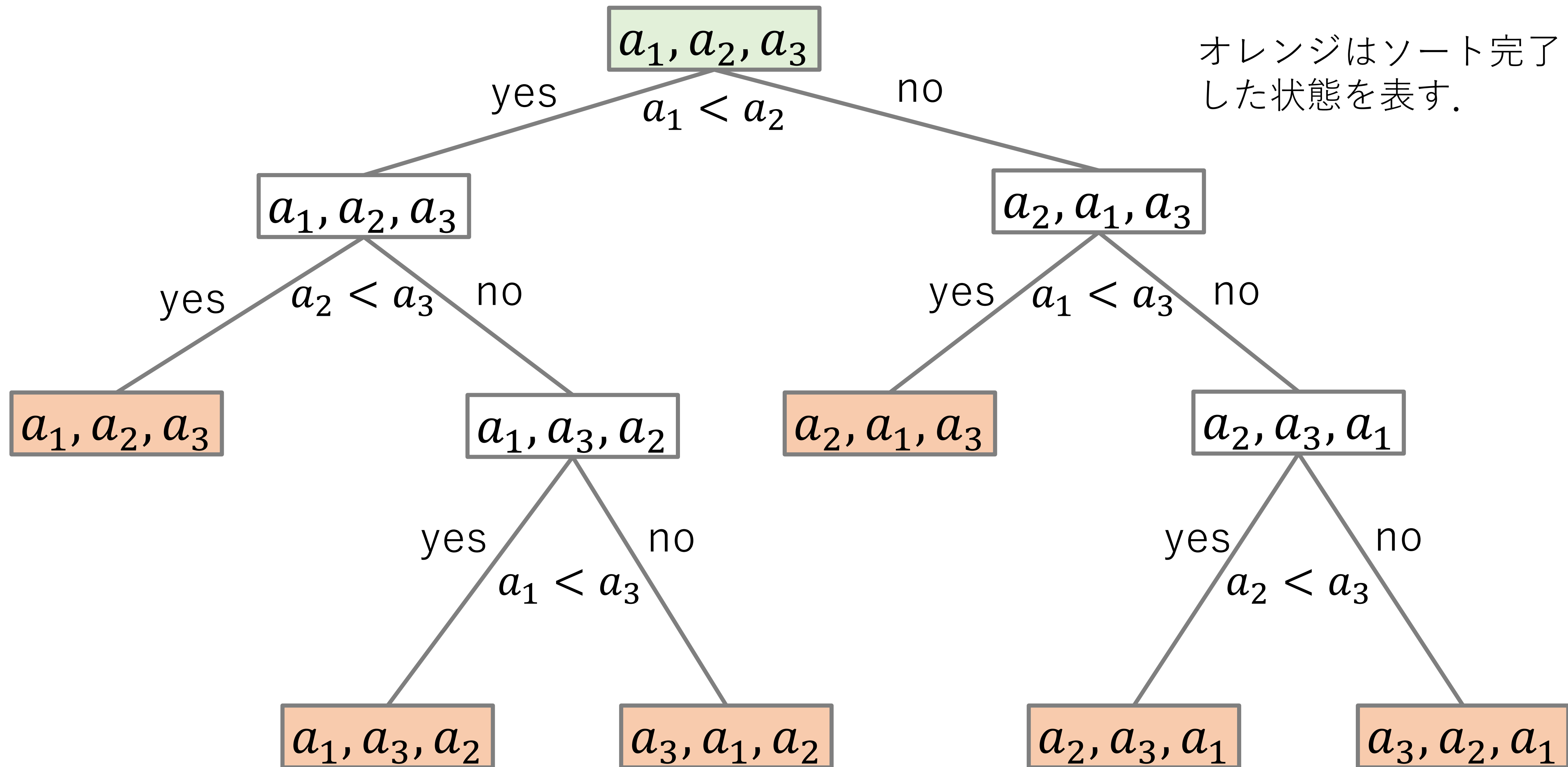
比較・交換を順次行うことで得られるソートの全ての起こりうる手順は決定木で表すことができる.

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



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

（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

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	0.36	39.8	4240	―	―
二分挿入ソート	0.11	0.37	31.9	3196	―	―
バブルソート	0.19	0.75	91.3	9415	―	―
シェーカーソート	0.16	0.61	74.8	7879	―	―
シェルソート	0.06	0.15	3.47	61.6	1079	―
クイックソート	0.05	0.12	1.67	22.0	266	3395
マージソート	0.11	0.26	3.41	45.4	552	7063
計数ソート	0.02	0.05	0.42	4.51	45.8	603

（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，どうしてその計算量なのか考えてみるといい勉強になるかも.

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かも.

コードチャレンジ：基本課題#6-a [1.5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#6-b [1.5点]

スライドを参考にしながら，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#6 [3点]

ソートを利用する問題.