制御情報システム工学科4年
選択科目：プログラミング特論

プログラミング特論

第４章　ソートを改良する

- ソート（整列）
  - 現代の標準的なプログラミング言語では，
    基本的な機能として組み込み済み．
    ⇒　自分で実装する機会はほぼなし．

- ソートアルゴリズムの研究
  - 1940年代から続き，今日までにさまざまな方法が開発．

- **アルゴリズムの基本的な考え方を知るための**
  とても良い教材．

4.1　問題を分割する

- 実社会においても，複雑に入り組んだ大きな問題を解決するのは困難．
- 問題を切り分けてサイズを小さくすると，解決の糸口が見えてくることも多い．

  配列のソートを例に，問題を分割することの有用性を理解．
  - アルゴリズムでも同じことがいえるため．

- 4.1.1　小さなソート
- 4.1.2　ソートされた配列のマージ
- 4.1.3　マージの計算量
- 4.1.4　小さな結果をまとめる
- 4.1.5　マージソートの計算量
- 4.1.6　ソートの計算量
- 4.1.7　分割統治法
- 4.1.8　再起を使ったマージソートの実装

4.1.1　小さなソート

- 小さなソートの問題（例：2つの数71と42のソート）
  - 42のほうが前，71が後ろ．

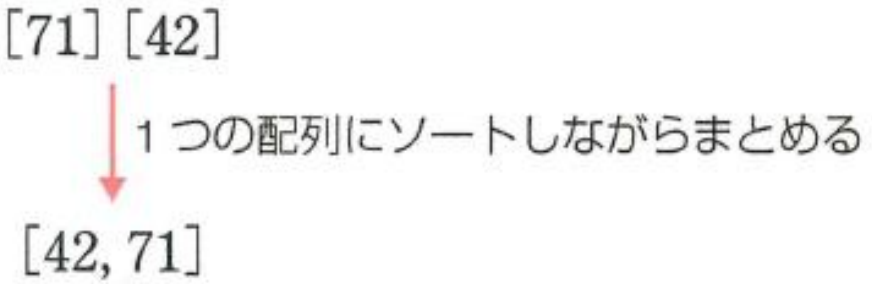

図 4.1　要素が 1 つの配列をソートしながらまとめる

- 並べ替える前の数それぞれ（例：71と42）を，要素が1つしかない配列だと認識．
  - これらはそれぞれ，ソート済み．
  - 要素が1つしかない配列なので，そのままでソート済み．
- 2つのソートされた配列を，1つのソートされた配列にまとめるという作業を考える．
  - 要素が1つの場合，数の並べ替えと同じなので簡単．

- それぞれの配列に要素が2つある場合

問題 4.1

2 つのソートされた配列をまとめて，1 つのソートされた配列を作る手順を考えよ．

- マージ(merge，「統合する」)

4.1.2 ソートされた配列のマージ

- 具体的な例を見ながら，マージの動きを確認．
- 図4.2）2つのソートされた配列 [42，71]，[39，70]を
  マージする手順

- **Step1**
  - ポインタは，それぞれの配列の先頭を指す．
  - まずこれらの要素を比較し，小さいほう(39)を新たな配列の先頭に置く．
  - (39)は出力用の配列に移ったので，赤ポインタを1つ進める．
- **Step2，3**
  - この作業の繰り返し．
- **Step4**
  - 赤ポインタが配列の右端から外れ，参照する要素がなくなる．
  - もう1つの青ポインタが指す数字が返される．

- 一連の手続きが済むと，2つの配列がマージされて1つのソートされた配列が完成．

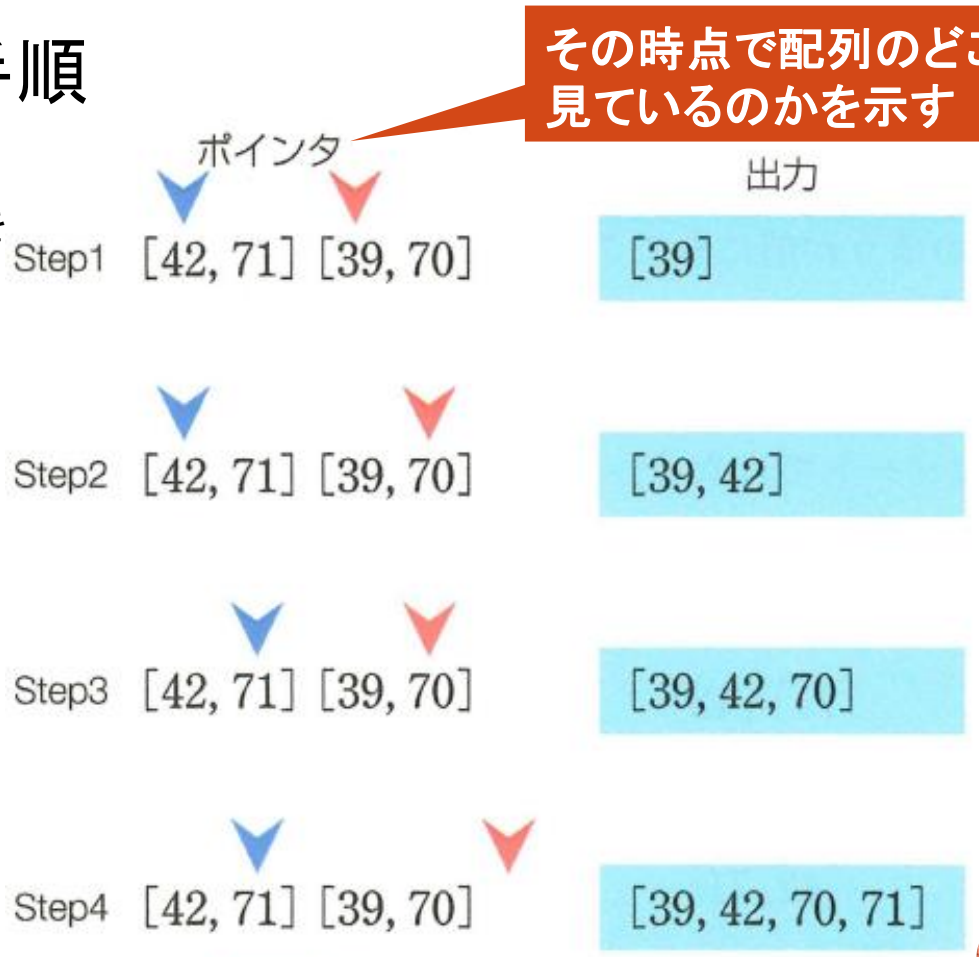

図 4.2 ソートされた配列のマージ

- 前スライドの手順
  - 入力の配列がそれぞれソートされていれば，
    長さに関係なし．

- 重要な点
  - ポインタがある場所の数値を比較するだけで良い．
  - 入力となる2つの配列が，
    「あらかじめソートされている」という
    条件があって成立．

4.1.3 マージの計算量

- **マージの作業に必要な計算量はどのくらい？**

- 「数の比較を何回するか」で考える.
- マージの各ステップ
  - それぞれのポインタが指す要素が比較され，
    小さいほうを指していたポインタが必ず次の要素へ移動.
- 2つの入力配列の長さをaとbすると，
  比較は，最悪の場合$a + b - 1$回
- できあがる配列の長さを$n = a + b$とすると，
  必要な計算量は$O(n)$
  - 2つのすでにソートされた配列をもとにして，
    1つのソートされた配列を作るのにかかる計算量.

4.1.4　小さな結果をまとめる

- マージの実装
  - *入力される配列が1つでも動作するように，*
    2つ目の引数のデフォルト値を空の配列にする．

- 2つのソートされた
  配列をまとめ上げる．

コード 4.1　2 つのソートされた配列を統合

```
1  def merge_arrays(left, right=[]):
2      res = []
3      i, j = 0, 0
4      n, m = len(left), len(right)
5      # どちらかの配列を調べ尽くしたらそこで終了
6      while i < n and j < m:
7          if left[i] < right[j]:
8              res.append(left[i])
9              i += 1
10         else:
11             res.append(right[j])
12             j += 1
13     # 残りはそのまま後ろに連結する
14     return res + left[i:] + right[j:]
```

- コード4.1（関数merge_arrays）を使い，
  与えられた配列をソートしたい！
- 条件）マージの入力は，それぞれソート済み．
  - 要素が1つの配列は，常にソートされていると考えられる．

- 与えられた配列を細かく分割していき，
  1つの数にすれば，これを2つセットにして
  マージへの入力とすることが可能．

- 配列を分割する単純な方法
  - 要素を先頭から2つずつ取り，それを入力とする.
  - 元の配列の長さが奇数の場合，余りが出る.
    - merge_arraysは引数が1つでも動作するので問題なし.

コード 4.2　配列を分割してマージへ渡す関数

```
1  def step(array):
2      res = []
3      for i in range(0, len(array), 2):
4          # 長さ 2もしくは 1の配列がスライスの結果を返る
5          res.append(merge_arrays(*array[i:i+2]))
6      return res
```

※何を手動で入力しなければならないかに注意.

課題1：コード4.2の関数の動きを確認（提出不要）

- 教科書P.058,59の例を参照し，関数の動きを確認.
- 長さ15の配列を生成し，すべての要素を長さ1の配列に変更したあと，関数stepを適用.
- 実行例

```
import random
random.seed(4)
my_array = [random.randint(0, 100) for i in range(15)]
my_array = [[v] for v in my_array]
step1 = step(my_array)
step1
```

```
[[30, 38], [13, 92], [50, 61], [11, 19], [2, 8], [51, 70], [37, 97], [7]]
```

- 関数stepを再適用

```
step2 = step(step1)
step2
```

```
[[13, 30, 38, 92], [11, 19, 50, 61], [2, 8, 51, 70], [7, 37, 97]]
```

- stepを再々適用

```
step3 = step(step2)
step4 = step(step3)
step4
```

```
[[2, 7, 8, 11, 13, 19, 30, 37, 38, 50, 51, 61, 70, 92, 97]]
```

- 手動で行った操作を関数化することで，
  マージソート（merge sort）を完成.

コード 4.3　マージソート

```
1  def merge_sort(array):
2      # すべての数をリストに変換する
3      res = [[v] for v in array]
4      while len(res[0]) != len(array):
5          res = step(res)
6      # リストの中にリストが入ってしまうのでこれを取り出す
7      return res[0]
```

- 教科書P.059の例を参照し，
  マージソートの結果を確認すること！

- 実行例
  - 新たにデータを作り，関数を適用

```
my_array = [random.randint(0, 100) for i in range(15)]
my_array
```

```
[11, 70, 38, 0, 37, 73, 90, 39, 97, 65, 24, 52, 54, 76, 36]
```

```
merge_sort(my_array)
```

```
[0, 11, 24, 36, 37, 38, 39, 52, 54, 65, 70, 73, 76, 90, 97]
```

4.1.5 マージソートの計算量

- 図4.3）4つの数が1つのソートされた配列へマージされる様子
  - 上から下へ計算ステップが進んでいく.
  - 最初はバラバラだった数が，マージによって2つが1つにまとまり，やがて全体が1つに集約.
- 図を下から上へと眺め，木（tree）を想像.
  - 木は上に枝（branch）を伸ばし，葉（leaf）をつける.
  - 根から葉までの経路が一本道.

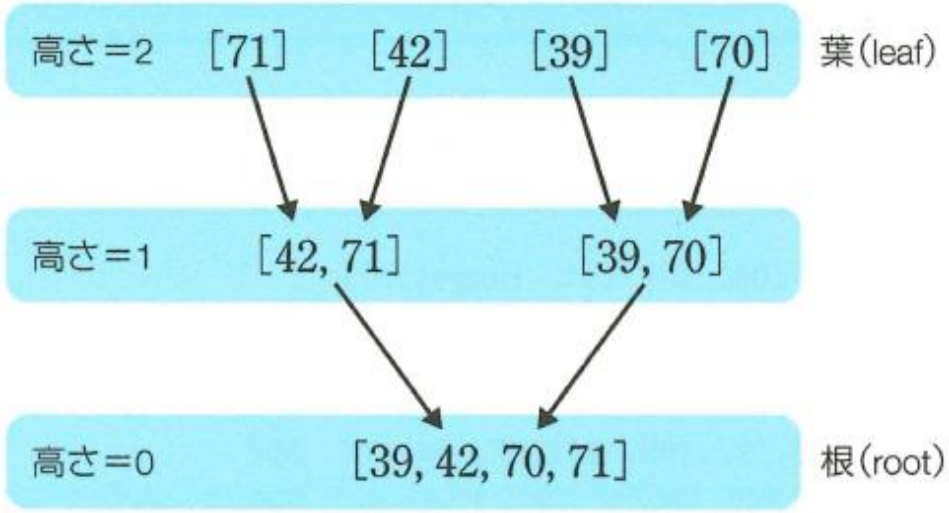

図 4.3　マージの様子を表現した木構造

- 木の高さに注目.
  - この木の高さは2
  - 下から木を見ていくと，1回の枝わかれで2つに分岐.
  - 高さ2の木は，4枚の葉をもつ.

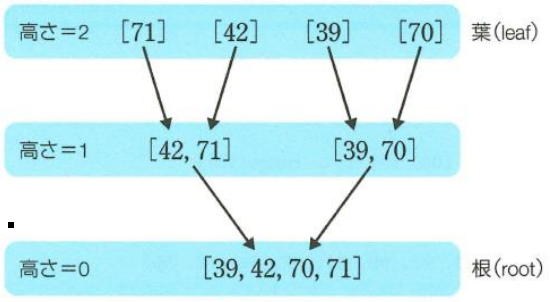

図 4.3　マージの様子を表現した木構造

- 二分木(binary tree，二進木)
  - 1回の枝わかれが高々2になる木
  - 二分木の高さがkだとすると，葉の数は最大で$2k$枚.

- ソートする入力配列のサイズをnとする.
- 1ステップの計算
  - 2つの配列をマージする計算.
- 木の高さkを調整し，$n \leq 2^k$となる木を用意.
- この木の高さは，$k = \log_2 n$
- 木の高さを1つ減らすために必要なマージの回数
  - 葉の枚数の半分となり，これは高々n回.
- マージソートの計算量は，$O(n \log n)$
  - 入力のサイズnに対して，これより速いソートのアルゴリズムはない.

4.1.6 ソートの計算量

- 入力のサイズnに対して，
  $O(n \log n)$より速いソートのアルゴリズムはない．
- 二分木を使って説明．
- 長さnの配列を想定し，各要素をa_iなどと表現．
- ソートの基本的な演算
  - 要素同士の比較と入れ替え．
- $i < j$なる添え字に対して，$a_i > a_j$であれば，
  2つの要素を入れ替え，小さな要素がより先頭に近づける．

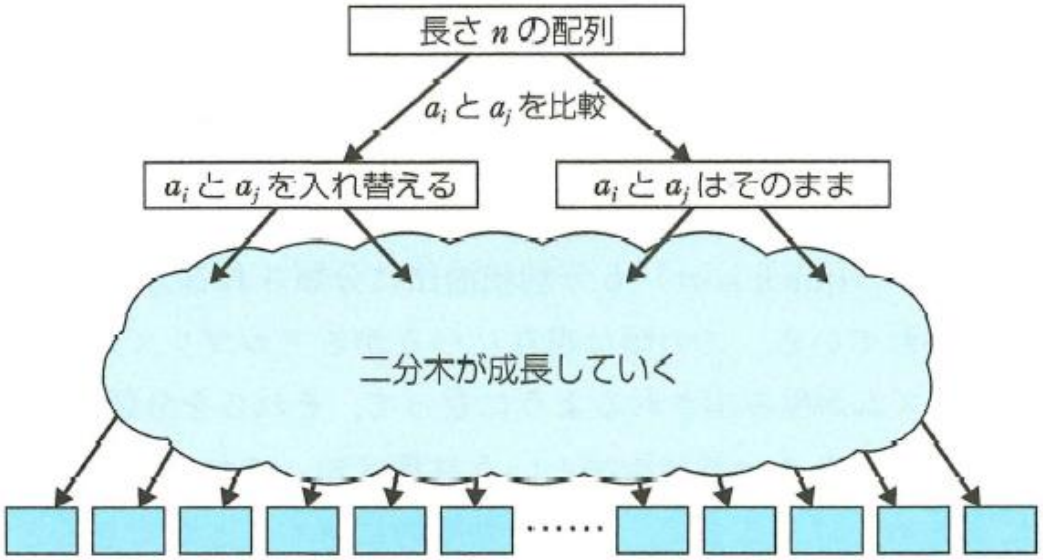

図 4.4　ソートに必要な計算量

- 長さnの配列の要素がすべて違うとすると，

  n個の要素を使って表現できる配列の種類は$n!$通り存在．
- 1回の比較演算で，配列には2つの可能性が発生．
  - 「比較した要素を入れ替えるか」，「そのままにするか」
- これを繰り返していく様子は，二分木として表現可能．
- ソートのアルゴリズムとしては，

  最終的にすべての可能性($n!$通り)に到達すること必要．
- このときの木の高さが，ソートアルゴリズムにどうしても必要な計算量．

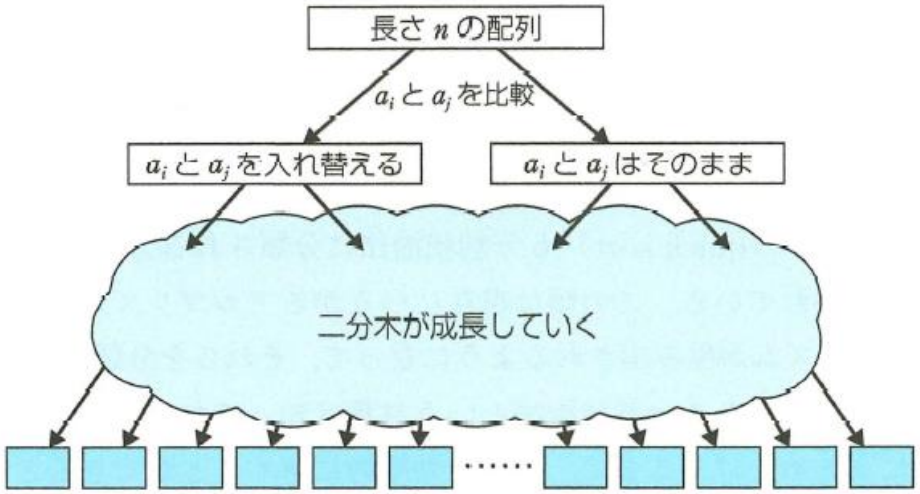

図 4.4　ソートに必要な計算量

- 図4.4の木の高さをkとすると，葉の枚数は$2k$.
- アルゴリズムはすべての可能性を実現できる必要があることから成立する不等式

$$2^k \geq n!$$

- 次式(4.1)：スターリングの近似

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \tag{4.1}$$

- 式(4.1)を使い，$2^k \geq n!$を変形（両辺の対数をとる）

$$\begin{aligned} k &\geq \log_2 n! \\ &\sim \log_2 \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \\ &= n\log_2\sqrt{2\pi n} + n\log_2 n - n\log_2 e \end{aligned} \tag{4.2}$$

- 計算量の漸近記法に従い，式(4.2)から次数が最大のものだけに注目.
- 第2項の$n\log_2 n$を取り出し，底の変換で出る定数係数を無視.

4.1.7　分割統治法

- マージソートの本質
  - 問題を小さく分割し，簡単な問題を解くことを
    積み重ねて全体の解を得る.
- 分割統治法(divide-and-conquer method)
  - クイックソート(quick sort) も分割統治法に分類される方法.
- 1945年ごろに，マージソートは考案.
- 現在につながるアルゴリズム研究の繋明期.
- その後，さまざまなアルゴリズムが生み出され，
  それらを分類する研究も発展.
- 重要なこと
  - 問題を分割して上手にまとめ上げることで，
    問題を効率的に解くことが可能.

【補足】分割統治法

- もっとも基本的なアルゴリズムの設計手法
- 現在知られている多くのアルゴリズムで使用
- 大雑把に言えば．．．
  - 与えられた問題を部分問題に分割して解いた後，その解を再構成して全体の解を得るという手法
- 分割統治法の概念を理解するための直感的な例
  - 図7.1の自動車を製造する場合

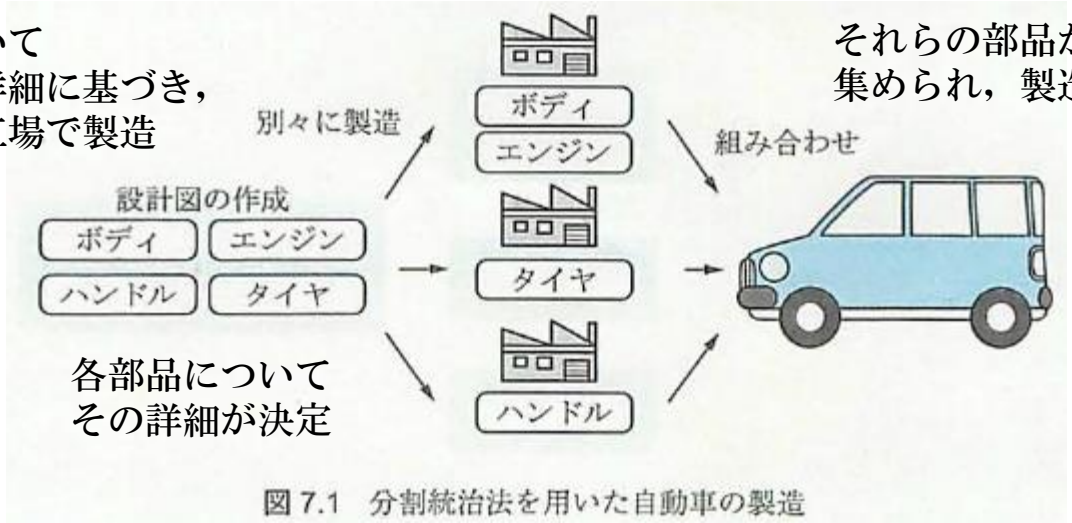

図 7.1　分割統治法を用いた自動車の製造

【補足】分割統治法を用いたアルゴリズムを構成する3つのステップ

- **問題を部分問題に分割することにより，**
  各部分問題を解くことが容易になり，
  問題を効率よく解くことが可能．
- **ステップ1：分割**
  - 問題をいくつかの部分問題に分割．
- **ステップ2：統治**
  - 分割された部分問題を解く．
  - 各部分問題は，再帰的に解かれることが多い．
    - 分割統治法のアルゴリズムは，
      再帰アルゴリズムとして記述されることが多い．
- **ステップ3：組合せ**
  - ステップ2で得られた部分問題の解をもとに，
    いくつかの計算を行い，問題全体の解を得る．

【補足】分割統治法によるマージソートのアイデア

- **分割の処理：最上部（a）**
  - 入力の列をほぼ均等な大きさの2つの列に分割
- **統治の処理：中部（b）**
  - 2つに分割された列をそれぞれ再帰的にソート
  - 分割された2つの列はソート済みの状態
- **組合せの処理：最下部（c）**
  - ソート済みの2つの列を1つのソートされた列に併合（マ-
  - 全体としてソートされた列を得る
    - マージ操作の実現には工夫が必要（詳細については後述）.

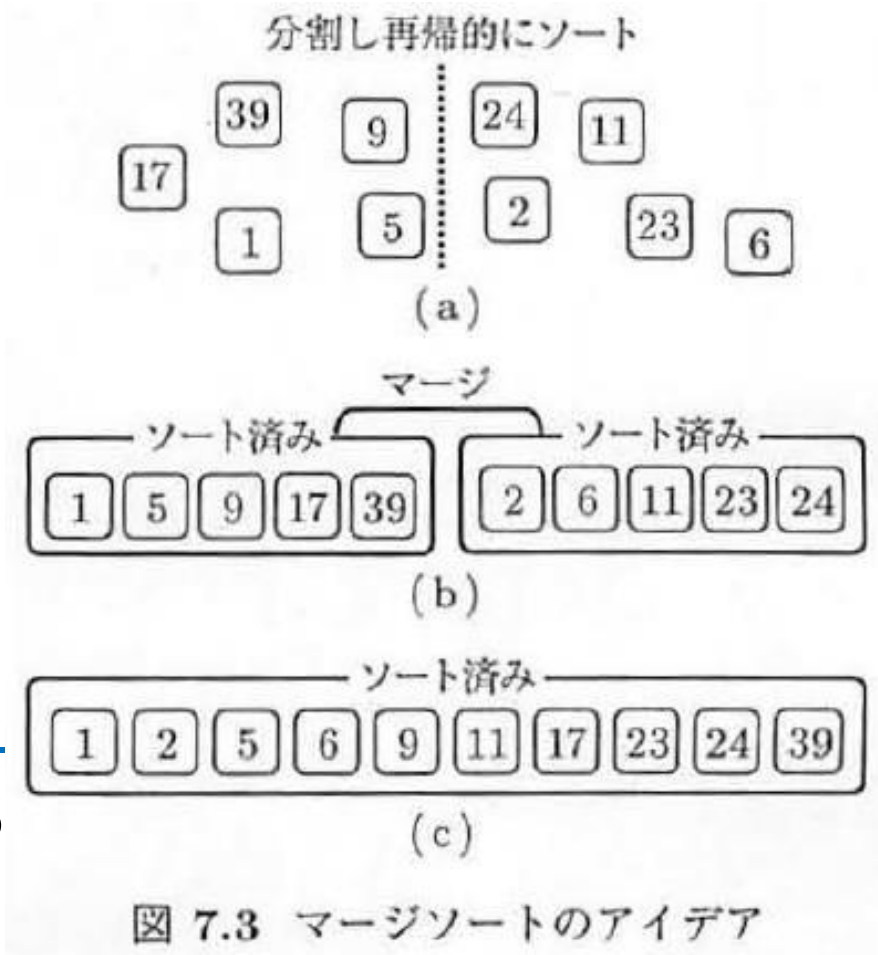

図 7.3 マージソートのアイデア

【補足】マージソート

- 分割統治法を用いた一つの例
  - ソートアルゴリズムの一つ

- クイックソートとの類似点
  - 入力のデータを2つに分割し，再帰的にソートを行う.
- クイックソートとの相違点

  （どのステップに重点を置くのか？）
  - クイックソート:　統治前の分割のステップ
  - マージソート:　　統治後の組合せのステップ

4.1.8　再帰を使ったマージソートの実装

- 一般的なマージソートの実装方法
  - 再帰を使った方法.

- 入力された配列を左右半分に分割し，
  それぞれに再びマージソートを適用.
- 分割の結果，配列の長さ1になったら，
  戻り値にする.

- 前述のコード4.3のエラーを修正
  - 受け取る配列が空のときエラー

課題2

- 再帰を使ったマージソートを実装，
  すなわち，任意の数の正の整数を格納した配列を
  ソートして結果を出力するプログラムを作成しなさい．

- 関数名merge_sort，引数（配列）array，配列の中央を表す添字mid_idx，元の配列の左半分left，元の配列の右半分rightとする．
- 注意
  - 教科書P.062の「コード4.4　マージソート（再帰）（Pythonコード）」を極力みないで作成すること．
  - Pythonがもつソートの関数やメソッドとプログラムの
    速度を比較するため，それぞれの実行時間も出力すること．

クイックソート

- マージソートの計算量： $O(n\log n)$
  - 理論的には，これ以上の改善が望めない．
- クイックソートの時間計算量： $O(n\log n)$

- Pythonでソートをする場合
  - 組み込み関数sortedなどの利用を推奨．

- 本章の目的
  - ×：ソートの速度改善．
  - ○：さまざまなアルゴリズムを学び，
    計算に関する知識を増やす．

- マージソートとクイックソートとの類似点
  - 入力のデータを小さく分割し，
    再帰的に全体をまとめ上げることでソートを行う.
- マージソートとクイックソートとの相違点
  （どのステップに重点を置くのか？）
  - クイックソート:　統治前の分割のステップ
  - マージソート:　　統治後の組合せのステップ

- 入力するデータの性質によって，
  計算時間が大きく変化することがある.
  - このような現象がなぜ起こるのか？
  - それがどのような示唆を与えるのか？

4.2節は，8章と9章への導入になる話題を含む.

クイックソートとは？

- 実用的にはもっとも高速に動作
- 再帰アルゴリズムとして記述.

- データを大まかに2つに分割し，
  分割した集合のデータが1つに
  なるまで繰り返すアルゴリズム

- 基準値（ピボット，pivot）
  - 分割の基準となる値

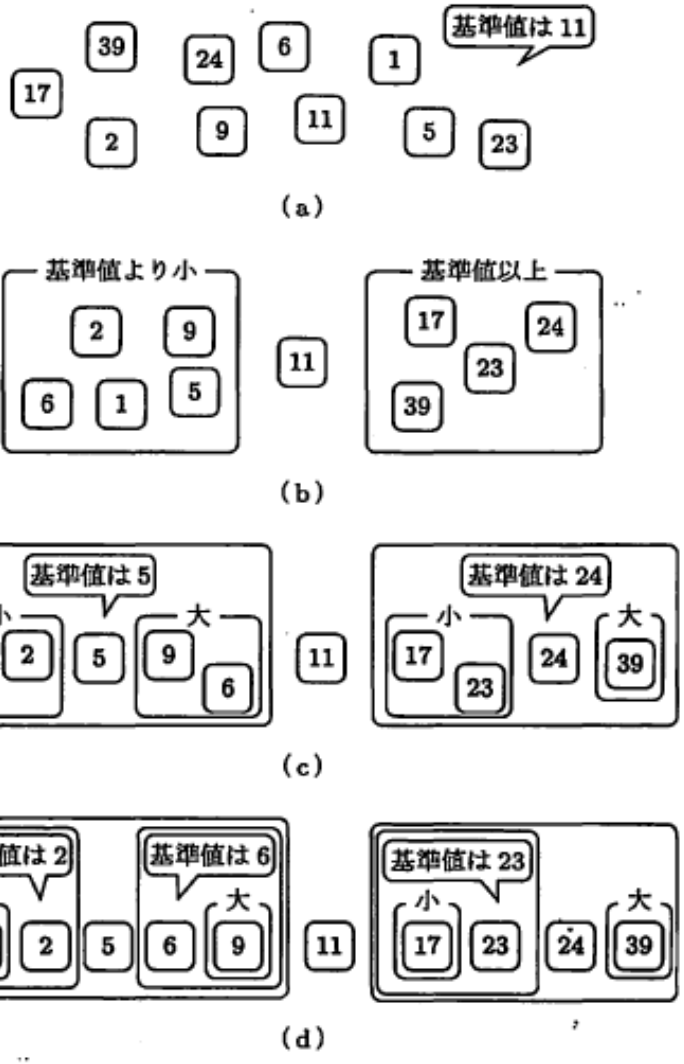

図 6.1　クイックソートのアイデア

クイックソートのアイデア

- 入力データの集合： $D = \{d_0, d_1, \ldots, d_{n-1}\}$
- (1) 集合Dに含まれる要素が1つならば,

  そのまま何もせずにアルゴリズムを終了.
- (2) 集合Dから適当に基準値となるデータd_k を1つ選択.
- (3) 集合Dに含まれる各データと基準値d_kを比較し,

  すべてのデータを,
  - d_kより小さいデータの集合　D_1
  - d_k以上のデータの集合　　D_2

  のいずれかに分割.
- (4) 集合D_1と集合D_2をそれぞれ再帰的にソート.
- (5) 再帰的なソートが済んだら,

  3つの集合D_1, $\{d_k\}$, D_2を

  この順番に連結したものを出力.

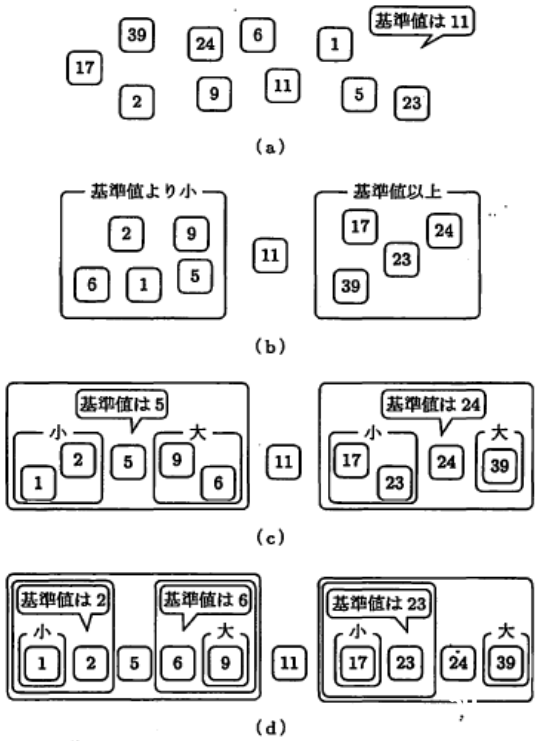

図 6.1　クイックソートのアイデア

4.2.1　クイックソートのアルゴリズム

- クイックソート
  - マージソートと同様に配列を小さく分割し，
    全体をまとめ上げることでソートを実行する方法.
  - 分割統治法の仲間.

- マージソートとの違い
  - 配列を分割するとき，ちょっとした工夫をする.

- アルゴリズムの概略（図4.5）
  - pivot（ピボット）と呼ばれる基準を適当に選択.
    - ひとまず，入力配列の最後の数.
  - 配列全体を走査し，
    pivotより小さかったら左，大きかったら右へと要素を移動.
    - pivotと等しい要素は，別の配列に集める.
  - 左右にできた新たな配列は，未ソート.
    → それぞれを再びクイックソートの入力とする.
    - pivotを集めた配列は，すべてが同じ要素
      → そのまま真ん中に残す.
  - 左右にできた配列に対して，再びクイックソートを適用.

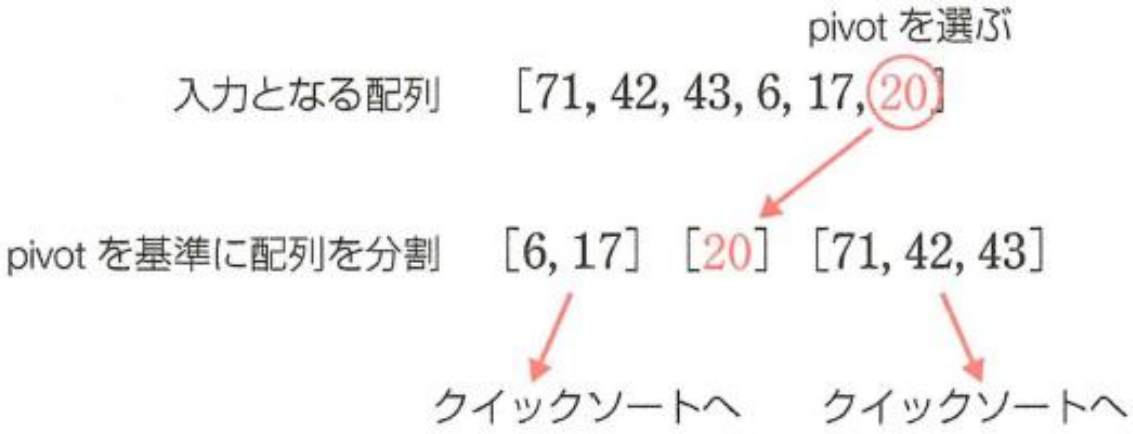

図 4.5 クイックソートのアルゴリズム

- 前述の手順を繰り返していけば，
  与えられた配列は最終的に要素が1つ以上で
  同じ数からなる配列に分割.

- クイックソート
  - 分割された配列の統合が，マージソートと違って簡単.
  - 分割するとき，pivotを基準に左右に値を
    振り分けているため，統合するときはそのまま統合.
  - 再帰呼び出しを使うと簡潔に実装可能.

4.2.2　クイックソートを実装する

- 注意点
  - 再帰呼び出しを必ず終了させる.
    → 関数が値を返すところに注意.

- 空の配列が入力として渡された場合
  - 空の配列をそのまま返すコードを関数の先頭に付加.

- 関数の最後
  - 左側と右側の配列にそれぞれ,
    クイックソートを適用して返す.
  - 空の配列の間に，pivotと等しい値が並んだ配列を
    そのまま返すコードを付加.

コード 4.5　クイックソート

```
1  def quick_sort(array):
2      # 空の配列はそのまま返す
3      if array == []:
4          return array
5      # 最後の要素をpivot にする
6      p = array[-1]
7      left = []
8      right = []
9      pivots = []
10     # pivot との関係で要素を分割する
11     for v in array:
12         if v < p:
13             left.append(v)
14         elif v == p:
15             pivots.append(v)
16         else:
17             right.append(v)
18     # 左と右は再び関数を適用して返す
19     return quick_sort(left) + pivots + quick_sort(right)
```

課題3：コード4.5の関数の動きを確認（提出不要）

- 適当な配列を用意し，プログラム実行．
  - バグがないか，特に再帰呼び出しがきちんと機能するかを確認．
  - エラーで再帰呼び出しが無限ループに陥った場合
    - Pythonにはそれを止める機能もあるが，結果が返ってこない場合はCtrl+Cによって実行を中止．

```
my_array = [random.randint(0, 100) for i in range(15)]
my_array
```

```
[55, 57, 20, 29, 39, 33, 5, 10, 5, 59, 80, 35, 66, 68, 82]
```

  - きちんと実装できていれば，ソートされた配列が返る．

```
quick_sort(my_array)
```

```
[5, 5, 10, 20, 29, 33, 35, 39, 55, 57, 59, 66, 68, 80, 82]
```

4.2.3 クイックソートの計算量

- クイックソートの計算量： $O(n\log n)$
- 図4.6
  - クイックソートにおいて，
    どのように配列が分割されるかを示す．
  - 木構造

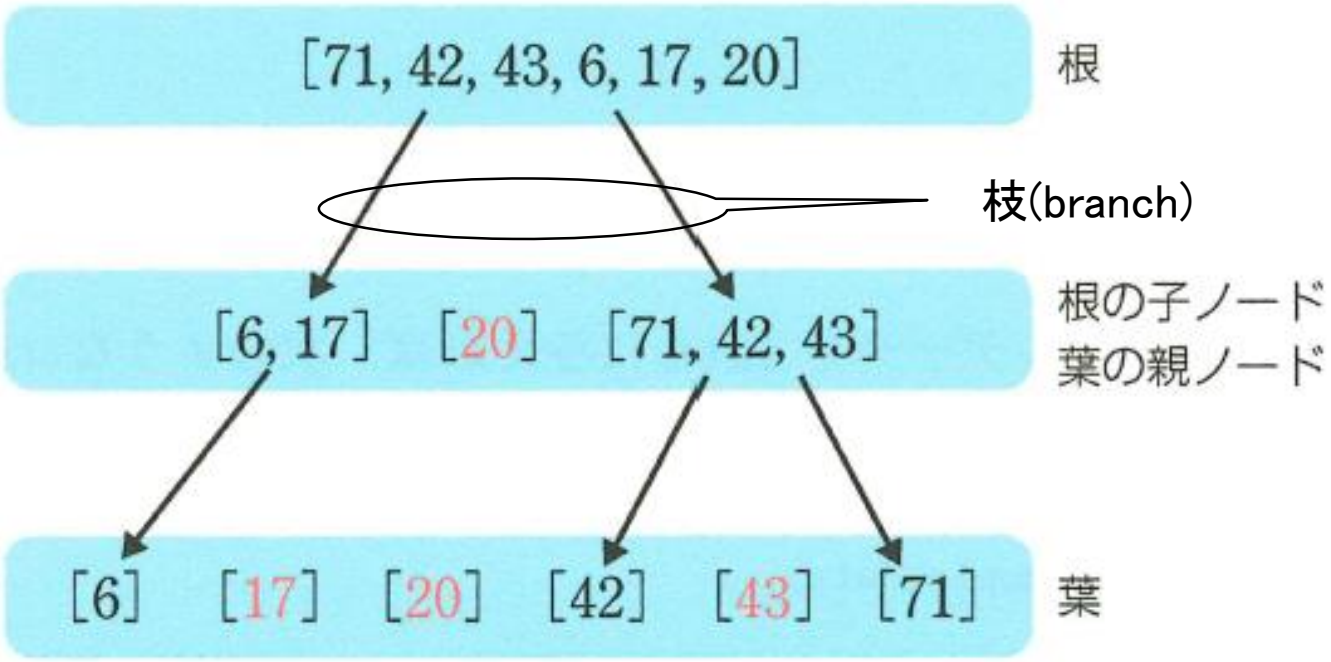

図 4.6　クイックソートの分割の様子

- クイックソートにおける分割
  - あるノードから見て，左側の子ノードと右側の子ノードへの分割.

  ⇒ 「二分木」で表現可能.
- pivotの選び方によってこの木の形が決定.
- ひとまず，pivotをうまく選べたとし，

  分割のたびに配列が半分程度の長さになったと仮定.
- マージソートのときと同じ考え方で，計算量は$O(n \log n)$
  - pivotをうまく選べたときという条件の場合

- **pivotをうまく選べないとどうなるか？**
  - 実際にソートにかかる時間を計測しながら，
    マージソートとの比較でこの点を考察.

4.2.4 マージソートとの比較

- 通常アルゴリズムの計算量の考え方
  - 入力データサイズに対して計算時間がどのように変化するか？
- この点では，マージソートとクイックソートは同じ計算量．
- 同じサイズのデータでもデータの性質が違うと，
  アルゴリズムによって計算時間が大きく変化．

- サンプルデータを用意．
  - 長さ2,000 の配列を100 個作り，
    配列の各要素は0～5,000 の中からランダムに選択．
  - 用意するサイズは，各自の計算機の能力に合わせ適当に調整可．

```
sample_data = []
for i in range(100):
    sample_data.append([random.randint(0, 5000) for i in range(2000)])
```

- 「ソートを実行する関数」と「データ」を引数にとり，実行時間を計測する関数を作成．
  - ソートの作業を引数にとった回数だけ繰り返す．

コード 4.6　ソートにかかる時間を計測する関数

```
1  import time
2
3  def performance_check(method, data, num=3):
4      s = time.time()
5      for i in range(num):
6          for v in data: method(v)
7      e = time.time()
8      return e - s
```

- 関数を引数methodに，サンプルデータをdataに与えれば，次のようなコードで時間を計測可能.

```
performance_check(merge_sort, sample_data)
```

```
2.0325491428375244
```

```
performance_check(quick_sort, sample_data)
```

```
1.2834510803222656
```

- 出力の単位は秒で，クイックソートがマージソートより若干速い.

4.2.5 クイックソートの弱点

- 入力となる配列がすでにソートされていると
  パフォーマンスが悪い.

- 次のようなコードで,
  サンプルデータをあらかじめソート.

```
sorted_data = []
for i in range(100):
    sorted_data.append(sorted([random.randint(0, 5000) for i in range(2000)]))
```

- このデータを引数に，再び計算時間の計測を実行．
- マージソート
  - 入力配列の長さは変わっていないため，
    ほとんど同じ時間で終了．
- クイックソート
  - パフォーマンスが大幅に劣化．

```
performance_check(merge_sort, sorted_data)
```

```
1.6679749488830566
```

```
performance_check(quick_sort, sorted_data)
```

```
55.67849087715149
```

4.2.6 PIVOTの選び方

- ソートされた配列に対して，
  クイックソートのパフォーマンスが大きく劣化する理由は？
- 問題を分割して解くアルゴリズム
  - 問題をうまく分割できるかどうかがパフォーマンスに大きく影響．

- マージソート
  - 入力配列を格納されている要素に関係なく分割．
  - 入力がソートされているかどうかに関係なく，
    いつも同じ分割になることを意味．
- クイックソート
  - 入力配列の最後の要素をpivotとして選択し，
    pivotとの比較で配列を分割．

- クイックソート
  - 入力配列がソートされていると，
    このとき選ばれるpivotは配列の中の最大値．
  - 結果として， pivot以外の要素は
    すべて左側の配列にまとめられる．

    ⇒ ほとんど分割されずに，次のステップへ引き渡す．
    分割され左側に残った配列の長さは，
    元の配列と1つしか違わない．

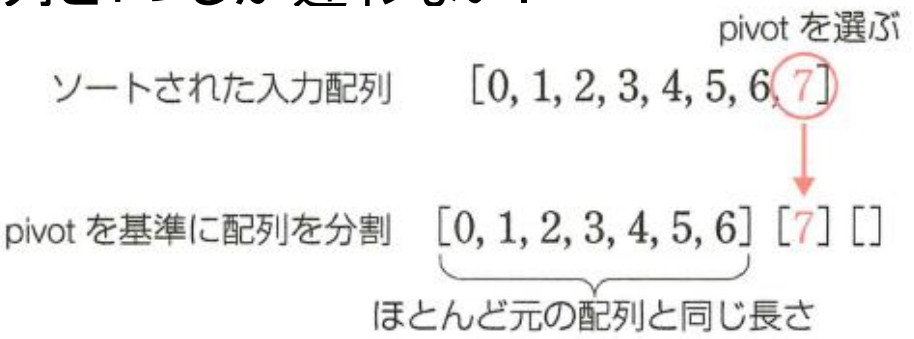

図 4.7　ソートされた配列に対するクイックソートの振る舞い

- 関数はn回呼び出されることになり，
  その都度残った配列の各要素とpivotの比較が必要．
- 比較回数は$(n-1)+(n-2)+\cdots+1=\frac{(n-1)(n-2)}{2}$となり，
  計算量は$O(n^2)$

課題4（提出不要）

- 4.2.4　マージソートとの比較
  - 入力するデータの性質によって，
    計算時間が大きく変化することがある．
    - このような現象がなぜ起こるのか？
    - それがどのような示唆を与えるのか？

    →　教科書を読み，理解して説明できるようになること．
- 4.2.6　pivotの選び方
  - 入力となる配列がすでにソートされていると，
    パフォーマンスが低下．
  - → 教科書の例を参考に，プログラムを実行して確認すること．

4.2.7 クイックソートの改良

- クイックソートの欠点を，
  pivotの選び方を変えることで克服.
- pivotをランダムに選ぶという戦略.

コード 4.7　pivot をランダムに選ぶクイックソート

```
1  import random
2
3  def quick_sort(array):
4      if not array:
5          return array
6      pivot = random.choice(array) # 変更点はここだけ
7      left = []
8      right = []
9      pivots = []
10     for v in array:
11         if v < pivot:
12             left.append(v)
13         elif v == pivot:
14             pivots.append(v)
15         else:
16             right.append(v)
17     return quick_sort(left) + pivots + quick_sort(right)
```

- ソートされた配列に対するパフォーマンスを調査.
- 再びマージソートと同じくらいの結果になるか？

```
performance_check(quick_sort, sorted_data)
```

```
1.6945829391479492
```

一般的なPIVOTの選び方

- クイックソートのもっとも速い動作
  - 基準値の選び方　⇒　データが均等に2分割できる値

- 【重要】基準値の一般的な選び方
  - (a) 対象の配列D[left],D[left+1],…,D[right]の中からランダム.
  - (b) 対象の配列の真ん中のデータ$D[\left\lfloor \frac{left+right}{2} \right\rfloor]$
  - (c) 対象の配列の左端D[left], 真ん中の$D[\left\lfloor \frac{left+right}{2} \right\rfloor]$,
    右端D[right]の3つのデータを並べて中央にくるデータ

課題5：PIVOTの選び方による弱点克服

- 4.2.7　クイックソートの改良
  - クイックソートの弱点
    - 入力配列がソートされていると，実行速度が著しく低下．
    - pivotの選び方を変えることで克服．
      - 戦略：pivotをランダムに選択．
      - 入力配列がソートされているかどうか不明．
  - 教科書P.068–069のコード4.7を実行し，
    教科書P.066のコード4.6を用いて，
    教科書P.064のコード4.5の結果と比較する
    プログラムを作成しなさい．
  - 余力がある学生へ
    - 【重要】基準値の一般的な選び方で紹介した，
      (b)と(c)の選び方にもチャレンジし，さまざまな配列データで
    - 選び方による効果を確認．

4.2.8　クイックソートが与える示唆

- クイックソート
  - pivotの選び方をランダムにするだけで，
    ソートされた配列に対してパフォーマンス低下を防止.
- pivotの選び方がランダムということは．．．
  - まったく同じデータに対しても，実行のたびに
    内部の計算過程は違ったものになる可能性あり.

⇒ ユークリッドの互除法やマージソートのような
アルゴリズムとは大きく違う.

- 乱択アルゴリズム（randomized algorithm）
  - アルゴリズムの実行にランダムな要素を含むもの.
  - 9章で詳しく扱う.

- クイックソート
  - pivotの選び方を変えると，
    同じ入力配列に対して計算時間がかなり変化．
- アルゴリズムの内部で起きる分岐において，
  いつも決まった選択肢ではなく，
  その都度別の選択肢を選べるような計算方法
  ⇒　計算が早く終わる可能性あり．
- 通常の計算
  - 数ある分岐の中から1つを選んで計算を進行．
  - よい選択肢を選ぶこともあれば，あまりよくない道を選ぶこともあり．
- さまざまな可能性をすべて試せるとしたら，
  計算量の考え方はどのように広がるのだろうか？
- 8章の内容
  - これらの疑問に対する答えと，問題の難しさに対するさらに深い知見．

課題6

- 教科書P.069の第4章の練習問題
  - 4.2
    - 標準ライブラリheapqにあるmergeを使ってマージソートを実装せよ.
  - 4.3
    - 受け取った配列が昇順にソートされているかどうかを
      判断するコードを作成せよ.
    - 教科書P.062のコード4.4の結果と比較すること.

【重要】アルゴリズム設計手法のまとめ

- **アルゴリズムを作成する場合に頻繁に用いられる設計手法を理解**
  - **自分で新たなアルゴリズムを考案する場合のヒント**
  - **効率のよいアルゴリズム作成の大きな手助け**
- アルゴリズムの設計手法
  - **分割統治法**
    - そのままでは解決できない大きな問題を小さな問題に分割し，その全てを解決することで，最終的に最初の問題全体を解決する手法
  - **グリーディ法**
    - 問題の要素を複数の部分問題に分割し，それぞれを独立に評価を行い，評価値の高い順に取り込んでいくことで解を得るという方法
  - **動的計画法**
    - 対象となる問題を複数の部分問題に分割し，部分問題の計算結果を記録しながら解いていく手法
  - **バックトラック法**
    - 問題の解を見つけるために，解の候補をすべて調べる（列挙する）ことを組織的にかつ効率よく行う手法
  - **分枝限定法**
    - 列挙木の各節点が表す選択肢において，その選択により問題の出力となる解が得られない場合はそれ以上の列挙の操作を中止（枝切り）し，列挙木の上のレベルに戻るという操作を行う方法