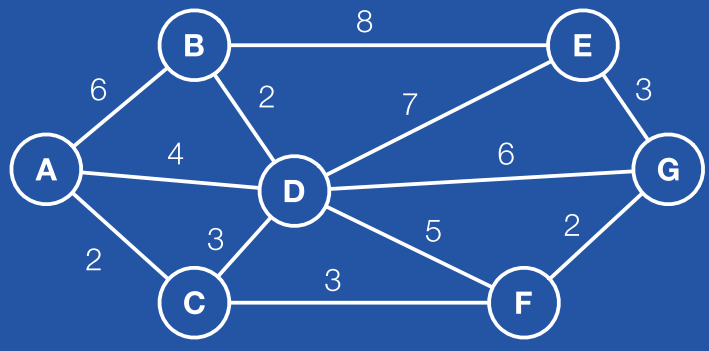


Pythonで体験してわかる アルゴリズムとデータ構造

```
def shortest_length(G, start):
    S={start:0}; D={}
    while len(S) > 0:
        x = select_min(S); m = S[x]; del S[x]; D[x] = m
        for (y, w) in edge(G, x):
            if y in S:
                if S[y] > m + w:
                    S[y] = m + w
            elif y not in D:
                S[y] = m + w
        print('仮', S, '確定', D)

def select_min(S):
    m = -1
    for a in S:
        if m == -1 or m > S[a]:
            x = a
            m = S[a]
    return x

def edge(G, x):
    return ([(b, G[(a, b)]) for (a, b) in G if a == x]
            + [(a, G[(a, b)]) for (a, b) in G if b == x])
```

西澤弘毅
Koki Nishizawa
森田　光
Hikaru Morita
[著]

近代科学社

Python
で体験してわかる
アルゴリズムとデータ構造

西澤弘毅
Koki Nishizawa
森田　光
Hikaru Morita
[著]

近代科学社

◈ 読者の皆さまへ ◈

平素より，小社の出版物をご愛読くださいまして，まことに有り難うございます．

(株)近代科学社は 1959 年の創立以来，微力ながら出版の立場から科学・工学の発展に寄与すべく尽力してきております．それも，ひとえに皆さまの温かいご支援があってのものと存じ，ここに衷心より御礼申し上げます．

なお，小社では，全出版物に対して HCD（人間中心設計）のコンセプトに基づき，そのユーザビリティを追求しております．本書を通じまして何かお気づきの事柄がございましたら，ぜひ以下の「お問合せ先」までご一報くださいますよう，お願いいたします．

お問合せ先：reader@kindaikagaku.co.jp

なお，本書の制作には，以下が各プロセスに関与いたしました：

・企画：山口幸治
・編集：山口幸治
・組版：藤原印刷 (LaTeX)
・印刷：藤原印刷
・製本：藤原印刷
・資材管理：藤原印刷
・カバー・表紙デザイン：藤原印刷
・広報宣伝・営業：山口幸治，東條風太

はしがき

人が電卓を使って計算することは，自動車を運転することに例えられる．人は，からだで道具を操作し目的を達成する．道具は人の能力を増すだけで，目的の成否は人の操作による．

21 世紀初頭の現在，多くのソフトウェアにより，人の操作は減り，人のできることが増している．表計算ソフトが日常の計算を簡単にし，さらに，ワープロ，メール，web などが加わり，パソコンがネットワークに結びついて，ますますたくさんのことができるようになった．自動車分野でも，文字通りの自動運転が目標になりつつある．

あらゆる操作，特に繰り返される操作は，あらかじめプログラムを作っておき，それを計算機に参照実行させるのが自動化の基本である．しかし，プログラムそのものは人が作る——人がプログラミングする——と考えられ，自動化できない聖域と思われてきた．ところが，今や，人並みに学習機能がある AI がその聖域を駆逐すると言われる．

その一方で，最先端の分野では，個人が独自に 1 からモノづくりをするのが当たり前になっている．また，プログラムを自作する世界では，人々が相互にプログラムのソース・コードを見せあい刺激を与えあう，オープン・ソースの環境で切磋琢磨しあう進歩が今後も続くであろう．

プログラミングは料理に例えられる．料理は様々な素材を使って美味しいものを作る．これに対し，プログラミングでは，データ構造を選択し，効率があがるようにアルゴリズム（算法）を考える．そのために，アルゴリズムとデータ構造については，人として基本的な知識とイメージを持つ必要があり，本書はその導入に位置づけられる．

本書は次のようなきっかけから生まれた．多くの類書は網羅的かプログラム満載で，1 から学ぶ学習者にとっては敷居が高い．その一方で多くは，アルゴリズムを数式または自然言語のままにし，学習者が計算機で試すのに手間をかけさせている．そこで著者らは，アルゴリズムとデータ構造を 15 の課題に絞り，各々の章の前半で原理と事例によって理解を促し，後半で Python 言語をベースにプログラムで確認できるようにした．

なお，著者のひとりの西澤が，前任の鳥取環境大学（現：公立鳥取環境大学）で高校生にコンピュータを教えるセミナーがあり，そこで名古屋孝

幸先生と検討した経路探索の資料を本書に盛り込んでいる．特に，名古屋先生には，資料の掲載許諾をいただき，また本書の理論面でのチェックもお引き受けいただいた．ここに感謝申し上げる．しかし，本書に誤りや表現不足があれば，それは著者の責に帰する所である．

また，本書は，既存のアルゴリズムとデータ構造の教科書について，近代科学社の山口幸治氏との議論の中から出てきた．伏してお礼申し上げる．

2018 年 3 月
著者らしるす

目　次

1章　なぜアルゴリズムが重要か

この章の目標

1. アルゴリズムとデータ構造とは何か，それぞれを大まかに説明できるようになる．
2. アルゴリズムとデータ構造を学ぶことがなぜ重要か，それぞれの重要性を説明できるようになる．

キーワード

アルゴリズム，データ構造，計算量，時間計算量，領域計算量

「アルゴリズム」と「データ構造」という言葉の意味は，大まかには以下のとおりである．

- アルゴリズム = 問題を解くための詳細な手順
- データ構造 = データの格納方法と，データの操作方法

これらを学ぶことがなぜ重要なのか，本章で説明していく．

1.1　アルゴリズムとは

アルゴリズム (algorithm) とは「問題を解くための詳細な手順」のことであり，ここでの「詳細な」とは，曖昧さがなく，コンピュータが間違いなく実行できる程度に詳細ということである．

たとえば，以下の問題を考えてみよう．

（問題）正の整数一つが入力 N として与えられるとき，1 から N までの整数の和を出力するにはどうしたらよいか？

この問題を解くアルゴリズムとして，以下の文を記しただけでは不十分である．

（アルゴリズム？）$1+2+\cdots+N$ を出力

なぜなら，上記の記述では $1+2+\cdots+N$ を「どのように」計算するかが明らかではないからである．たとえば，1 に 2 から N までを順に加えるのか，逆に N に $N-1$ から 1 までを順に加えるのか，など手順はいろいろ考えられる．アルゴリズムとは「何を求めるか」ではなく「どのように求めるか」を記したものなのである．

この問題を解くアルゴリズムは少なくとも以下のように二つある．

（アルゴリズム 1）　1 に 2 から N までを順に加えて，その結果を出力
（アルゴリズム 2）　$\frac{N(N+1)}{2}$ を出力

アルゴリズム 2 の出力が問題の答えに一致することは，図を使って理解できる．図 1.1 では，1 から N までの整数の和にあたる部分を着色部の面積で表している．白い部分は，それと上下が逆だが同じ面積の領域である．二つの領域を合わせた面積は，長方形の面積なので $N(N+1)$ と簡単に求められ，着色部の面積は長方形の半分である $\frac{N(N+1)}{2}$ として求められる．

1.2　アルゴリズムを学ぶことがなぜ重要か

アルゴリズムを学ぶことが重要な理由は，結果が同じでもアルゴリズムが異なれば，計算の効率が異なる可能性があるからである．

実際，前節のアルゴリズム 1 とアルゴリズム 2 は，どちらも同じ問題を解

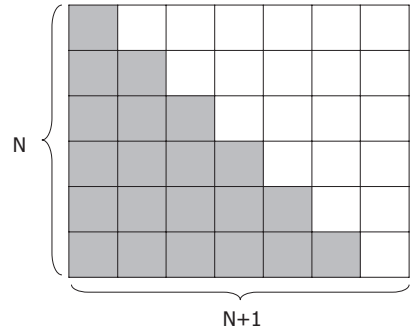


図 1.1　1 から N までの和（$N = 6$ の例）

いていることがわかるが，計算の効率は異なる可能性がある．それは，四則演算の使用回数に違いがあるためである．アルゴリズム 1 は，足し算を $N - 1$ 回実行しているが，掛け算と割り算は実行していない．アルゴリズム 2 は，足し算と掛け算と割り算を 1 回ずつ実行している．もし，足し算と掛け算と割り算の計算コストが同じであるならば，アルゴリズム 2 の方が効率が良いと言える．一方，掛け算と割り算の計算コストが足し算に比べて非常に大きい状況ならば，アルゴリズム 1 の方が効率が良いと言える．

▶[計算コスト]
計算のために必要な時間やメモリ領域の損失を指す．

したがって，今後，問題を正しく解けるアルゴリズムが設計できたからといって，満足してはいけない．同じ問題を解けるアルゴリズムは複数あるかもしれない．解いた結果は同じでも，計算時間が少なくて済むものや，計算に必要な記憶領域が少なくて済むものもある．

▶[計算]
ここで言う計算とは，必ずしも数値計算のことではなく，機械的な処理全般のことを指す．

また，与えられたデータの傾向によってその効率が変わってくる場合もある．アルゴリズムの中で用いてよい命令が制限されると実行できなくなるものもある．そのようなアルゴリズムによる違いを学び，効率も考えてアルゴリズムを設計できるようにならなくてはならない．

1.3　データ構造とは，データ構造がなぜ重要か

データ構造 (data structure) とは，大まかに言えばデータの格納方法と操作方法を決めたもののことである．コンピュータ内の記憶領域である**メモリ** (memory) は，基本的には番号づけられて並んだ領域であるが，そこに対してデータをどのように格納するのかということと，格納されているデータに対してどのような操作を許すかということは，慎重に決めなければならない．その選択を誤ると，アルゴリズムが遅くなったり記憶領域が無駄になったりしてしまう場合がある．

▶[データ構造の定義]
正確には，メモリに言及せず集合や写像などで数学的に定式化しただけの論理的なデータ構造と，メモリにまで言及する物理的なデータ構造が存在する．本書では，両方を総称してデータ構造と呼ぶことにする．

例として，配達システムの架空データを図 1.2 に示す．データ構造 (a) は，注文番号と宛先の住所と氏名をセットにして注文を列挙している．この方法でデータを格納すると，同じ宛先への注文が複数回あった場合に，

全く同じ住所と氏名が何箇所にも現れることになり，領域に無駄ができる．一方，データ構造 (b) は，宛先記号という記号を導入して表を二つに分けている．宛先記号という新しいデータが格納されているにもかかわらず，記憶領域は 20 マスで済んでおり，データ構造 (a) の 21 マスよりも少ない．このようにデータの格納方法の違いによって，記憶領域の使用効率は変わる．

データ構造 (a)

注文番号	住所	氏名
001	○県△市1-1-1	佐藤様
002	○県◇市2-2-2	鈴木様
003	○県◇市2-2-2	鈴木様
004	○県△市1-1-1	佐藤様
005	○県◇市2-2-2	鈴木様
006	○県△市1-1-1	佐藤様
007	○県△市1-1-1	佐藤様

データ構造 (b)

注文番号	宛先記号
001	A
002	B
003	B
004	A
005	B
006	A
007	A

宛先記号	住所	氏名
A	○県△市1-1-1	佐藤様
B	○県◇市2-2-2	鈴木様

図 1.2　データ構造の違い

また，データ構造においてはデータの操作方法も重要となる．図 1.2 のデータ構造 (a) を用いる場合は，実際の配達の担当者には，各行の内容をそのまま表示すれば宛先は一目瞭然である．一方，データ構造 (b) を用いる場合は，配達の担当者は宛先記号だけを見てもどこに配達すればよいのかわからない．そこで，配達の担当者に対しては，二つの表を組み合わせて，実際の住所や氏名を示してあげる必要があるだろう．その組み合わせの操作に非常に長い時間がかかるのであれば，データ構造 (b) にもデメリットがあることになる．このようにデータの格納方法と操作方法をまとめて全体として適切なデータ構造を決めなければならない．

1.4　次章以降の構成

2 章では，具体的にアルゴリズムを記述する方法について説明する．いきなり正確なアルゴリズムを記述するのは難しいので，図などのあいまいな表現から徐々に具体化していく流れを示す．

▶[正確なアルゴリズム]
数学的には，アルゴリズムは**計算可能性 (computability)** という概念で厳密に定められている．

3 章以降では，世の中ですでに知られているさまざまなアルゴリズムを紹介する．ただし，重要なことは，それらを丸暗記することではなく，アルゴリズムの大まかな種類や注意点を学んで，読者それぞれが解きたい問題に対して，自分でアルゴリズムを設計できるようになることである．本書では，アルゴリズムで解く「問題」の例として，整列と探索を多く取り上げる（図 1.3）．**整列（ソート，sort）**とは並べ替えのことで，たとえば自分のメールソフトの受信トレイに入っているメールを古い順に並べ替えたり，サイズの大きい順に並べ替えたりすることである．**探索 (search)** と

は，データの集まりから特定の値（あるいは特定の条件を満たす値）を探すことで，たとえばメールのうち差出人が A さんのものだけを探すことである．整列や探索というのはあまりに基本的な問題に思えるかもしれないが，いろいろなアルゴリズムを比較したり効率を分析するには，あつかう例が整列のようにシンプルな方が，本質がわかりやすくなるのである．

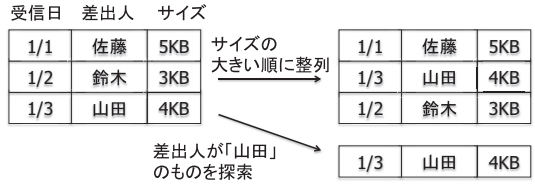


図 1.3　整列と探索の例

8 章では，アルゴリズムの効率を評価する尺度として**計算量 (complexity)** を学ぶ．特に計算にかかる時間を評価した尺度は**時間計算量 (time complexity)**，計算に必要となるメモリ内の記憶領域を評価した尺度は**領域計算量 (space complexity)** と区別して呼ぶ．

では，本章のアルゴリズム 1,2 の時間計算量は，それぞれどのような値になるのだろうか？まず，時間計算量とは「計算にかかる時間を評価した尺度」ではあるが，「何分何秒」といった正確な時間を用いることはできないことに注意しよう．なぜなら計算にかかる時間というものは，アルゴリズムやそれを実装したソフトウェアだけでなく，ハードウェアの性能にも依存するからである．したがってアルゴリズムを評価する尺度には適していない．そこで時間そのものではなく，アルゴリズムを構成する基本命令の実行回数で評価することが多い．整数の四則演算を基本命令とする場合は，アルゴリズム 1 では $N-1$ 回，アルゴリズム 2 では 3 回，基本命令が実行されると言える．

10 章以降でデータ構造について述べる．本章では具体的な配達システムや宛先記号を例に挙げたが，本書の残りの章では，もう少し抽象的なデータ構造について紹介し，データ構造の選択の要点が本質的にわかるようにする．

1.5　練習問題

1. 「正の整数一つが入力 N として与えられるとき，1 から始めて小さい順に N 個の奇数を足した数を出力するにはどうしたらよいか」という問題に対して，アルゴリズムを二つ考える．アルゴリズムの一つは「＋･･･＋」を許した数式として書け．また，その二つのアルゴリズムがどちらもこの問題の答えになることを説明せよ．
2. 以下のデータに，宛先記号という記号を導入して表を二つに分けて整理せよ．

注文番号	住所	氏名
001	○県△市1-1-1	佐藤様
002	○県◇市2-2-2	鈴木様
003	○県◆市3-3-3	山田様
004	○県△市1-1-1	佐藤様
005	○県◇市2-2-2	鈴木様
006	○県◇市2-2-2	鈴木様
007	○県◇市2-2-2	鈴木様
008	○県◆市3-3-3	山田様
009	○県◇市2-2-2	鈴木様
010	○県◆市3-3-3	山田様

1.6　1 章のまとめ

1. アルゴリズムとデータ構造とは何か，それぞれ大まかに説明できるようになった．
   - アルゴリズムとは，「問題を解くための詳細な手順」のことであり，「何を求めるか」ではなく「どのように求めるか」を記したものである．
   - データ構造とは，データの格納方法と操作方法を決めたものである．
2. アルゴリズムとデータ構造を学ぶことがなぜ重要か，それぞれ説明できるようになった．
   - 実行結果が同じでも，アルゴリズムが異なると計算の効率が異なる場合があるためである．
   - 格納しているデータの全体が同じでも，格納方法が変わると記憶領域の使用効率が変わり，操作方法が変わると計算の効率が変わるためである．

1.7 Python 演習

本書の各章は，アルゴリズムの説明部分だけを読んでも理解できるように構成されているが，学んだアルゴリズムを実際にコンピュータ上で実行させたい人のために，プログラムも紹介している．本書では，プログラミング言語として **Python（パイソン）**を用い，各章に「Python 演習」という節を設けている．本章の Python 演習では，すでに Python の開発環境を整えた読者を対象に，Python の基本的なデータと命令について紹介する．

▶[Python の開発環境]
Python の導入方法と，プログラミングから実行までの手順は付録で紹介しているので，まずは付録を確認してほしい．

1.7.1 オブジェクトと型

Python におけるデータや処理の実体を総称して**オブジェクト (object)** と呼ぶ．各オブジェクトは**型 (type)** と呼ばれる「オブジェクトの種類」のいずれかに属している．Python の主な型と，その型に属するオブジェクトの例などを表 1.1 に示す．

▶[型]
各オブジェクトの型の名前は，Python インタプリタで type 関数を各オブジェクトに適用すれば自分で確認できる．

表 1.1 Python の主な型

型	オブジェクトの例	オブジェクトの性質		
int	0 や 1 や-2			
float	0.0 や-1.5			
bool	True と False のみ			
NoneType	None のみ			
str	'' や'abc'	シーケンス	イテラブル	
tuple	() や (2,) や (1,2,'a')	シーケンス	イテラブル	
range	range(0,4)	シーケンス	イテラブル	
list	[] や [2] や [1,2,'a']	シーケンス	イテラブル	ミュータブル
set	set() や{2}や{1,2,'a'}		イテラブル	ミュータブル
dict	{}や{1:'a',2:'b'}		イテラブル	ミュータブル

型が重要なのは，型によってオブジェクトに適用できる演算が異なるためである，たとえば，int 型には整数，float 型には小数，str 型には**文字列 (string)** が，それぞれ属している．+演算は整数や小数に対しては足し算を意味するが，文字列に対しては連結を意味する．

```
>>> 1 + 2
3
>>> '1' + '2'
'12'
```

bool 型には真偽値の True（真）と False（偽）のみが属する．NoneType 型は「値が指定されていない」ことを意味する値 None のみが属する特殊な型である．

tuple 型のオブジェクトは**タプル (tuple)**，list 型のオブジェクトは**リスト (list)** とも呼ばれる．どちらも有限個のオブジェクトからなる列を表すが，後述するミュータブルという性質があるかどうかが主な違いである．set 型のオブジェクトは**集合 (set)** とも呼ばれ，タプルやリストと異なり要素の順序と重複を無視する．このことを比較演算子==を使って確かめよう．==は，二つのオブジェクトの見た目が異なっていても，意味が同じであれば True を返す演算子である．==は連続して使えることにも注意しよう．

```
>>> [3,2,3] == [2,3,3]
False
>>> {3,2,3} == {2,3,3} == {2,3}
True
```

型が異なっていても，より広い種類への型変換が自動的に行われて「意味が同じ」と判定されることがある．

```
>>> True == 1 == 1.0
True
```

range 型は 3 章で，dict 型は 13 章で解説する．

1.7.2　関数と変数

Python に実行させたい「処理」を記述して名前をつけたものを**関数 (function)** という．1.1 節で例に挙げた整数の和を求めるアルゴリズム 1,2 をそれぞれ関数 sum1 と sum2 として実装した Python プログラムをソースコード 1.1 に示す．

ソースコード 1.1　sum_of_numbers.py

```
1 def sum1(n):
2     s = 1
3     for i in range(2, n + 1):
4         s = s + i
5     return s
6
7 def sum2(n):
8     return n * (n + 1)//2
```

関数を呼び出して実行するには，関数名のあとに処理に必要な情報をカッコに囲んで与えればよい．この「処理に必要な情報」を**引数 (argument)** と呼ぶ．今回の関数 sum1 と sum2 の引数は正の整数である．引数を複数取る関数もある．

```
>>> from sum_of_numbers import sum1,sum2
>>> sum1(4)
10
>>> sum2(4)
10
```

関数は def 文で定義し，def のあとに関数名と，引数の名前（今回の関数 sum1 と sum2 の場合は n）を記述する．実行させる処理の内容は:（コロン）の次の行からインデント（字下げ）をして記述しなければならない．

▶[インデント]
本書ではインデントは半角 4 文字で統一しておく．インデントは tab キーではなくスペースキーで行うこと．

return 文は関数の実行終了と返り値の指定をする文である．**返り値 (return value)**（あるいは**戻り値**）とは，関数の実行が終了した際にその関数を呼び出した側に伝えることのできる情報である．ソースコード 1.1 の関数 sum1 と sum2 の返り値はいずれも 1 から n までの和である．したがって，どちらも同じ結果を返す．

「名前=オブジェクト」という文でオブジェクトに名前をつけることができ，その名前のことを**変数 (variable)** という．

▶[変数]
変数名に使えない文字や単語もあるので，詳しくは「The Python Language Reference」を参照すること．

正式には変数がオブジェクトを**参照する (refer)** という．

▶[参照]
変数にオブジェクトを「代入」するという表現を使うこともある．しかし，変数はオブジェクトを格納する箱のようなものではなく，オブジェクトにつける名前でしかないので注意すること．詳しいことは後述する．

変数名は定義された関数の中でしか有効でない．たとえば，ソースコード 1.1 の中の n は変数であるが，sum1 が受け取る n と sum2 が受け取る n は区別される．

1.7.3 オブジェクトの性質

表 1.1 で**シーケンス (sequence)** と書かれているオブジェクトはいずれも，要素を $0, 1, 2, \ldots$ と番号付けて格納しているオブジェクトであり，後ろに [番号] をつけることで個々の要素を評価したり，+演算で連結したりすることができる．後半の章で紹介するスライスという演算も使える．

```
>>> x = [10,20,30]
>>> x[1]
20
```

イテラブル (iterable) なオブジェクトは，格納している要素一つ一つについて同じ処理を繰り返して行わせるための，for 文が使える．詳しいことは 3 章以降で紹介するが，要素の数を len 関数で確認したり，特定のオブジェ

クトの有無を in 演算で判定したりもできる．ソースコード 1.1 の 3,4 行目は，変数 i に 2 から n の値を順に代入し，そのたびに文 s=s+i を実行させるという意味である．これにより関数 sum1 はアルゴリズム $1+2+\cdots+n$ を実現している．

ミュータブル (mutable) なオブジェクトは，オブジェクトの中身を変更できる．

▶[ミュータブル]
「変数がどのオブジェクトを参照しているか」という対応付けを変更するのではなく，オブジェクトそのものの中身を変更できることが特徴である．

ミュータブルなオブジェクトを変数で参照する場合は注意が必要である．なぜなら，同じオブジェクトを複数の変数が参照することもでき，ある変数の参照するオブジェクトの変更が，他の変数にも反映されることがあるからである．たとえば，以下の変数 x,y は同じリストを参照している．ここで x の中身を変更すると，y の中身も変更されることがわかる．

```
>>> x = [10,20,30]
>>> y = x
>>> x[1] = 40
>>> y
[10, 40, 30]
```

なお，プログラム中で二つの変数が参照しているオブジェクトが同じかどうか確認したい場合は，比較演算子 is を使えばよい．実はオブジェクトには製造番号のような番号がつけられており，is は両者の製造番号が同じ場合に True を返す演算子である．

```
>>> x = [1,2,3]
>>> y = x
>>> x is y
True
```

ミュータブルなオブジェクトにはデメリットもある．集合は「重複」を無視するが，重複とは下のように「意味が同じこと」を指す．

```
>>> {1, 1.0, True} == {1}
True
```

仮に，ミュータブルなオブジェクトを集合に入れてしまうと，その値を変更するたびに重複の判定が必要になるため，ミュータブルなオブジェクトは集合には入れられない決まりになっている．タプルはミュータブルではないので，集合に列を入れたい場合は，タプルを入れればよい．

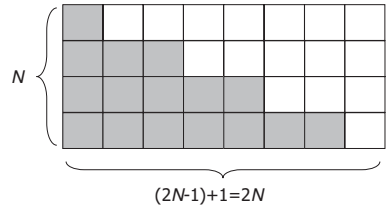
1.8　練習問題正解例

練習問題 1 の正解例

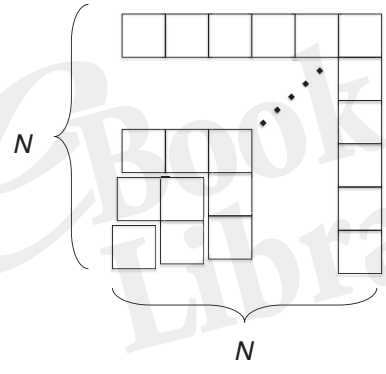
（アルゴリズム 1）　1 に 3 から $2N-1$ までの奇数を順に加えて，その結果を出力

（アルゴリズム 2）　N^2 を出力

以下の図（$N=4$ の例）では，数を四角の集まりとして表し，1 から始めて小さい順に N 個の奇数を足したものが，$N \times N$ 個になることを確認できる．



また，以下の図でも，N 個の奇数を足したものが $N \times N$ の正方形になることを確認できる．



練習問題 2 の正解例

注文番号	宛先記号
001	A
002	B
003	C
004	A
005	B
006	B
007	B
008	C
009	B
010	C

宛先記号	住所	氏名
A	○県△市1-1-1	佐藤様
B	○県◇市2-2-2	鈴木様
C	○県◆市3-3-3	山田様

2章　アルゴリズムを表現する様々な方法

この章の目標

アルゴリズムを，フローチャート，擬似コード，Python プログラム，などの異なる手段で表現できるようになる．

キーワード

フローチャート，擬似コード，逐次実行，条件分岐，反復

前章で，アルゴリズムとは「問題を解くための詳細な手順」であることがわかった．本章では，具体的にアルゴリズムを記述する方法をいくつか紹介する．

本章の中心的な例として，整数の除算の方法を取り上げることにする．ここでいう整数の除算とは，割る数と割られる数がどちらも整数であるだけでなく，答えも小数点以下を無視して整数部分のみ（商）を求める割り算のことである．また，割られる数も割る数も正の整数だけを対象とする．

小学校で習う方法では，乗算の部分を簡単にするために割られる数を一度に扱わず，桁ごとに段階的に求めていくことになっている．一方，本章では，乗算は何桁でも瞬時に求められると仮定し，割られる数を桁で区切ることはしないものとする．

2.1　言葉による表現

整数の除算は誰もが小学校で習う方法だが，言葉で説明しようとすると意外と複雑な手順であることがわかる．とりあえず手順を言葉で表したものが図 2.1 である．

> 割られる数を m として受け取り，割る数を n として受け取り，$n \times a \leqq m$ を満たすような a のうちで最大の値を商として出力して終了する

図 2.1　アルゴリズムとして不十分な整数の除算の説明の例

しかし，これはアルゴリズムとしては不十分である．理由は，「$n \times a \leqq m$ を満たすような a のうちで最大の値」をどのように求めるのかが不明だからである．したがって，その部分を具体的に手順化する必要がある．

2.2　図による表現

最初からアルゴリズムを厳密に記述することが難しい場合，とりあえず図で表現してみることも一つの方法である．ここで「$n \times a \leqq m$ を満たすような a のうちで最大の値」を求める際に，自分が何を行なっているか自己分析してみよう．とりあえず入力を限定して，$13 \div 3$ の場合を想像してみることにする．多くの人は，頭の中で $3, 6, 9, \ldots$，と割る数 3 の倍数を列挙し，どの段階で 13 を超えるか，確認する作業をしているはずである．これを図にすると図 2.2 のようになるだろう．

図 2.2 が表している手順は以下のとおりである．まず商 a の候補として 1 を仮定し，$n \times a$ が m を超えているかどうか計算して確認する．この場

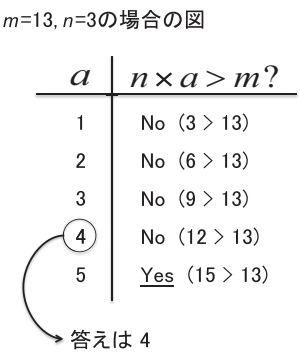


図 2.2　アルゴリズムとして不十分な整数の除算 $(m \div n)$ の図の例

合は $3 > 13$ は成り立たないので，次に a を 2 にして確認する．この場合も成り立たないので，次に a を 3 にして確認する．このように a を $3, 4, 5$ と増やしながら，$n \times a$ が m を超えているかどうかを確認していく．a を 5 にした時，初めて $n \times a$ が m を超えたことがわかる．そうしたら手順をやめ，直前の $a = 4$ を答えとする．

整数の除算を図で表したことによって，いきなり文章で表そうとするよりは具体的な手順が明らかになった．それが図で表現することの利点である．注意点としては，あくまでも図で表した入力例（図 2.2 の場合は $m = 13, n = 3$ のこと）についてしか表現できていないことがある．他の入力例に対しても手順として有効かどうか確信は持てないだろう．したがって図もアルゴリズムの表現としては不十分である．

2.3　フローチャートによる表現

図による表現はアルゴリズムとしては不十分だが，重要な情報を浮き立たせてくれることが多い．図 2.2 でいえば，「$a = 1$ という初期設定」「a を 1 増やす処理」「$n \times a > m$ についての確認」などが重要な情報である．これらをどのような順番で行なっているか，矢印を使って表現してみよう．今度は特定の入力例に関する図ではなく，任意の入力に対して計算方法が確実に定まるように記述してみる．そのような処理の流れを表す図を**フローチャート (flowchart)** という．整数の除算のフローチャートを図 2.3 に示す．

フローチャートには重要な仕組みが三つ現れている．一つ目は，計算は「開始」から始まり，特に矢印で示されていない場合は上から順に実行されるという点である．この仕組みを**逐次実行 (sequential execution)** という．

二つ目は，$n \times a > m$ と書かれたひし形である．ひし形は，中に書かれた条件式が成り立つ場合は「Yes」の先に実行を移し，そうでない場合は「No」の先に実行を移すことを表す．このように，状況に応じて行うべき計算を変える仕組みを**条件分岐 (conditional branch)** という．$n \times a > m$ な

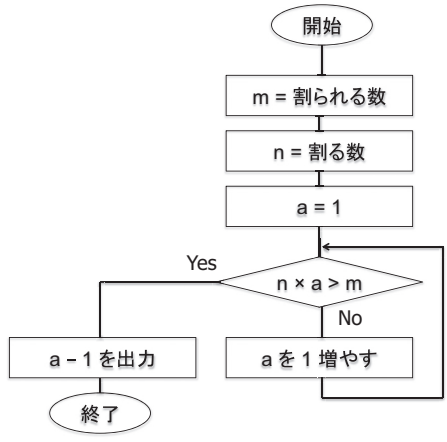


図 2.3　整数の除算のアルゴリズムのフローチャート

どの条件式の部分があまりに複雑になる場合は，必要な計算ができるだけ条件分岐の前までに済むような記述にするか，その条件式の内容を計算するアルゴリズムは別途記述するなどすべきだろう．

三つ目は，「a を 1 増やす」の下からひし形の上に矢印が戻っている部分である．この仕組み自体は，必ず特定の箇所に移行する**無条件ジャンプ** (unconditional jump) という仕組みだが，それによってひし形の条件分岐と「a を 1 増やす」の処理を何回か繰り返す意図を実現できることが重要である．処理の繰り返しを**反復** (iteration) ともいう．

上記の「逐次実行」「条件分岐」「反復」はアルゴリズムを詳細に記述する際の「詳細さ」の一つの目安になる構造であり，**制御構造** (control structure) と呼ばれる構造に属する．アルゴリズムを記述する際には，手順の中に隠れている制御構造についてはできるだけ明確に記述することが望ましい．

2.4　擬似コードによる表現

整数の除法をフローチャートとして表現できたところで，あらためて言葉で表現し直してみよう．ただし今度は，図 2.1 のように無理に 1 文で表現しようとせず，フローチャートで長方形やひし形に区切られていた各命令を，箇条書きのように 1 行ずつ区切って言葉にしてみよう．図 2.3 のフローチャートには長方形とひし形が合計 6 個あるので，1. から 6. の箇条書きにして表現する．箇条書きとして書き直した整数の除算のアルゴリズムが図 2.4 である．

図 2.4 は単なる箇条書きではあるが，フローチャートを基にして記述したため，手順が明確に表されている．このように「プログラムに近い程度に詳細に書かれた箇条書き」を**擬似コード** (pseudocode) という．

1. 割られる数を m として受け取る
2. 割る数を n として受け取る
3. 商の候補として $a = 1$ を用意する
4. $n \times a > m$ であれば，手順 6. に移る
5. そうでなければ，a の値を 1 増やして上記 4. の手順に戻る
6. $a - 1$ を商として出力して終了する

図 2.4　整数の除算のアルゴリズムの擬似コード

2.5　アルゴリズムの正しさの保証

整数の除法の手順をフローチャートや擬似コードとして表現したことで，とりあえずアルゴリズムとして曖昧な点のない表現が得られた．しかし，このアルゴリズムで本当に除法の商が正しく求まるかどうかは，明らかではないことに注意しよう．

得られたアルゴリズムやプログラムが求めるべき答えを正しく導出しているかどうかを，どう確かめたらよいか，という問題は，専門家の間でも未だ研究中の課題である．アプローチとしては，有限個の入力例を試して出力が正しいかどうか確認するという**テスト** (test) と呼ばれる手法が伝統的であるが，これはあらゆる入力に対して確認したことにはならないのは明らかであろう．そこでアルゴリズムが満たすべき**仕様** (specification) をまず数学的に厳密に記述し，得られたアルゴリズムがどんな入力に対しても仕様を満たすことを数学的に証明するアプローチがとられる．数学，論理学，計算機などの力を総動員して正しさの保証を目指す**形式手法** (formal method) という手法も近年盛んに研究されている．

上述の手法を含め，アルゴリズムやソフトウェアを高い品質で開発する方法の研究分野は**ソフトウェア工学** (software engineering) と呼ばれ，現在でも活発に研究が行われている．

2.6　練習問題

1. 三つの整数 x, y, z を入力として受け取り，三辺の長さが x, y, z であるような三角形について「A. 正三角形」「B. 正三角形ではない二等辺三角形」「C. 正三角形でも二等辺三角形でもない普通の三角形」「D. 三角形ではない」のうち正しい答えを出力するアルゴリズムを考え，フローチャートとして記述せよ．ここでひし形の中の条件式としては，「x」「y」「z」「$+$」「$>$」「$<$」「$\leqq$」「$\geqq$」「$=$」「かつ」「または」という記号や用語しか使ってはいけないものとする．ただし，三つの入力 x, y, z は整数ということしか決まっておらず，正の数が入力されるとは限らないことに注意すること．もちろん長さが正でない場合は三角形ではない．
2. 上記 1. のフローチャートを擬似コードとして表現せよ．ただし，フローチャートに現れる長方形やひし形の一つずつに対し，箇条書きの番号の一つずつに対応させよ．

2.7　2 章のまとめ

1. 考えたいアルゴリズムを，言葉，図，フローチャート，擬似コード，と段階的に表現していくことで，徐々に具体的に表現できていくことを理解できた．
2. 逐次実行，条件分岐，反復，などの制御構造については特に気をつけて具体的に記述すべきであることが理解できた．

2.8 Python 演習

本章の Python 演習では，制御構造の「逐次実行」「条件分岐」「反復」について Python プログラムとして実現する方法を説明する．

2.8.1 逐次実行

Python では，インタプリタ上の入力でも Python プログラム（.py ファイル）でも，行ごとに命令が分かれていることが基本であり，上の行から順に命令を逐次実行していくことになっている．ただし複数の命令を;（セミコロン）で区切って 1 行に記述することもできる．

```
>>> x = 1 ; x = 2
>>> x
2
```

また，() や [] などのカッコ類の内部では，途中で改行しても命令は続いているものとみなされる．

```
>>> (1 +
... 2)
3
```

インタプリタ上で，改行しても命令が続いていると判断されている場合は，... が左端に表示される．インタプリタ上で複数行にわたる命令の終了を宣言するためには，最後の... の後でもう一度改行すればよい．

```
>>> def test(n):
...     return n+1
...
>>>
```

2.8.2 条件分岐

条件分岐を実現するには if 文を用いる．if 文には，キーワード else による else 節を付けることもできる．if 文の意味をフローチャートで説明した図が図 2.5 である．

キーワード if の直後の条件が成り立つ場合は，直後のインデント部分（ブロック A）が実行される．条件が成り立たない場合は，else 節があればブロック B が実行され，なければ何も実行されない．ブロックは複数行にわたっても構わない．

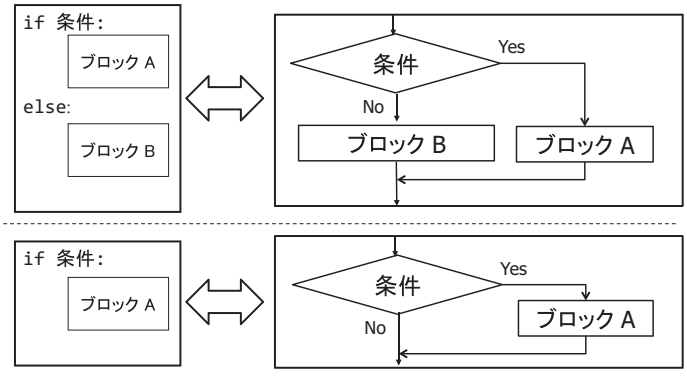


図 2.5　if 文のフローチャート

ここで，ブロックの中に `if` 文が含まれても構わない，ということに注意しよう．たとえば図 2.6 のように，点線部のブロック内に `if` 文を入れることもできる．その場合，インデントが深くなりプログラムが読みにくくなるため，`elif` というキーワードが用意されており，図 2.6 の最も左側の図のように単純に書き直すことができる．

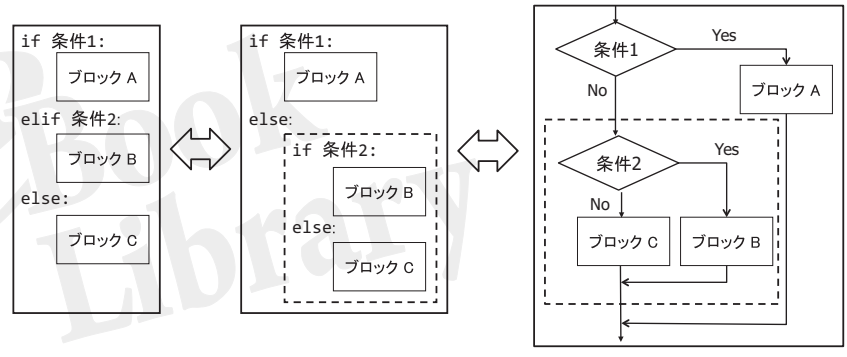


図 2.6　elif 節のフローチャート

2.8.3　反復

Python には，反復を実現する構文がいくつか用意されている．本節では `while` 文について説明する．`while` 文のフローチャートが図 2.7 である．

ここで，ブロック A が実行されるのは，条件が「成り立つ場合」であることに注意しよう．たとえば，図 2.3 で表現した整数の除法のフローチャートでは，「$n \times a > m$」が成り立たないときに「a を 1 増やす」という処理を繰り返している．これを `while` 文で表現するには，繰り返す条件として，逆の「$n \times a \leqq m$」という式に書き改める必要がある．記述した Python プログラムをソースコード 2.1 に示す．

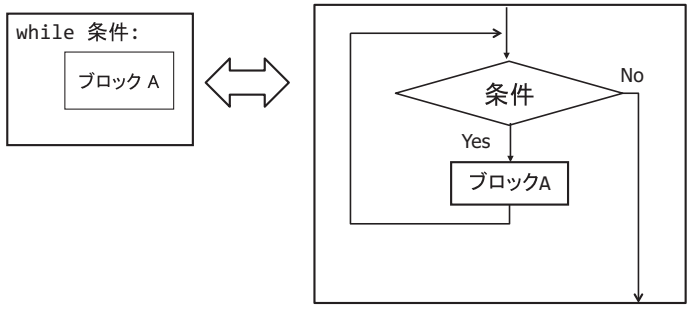


図 **2.7** while 文のフローチャート

ソースコード **2.1** division.py

```
1 def divide(m,n):
2     a = 1
3     while n * a <= m:
4         a = a + 1
5     return a - 1
```

このプログラムの動作を確認しよう．ただし，最終結果を出力させるだけでは，途中経過がアルゴリズムどおりかはわからない．そこで標準出力をさせる `print()` 関数とセミコロンを使って，`while` 文のブロックの中に変数 `a` の値を出力させる命令を追加したものがソースコード 2.2 である．

ソースコード **2.2** division_print.py

```
1 def divide(m, n):
2     a = 1
3     while n * a <= m:
4         print(a) ; a = a + 1
5     return a - 1
```

このプログラムの動作を確認してみると．以下のとおり，最終結果の 4 の前に 1 から 4 が出力されている．このことから，アルゴリズムどおりに a が 1 から順に増えていることがわかる．

```
>>> from division_print import divide
>>> divide(13, 3)
1
2
3
4
4
```

if 文や while 文の条件式の中で記述できる比較演算子を表 2.1 に示す.

表 2.1　Python の比較演算子

式	意味
x < y	x は y より小さい
x <= y	x は y より小さいか等しい
x > y	x は y より大きい
x >= y	x は y より大きいか等しい
x == y	x は y と等しい
x != y	x は y と等しくない
x is y	x は y と同じオブジェクト
x is not y	x は y と同じオブジェクトではない
x in y	x は y に含まれる
x not in y	x は y に含まれない

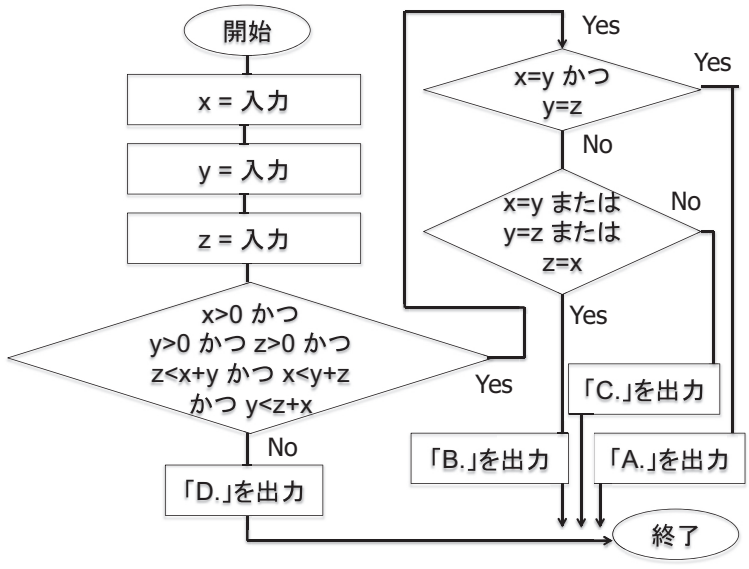
また, 条件式を組み合わせるために使用できる論理演算子を表 2.2 に示す.

表 2.2　Python の論理演算子

式	意味
x or y	x と y の少なくとも一つが True ならば True
x and y	x と y の両方が True ならば True
not x	x が False ならば True

2.9　練習問題正解例

練習問題 1 の正解例



練習問題 2 の正解例

1. 入力を x として受け取る
2. 入力を y として受け取る
3. 入力を z として受け取る
4. $x > 0$ かつ $y > 0$ かつ $z > 0$ かつ $z < x + y$ かつ $x < y + z$ かつ $y < z + x$ ならば，手順 6. に移る
5. そうでなければ，「D.」を出力して終了する
6. $x = y$ かつ $y = z$ ならば，手順 10. に移る
7. そうでなければ，$x = y$ または $y = z$ または $z = x$ ならば，手順 9. に移る
8. そうでなければ，「C.」を出力して終了する
9. 「B.」を出力して終了する
10. 「A.」を出力して終了する

3章　アルゴリズムを比べる方法

この章の目標

1. 整列のアルゴリズムにはどのようなものがあるか，複数の例を挙げられるようになる．
2. 入力の内容によって計算回数の大小が変わってくる場合の比較の仕方を説明できるようになる．

キーワード

選択ソート，最悪計算量，平均計算量

前章までで，二つのアルゴリズムを比較するためには，基本命令の計算回数などを基準とすればよいことがわかった．しかし，入力の内容によって計算回数が変化する場合は，どのように比較したらよいだろうか．本章では，整列のアルゴリズムを二つ紹介し，計算回数が変化する場合の比較の仕方を説明する．

3.1　整列（ソート）とは

並べ替えのことを整列（ソート）と呼ぶ．本章では理解しやすくするために「整数を昇順に並べる並べ替え」のみを扱う．

ここで**昇順 (ascending order)** とは，小さい値ほど先頭に近くなるように並べる方法で，**降順 (descending order)** とは大きい値ほど先頭に近くなるように並べる方法のことである．先頭とは左端のこととする．

記号で表すと，入力は正の整数 n と，n 個の整数からなる整数列 $A[0], A[1], ..., A[n-1]$ となる．

本書では列を 0 番目から数えることを基本とする．また，列には [ ] をつけて $[8, 4, 3, 9, 6]$ のように表す．したがってこの列を昇順に並べ替えると $[3, 4, 6, 8, 9]$ となる．

▶[並べ替えの可能性]
一般には，整数以外のものでも，全順序 (total order) が定義されていればどのようなデータ集合でも並べ替えることは可能である．全順序とは，二項関係 $\leq$ で任意の要素 x, y, z について，反射律 ($x \leq x$) と推移律（$x \leq y$ かつ $y \leq z$ ならば $x \leq z$）と反対称律（$x \leq y$ かつ $y \leq x$ ならば $x = y$）と完全律（$x \leq y$ または $y \leq x$）が成り立つもののことをいう．

▶[並べ替えの種類]
一般には，同じデータ集合でも大小の定義が異なれば並べ替えた結果も異なる．たとえばメールを並べ替えるにしても，受信日時の古い順に並べ替えるのか，サイズの小さい順に並べ替えるのか，などの基準によって結果が異なるということである．

3.2　選択ソートとは

本節では，整列アルゴリズムの基本的な例として選択ソートを説明する．**選択ソート (selection sort)** とは，図 3.1 のようなアルゴリズムである．

1. ソート対象の列の中から最小値を見つけ，その値と先頭を交換する．
2. 先頭を除いた残りの列をソート対象とし，1. に戻る．
   ソート対象の列の要素が 1 個になったら停止する．

図 3.1　選択ソートのアルゴリズム

たとえば，入力が $n = 5$ と列 $[8, 4, 3, 9, 6]$ の場合を手順を図 3.2 にように示す．本章では，アルゴリズムの計算量を測る手段として，アルゴリズム中で実行された比較と交換の回数に着目する．

このアルゴリズムのポイントは，手順の 1. で先頭に移動した最小値は，もう移動させる必要がないことである．なぜなら昇順では最小値は必ず先頭に来なければならないからである．その手順を残りの列にも適用し，「残りの中の最小値を残りの中の先頭と交換する」という手順を繰り返すというわけである．

注意しなければならないのは，図 3.2 の手順②である．ここでは偶然，先

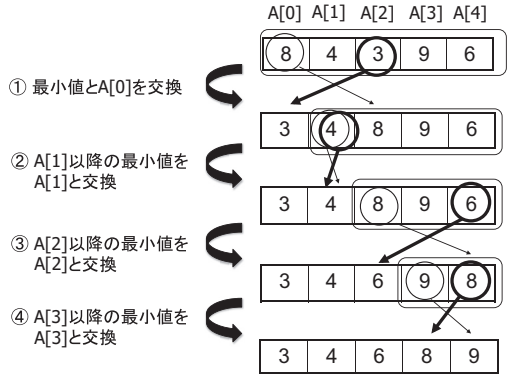


図 3.2　選択ソートの流れ（入力例 [8,4,3,9,6]）：丸印は比較対象，矢印は交換

頭の 4 自体が最小値でもあり，「先頭と最小値の交換」は不要であることは明らかである．しかし図 3.1 の選択ソートのアルゴリズムにはそのような例外処理は書かれていないので，ここでは「4 と 4 の交換」は行なったとし，交換の回数にも数える．

忘れてはならないのは，途中で必要となる「列から最小値を見つけるアルゴリズム」をどうするか，ということである．ここでは，スポーツ等で行われるトーナメント戦を参考にする．今回は整数の最小値を求めたいので，「整数と整数が比較という名の試合をして，小さい方が勝ち進む」というトーナメント戦を考え，図 3.3 のような偏ったトーナメントにする．この図の厳密なアルゴリズムの記述は省略する．

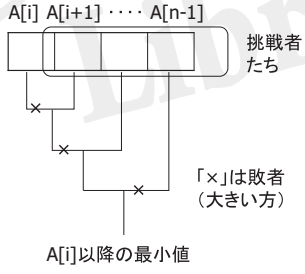


図 3.3　最小値の求め方の流れ

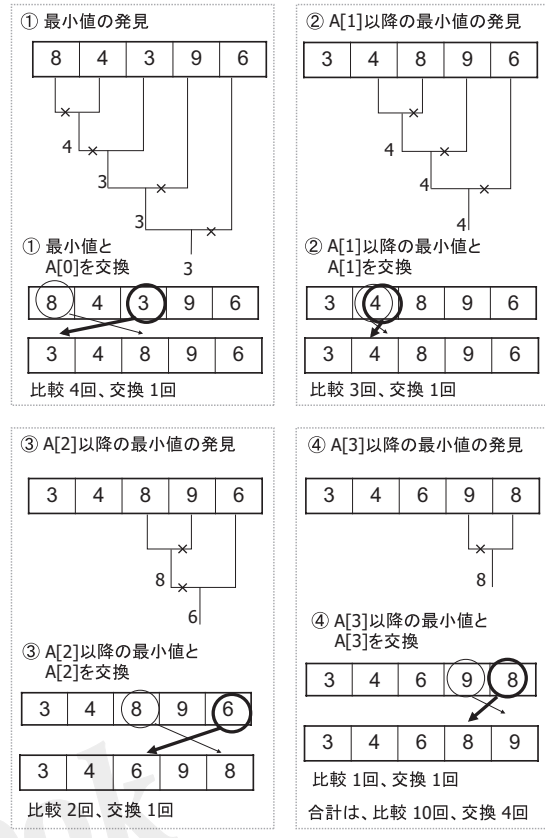
例題 3.1

図 3.2 の列 $[8, 4, 3, 9, 6]$ の選択ソートの①から④の経過を図に描け．また，①から④のそれぞれの中での「比較の回数」と「交換の回数」を答えよ．最後に，「比較の合計回数」と「交換の合計回数」も答えよ．

▶[不要な交換]
もちろん，「先頭が最小値でもある場合は交換しない」という例外処理を付け加えた新たなアルゴリズムを考えることはできるが，例外処理が増えすぎるとかえって計算時間がかかる可能性もある．

▶[偏ったトーナメント]
トーナメントの構造が平等でなくても優勝者が最小値には違いないので問題はない．こうなるとトーナメントというよりは挑戦者に対する暫定最小値の防衛戦とみなした方がよいかもしれない．不公平な形にした理由は，この方がアルゴリズムとして単純になるからである．また，下へ下へと加筆しやすいように図 3.3 は通常のトーナメントとは上下を逆にしたがアルゴリズムとしては関係ない．

例題 3.1 の正解例



3.3　選択ソートの随時交換版とは

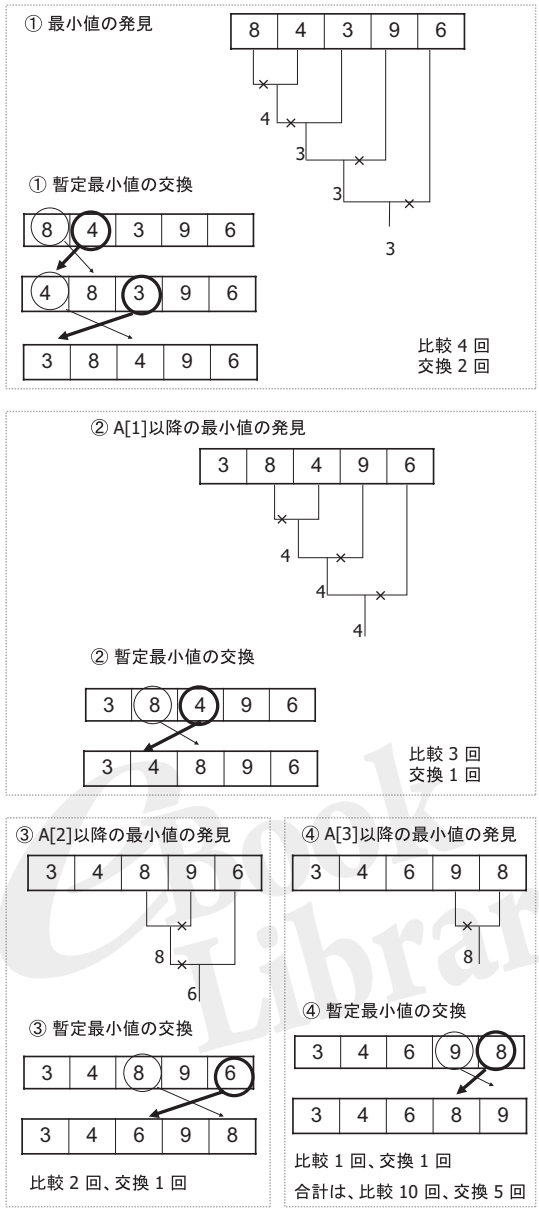
前節の例題 3.1 で選択ソートの比較と交換の回数を数えたが，列の長さが n であれば必ず，比較が合計 $1+2+\cdots+(n-1)=\frac{n(n-1)}{2}$ 回，交換が合計 $n-1$ 回になることに気づいたであろう．

次に，「先頭との交換を，最小値を求めた後ではなく，先頭より小さい数が見つかるたびに行う」という方針に変更したものを扱い，それを「選択ソートの随時交換版」と呼ぶことにする．こちらの比較と交換の回数はどうなるだろうか．

例題 3.2

列 $[8,4,3,9,6]$ の選択ソートの随時交換版の，図 3.2 の①から④の経過を図に描け．また，①から④のそれぞれの中での「比較の回数」と「交換の回数」を答えよ．最後に，「比較の合計回数」と「交換の合計回数」も答えよ．

例題 3.2 の正解例



次節の練習問題を解いてみると，随時交換版の交換の合計回数は入力データによって変わることがわかる．計算回数の最小値どうしで比較すると，随時交換版の方が少なくなり，最大値どうしで比較すると選択ソートの方が少なくなる．最大になる場合の計算量を**最悪計算量** (worst case complexity) と呼ぶ．アルゴリズムを評価する場合には「どんな場合の計算量を評価するか」ということまで決めなければならない．

▶[計算量の種類]
他に，入力の確率分布を想定して求める**平均計算量** (average case complexity) を用いることもある．ちなみに 8 章で述べる漸近的計算量を用いれば，選択ソートとその随時交換版のいずれも比較回数は $O(n^2)$ で交換回数は $O(n)$ なので同等といえる．

3.4　練習問題

1. 列 [9,2,7,4,5] の選択ソートの①から④の経過を図に描け．また，①から④のそれぞれの中での「比較の回数」と「交換の回数」を答えよ．最後に，「比較の合計回数」と「交換の合計回数」も答えよ．
2. 列 [9,2,7,4,5] の選択ソートの随時交換版の①から④の経過を図に描け．また，①から④のそれぞれの中での「比較の回数」と「交換の回数」を答えよ．最後に，「比較の合計回数」と「交換の合計回数」も答えよ．
3. 列 [2,4,5,7,9] の選択ソートの随時交換版の①から④の経過を図に描け．また，①から④のそれぞれの中での「比較の回数」と「交換の回数」を答えよ．最後に，「比較の合計回数」と「交換の合計回数」も答えよ．

3.5　3 章のまとめ

1. 整列のアルゴリズムにはどのようなものがあるか，複数の例を挙げられるようになった．

   - 選択ソート
   - 選択ソートの随時交換版

2. 入力の値によって計算回数が変化する場合があるため，アルゴリズムを評価する場合には最悪計算量や平均計算量など「どんな場合の計算量を評価するか」まで決めなければならない．

3.6 Python 演習

3.6.1 選択ソートの Python プログラム

本章で扱った選択ソートを Python プログラムとして実装したものをソースコード 3.1 に示す．実際に，このプログラムの関数 `sort` をインタプリタ上で実行すると，以下のように並べ替えを実行できる．

ソースコード **3.1** selection_sort.py

```
1  def sort(A):
2      for i in range(0, len(A) − 1):
3          select_min(A, i)
4
5  def select_min(A, i):
6      min = i
7      for j in range(i + 1, len(A)):
8          if A[min] > A[j]:
9              min = j
10     A[i], A[min] = A[min], A[i]
```

```
>>> from selection_sort import sort, select_min
>>> A=[8,4,3,9,6]
>>> sort(A)
>>> A
[3, 4, 6, 8, 9]
```

ソースコード 3.1 が，なぜ図 3.1 で示した選択ソートのアルゴリズムを実装していると言えるのか，確認していこう．まずは，ソースコード 3.1 の意味を「行単位で」日本語に直したものを図 3.4 に示した．Python のキーワードを確認しながら対応を理解していこう．

2 行目に現れる式 `len(A)` はリスト `A` の長さ（つまり本文中の n）を計算する式である．インタプリタで直接 `len` を使って動作を確認できる．

```
>>> len([8,4,3,9,6])
5
```

式 `range(x,y)` の値は `x` から `y-1` までの整数区間を意味するようなオブジェクトで，`range` 型に属する．`range` 型の値は 1 章で述べたようにイテラブルである．`y` までではなく `y-1` までを意味するので，式 `range(0,len(A)-1)` の値は `0` から `len(A)-2` までを意味する．

`for` 文は 2 章で述べた反復を実現するための文法の一つで，イテラブル

```
1   sort とは，引数 A について以下を行う関数とする．
2       i に，0 から len(A)-2 までを順に代入し，そのたびに
3           select_min(A,i) を実行する．
4
5   select_min とは，引数 A と変数 i について以下を行う関数とする．
6       min に i の値を代入する．
7       j に，i+1 から len(A)-1 までを順に代入し，そのたびに
8           もし A の min 番目より j 番目が小さいなら
9               min に j の値を代入する．
10      A の min 番目と i 番目を交換する．
```

図 3.4　selection_sort.py の意味

オブジェクトと組み合わせると簡単に反復を実現できることが特徴である．今回の表現「for i in イテラブルオブジェクト:」は，そのイテラブルオブジェクトの要素を順々に変数 i に代入し，そのたびに繰返し部分（インデントされている部分）を実行せよ，という意味になる．

```
>>> for i in range(0,4):
...     print(i)
...
0
1
2
3
```

これで Python プログラムと日本語との対応は，3 行目までは理解できただろう．この 1 行目〜3 行目の部分が，図 3.2 の流れ全体に対応していることを確認しよう．

3 行目に現れる関数 select_min は 5 行目以降によって定義されており，じつは，式 select_min(A, i) が「A の i 番目以降の中から最小値を見つけ，A の i 番目と交換する」という意味になるように意図されているのである．ということは，変数 i が表している値が，「最小値を探そうとしている範囲の先頭の位置」であればよい．このプログラムでは，0 から長さ −2（つまり最後尾の一つ前）までの値を順に代入しているので，まさに図 3.2 のとおりになっていることがわかる．

次に，Python プログラムと日本語との対応について，5 行目から確認を再開しよう．前述の range の解説を踏まえれば，7 行目の式 range(i+1,len(A))

の値が i+1 から len(A)-1 までの整数区間を意味することは明らかである．

8 行目に現れる if 文は，2 章で述べたとおり条件分岐を実現している．

10 行目では，代入文の左辺と右辺に複数の変数が用いられている．これは複数の代入文を同時に実行できる記法であり，これにより A[i] と A[min] を交換する命令を 1 行で書くことができている．実際は，右辺が複数の変数からなる代入は**タプルパッキング** (tuple packing) と呼ばれ，自動的にタプルにまとめて代入してくれる機能である．逆に，左辺が複数の変数からなる代入は**シーケンスアンパッキング** (sequence unpacking) と呼ばれ，右辺のシーケンスオブジェクトの要素を自動的に変数に分けて代入する機能である．今回のプログラムでは，このタプルパッキングとシーケンスアンパッキングを同時に使うことによって，複数の代入文を同時に実行できている．

▶[シーケンスアンパッキング]
ちなみにシーケンスアンパッキングは，シーケンスオブジェクトならタプル以外の型でも利用できる．

以上により，Python プログラムと日本語との対応については 10 行目まで確認できた．

5 行目以降で定義されている select_min(A, i) は，意図どおりに「A の i 番目以降の中から最小値を見つけ，A の i 番目と交換する」という意味になっている．というのも，変数 min が「A の i 番目以降の中の最小値が何番目にあるか」を意味しており，変数 j が図 3.3 の挑戦者の位置を表しているからである．j には i+1 から長さ −1 までの値を順に代入しているので図と合致している．

それでは改めて関数 sort の動きを確認してみよう．まず，3 行目の select_min(A, i) の右に;print(A) と書き足してみよう．ただし，Python プログラムのファイルを上書きした後は，Python インタプリタも再起動して import し直そう．これで select_min(A, i) の直後に A の内容を表示させることができる．以下のように図 3.2 と同じ列を入力してみると，途中で列が図 3.2 と同じように変化しているのを出力で確認できる．

```
>>> from selection_sort import sort
>>> A = [8,4,3,9,6]
>>> sort(A)
[3, 4, 8, 9, 6]
[3, 4, 8, 9, 6]
[3, 4, 6, 9, 8]
[3, 4, 6, 8, 9]
```

3.6.2 選択ソートの随時交換版の Python プログラム

選択ソートの随時交換版のプログラムは，選択ソートのプログラムを一

部書き換えてソースコード 3.2 のように書くことができる．「先頭との交換を，最小値を求めた後ではなく，先頭より小さい数が見つかるたびに行う」という方針なので，最小値の位置を表す変数 `min` は不要となる．代わりに `i` 番目が暫定最小値の位置を表し，`j` 番目の挑戦者に負けるたびに交換を実行するようになっている．

ソースコード 3.2　selection_sort_exchange.py

```
1 def sort(A):
2     for i in range(0, len(A) - 1):
3         select_min(A, i)
4
5 def select_min(A, i):
6     for j in range(i + 1, len(A)):
7         if A[i] > A[j]:
8             A[j], A[i] = A[i], A[j]
```

このプログラムの動作を確認するために，3 行目の `select_min(A, i)` の右に `;print()` と書き足し，8 行目の `A[j], A[i] = A[i], A[j]` の右に `;print(A)` と書き足してみよう．

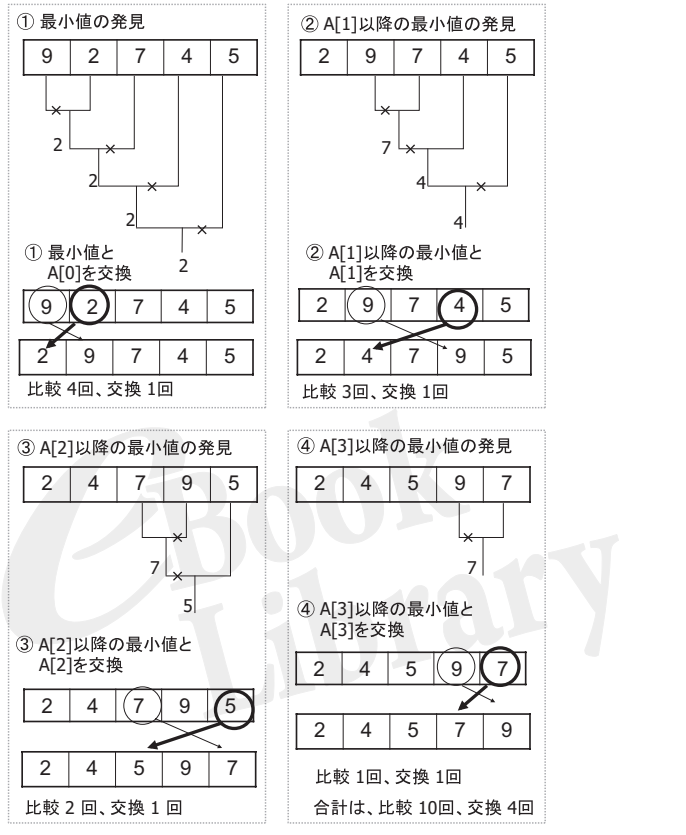
これで 8 行目で交換を行うたびに `A` の内容を表示させることができ，また最小値が確定するごとに改行を入れて出力を見やすくすることができる．以下のように例題 3.2 と同じ列を入力してみると，その正解例と同じように列が変化しているのを出力で確認できる．

```
>>> from selection_sort_exchange import sort
>>> A = [8,4,3,9,6]
>>> sort(A)
[4, 8, 3, 9, 6]
[3, 8, 4, 9, 6]

[3, 4, 8, 9, 6]

[3, 4, 6, 9, 8]

[3, 4, 6, 8, 9]

>>>
```

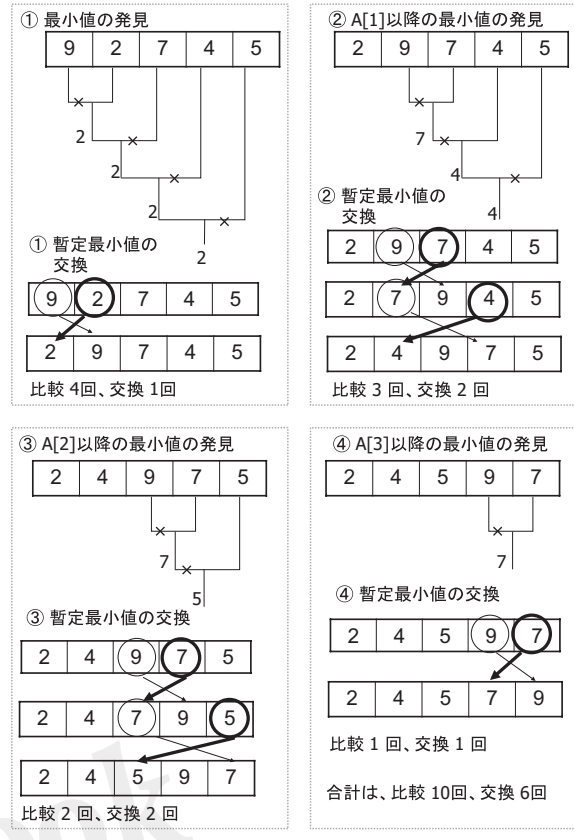
3.7　練習問題正解例

比較の合計回数は，選択ソートでも随時交換版でも，入力の長さが 5 であれば必ず $4+3+2+1=10$ で 10 回となる．一方，交換の合計回数は，選択ソートでは入力の長さが 5 であれば必ず 4 回だが，随時交換版では練習問題 2 のように多くなるときもあれば，練習問題 3 のように 0 回のときもあるということがわかる．入力が [9,7,5,4,2] と降順だった場合，随時交換版の交換回数は 10 回となることも想像できるだろう．

練習問題 1 の正解例



練習問題 2 の正解例



練習問題 3 の正解例