2019 年 3 月 20 日

『データ構造とアルゴリズム（データサイエンス大系）』

（川井明・梅津高朗・高柳昌芳・市川治 共著，学術図書出版社）

章末問題解答

第 2 章

2-1 フローチャート

a と b の大小関係を比較し出力するアルゴリズムである．$a = b$ の場合，分岐先は [出力：$b > a$] になるため，正しくない結果が出力される．修正する場合，[出力：$b > a$] を [出力：$b \geq a$] とすればよい．

2-2 フローチャート（**START 法**）

① 傷病者に〈歩行可能か〉を確認する．可能であれば，負傷の有無をチェックし，負傷の場合，緑グループに入れ，負傷がない場合，無傷のグループに入れる．歩行不可の場合②へ．

② 傷病者の〈呼吸有無〉を確認する．呼吸があれば③へ，呼吸がなければ，ただちに〈気道確保し，自発呼吸できるか〉確認する．自発呼吸ができなければ，黒グループに入れ，できる場合，赤グループに入れる．

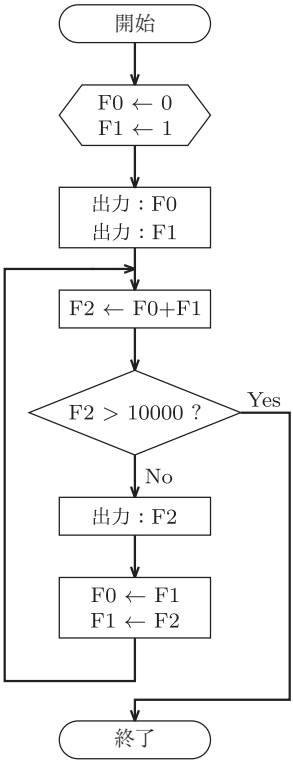
③ 傷病者の呼吸数が〈30 回/分以上か〉確認する．30 回/分以上の場合，赤グループに入れ，それ以下の場合④へ．

④ 傷病者の呼吸数が〈10 回/分未満か〉確認する．未満の場合，赤グループに入れ，それ以上の場合，⑤へ．

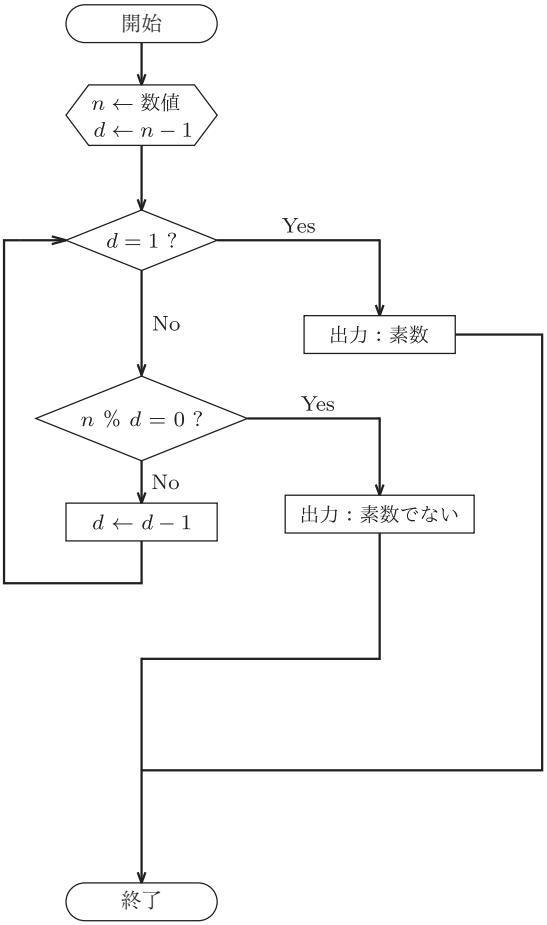
⑤ 傷病者の脈拍が〈120 回/分以上，または触れないか〉確認する．該当の場合，赤グループに入れ，該当しない場合⑥へ．

⑥ 傷病者に簡単な指示を与え〈応えられるか〉確認する．指示に応えられる場合，黄グループに入れ，応えられない場合は赤グループに入れる．

2-3 フィボナッチ数作成アルゴリズム



2-4 素数判定アルゴリズム



2-5 オーダーの計算

(1) $\mathcal{O}(N^2 + 2N + 1) = \mathcal{O}(N^2)$

(2) $\mathcal{O}(N(N+1)) = \mathcal{O}(N^2 + N) = \mathcal{O}(N^2)$

(3) $\mathcal{O}(N^2 + 2^N) = \mathcal{O}(2^N)$

N が大きくなると，N^2 より 2^N の方が大きくなるため．

(4) $\mathcal{O}(N^2 + N \log N) = \mathcal{O}(N^2)$

N が大きくなるにつれ，$\log N$ は N より小さくなっていく．そのため，$N \times N$ の方が大きくなる．

プログラム全体の計算量を考えた場合，前処理を行う部分と，目的とする処理を行う部分など，前半と後半の足し算などのような形で表されることが多々ある．そういった場合，最も大きい部分のオーダーがプログラム全体のオーダーとなる．それはすなわち，もしそのプログラムに対して計算時間を縮める改良を加える必要がある場合，一般に，オーダーが高くなっている部わから着手すべきだということを意味する．

2-6 **オーダーの計算**

(1) $\mathcal{O}(N)$

すべてのデータを少なくとも 1 回ずつ確認しないと，最低平均気温を見つけることはできない．どこかを読み飛ばした場合，読み飛ばしたところに真の最低気温が見落とされている可能性が常にある．実用的には，夏場のデータなど，そこに最低平均気温が紛れ込んでいることはなさそうなところを読み飛ばすという方針があり得るかもしれないが，外れ値を見落とす危険性とのトレードオフになる．なお，そのようにして仮にデータの半分を読み飛ばしても，オーダーは $\mathcal{O}(N/2)$ で，これは結局，$\mathcal{O}(N)$ のままである．

(2) $\mathcal{O}(\log N)$

37 ページの例題 2.3(3) と同様の議論が成り立ち，このアルゴリズムは二分探索と呼ばれる (第 6 章「6.3 二分探策」を参照)．

N 人分のデータから，X 点を取った者がいるかどうかを調べるとする．N が奇数ならちょうど真ん中，N が偶数ならちょうど真ん中の 2 つの内のいずれかの値を M とする．M と X を比較して $X = M$ ならば，X が見つかったので探索終了となる．そうでないなら，$X < M$ ならば，X があるとすれば M より前，$M < X$ ならば，X があるとすれば M より後だとわかる．どちらにしても，探すべき範囲は $N/2$ より小さくなる．よって，高々 $\log_2 N$ 回の比較で探索は完了する．

(3) $\mathcal{O}(\log N)$

直感的には，数え上げるのに $\mathcal{O}(N)$ かかりそうに思える．たとえ

ば，もし，N 人全員が 50 点のときに，50 点を取った人数を調べようとすると，N 人全員を数えることになり，N 手必要，など．しかし，データがソートされているのであれば，もっと効率的なやり方がある．

前述の二分探索を使えば，N 個のデータの内，ある値 X が最初に現れる順番 S と，最後に現れる順番 E のどちらもそれぞれ $\mathcal{O}(\log N)$ の手数で調べられる．そして，データはソートされているので，S 番目から E 番目はすべて X で，それ以外には X はない．よって，X の個数は，$E - S + 1$ 個だと求まる．

(4) $\mathcal{O}(\log N)$

ある階級の最小値，最大値が与えられたとき，ソート済みのデータのどこからどこまでがその階級に属するかは，(3) と同様に $\mathcal{O}(\log N)$ の手数の探索 2 回で求められる．100 点満点のテストなので，階級幅を 1 としても，階級の個数は 0 点から 100 点までの 101 個しかないため，度数分布表の作成には，$\mathcal{O}(\log N \times 2 \times 101)$ の手数がかかる．なお，ソートされていないデータの度数分布表の作成の場合にはまた話が違ってくる点は，注意が必要な落とし穴だ．第 7 章で紹介するとおり，ソートには $\mathcal{O}(N \log N)$ の手数が掛かる．よって，ソートされていないデータをソートしてから上述の方法で度数分布表を作った場合には，$\mathcal{O}(N \log N + \log N)$ で，トータルとしては $\mathcal{O}(N \log N)$ の手数が掛かる．しかし，ソートされていないデータを順に見ていって，各階級に属する個数を適切なデータ構造とアルゴリズムを用いて数え上げていくような方法を使うと，$\mathcal{O}(\log N)$ で済み，こちらの方が速い．

データの状態や，何をどこまでやるかに依って，最適なアルゴリズムが変わってくることはよくある話なので注意されたい．

2-7 ハノイの塔

上から N 段分の円盤を移動させることを考える．説明のため，移動させたい N 段分が刺さっている柱を「ここ」，それらを移したい先の柱を

「そこ」，そのいずれでもないもう 1 本の柱を「よそ」と呼ぶことにする．ヒントの通り，$N \geq 2$ のとき，$N-1$ 段のハノイの塔を最小の手数で解く手順がわかっていたと仮定する．すると，上から N 段分の円盤を動かす手順は以下のようになる．

1. 上から $N-1$ 段分をいったん，ここからよそへと避ける
2. N 段目の円盤を，ここからそこへと移す
3. よそに避けておいた上から $N-1$ 段分を，そこに移す

この手順が，N 段分の円盤を移動させる最小の手順である．N 段目の円盤を動かすためには，その上に乗っているすべての円盤をいったんどこかへ避ける必要があるため，この手順の 1 は避けられない．また，2 で N 段目の円盤を移した後，その上に乗っていた $N-1$ 段分を戻してやる必要があり，3 の手順も必要である．そして，手順通りに実施することで，N 段分の円盤を移動させられることから，この手順が最小だと示せる．さて，ヒントの通り，N 段のハノイの塔を解く最小手順を $h(N)$ とすると，上記の手順で N 段分を移動させるのに必要な手数は，$h(N-1)$ を用いて以下のように書ける．

$$h(N) = h(N-1) + 1 + h(N-1) = 2h(N-1) + 1$$

また $h(1)$ は明らかに 1 なので，以下のような漸化式を解けばよい．

$$h(1) = 1$$
$$h(N) = 2h(N-1) + 1$$

式を変形すると，

$$h(1) + 1 = 2$$
$$h(N) + 1 = 2\{h(N-1) + 1\}$$

よって，

$$h(N) = 2^N - 1$$

求めるオーダーは，

$$\mathcal{O}(h(N)) = 2^N$$

ちなみにプログラムの形でこのアルゴリズムを書き下すと以下のようになる.

```
# ハノイの塔の解法を表示する関数
# koko から soko へ n 段を移動させる手順を表示する.
# koko と soko には，1，2，3 の整数値のいずれかを指定する
def solve_Hanoi(koko, soko, n):
    # yoso の値として，1，2，3 のうち，koko でも soko でもないものを選ぶ
    if koko == 1:
        if soko == 2:
            yoso = 3
        else:
            yoso = 2
    elif koko == 2:
        if soko == 1:
            yoso = 3
        else:
            yoso = 1
    else:
        if soko == 1:
            yoso = 2
        else:
            yoso = 1

    # 2段以上の場合は,いったん,n-1段分をyoso へ避ける
    if n > 1:
         solve_Hanoi(koko, yoso, n - 1)

    # 円盤n を，koko から soko へと移す
    print("柱{}から柱{}へ円盤{}を移す".format(koko, soko, n))

    # 2段以上の場合は,避けてあったn-1段分を
    # soko へ移動した円盤 n の上に移動させる
    if n > 1:
         solve_Hanoi(yoso, soko, n - 1)

# 柱 1から柱 2へ移動する，5段のハノイの塔を解かせてみる
solve_Hanoi(1, 2, 5)
```

この解法を可視化したければ，以下のようなプログラムで実現できる．

```
N = 5

# 柱と円盤の状態を初期化
# 1の柱にだけN〜1 のサイズの円盤が突き刺さっている状態にする
posts = {
    1: [i for i in range(N, 0, -1)],
    2: [],
    3: []
}

# 柱と円盤の状態を表示
def output_state():
    # 上から順に柱と円盤の状態を表示
    for y in range(N, 0 - 1, -1):
        # y 段目の状態を表示
        print("␣", end = "")
        # それぞれの柱についてy 段目の状態を表示
        for post in [1, 2, 3]:
            # y 段目に円盤が刺さっていれば size をその円盤のサイズに
            if y < len(posts[post]):
                size = posts[post][y]
            else:
                size = 0
            # 柱を「|」,円盤を「|」の両側にsize 個の「=」として表示する
            #「|」を揃えるため,N - 円盤のsize 分の空白を前後に付ける
            disc = "=" * size
            space = "␣" * (N - size)
            print(space, disc, "|", disc, space, sep = "",
                end = "")
        print()
    #「#」を必要個数並べて地面を表現する
    print("#" * (N * 3 * 2 + 4))
    print()
```

```
33
34 # 円盤n を柱 koko から，柱 soko へと移す
35 def move_disc(koko, soko, n):
36     print("柱から柱へ円盤を移す{}{}{}".format(koko, soko, n))
37     size = posts[koko].pop()
38     # 念のためのチェック．バグがなければ，この
           if 文の条件が成立することはない
39     if size != n:
40         raise Exception("想定しない円盤が刺さっています
               :␣{},␣{}".format(size, n))
41     posts[soko].append(n)
42
43     output_state()
44
45 def solve_Hanoi(koko, soko, n):
46     if koko == 1:
47         if soko == 2:
48             yoso = 3
49         else:
50             yoso = 2
51     elif koko == 2:
52         if soko == 1:
53             yoso = 3
54         else:
55             yoso = 1
56     else:
57         if soko == 1:
58             yoso = 2
59         else:
60             yoso = 1
61
62     if n > 1:
63          solve_Hanoi(koko, yoso, n - 1)
64
65     move_disc(koko, soko, n)
66
67     if n > 1:
68          solve_Hanoi(yoso, soko, n - 1)
```

```
69
70 # 柱 1から柱 2へ移動する，5段のハノイの塔を解かせてみる
71 output_state()
72 solve_Hanoi(1, 2, n)
```

第 3 章

3-1 配列とリンクリスト

配列．データの型が単純で，数も既知の場合，配列を用いるべき．メモリの浪費が少なく，かつ高い演算効率で操作できる．

3-2 配列とリンクリスト

リンクリスト．データの型が複雑で，数もわからない場合，リンクリストを用いるべき．複雑なデータ構造を実現できるほか，必要な分だけメモリを使うため，メモリの浪費が少ない．

3-3 配列 (リスト) の操作

```
1  a = []
2  while True:
3      b = int(input())
4      if b == -1:
5          break
6      a.append(b)
7  print(max(a))
8  print(min(a))
9  print(sum(a))
10 print(sum(a)/len(a))
```

3-4 リンクリストのポインタ

```
1  class Llist:
2      def __init__(self, x):
3          self.data = x
4          self.preppt = None #後戻りポインタ
5          self.nextpt = None
6
7  A1 = Llist(1)
```

```
A2 = Llist(2)
A1.nextpt = A2
A2.preppt = A1 #後戻りポインタにアドレス格納
print(A1.data, A1.nextpt.data)
print(A2.preppt.data, A2.data)
```

3-5 文字列の辞書順比較

以下のようなプログラムを書いて試してみるとよい．

```
word1 = input("1つ目の単語を入力して下さい>␣")
word2 = input("2つ目の単語を入力して下さい>␣")
if word1 > word2:
    earlyer = word1
else:
    earlyer = word2
print("{}の方が先.先頭の文字コードは{}と{}".format(earlyer
    , ord(word1[0]), ord(word2[0])))
```

たとえば，apple と Banana を比較すると以下のようになる．

```
1つ目の単語を入力して下さい> apple
2つ目の単語を入力して下さい> Bababa
Bababa の方が先．先頭の文字コードは 97 と 66
```

これは，63 ページのコラム「辞書順の罠 1」で紹介した通り， 文字コードがおおむね，数字 → 大文字アルファベット → 小文字アルファベット → ひらがな → カタカナ → 漢字，の順に小さい番号から振られているためである． 大文字，小文字を区別せずに並び替えたい場合などには，すべてを大文字か小文字に統一してから並び替えるなどの工夫が必要となる．

もう 1 つ注意が必要なのは，64 ページのコラム「辞書順の罠 2」で紹介した，文字列として表現された桁数の異なる数字を比較する場合である．以下のように，数値としては 3 の方が 100 より小さいが，文字列として比較すると 1 文字目の 3 と 1 の文字コードが比較され，このような結果になる．

```
1つ目の単語を入力して下さい> 100
2つ目の単語を入力して下さい> 3
100の方が先.先頭の文字コードは 49と 51
```

3-6 ハミング距離の計算

プログラムは以下のように書ける．ここでは，文字列の長さが一致せず，どちらかが短いような場合には，短い方の文字列が終わって以降の文字は一致しないと見なすようにしてある．

```
a = "science"
b = "scene"

# ハミング距離を 0に初期化
humming_dist = 0

# 1文字目から順に比較
i = 0

# どちらかの単語の末尾までまだ到達していなければ
while i < len(a) or i < len(b):
    # どちらかの単語の末尾を越えているか,
        i 文字目が一致しなければハミング距離を 1 増やす
    if i >= len(a) or i >= len(b) or a[i] != b[i]:
        humming_dist += 1
    i += 1
print(humming_dist)
```

3-7 編集距離の計算

```
str_a = input("1つ目の文字列>␣")
str_b = input("2つ目の文字列>␣")

# 何度も使うので入力された文字列の文字数を変数に覚えておく
len_a = len(str_a)
len_b = len(str_b)

# len_a 列 × len_b 行の 2次元リスト（表),
    d_table を作る．すべての要素は 0としておく
```

```
d_table = []
for i in range(len_b + 1):
    d_table.append([0] * (len_a + 1)) # 配列*整数値, でその配列を整数値回繰り返した配列が得られる

# 編集距離が自明な部分を初期化
#（表の最初の列と最初の行は, 文字を追加する処理のみでたどり着ける）
for i in range(len_b + 1):
    d_table[i][0] = i
for j in range(len_a + 1):
    d_table[0][j] = j

# 表の残りの部分は左上から順に埋めて行く
for i in range(1, len_b + 1):
    for j in range(1, len_a + 1):
        # 上, 左の交点経由での, その交点までの距離
        d_from_top = d_table[i][j - 1] + 1
        d_from_left = d_table[i - 1][j] + 1

        # 左上の交点経由での, その交点までの距離
        if str_a[j - 1] == str_b[i - 1]:
            # 削除・挿入する文字が同じ場合は手数は不要なので増やさない
            d_from_top_left = d_table[i - 1][j - 1]
        else:
            # 異なる場合は置換が必要で 1手増える
            d_from_top_left = d_table[i - 1][j - 1] + 1

        # 3つの内, 最小のものがその交点までの最短距離になる
        d_table[i][j] = min([d_from_top, d_from_left, d_from_top_left])

print("各交点までの距離の表は")
for d_row in d_table:
    print(d_row)

print("編集距離は", d_table[len_b][len_a])
```

このプログラムはわかりやすさを重視して，効率を犠牲にしている．各行ごとに d_table の中を埋めていく計算は，その直前の行のデータだけがあれば十分である．よって，1 行計算するごとにそれより前の部分のデータを棄てても大丈夫なので，編集距離だけが必要な場合，表全体を保持するメモリは必要ではない．高々 2 行分のメモリがあれば算出できる．

第 4 章

4-1 スタックに対する操作

(1)

```
5
4
3
2
1
```

(2)

```
# ソースコード (4.1(2)の解答)
A = []
push(A,1)
print(pop(A))
push(A,2)
push(A,3)
print(pop(A))
push(A,4)
push(A,5)
print(pop(A))
print(pop(A))
print(pop(A))
```

4-2 スタックを使った加減算

考え方：

オペランド (演算対象) を保存するスタックに対して pop 操作を行うと，式の後方オペランドが取り出される．連続 2 回 pop すれば，2 個のオペランドが取り出され，二項演算が可能となる．

オペレータ (演算子) スタックに対して pop 操作を行えば，前述 2 個のオペランドのための演算子が取り出される．

問題は，さらに 1 個前方のオペレータが”−” の場合を考えなければならない．

すなわち，$a-b+c$ の場合，上記の操作では，取り出された 2 個のオペランドは b,c で，$b+c$ を足し算するわけにはいかない．

ここでの工夫は，オペレータスタックに対する「先読み」である．つまり，オペレータスタックに対して 2 回目 pop を行い，b の前のオペレータを確認する．それが”−” であれば，b を $-b$ にし，$-b+c$ を演算すればよい．ここで 1 点注意してほしい．先読みした”−” をすでに b に適用したため，スタックに戻すときに”+” にしなければならない．

以下のソースコードを参考にしてください．

```
# ソースコード (4.2の解答)
A = []   # オペランドを覚えるためのスタック
B = []   # オペレータを覚えるためのスタック

while True:  # =が入力されるまで式をスタックに読み込み続ける
    n = input()
    if n == "=":
        break
    if n == "+" or n =="-":
        push(B, n)
    else:
        push(A, n)

while True:
    operand2 = int(pop(A)) # 最後尾のオペランドを取り出す
    operand1 = int(pop(A)) # その次のオペランドを取り出す
    operator = pop(B)  # 最後尾のオペレータを取り出す
    op_next = ""  # オペレータを先読み用の変数を初期化
    if not isEmpty(B):  #
        Bスタックが空でなければオペレータを先読み
        op_next = pop(B)
        push(B, "+")  # 先読みしたら戻す,戻すオペレータは”+”

    if op_next=="-":  # 先読みしたオペレータが”-”なら,
```

```
        operand1 符号を逆
24      operand1=-operand1
25
26  if operator == "+":  # 足し算
27      operand1 = operand1 + operand2
28  else:    # 引き算
29      operand1 = operand1 - operand2
30
31  if not isEmpty(B):  # オペレータスタックが空でなければ
        ,繰り返す
32      push(A, operand1) # 演算した結果をオペランドスタッ
            クに戻す
33  else:    # オペレータスタックが空であれば,式計算終了
34      print(operand1)  # 結果を出力してブレイク
35      break
```

4-3 キューに対する操作

(1)

```
1 1
2 2
3 3
```

(2)

```
1 1
2 2
3 3
```

第 5 章

5-1 略

5-2 略

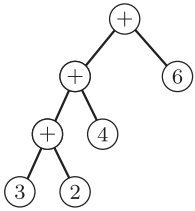
5-3

```
1 class Node:
2     def __init__(self, label):
3         self.label = label
4         self.left = None
5         self.right = None
6         self.super = None  #親へのポインタを追加
7
```

```
8  Node_A = Node(1)
9  Node_B = Node(2)
10 Node_A.left = Node_B
11 Node_B.super = Node_A  #.super に親のアドレスを保存
12 print(Node_A.left.label)
13 print(Node_B.super.label)
```

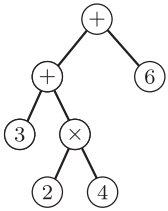
5-4 (1) $3+2+4+6$ (中置記法)

$32+4+6+$（RPN）



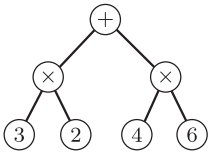
(2) $3+2\times4+6$ (中置記法)

$324\times6+$（RPN）



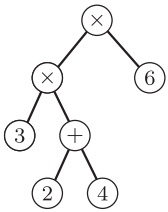
(3) $3\times2+4\times6$ (中置記法)

$32\times46\times+$（RPN）



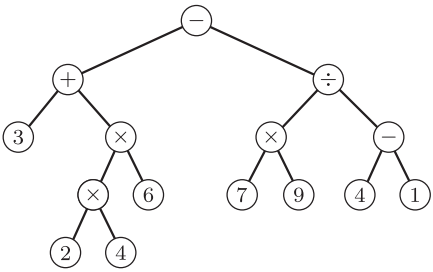
(4) $3 \times (2 + 4) \times 6$ (中置記法)

$324 + \times 6 \times$（RPN）



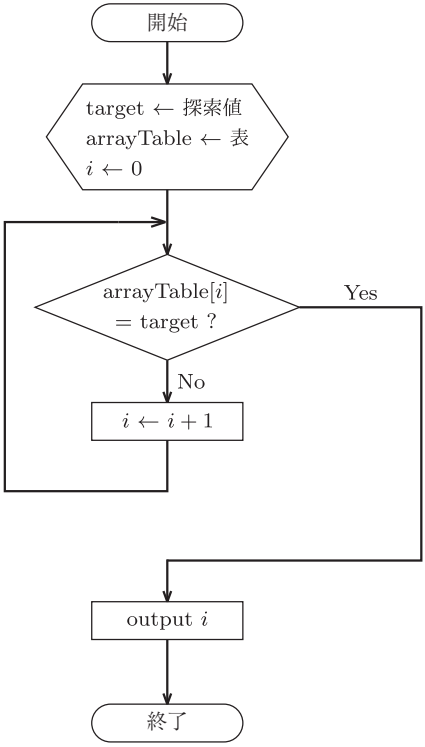
(5) $3 + 2 \times 4 \times 6 - 7 \times 9 \div (4 - 1)$ (中置記法)

$324 \times 6 \times +79 \times 41 - \div -$（RPN）

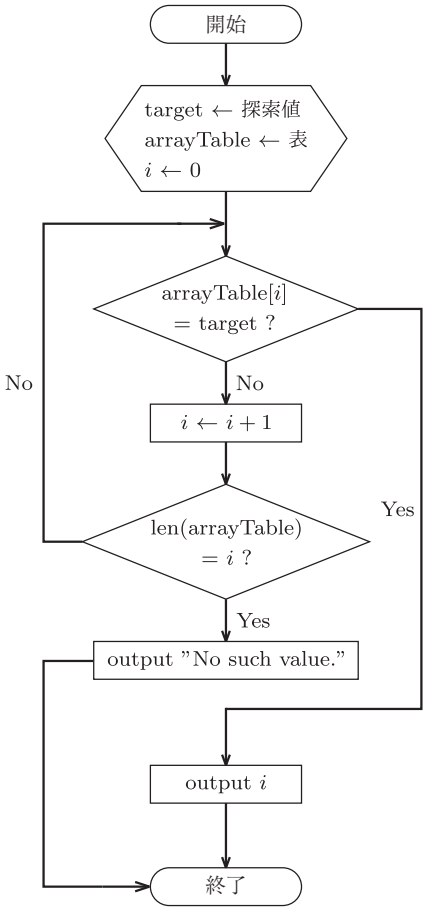


第 6 章

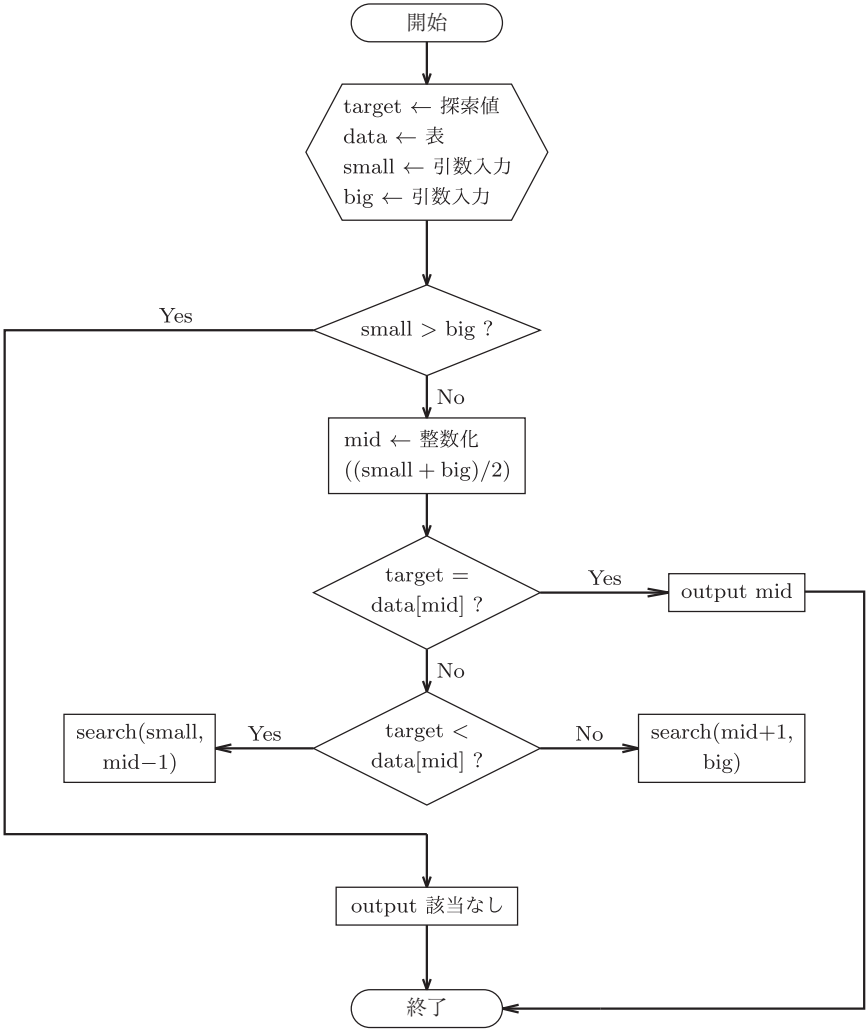
6-1 ソースコード 6.1 を例に作ったフローチャート



ソースコード 6.1 を改善後 (***6-3***) を例に作ったフローチャート



6-2 ソースコード 6.2 の search 関数を例に作ったフローチャート.
再帰呼び出しを行うプログラムの制御流れが複雑になることが多く，詳細なフローチャートは書きにくい場合が多い.



6-3 表を走査するインデックス i の最大値は len(arrayTable)−1 である．表の走査が完了した直後に i の値が len(arrayTable) と同値になるため，このタイミングを検出して 'No such value.' を出力すればよい．

```
1 arrayTable = [1, 2, 3, 4, 5] # 表を作成
2
3 target = 6 # 探索する値
4 i=0
```

```
while(i<len(arrayTable)):
    if(arraryTable[i]==target):
        print('Key␣(', target, ')␣has␣been␣found,␣index
            :', i)
        break;
    i+=1
    if(i==len(arrayTable)):
        print('No␣such␣value.')
```

6-4 while 文で走査開始するため，その前にカウンター変数 count を設け，if 文で比較する前にカウンターを 1 加算し，走査完了後，count 値を出力すればよい．ただし，if 文の後にカウンター加算文を置くと，検索成功時 break 文が実行されるため，カウント回数が足りなくなることを注意してほしい．

```
arrayTable = [1, 2, 3, 4, 5] # 表を作成

target = 3 # 探索する値
i=0
count=0
while(i<len(arrayTable)):
    count+=1
    if(arraryTable[i]==target):
        print('Key␣(', target, ')␣has␣been␣found,␣index
            :', i)
        break;
    i+=1
print('探索回数:', count)
```

6-5 ***6-4*** では，count 変数を外で設けたが，外部で宣言した変数を関数の中でアクセスするとエラーになる．グローバル変数などで対応することも可能であるが，バグの温床になるため推奨しない．この例では，外部で count をリストとして宣言し，探索開始前に append(0) でカウンター値をセットする．関数 search が呼び出されるたびに count[0] を加算し，探索終了後にカウンター値を出力するとともに削除する．ここで注意してほしいことは，探索終了後にカウンター値を削除しなければ，2 回目以降

の検索回数のスタート値になってしまうことである．

```
data = [1, 3, 5, 6, 7, 8, 9, 10, 11, 13]
start = 0
end = 9
count=[]

def search(small, big):
    if(small > big):
        return
    count[0]+=1
    mid=(small+big)//2
    if(target == data[mid]):
        return mid
    elif(target<data[mid]):
        return search(small, mid-1)
    else:
        return search(mid+1,big)

while(True):
    target = int(input('Input␣number␣(End␣=␣-1):'))
    if(target==-1):break

    count.append(0)
    print('Searching', target, 'in', data)
    result=search(start, end)
    print('探索回数:', count.pop())
    print('Key␣(', target, ')␣has␣been␣found,␣index:',
          result)
```

6-6 素数かどうかを判定する関数 isPrime および素数リストを生成する関数 AssendPrime を先に用意し，線形探索の部分はおおむねソースコード 6.1 と似ている．

```
PrimeNum=[]

def isPrime(n):
    if n<2:
        return False
```

```
6     for i in range(2,int(n**0.5)+1):
7         if n%i==0:
8             return False
9     return True
10
11 def AssendPrime(start, end):
12     a=start
13     while(a<=end):
14         if isPrime(a):
15             PrimeNum.append(a)
16         a+=1
17
18 def LinearSearch(target):
19     count=0
20     i=0
21     while(i<len(PrimeNum)):
22         count+=1
23         if(PrimeNum[i]==target):
24             print('Key␣(', target, ')␣has␣been␣found,␣
                   index:', i)
25             print(count)
26             break;
27         i+=1
28         if(i==len(PrimeNum)):
29             print('No␣such␣value.')
30             print(count)
31
32 AssendPrime(0,100000)
33 while(True):
34     t=int(input('Input␣number␣(End␣=␣-1):'))
35     if(t==-1):
36         break
37     LinearSearch(t)
```

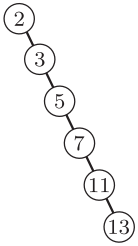
6-7

```
1 PrimeNum=[]
2 count=[]
3
```

```
4  def isPrime(n):
5      if n<2:
6          return False
7      for i in range(2,int(n**0.5)+1):
8          if n%i==0:
9              return False
10     return True
11
12 def AssendPrime(start, end):
13     a=start
14     while(a<=end):
15         if isPrime(a):
16             PrimeNum.append(a)
17         a+=1
18
19 def BinarySearch(target, small, big):
20     if(small>big):
21         return
22     count[0]+=1
23     mid=(small+big)//2
24     if(target==PrimeNum[mid]):
25         return mid
26     elif(target<PrimeNum[mid]):
27         return BinarySearch(target, small, mid-1)
28     else:
29         return BinarySearch(target, mid+1, big)
30
31
32 AssendPrime(0,100000)
33 while(True):
34     t=int(input('Input␣number␣(End␣=␣-1):'))
35     if(t==-1):
36         break
37     count.append(0)
38     print('Searching', t, 'in␣PrimeNum')
39     result=BinarySearch(t, 0, len(PrimeNum)-1)
40     print('探索回数:', count.pop())
41     if result==None :
42         print('No␣such␣value')
```

```
43      else:
44          print('Key␣(', t, ')␣has␣been␣found,␣index:',
                result)
```

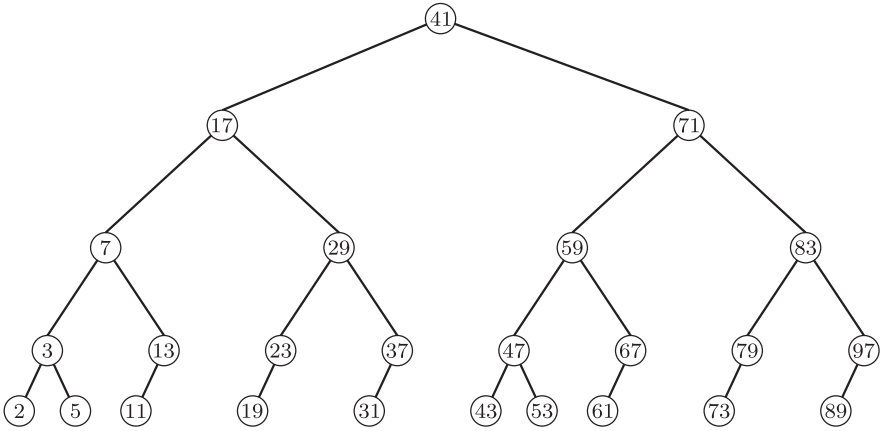
6-8 2 を根とする右への一本木になる．非常にバランスの悪い 2 分木である．下図は最初の 6 要素で作ったイメージである．



6-9 100 までの素数：

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]

完全二分木（狭義）の場合，ノード総数は $2^n - 1$ でなければならない．100 までの総数計 25 個のため，狭義の完全二分木を作成できない．最大レベルの節点が左詰めとなる完全二分木（広義）は下図である．

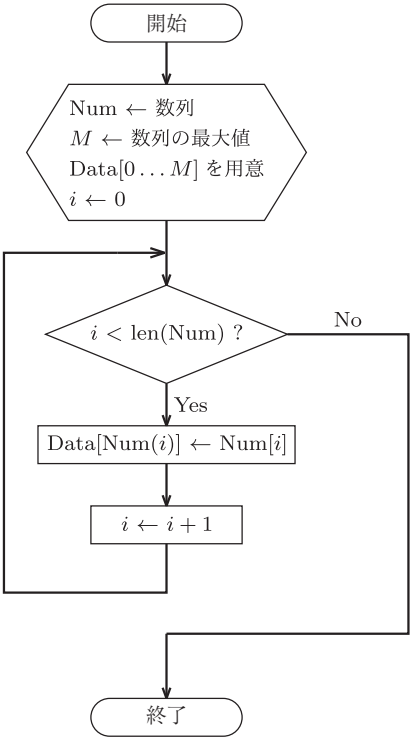


第 7 章

7-1 数列 A, B いずれもクイックソートにとって最悪な構造であるため，クイックソートとバブルソートと同じく $\mathcal{O}(N^2)$ の計算量となった．

	数列 A	数列 B	数列 C
バブルソート	190	190	190
クイックソート	190	190	78

7-2 バケツソートのフローチャート



7-3 バケツソートの時間計算量は $\mathcal{O}(N)$ である．数列を最初から最後まで 1 回だけの走査で終わる．

```
1 Num=[17, 3, 10, 9, 11, 4, 12, 5, 20, 7, 2, 19, 16, 1,
      6, 18, 14, 8, 13, 15]
2 M=20
3 Data = [-1]*(M+1)
```

```
count = 0

for i in range(len(Num)):
    count+=1
    Data[Num[i]]=Num[i]

print(count)
print(Data)
```

第 8 章

8-1 自作のハッシュ関数

以下のようなプログラムを実際に書いて試してみよう．

```
def my_hash(word):
    hash_value = 0
    for c in word:
        hash_value += ord(c)
    return hash_value

print(my_hash("まどか"))
print(my_hash("かまど"))
print(my_hash("かどま"))
print(my_hash("ひこね"))
print(my_hash("ひたち"))
print(my_hash("いわた"))
print(my_hash("かんご"))
print(my_hash("おやつ"))
```

このプログラムを実行すると，`my_hash()` で計算したこれらの単語のハッシュ値は，すべて一致する． 最初の 3 つに関しては自明で，文字コードを足しているだけなので，ひらがなの順序が入れ替わっても合計は変わらない． それ以外の単語については，偶然の一致だが，いずれかの文字コードが小さい分， 他の文字コードがちょうどそれを埋め合わせる分だけ大きいような関係が成立していて，合計が一致している (実のところ，本書執筆のため，`my_hash()` の値が一致する単語を列挙するプロ

グラムを作り，”ひこね”と一致するものを例示したのが章末問題のヒントである)．

Python の `hash()` で求めた場合は，それぞればらばらの値になる． 何かのアルゴリズムを実装していてハッシュ値が必要な場合は，Python の `hash()` 関数のように， きちんと設計されていると信用できるものを使うように気を付けられたい．

8-2 連鎖法を用いたハッシュテーブルの実装

まず，第 4 章の内容を元に，下記のような，最小限の機能だけを持つ Llist(リンクリスト) クラスを作る． このリンクリストは，要素を追加する `append()` のみを持ち， リンクリスト内の各要素は `Llistnode` クラスのオブジェクトとして保持する．

```
1  class Llistnode:
2      def __init__(self, x):
3          self.data = x
4          self.nextpt = None
5
6  class Llist:
7      def __init__(self):
8          self.head = None
9
10     def append(self, x):
11         new_head = Llistnode(x)
12         new_head.nextpt = self.head
13         self.head = new_head
14         return new_head
```

これを使って，題意を満たすようなハッシュテーブルは以下のように定義できる． キーと値のペアを記録するために `my_hash_node` クラスを使い，そのオブジェクトをキーのハッシュ値ごとに用意したリンクリストにデータを格納する． `my_get_node()` 関数で， 指定した `key` をキーに持つ

```
1  class my_hash_node:
2      def __init__(self, key, data):
```

```
3         self.key = key
4         self.data = data
5
6  # ハッシュテーブルの大きさ
7  my_hash_table_size = 10
8
9  # ハッシュテーブルを空のリンクリストで初期化
10 myhashtable = []
11 for i in range(my_hash_table_size):
12     myhashtable.append(Llist())
13
14 # このハッシュテーブルで使うハッシュ関数．0〜
       my_hash_table_size - 1の値を返す
15 def my_hash(key):
16     return hash(key) % my_hash_table_size
17
18 # 指定されたキーを持つリンクリスト，そのキーの直前の要素（なけ
       ればNone），そのキーと値のペアのオブジェクト（なければNone
       ）を探して返す
19 def my_get_node(key):
20     llist = myhashtable[my_hash(key)]
21     prev_pt = None
22     pt = llist.head
23     while pt != None:
24         if pt.data.key == key:
25             return (llist, prev_pt, pt)
26         prev_pt = pt
27         pt = pt.nextpt
28     return (llist, None, None)
29
30 # キーと値のペアをハッシュテーブルに追加する
31 def my_put(key, data):
32     llist, prev_pt, pt = my_get_node(key)
33     if pt != None:
34         # 既に
               key に対応する値が保存されていれば，それを上書き
35         pt.data.data = data
36     else:
37         # そうでなければリンクリストに値を追加
```

```
38          llist.append(my_hash_node(key, data))
39
40  # 指定したキーを持つデータが登録されているかどうか
41  def my_is_in(key):
42      llist, prev_pt, pt = my_get_node(key)
43      return pt != None
44
45  # 指定したキーを持つデータがあれば取得する
46  def my_get(key):
47      llist, prev_pt, pt = my_get_node(key)
48      if pt != None:
49          return pt.data.data
50      else:
51          return None
52
53  # 指定したキーと関連する値をハッシュテーブルから削除
54  def my_del(key):
55      llist, prev_pt, pt = my_get_node(key)
56      if pt != None:
57          # キーに該当するデータが存在すれば
58          if prev_pt == None:
59              # リンクリストの直前のデータがあれば,その「次の値」
                    を繋ぎ替え
60              llist.head = pt.nextpt
61          else:
62              # なければ（リンクリストの先頭なら),先頭を繋ぎ替
                    え
63              prev_pt.nextpt = pt.nextpt
```

8-3 単語の登場回数を数える

以下のようなプログラムで数えられる．ただし単純にスペースで区切られたものを単語としているため，このままでは，ピリオドやカンマも単語の一部として扱ってしまうなど，実用するにはいくらかの改善が必要である．プログラムを改善するか，読み込ませるデータを整形することにするか，どちらがよりよいやり方になるかは場合によって変わってくる．

```
file = open('source.txt')

table = {}
for line in file:
    for word in line.split():
        word = word.lower()
        if word in table:
            table[word] += 1
        else:
            table[word] = 1

for word in table:
    count = table[word]
    print("{}が{}回".format(word, count))
```

第 9 章

9-1 図 9.1

$V =$ { 東京, 名古屋, 米原, 京都, 新大阪, 長野, 金沢 }

$E =$ {(東京, 名古屋), (名古屋, 米原), (米原, 京都), (京都, 新大阪), (東京, 長野), (長野, 金沢), (金沢, 米原)}

図 9.2

$V =$ { 鈴木, 佐藤, 山本, 伊藤, 渡辺, 田中, 高橋 }

$E =$ {(鈴木, 佐藤), (鈴木, 山本), (鈴木, 伊藤), (佐藤, 山本), (佐藤, 伊藤), (山本, 伊藤), (伊藤, 渡辺), (渡辺, 田中), (渡辺, 高橋), (田中, 高橋)}

9-2 DAG の 2 つの条件

1. すべての辺が有向辺
2. 閉路を 1 つも有さない

のうち，条件 1 はすべてのグラフが満たしている．

2 の閉路について考えると（B）では GDP →輸入額→輸出額 という閉路が存在する．（A）と（C）については 1 つも閉路が存在しない．よって

（A）は DAG

（B）は DAG ではない（閉路が存在する）

（C）は DAG

9-3 (A)

隣接行列 $\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$

隣接リスト (重み w はすべて 1 なので省略)

$v = 0 \to v = 1$

$v = 1 \to v = 0 \to v = 2 \to v = 3$

$v = 2 \to v = 1 \to v = 3$

$v = 3 \to v = 1 \to v = 2$

(B)

隣接行列 $\begin{pmatrix} 0 & 1 & 2 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 3 & 0 \end{pmatrix}$

隣接リスト $v = 0 \to (v = 1, w = 1) \to (v = 2, w = 2)$

$\to (v = 3, w = 1) \to (v = 4, w = 1)$

$v = 1 \to (v = 0, w = 1) \to (v = 1, w = 1)$

$v = 2 \to (v = 0, w = 2) \to (v = 3, w = 1)$

$v = 3 \to (v = 0, w = 1) \to (v = 2, w = 1)$

$v = 4 \to (v = 0, w = 1) \to (v = 5, w = 3)$

$v = 5 \to (v = 4, w = 3)$

(C)

隣接行列 $\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$

隣接リスト (重み w はすべて 1 なので省略)

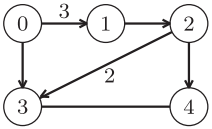
$v = 0 \to v = 1$

$v = 1 \to v = 2$

$v = 2 \to v = 3$

$v = 3 \to v = 0$

9-4 (A) 

(B) 

9-5 (1) 深さ優先探索：$0 \to 1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8$

幅優先探索：$0 \to 1 \to 2 \to 10 \to 3 \to 6 \to 8$

(2) 隣接行列と節点名の配列は

```
# グラフF の隣接行列
adjMat = [[0,1,1,0,0,0,0,0,0,0,1],
          [1,0,0,0,0,0,0,0,0,0,0],
          [1,0,0,1,0,0,1,0,0,0,0],
          [0,0,1,0,1,0,0,0,0,0,0],
          [0,0,0,1,0,1,0,0,0,0,0],
          [0,0,0,0,1,0,0,0,0,0,0],
          [0,0,1,0,0,0,0,1,1,0,0],
          [0,0,0,0,0,0,1,0,0,0,0],
          [0,0,0,0,0,0,1,0,0,1,1],
          [0,0,0,0,0,0,0,0,1,0,0],
          [1,0,0,0,0,0,0,0,1,0,0]]
# グラフF の節点名の配列
V = ['0', '1', '2', '3', '4', '5', '6', '7', '8',
    '9', '10']
```

深さ優先探索の実行

```
history = DepthFirstSearch(adjMat, 0, 8)
printHistory(history, V)
```

幅優先探索の実行

```
history = BreadthFirstSearch(adjMat, 0, 8)
printHistory(history, V)
```

(3) このグラフの最短経路は $0 \to 10 \to 8$ である．
深さ優先探索の場合，得られた探索結果から構築できる 0 と 8 の間を結ぶ経路は $0 \to 2 \to 6 \to 8$ であり，これは最短経路ではない．そもそも接点 10 は通過すらしていない．
幅優先探索の場合，$0 \to 10 \to 8$ の最短経路に沿った経路を通過している．