1-1 漸化式

例題 1 ${}_nC_r$ を求める

n個の中からr個を選ぶ組み合わせの数${}_nC_r$を求める.

たとえば，a，b，cという3個の中から2個選ぶ組み合わせは，ab，ac，bcの3通りある．一般に，n個の中からr個を選ぶ組み合わせの数を${}_nC_r$と書き，次の式で定義される．なお，$n!$は$n\cdot(n-1)\cdot(n-2)\cdots2\cdot1$という値．Cはの頭文字からとっている．

$$ {}_nC_r = \frac{n!}{r!(n-r)!} $$

この式で，このまま計算した場合は，大きなnの値に対し$n!$でオーバーフローする危険性がある．たとえば，

$$ {}_{10}C_5 = \frac{10!}{5!\cdot5!} = \frac{3628800}{120\cdot120} = 252 $$

となり，最終結果はオーバーフローしない値でも，int型なら10!のところでオーバーフローしてしまう．

$$ {}_nC_r \text{は} \begin{cases} {}_nC_r = \dfrac{n-r+1}{r}\,{}_nC_{r-1} & \text{（漸化式）} \\ {}_nC_0 = 1 & \text{（0次の値）} \end{cases} $$

という漸化式を用いて表現することでもできる．

漸化式とは，自分自身（${}_nC_r$）を定義するのに，1次低い自分自身（${}_nC_{r-1}$）を用いて表し，0次（${}_nC_0$）はある値に定義されているというものである．

こうした漸化式をプログラムにする場合は，繰り返しまたは再帰呼び出し（**第4章 4-1**）を用いて表現することができる．ここでは，繰り返しを用いて表現する．漸化式を繰り返しで表現する場合，0次の値を初期値とし，それに係数（$(n-r+1)/r$）をrの値を1から始め，繰り返しのたびに＋1しながら順次掛け合わせて行えばよい．

$$ {}_nC_r = 1 \cdot \underbrace{\frac{n-1+1}{1}}_{{}_nC_0} \cdot \underbrace{\frac{n-2+1}{2}}_{{}_nC_1} \cdot \underbrace{\frac{n-3+1}{3}}_{{}_nC_2} \cdots \frac{n-r+1}{r} $$

このような方法だと，かなり大きなnに対してもオーバーフローしなくなる．

❗ 一般の言語では整数型でのオーバーフローが起こるが，Pythonでは整数型でのオーバーフローは起きない．

プログラム Rei1

```
# ---------------------------
# *     漸化式（nCrの計算）     *
# ---------------------------

def combi(n, r):
    p = 1
    for i in range(1, r + 1):
        p = p * (n - i + 1) // i
    return p

for n in range(0, 6):
    result = ''
    for r in range(0, n + 1):
        result += '{:d}C{:d}={:<4d}'.format(n, r, combi(n, r))
    print(result)
```

実行結果

```
0C0=1
1C0=1   1C1=1
2C0=1   2C1=2   2C2=1
3C0=1   3C1=3   3C2=3   3C3=1
4C0=1   4C1=4   4C2=6   4C3=4   4C4=1
5C0=1   5C1=5   5C2=10  5C3=10  5C4=5   5C5=1
```

参考 print関数のendパラメータとf文字列リテラル

print関数はデフォルトで改行を行うが，endパラメータを使えば改行しないようにすることができる．しかし，endパラメータを認めていない処理系があるので，本書ではresult変数に出力結果を連結し，改行時にprint(result)を行う方式にしてある．

また，フォーマット済み文字列リテラル（f文字列リテラル）を使えば，format関数に似た結果が簡便にできる．しかし，f文字列リテラルを認めていない処理系があるので，本書ではformat関数を使用した．

例題1のプログラムをendパラメータさらにf文字列リテラルを使えば以下のようになる．

```
for n in range(0, 6):
    for r in range(0, n + 1):
        print(f'{n:d}C{r:d}={combi(n,r):<4d}', end='')
    print()
```

2-2 数値積分

例題 9 台形則による定積分

関数$f(x)$の定積分$\int_a^b f(x)dx$を台形則により求める.

関数$f(x)$の定積分を解析的（数学の教科書に出ている方法）に数式として求めるのではなく，微小区間に分割して近似値として求める方法を数値積分という.

台形則により，$\int_a^b f(x)dx$を求める方法を示す.

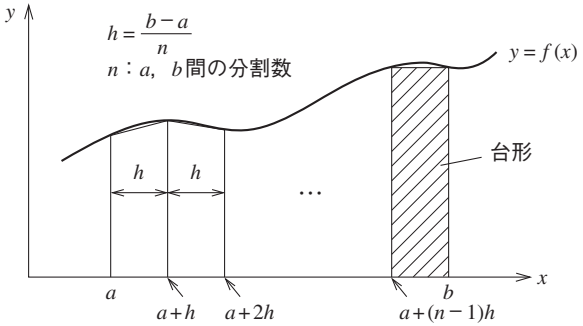

図 2.3 台形則

上図に示すように，a，b区間をn個の台形に分割し，各台形の面積を合計すると，

$$\int_a^b f(x)dx = \frac{h}{2}\{f(a)+f(a+h)\} + \frac{h}{2}\{f(a+h)+f(a+2h)\}$$
$$+\frac{h}{2}\{f(a+2h)+f(a+3h)\}+\cdots+\frac{h}{2}\{f(a+(n-1)h)+f(b)\}$$
$$= h\left\{\frac{1}{2}\{f(a)+f(b)\}+f(a+h)+f(a+2h)+\cdots+f(a+(n-1)h)\right\}$$

となる.

プログラム Rei9

```
# ----------------------------
# *       台形則による定積分       *
# ----------------------------

import math

def f(x):    # 被積分関数
    return math.sqrt(4 - x*x)

a, b = 0.0, 2.0    # 積分区間
n = 50             # a-b 間の分割数
h = (b - a) / n    # 区間幅
x, ds = a, 0.0

for k in range(1, n):
    x += h
    ds += f(x)
s = h * ((f(a) + f(b))/2 + ds)
print('   /{:f}'.format(b))
print('   |  sqrt(4-x*x) ={:f}'.format(s))
print('   /{:f}'.format(a))
```

実行結果

```
   /2.000000
   |  sqrt(4-x*x) =3.138269
   /0.000000
```

練習問題 9 シンプソン則による定積分

関数$f(x)$の定積分$\int_a^b f(x)dx$をシンプソン則により求める.

台形則ではx_0～x_2の微小区間を直線で近似しているのに対し，シンプソン則では2次曲線を用いて近似する.

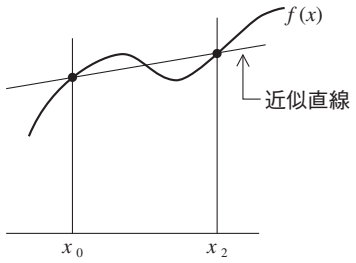

図 2.4 台形則

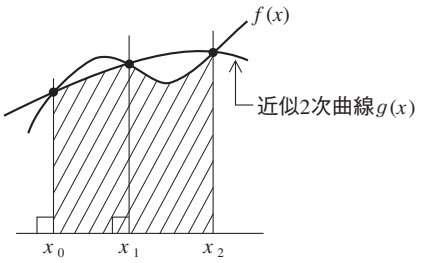

図 2.5 シンプソン則

3-2 シェル・ソート

例題 19 基本挿入法

基本挿入法により，データを昇順（正順）にソートする．

基本挿入法の原理は数列$a_0 \sim a_{n-1}$の$a_0 \sim a_{i-1}$がすでに整列された部分数列であるとして，a_iがこの部分数列のどの位置に入るかを調べてその位置に挿入することである．これをiを1から$n-1$まで繰り返せばよい．

a_iが部分数列のどこに入るかは，部分数列の右端のa_{i-1}から始め，a_iが部分数列の項より小さい間は交換を繰り返せばよい．

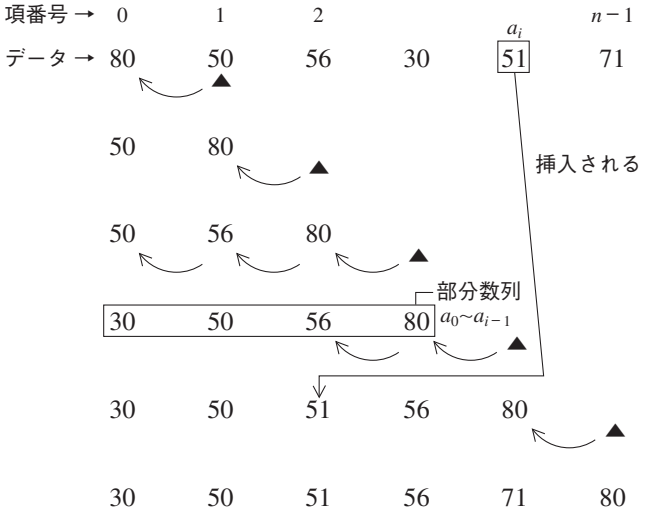

図 3.6 基本挿入法

たとえば，30，50，56，80という部分数列のどこに51が入るかを調べるには，まず80と比較し，51の方が小さいので80と51を変換する．次に56と比較し，51の方が小さいので56と51を交換する．次に50と比較し，51の方が大きいので51が入る位置はここということになり，このパスの処理を終了する．

プログラム Rei19

```
# ---------------------
# *      基本挿入法      *
# ---------------------

import random

N = 100   # データ数
a = [random.randint(1, 1000) for i in range(N)]    # N個の乱数

for i in range(1, N):
    for j in range(i - 1, -1, -1):
        if a[j] > a[j + 1]:
            a[j], a[j + 1] = a[j + 1], a[j]
        else:
            break

print(a)
```

①

①部は次のように書くこともできる．

```
    t = a[i]
    j = i - 1
    while  j >= 0 and a[j] > t :
        a[j + 1] = a[j]
        j -= 1
    a[j + 1] = t
```

番兵を用いれば，もっと簡単な表現ができる．（3-3参照）

実行結果

```
[17, 25, 27, 30, 52, 65, 71, 72, 124, 130, 152, 164, 173, 174,
178, 186, 188, 188, 191, 194, 196, 202, 208, 220, 222, 225, 247,
253, 257, 271, 288, 300, 301, 312, 316, 324, 324, 329, 334, 342,
371, 371, 372, 379, 391, 392, 393, 397, 414, 417, 426, 431, 435,
445, 466, 483, 484, 505, 526, 574, 586, 597, 599, 632, 645, 655,
661, 665, 666, 711, 722, 722, 736, 742, 744, 760, 765, 777, 788,
789, 799, 808, 817, 819, 838, 850, 854, 855, 880, 881, 884, 919,
926, 933, 938, 942, 973, 977, 987, 995]
```

4-4 ハノイの塔

例題 29 ハノイの塔

ハノイの塔問題を再帰を用いて解く.

ハノイの塔の問題は，再帰の問題でよく取り上げられる．ハノイの塔の問題は次のように定義される.

図4.10に示す3本の棒a，b，cがある．棒aに，中央に穴が空いたn枚の円盤が大きい順に積まれている．これを1枚ずつ移動させて棒bに移す．ただし，移動の途中で円盤の大小が逆に積まれてはならない．また，棒cは作業用に使用するものとする.

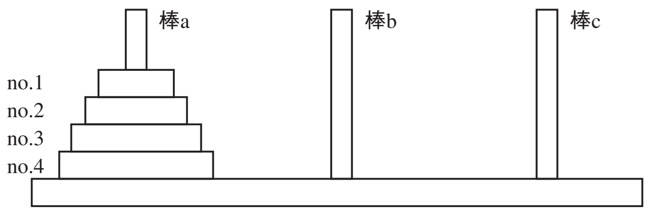

図 4.10 ハノイの塔

棒aの円盤が1枚なら，

a → b ―――― ⓐ

と移す．棒aの円盤が2枚なら，**図4.11**に示すように

1. a → c
2. a → b
3. c → b

（以上 ―――― ⓑ）

の順に移す.

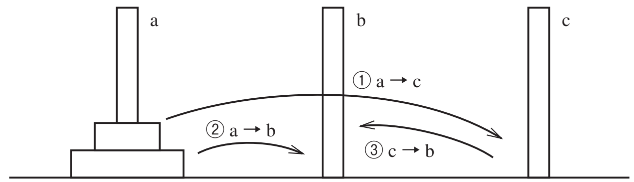

図 4.11 2枚の円盤の移動

これがハノイの塔の基本操作になる．なぜ基本操作になるかはすぐにわかる.

さて，棒aの円盤が3枚の場合を考えてみよう．**図4.12**において，棒aの上の2枚の円盤を△とし，

1. aの△を　a → c　に移せたとし
2. 下の1枚を　a → b　に移し，
3. cの△を　c → b　に移せたとする.

（以上 ―――― ⓒ）

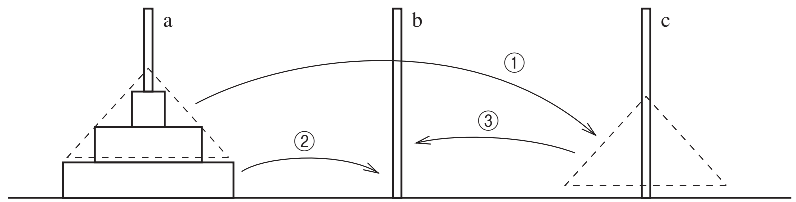

図 4.12 3枚の円盤の移動

ここで，△の移動は円盤の1枚ずつの移動ではないから，△の移動を1枚ずつの移動に置き換えなければならない．つまり，aの△をa→cに移す動作は，

a→b
a→c
b→c

となる．この操作は，2枚の円盤をaからbに移したβの解の目的の棒bを棒cに置き換えたものに他ならない．つまり，

a → b a → c b → c	swap b, c →	a → c a → b c → b	ⓑ解

5-1 スタック

例題 32 プッシュ／ポップ

スタックにデータを積む関数pushとデータを取り出す関数popを作る.

データを棚（stack）の下部から順に積んでいき，必要に応じて上部から取り出していく方式（last in first out：後入れ先出し）のデータ構造をスタック（stack：棚）という.

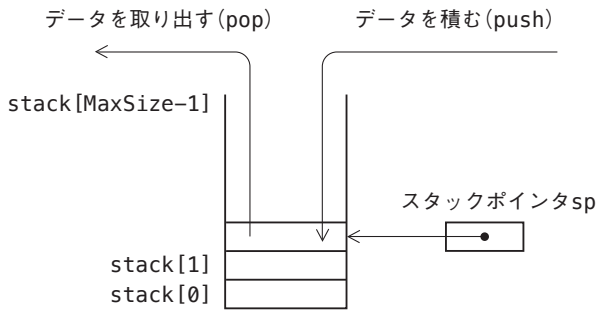

図 5.1 プッシュ／ポップ

データをスタックに積む動作をpush，スタックから取り出す動作をpopと呼ぶ.スタック上のデータがどこまで入っているかをスタックポインタspで管理する.

データがスタックにpushされるたびにspの値は＋1され，popされるたびに－1される.

スタック・ポインタspの初期値は最初0に設定しておき，データをpushするときは現spが示す位置にデータを積んでからspを＋1し，データをpopするときは，spの値を－1してからそれが示す位置のデータを取り出すものとするとする.

したがって，spが0の状態でpopしようとする場合は，スタックは「空の状態」であるし，spがMaxSizeの状態でpushしようとする場合はスタックは「溢れ状態」である.

プログラム Rei32_1

```
# ------------------
# *    スタック     *
# ------------------

def push(n):    # スタックにデータを積む手続き
    global sp
    if sp < MaxSize:
        stack[sp] = n
        sp += 1
        return 0
    else:
        return -1   # スタックが一杯のとき

def pop():          # スタックからデータを取り出す手続き
    global sp
    if sp > 0:
        sp -= 1
        return 0, stack[sp]
    else:
        return -1, 0 # スタックが空のとき

MaxSize = 100         # スタック・サイズ
stack = [0 for i in range(MaxSize)]   # スタック
sp = 0                # スタック・ポインタ

while (c := input('i:push, o:pop, e:end ?')) != 'e':
    if c == 'i':
        data = input('data?')
        ret = push(data)
        if ret == -1:
            print('スタックが一杯です')
    if c == 'o':
        ret, n=pop()
        if ret == -1:
            print('スタックは空です')
        else:
            print('pop --> {:s}'.format(n))
```

実行結果

```
i:push, o:pop, e:end ?i
data?21
i:push, o:pop, e:end ?i
data?56
i:push, o:pop, e:end ?o
pop --> 56
i:push, o:pop, e:end ?o
pop --> 21
i:push, o:pop, e:end ?o
スタックは空です
i:push, o:pop, e:end ?e
```

6-7 ヒープ・ソート

例題 47 ヒープ・ソート

ヒープ・ソートによりデータを降順に並べ換える.

ヒープ・ソートは大きく分けると次の2つの部分からなる.

① 初期ヒープを作る
② 交換と切り離しにより崩れたヒープを正しいヒープに直す

②の部分を詳しく説明すると次のようになる.

・n個のヒープデータがあったとき，ルートの値は最小値になっている．このルートと最後の要素（n）を交換し，最後の要素（ルートの値）を木から切り離す.
・すると，$n-1$個のヒープが構成されるが，ルートのデータがヒープの条件を満たさない．そこでルートのデータを下方移動（**練習問題46**）し，正しいヒープを作る.

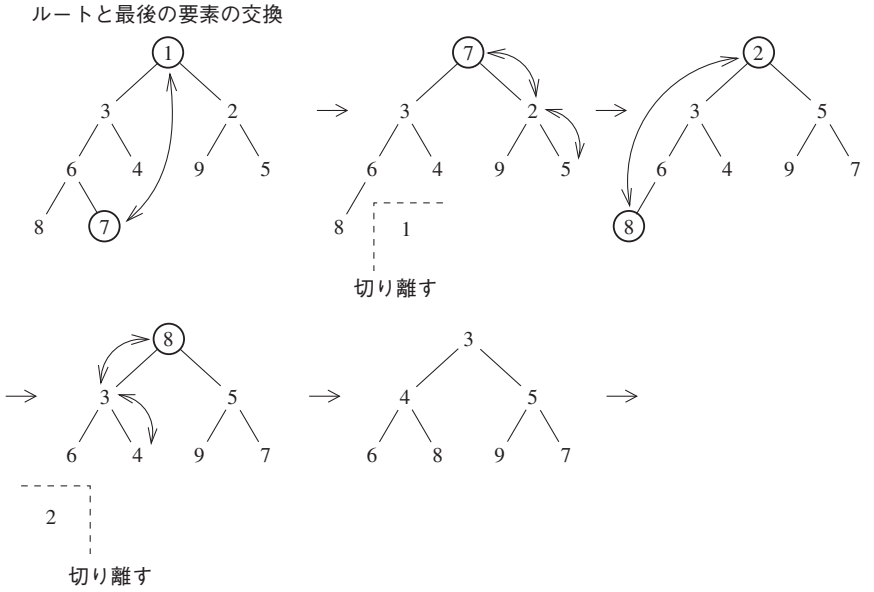

図 6.14 ヒープ･ソート

・$n-1$個のヒープについて，ルートと最後の要素（$n-1$）を交換し，最後の要素を木から切り離す.

以上を繰り返していけば，n，$n-1$，$n-2$，…に小さい順にデータが確定されるとともに，ヒープのサイズが1つずつ小さくなっていき，最後に整列が終了する.

プログラム Rei47

```
# ----------------------
# *    ヒープ・ソート    *
# ----------------------

heap = [0 for i in range(100)]
n = 1
while (data := input('data?')) != '/':
    heap[n] = int(data)
    s = n
    p = s // 2     # 親の位置
    while s >= 2 and heap[p] > heap[s]: # 上方移動
        heap[p], heap[s] = heap[s], heap[p]
        s = p
        p = s // 2
    n += 1

m = n - 1       # n の保存
while m > 1:
    heap[1], heap[m] = heap[m], heap[1]
    m -= 1            # 木の終端を切り離す
    p = 1
    s = 2 * p
    while s <= m:
        if s < m and heap[s + 1]< heap[s]:   # 左と右の小さい方
            s += 1
        if heap[p] <= heap[s]:
            break
        heap[p], heap[s] = heap[s], heap[p]
        p = s          # 親と子の位置の更新
        s = 2 * p

print(heap[1:n])
```

実行結果

```
data?1
data?5
data?3
data?9
data?2
data?/
[9, 5, 3, 2, 1]
```

第7章 グラフ(graph)

7-1 グラフの探索（深さ優先探索）

例題 50 深さ優先探索

深さ優先によりグラフのすべての節点を訪問する.

深さ優先探索（depth first search：縦型探索ともいう）のアルゴリズムは次の通りである.

・始点を出発し，番号の若い順に進む位置を調べ，行けるところ（辺で連結されていてまだ訪問していない）まで進む.
・行き場所がなくなったら，行き場所があるところまで戻り，再び行けるところまで進む.
・行き場所がすべてなくなったら終わり（来た道を戻る）.

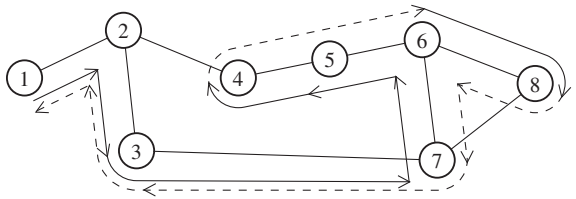

図 7.5 深さ優先探索

たとえば，節点3での次に進む位置のチェックは次のようにして行われる.

① 節点1について調べる．連結していないので進めない.
② 節点2について調べる．すでに訪問しているので進めない.
③ 節点3について調べる．連結していない（隣接行列の対角要素を0にしてある）ので進めない.
④ 節点4，5，6について調べる．連結していないので進めない.
⑤ 節点7について調べる．条件を満たすので節点7へ進む.

プログラムにおいてi点→j点へ進めるかは，隣接行列`a[i][j]`が1でかつ訪問フラグ`v[j]`が0のときである．訪問フラグはj点への訪問が行われれば`v[j]`は1とする.

プログラム Rei50

```
# ---------------------------------
# *      グラフの探索（深さ優先）      *
# ---------------------------------

N = 8   # 点の数

a = [[0, 0, 0, 0, 0, 0, 0, 0, 0],  # 隣接行列
     [0, 0, 1, 0, 0, 0, 0, 0, 0],
     [0, 1, 0, 1, 1, 0, 0, 0, 0],
     [0, 0, 1, 0, 0, 0, 0, 1, 0],
     [0, 0, 1, 0, 0, 1, 0, 0, 0],
     [0, 0, 0, 0, 1, 0, 1, 0, 0],
     [0, 0, 0, 0, 0, 1, 0, 1, 1],
     [0, 0, 0, 1, 0, 0, 1, 0, 1],
     [0, 0, 0, 0, 0, 0, 1, 1, 0]]

v = [0 for i in range(N + 1)]   # 訪問フラグ

def visit(i):
    global result
    v[i] = 1
    for j in range(1, N + 1):
        if a[i][j] == 1 and v[j] == 0:
            result += '{:d}->{:d}  '.format(i, j)
            visit(j)

result = ''
visit(1)
print(result)
```

実行結果

```
1->2  2->3  3->7  7->6  6->5  5->4  6->8
```

8-5 立体モデル

立体の基本図形で比較的簡単に作れるものとして，錐体，柱体，回転体が考えられる．

錐体を生成するために必要なデータは，底面の各点の座標(x_1, z_1)，(x_2, z_2)，…，(x_n, z_n)と，頂点の$x-z$平面への投影点(x_c, z_c)および高さhである（**図 8.23**）．

柱体を生成するために必要なデータは，底面の各点の座標(x_1, y_1)，(x_2, y_2)，…，(x_n, y_n)と，高さhである（**図 8.24**）．

回転体はたとえば，**図 8.25** のようなa～hで示される 2 次元図形を，y軸の回りに回転させることにより生成できる．

各点のy座標と，y軸からの距離（半径）rがわかっていれば，各点がy軸回りにθ回転したときの座標は，次式で示される．

$$\begin{cases} x = r\cos(\theta) \\ y = y \\ z = r\sin(\theta) \end{cases}$$

この点を回転変換（軸測投影のところで示した式）して回転していけばよいが，a～h点を回転させた軌跡を描いただけでは，単に 8 個の楕円が描けるだけで，とても立体には見えない．そこで$a \to b \to c \cdots h$を結んだ直線（稜線）をある回転角度ごとに何箇所かに描くことにする．

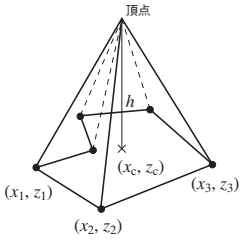

図 8.23 錐体

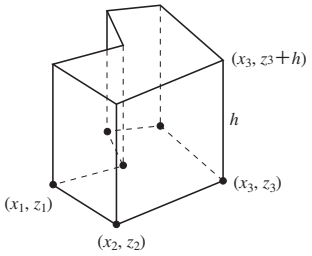

図 8.24 柱体

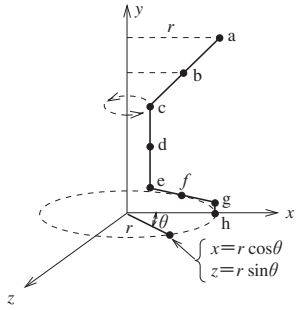

図 8.25 回転体

例題 59 回転体モデル

回転体モデルによる立体を軸測投影で表示する．

プログラム Rei59

```
# ------------------------------------
# *      回転体モデル（ワイングラス）      *
# ------------------------------------

!pip3 install ColabTurtle
from ColabTurtle.Turtle import *

initializeTurtle(initial_window_size=(640, 640), initial_speed=13)
hideturtle()
width(2)
bgcolor('white')
color('blue')

def rotate(ax, ay, az, x, y, z):     # 3 次元回転変換
    x1 = x*math.cos(ay) + z*math.sin(ay)    # y 軸回り
    y1 = y
    z1 = -x*math.sin(ay) + z*math.cos(ay)
    x2 = x1                                 # x 軸回り
    y2 = y1*math.cos(ax) - z1*math.sin(ax)
    px = x2*math.cos(az) - y2*math.sin(az)  # z 軸回り
    py = x2*math.sin(az) + y2*math.cos(az)
    return px, py

y = [180, 140, 100, 60, 20, 10, 4, 0]    # 高さ
r = [100, 55, 10, 10, 10, 50, 80, 80]    # 半径

ax = math.radians(35)
ay = math.radians(0)
az = math.radians(20)

for k in range(len(y)):     # y 軸回りの回転軌跡
    for n in range(0, 361, 10):
        x = r[k] * math.cos(math.radians(n))
        z = r[k] * math.sin(math.radians(n))
        px, py = rotate(ax, ay, az, x, y[k], z)
        if n == 0:
            penup()
        else:
            pendown()
        goto(px + 320, 320 - py)

for n in range(0, 361, 60):   # 稜線
    for k in range(len(y)):
        x = r[k] * math.cos(math.radians(n))
        z = r[k] * math.sin(math.radians(n))
        px, py = rotate(ax, ay, az, x, y[k], z)
```

8-12 Matplotlib を使った 3D 表示

1. 3D 描画手順

Matplotlib で 3D 表示する描画手順は以下である.

・描画エリア figure の作成

```
fig = plt.figure(figsize=(8, 8))
```

・3D の subplot を追加

```
ax = fig.add_subplot(projection='3d')
```

・データの作成

```
x, y, z= それぞれの値
```

・グラフの作成

```
ax.plot(x, y, z)
```

・グラフの描画

```
plt.show()
```

! Matplotlib バージョン 3.2.0 以前は,「from mpl_toolkits.mplot3d import Axes3D」で Axes3D クラスをインポートしなければならなかったが, 現在のバージョンでは不要.

3 次元座標は右手系と左手系があり, その中で y を上方向にするか z を上方向にするかで分かれる. 本書では右手系で y を上にする座標で解説している.

Matplotlib の 3 次元座標は, z 軸が上方向になる右手座標である. このため y 軸の正の向きは奥方向になる.

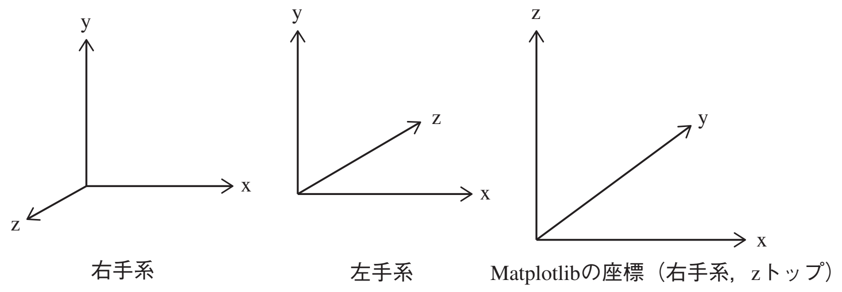

図 8.63

例題 67 立方体

Matplotlib を使って立方体を描く.

立方体の辺を描くには, 一筆書きできる直線群に分ける. 立方体の場合は①～④の直線群に分け, データを 2 次元リストに格納する. ①群は (0,0,0) から 10 の点を矢印の順にたどる.

① ② ③ ④

x = [[0, 1, 1, 0, 0, 0, 1, 1, 0, 0], [1, 1], [1, 1], [0, 0]]
y = [[0, 0, 1, 1, 0, 0, 0, 1, 1, 0], [0, 0], [1, 1], [1, 1]]
z = [[0, 0, 0, 0, 0, 1, 1, 1, 1, 1], [1, 0], [1, 0], [1, 0]]

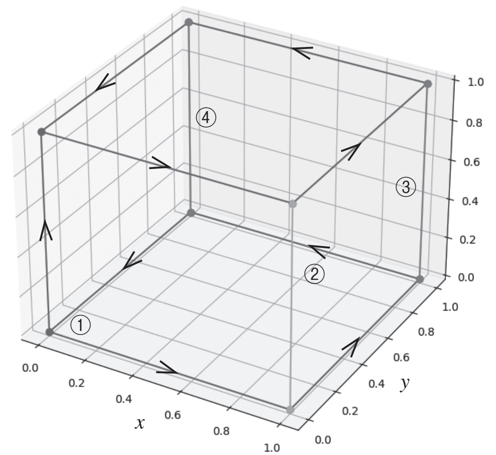

図 8.64

9-6 21を言ったら負けゲーム

例題 74 **コンピュータと対戦**

21を言ったら負けゲームのルールは「1～21の数字を交互に言い合う」，「1度に言える数は，連続して3つまで」，「21を言ったら負け」である．先手を人間，後手をコンピュータとする「21を言ったら負けゲーム」を作る．

「21」を言ったら負けということは相手に「20」を言われたら負けということになる．「20」を抑えるにはその前に「16」を抑える必要があり，…と続けていけば，最初に「4」を抑えた方が必ず勝つことになる．ここで連続して3つまで言えるというルールが重要になる．先手が「1」と言えば，後手は「2,3,4」と言い，先手が「1,2」と言えば，後手は「3,4」と言い，先手が「1,2,3」と言えば，後手は「4」と言えば良く，必ず後手は「4」を抑えることができる．以後，後手は「8,12,16,20」と抑えていけばよい．先手後手どちらが有利かといえば，先手有利のゲームの方が多い中で「21を言ったら負けゲーム」は数少ない後手必勝ゲームである．ちなみに，プロ棋士の将棋では先手の勝率は約52%（理論値でなく経験値）とわずかながら先手が有利だそうである．

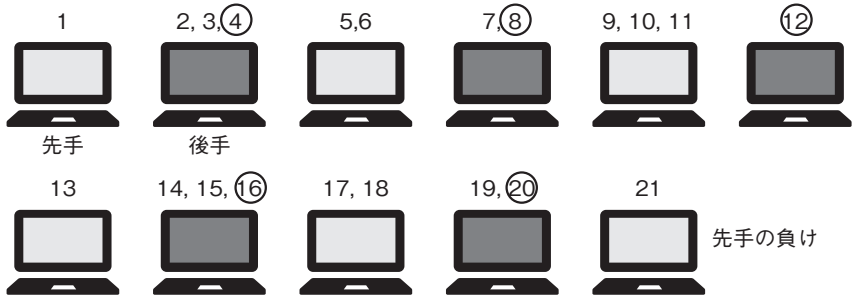

図 9.18 21を言ったら負けゲーム

人間の答えの入力と，最後の数字の取得

先手の人間の答えは「1,2」や「1,2,3」などの文字列で入力することにする．この文字列から最後の数字を取り出す．最後の数字の入力パターンは以下の4種類である．「,」で区切った複数の数字列で最後が1桁の場合（①）と2桁の場合（②），単独の数字で1桁の場合（③）と2桁の場合（④）．

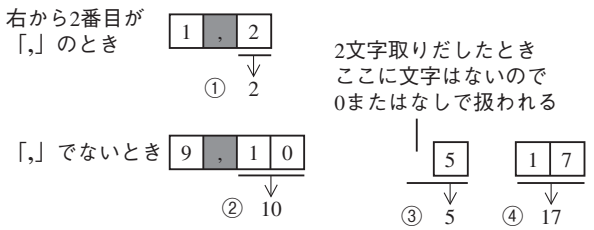

図 9.19

この4つのパターンを条件分けするには「文字列の右から2番目が「,」なら」という単純な判定でうまくいく．

プログラム Rei74

```
# ---------------------------------
# *      21 を言ったら負けゲーム      *
# ---------------------------------

ans = 1
count = 1
while ans < 20:
    kotae = input(' 数字を入力して ')
    N = len(kotae)
    if kotae[N - 2] == ',':      # 人間の答えの最後の数字を取り出す
        ans = int(kotae[N - 1])
    else:
        ans = int(kotae[N - 2:N])
    result = ' 僕の答えは '
    while ans < count * 4:       # コンピュータの答え
        ans += 1
        result += '{:3d}'.format(ans)
    print(result)
    count += 1
print(' 僕が 20 と言ったので君の負け ')
```

実行結果

```
数字を入力して 1,2,3
僕の答えは   4
数字を入力して 5
僕の答えは   6   7   8
数字を入力して 9,10
僕の答えは  11  12
数字を入力して 13
僕の答えは  14  15  16
数字を入力して 17,18
僕の答えは  19  20
僕が 20 と言ったので君の負け
```