

アルゴリズム入門 #7

地引 昌弘

2021.11.18

はじめに

今回は、配列を利用して、自然現象や社会現象をコンピュータ上で簡潔に模擬する (これを、シミュレーション (Simulation) と呼びます) 手法の一つであるセル オートマトンを取り上げます。

1 前回の演習問題の解説

1.1 演習 6-1 — 配列の生成

これは簡単にコードだけを示します:

```
!pip install ita          # Google Colaboratory へのログイン毎に 1 度実行して下さい。
import ita

def arraynew1():          # (a)
    a = ita.array.make1d(10)
    for i in range(len(a)): # 配列の長さを返す len 関数を利用しましょう (以下同じ)
        a[i] = 10 - i
    print(a)

def arraynew2():          # (b)
    a = ita.array.make1d(10)
    for i in range(len(a)):
        a[i] = i % 2
    print(a)

def arraynew3():          # (c)
    a = ita.array.make1d(10)
    for i in range(len(a)):
        if (i - 4 >= 0):
            a[i] = i - 4
        else:
            a[i] = - (i - 4)
    print(a)

def arraynew4():          # (d)
    a = ita.array.make1d(10)
    for i in range(len(a)):
        if (i >= 5):
            a[i] = 0
        else:
            a[i] = 1
    print(a)
```

(a) と (b) は簡単なクイズという感じですね。(c) については、絶対値を求める `abs` 関数を使う方法もあります (`a[i] = abs(i-4)`)。

1.2 演習 6-2 — 配列の取り扱い

演習 6-2 も擬似コードを省略して、Python のコードだけ記します。まず最大:

```
def arraymax(a):
    max = a[0]
    for i in range(len(a)):
        if a[i] > max:
            max = a[i]
    return(max)
```

このような形で配列を使う場合は、通常「取り敢えず変数 `max` に最初の値を入れておき、より大きい値が出てきたら入れ換える」手順を採用します。配列の先頭の添字番号は、0 であることに注意して下さい。

次は最大値が何番目に出て来るかなので、「何番目か」も変数に記録しておき、最大値を更新した際は両者を同時に更新します:

```
def arraymaxno1(a):
    max = a[0]
    pos = 0
    for i in range(len(a)):
        if a[i] > max:
            max = a[i]
            pos = i
    return(pos)
```

最大値を一つだけ記録するのは、単に変数へ入れるだけで可能ですが、最大値が複数あった場合に、その位置を全部打ち出すには、1) まず最大値を求め、2) その最大値と等しいものがあれば位置を打ち出す、という形で 2 回ループを回す必要があります:

```
def arraymaxno2(a):
    max = a[0]
    for i in range(len(a)): # search for the maximum value
        if a[i] > max:
            max = a[i]
    for i in range(len(a)):
        if a[i] == max:
            print(i)
```

平均値より小さい値を出力するのもこれと同様です:

```
def arrayavgsmaller(a):
    sum = 0.0
    for i in range(len(a)): # calculating the average value
        sum += a[i]
    avg = sum / len(a)
    print(avg) # output the average value
    for i in range(len(a)):
        if a[i] < avg:
            print(a[i])
```

Python では、割り算は常に実数 (浮動小数点) として計算されますが、言語によっては (例えば、Ruby など)、`sum` の初期値を 0 にすると (つまり、整数)、切捨て除算 (商の小数部分を切り捨て) として扱われてしまうので、注意して下さい。

1.3 演習 6-3 — 素数発見方法の改良

まずは、前回取り上げた素朴な素数発見方法を再掲しておきます:

```
import time

def isprime1(n):
    prime = True
    for i in range(2, n):          # 終値 = n-1
        if n % i == 0:
            prime = False
    return(prime)

def primes1a(n):
    start = time.process_time()   # measurement of start time
    cnt = 0
    for i in range(2, n+1):       # 終値 = n
        if isprime1(i):
            cnt += 1              # counting the number of prime numbers
    print(cnt)
    finish = time.process_time()  # measurement of finish time
    print("%g" % (finish - start)) # display of required time
```

この方法では、与えられた数 n を $2 \sim n-1$ の各数で割り算し、余りがあるかどうかを調べています。これをあるマシンで動かしてみたところ、 $2 \sim 150,000$ までの間に存在する素数の個数を調べるのに、693.1 秒掛かりました。ここから先は、この素数発見プログラムの高速化を試みて行きましょう。

最も素朴な素数判定 `isprime1` 関数は、「割り切れる」と分かってもそこで計算を止めず、 n の手前まで割り算を続けるため、早い段階で割り切れた数に対しては非常に無駄が大きいのと思われます。そこで、改良版を作ってみましょう:

```
def isprime2(n):
    for i in range(2, n):          # 終値 = n-1
        if n % i == 0:
            return(False)
    return(True)
```

こちらは、割り切れると分かったら直ちに `return` で「いいえ」を返すので、無駄な割り算をしなくて済みます。ここで、 n として 2 を渡された場合の動作を少し補足しておきます。ループの範囲は 2 から $n-1$ までなので、この場合は範囲が 2 から 1 までとなり、1 度もブロックを実行することなくループを抜けることになります。上の `primes1a` 関数を、こちらを使うように直したところ、その所要時間は 56.2 秒でした。つまり速度が 12 倍以上速くなったわけです。

さらに考えると、割り算を $n-1$ までやる必要はなく、 \sqrt{n} まで調べても割り切れなければ、それ以上の数でも割り切れないことが分かります (\sqrt{n} よりも大きい因数があるならば、それと対になる小さい因数が必ずありますよね)。そこで素数判定を次のように改良します:

```
import math

def isprime3(n):
    if n <= 3:    # for 文の range 関数は始値が 2 なので、終値が 3 未満 (math.sqrt() 部分が 2 未満) は別扱い
        return(isprime2(n))
    for i in range(2, int(math.sqrt(n))+1):    # 終値の+1 に注意
        if n % i == 0:
            return(False)
    return(True)
```

この方法による素数判定は、与えられた n に対して、`math.sqrt(n)` を超えるまで続ける必要があります。しかし、`range` 関数に終値 `stop` を指定した場合は、カウンタが `stop - 1` になるとループを抜けてしまいます。つまり、調査に使う因数が `math.sqrt(n)` より 1 少ない時点で、判定を止めてしまうわけです。よって今回は、`for` ループの終値に `+1` する必要があることに注意して下さい。また、`math.sqrt(n)` 後の値が 2 以上になる数 n は 4 以上なので、3 以下については別途調べる必要があります (かと言って、`for` ループのカウンタを 1 から始めると、1 で割ることになるため、全ての数が非素数と判定されてしまうので要注意です)。これで試してみると、所要時間はなんと 0.31 秒まで縮まりました。

次に、2 は素数であり、2 の倍数は素数でないことが分かるので調べる必要はない、ということを利用しましょう。これより、3 以上の奇数だけで割り算を試みる改良版の素数判定を作ってみます:

```
def isprime4(n):
    # for 文の range 関数は始値が 3 なので、終値が 4 未満 (math.sqrt() 部分が 3 未満) は別扱い
    if n <= 8:
        return(isprime3(n))

    # math.sqrt(n) 以下の奇数でしか割り算をしない (始値 3 から 2 ずつ増やす → 常に奇数)
    for i in range(3, int(math.sqrt(n))+1, 2):
        if n % i == 0:
            return(False)
    return(True)

def primes4(n):
    start = time.process_time()
    cnt = 0
    if (n >= 2):
        cnt += 1

    # 判定処理 (isprime4 関数) へは奇数しか渡さない (始値 3 から 2 ずつ増やす → 常に奇数)
    for i in range(3, n+1, 2):
        if isprime4(i):
            cnt += 1
    print(cnt)
    finish = time.process_time()
    print("%g" % (finish - start))
```

判定処理 (`isprime4` 関数) は 3 から始まるので、2 については別途追加する必要があります。さらに `isprime3` 関数と同様、 \sqrt{n} 後の値が 3 以上になる数は 9 以上なので、8 以下についても別途調べる必要があります。この版の所要時間は 0.16 秒でした。

もう少し頑張り、2と3より大きい素数は6の倍数±1だけ(それ以外は2と3の倍数になってしまう)、ということを利用して、さらに調べる数を減らしてみましょう(ループの境界条件に注意して下さい):

```
def isprime5(n):
    # for 文の range 関数は始値が6なので、終値が8未満(math.sqrt())部分が6未満)は別扱い
    # (7ではなく8となる理由は本文参照)
    if n <= 35:
        return(isprime4(n))

    # math.sqrt(n) 以下の6の倍数でしか割り算をしない(始値6から6ずつ増やす→常に6の倍数)
    for i in range(6, (int(math.sqrt(n))+1) + 1, 6):    # 終値の再増分に注意
        if n % (i-1) == 0:
            return(False)
        if n % (i+1) == 0:
            return(False)
    return(True)

def primes5(n):
    start = time.process_time()
    cnt = 0
    if (n >= 3):
        cnt += 2
    elif (n == 2):
        cnt += 1

    # 判定処理(isprime5関数)へは6の倍数しか渡さない(始値6から6ずつ増やす→常に6の倍数)
    for i in range(6, (n+1) + 1, 6):    # 終値の再増分に注意
        if isprime5(i-1):
            cnt += 1
        if ((i+1) <= n) and isprime5(i+1):
            cnt += 1
    print(cnt)
    finish = time.process_time()
    print("%g" % (finish - start))
```

今回は、forループの終値が、以前のプログラムよりさらに1増えていることに注意して下さい。その理由についてですが、このプログラムは主に下記の動作を行いません。

1. 6の倍数 $i (= 6x)$ を取り出す(この i が基準となる)。
2. $i - 1 (= 6x - 1)$ を調べる。
3. $i + 1 (= 6x + 1)$ を調べる。

ここで、 n として丁度 $6a - 1$ を渡された場合を考えます。 $6a - 1$ は6の倍数-1に該当するので、上記2より `isprime5` 関数が調べる対象です。しかし、`primes5` 関数内でカウンタ i の終値を以前と同じにしまうと、カウンタは $(6a - 1) + 1 = 6a$ 未満の6の倍数しか取らないため、このforループは“ $i = 6(a - 1) = 6a - 6$ ”で終わってしまいます(\because 6から始まり6ずつ増やしているので、 a を1から増やして行くと考えて下さい)。つまり、 $6a - 1$ は上記2の判定対象であるにもかかわらず、 i が基準となる $6a$ を取る前にforループが終了するため、判定対象から外れてしまうというわけです。よって、 $6a - 1$ のような数値も判定対象から外れないようにするため、終値を1増やしています。こうしておけば、このforループは基準となる“ $i = 6a$ ”まで回るため、 $6a - 1$ も判定の対象になります(`isprime5`関数内のforループも同じ考えに基づいています)。isprime3関数もそうでしたが、境界条件を正しく設定することは少々面倒ですね。ところで最後は、2~150,000までの間に存在する素数を調べるのに要する時間を、0.13秒まで縮めることができました。

1.4 演習 6-4 — 配列を用いた素数発見方法の改良

この演習では、アルゴリズム単体だけではなく、処理に適したデータ型を用いることでアルゴリズムをさらに改良し、処理の高速化を目指します。

まず最初は、これまでに見つかった素数を配列に覚えておく方法です。素数が見つかるにつれ、配列を伸ばして行く必要があるため、`insert` 関数を利用します:

```
def isprime6(a, n):
    limit = int(math.sqrt(n))
    for i in range(len(a)):

        # 既に発見した素数で割り切れた場合は、素数ではないと判定。
        if n % a[i] == 0:
            return(False)

        # 同、割り切れなかった場合は、
        # 割り算をした素数が候補の数 n の平方根以上であれば、
        # それ以上調べても割り切れないことが分かっているので、素数と判定。
        # 平方根未満であれば、次に発見した素数を試すため、ループを継続する。
        if a[i] >= limit:
            a.insert(len(a), n)
            return(True)

    # 候補の数 n が 2 の場合 (つまり、初めて isprime6 が呼び出された場合) は、
    # 素数を記録している配列 a が空なので、この処理に至る。
    # n が 3 以上の場合は、a が空ではないので、この処理に来ないはず。
    a.insert(len(a), n)
    return(True)

def primes6(n):
    start = time.process_time()
    cnt = 0
    a = []    # 事前に配列の大きさを決められないので、作成時の要素数は取り敢えず 0 にしておく。
    for i in range(2, n):
        if isprime6(a, i):
            cnt += 1
    print(cnt)
    finish = time.process_time()
    print("%g" % (finish - start))
```

素数判定メソッド `isprime6` の動作が少々分かりにくいので、コードの意味を説明するコメントを入れてあります (本格的なプログラムを書く際は、必ずこのようなコメントを入れます)。 `isprime6` は、素数の入った配列 `a` を受け取り、そこから順に素数 `i` を取り出して、候補の数 `n` が割り切れるかどうかを調べて行きます。その処理を続け、これまでに取り出した素数で割り切れないまま、新たに取り出した素数が候補の数の平方根 `limit` より大きくなると、以後に取り出す素数では割り切れないことが明らかなので、即座に「素数である」という答えを返します (候補の数を素数として記録します)。これも、候補の数が 2 の場合だけは、素数の入った配列がまだ作られていない (と言うか、要素が入っていない) ので、他と区別する必要があります。この方法だと、「2 や 3 の倍数を除外」といった細かい工夫をせずとも、0.22 秒で 2 ~ 150,000 までの間に存在する素数を調べられました (Python では、配列の長さを一つずつ伸ばすという処理が、どうも遅いようです)。

最後は、「エラトステネスのふるい」です。この方法は、これまでと大幅に違う方法です:

```
def primes7(n):
    # 巨大な配列を生成するのは時間が掛かり、ここではアルゴリズムの実行時間を比べることが目的なので、
    # 今回は事前に配列を用意しておくこととした。
    a = ita.array.make1d(n+1)      # 要素数+1 に注意
    start = time.process_time()
    cnt = 0
    for i in range(2, n+1):        # 終値 = n (n まで調べる → 添え字番号の最後 = n)
        if a[i] == 0:
            cnt += 1

            # i より後ろにある i の全倍数に対して印を付ける。
            # 始値 = i より後ろにある最初の i の倍数 = i+i, ステップ = i の倍数毎 = i
            for j in range(i+i, n+1, i):
                a[j] = 1
    print(cnt)
    finish = time.process_time()
    print("%g" % (finish - start))
```

まず最初に、添字番号が $0 \sim n$ の配列 a を作り (要素数は $n+1$)、各要素の値を 0 にしておきます。今回は、“添字番号 = 調べる数字” としたかったので、配列の要素数を一つ増やしました。次に、 2 から始めて各候補の数値 i に対し、 $a[i]$ が 0 ならば素数なので記録するとともに、その素数の全倍数 j について $a[j]$ を 1 にします。これにより、割り算をせずとも i より大きい非素数が、次々とふるい落とされて行きます。この方法は非常に高速で、 $150,000$ 以下の素数を調べるのに要した時間は、なんと 0.03 秒です¹。最初の素朴版が数百秒のレベル、こちらが数十ミリ秒のレベルなので、両者を比較すると **10,000 倍!!!** も速くなったわけです (今回の例で言えば、 $20,000$ 倍)。日常的な世界では、「 $10,000$ 倍の差」というものはなかなかありません。我々の歩く速度はおよそ 4 km/h ですが、 $40,000$ km/h というのは地球の重力圏を脱出するのに必要な速度です。この例に比べて、プログラムの動作速度、つまりアルゴリズムの世界は、その良し悪しにより簡単に「ものすごい差」が付いてしまいます (まあ、これを学ぼうというのが、本講義の趣旨なのですが)。

¹因みに、配列の生成込みの時間は 0.13 秒でした。それでも、前回作成した最速のアルゴリズム (6 の倍数 $\pm 1 \dots$) と同じですね。

2 2次元配列

これまでに出て来た配列は全て1次元配列(各データが1列に並んでいるだけ)でしたが、2次元配列(配列の配列)を利用したい場合もあります。Pythonでは、1次元配列と同様に2次元配列も扱うことができます。「2次元配列」は、「縦横2次元に要素が並んでいる」というイメージですが、実際には図1のように、配列の各要素が配列という構造になっています(とは言え、縦横2次元のイメージでも実用上は差し支えありません)。

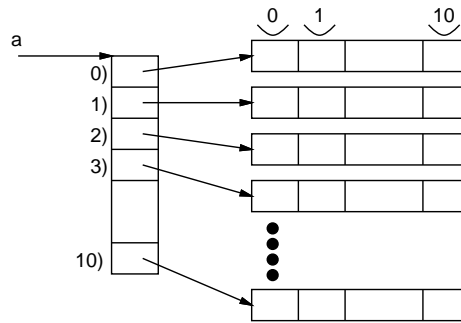


図 1: 2次元配列

2次元配列 `a` の各要素は、行の並び方向(上下方向)の添字番号 `i` と、列の並び方向(左右方向)の添字番号 `j` を用いて、`a[i][j]` と指定します。2次元配列の生成は、1次元配列と同じく、全ての値を直接「`[[a,b], [c,d]]`」のように指定することで生成できます。しかし、この方法では、配列が大きくなるととても面倒です。1次元配列の作成では、まず `ita.array.make1d` 関数を用いて配列を用意し、その後で `for` ループを用いて値を設定しました。2次元配列についても同様な方法を利用しましょう²。但し、2次元配列に対して配列の長さを返す `len` 関数を利用する場合は、注意が必要です。`len` 関数は配列に格納された要素数を返すのですが、2次元配列は `[[a,b], [c,d]]` のように格納されるので(図1)、2次元配列に `len` 関数を適用した場合は、結果として行方向の長さが返されることになります(つまり、`a[i][j]` の `i` に該当する長さ)。多くの場合は、行方向・列方向に沿って処理をする方が見通しが良いので、下記のように変数 `size_i`, `size_j` を利用するのが無難です:

```
a = ita.array.make2d(size_i, size_j)    # size_i 行 size_j 列の配列を作成 (各要素の値 = 0)
for i in range(size_i):
    for j in range(size_j):
        a[i][j] = ...                    # 2次元配列の各要素に値を設定
```

最後に:

配列は、様々なプログラムで利用される基本的なデータ構造です。従って、各プログラミング言語では、配列を効率的に処理するために、様々な関数や構文が用意されています。本講義の狙いから外れるため、ここではそれらを逐一取り上げることはしませんが、教科書の5.6節「【発展】配列の様々な機能」および5.7節「【発展】配列とコピー」は、必ず目を通して置いて下さい。

演習 7-1 `ita.array.make2d` 関数を用いて 5×5 の2次元配列を生成し、`for` ループを用いて (a)~(d) の値を設定して下さい。

(a)	<table border="1"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>-1</td><td>0</td><td>1</td><td>2</td><td>3</td></tr><tr><td>-2</td><td>-1</td><td>0</td><td>1</td><td>2</td></tr><tr><td>-3</td><td>-2</td><td>-1</td><td>0</td><td>1</td></tr><tr><td>-4</td><td>-3</td><td>-2</td><td>-1</td><td>0</td></tr></table>	0	1	2	3	4	-1	0	1	2	3	-2	-1	0	1	2	-3	-2	-1	0	1	-4	-3	-2	-1	0
0	1	2	3	4																						
-1	0	1	2	3																						
-2	-1	0	1	2																						
-3	-2	-1	0	1																						
-4	-3	-2	-1	0																						
(b)	<table border="1"><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr><tr><td>1</td><td>2</td><td>4</td><td>8</td><td>16</td></tr><tr><td>1</td><td>3</td><td>9</td><td>27</td><td>81</td></tr><tr><td>1</td><td>4</td><td>16</td><td>64</td><td>256</td></tr></table>	1	0	0	0	0	1	1	1	1	1	1	2	4	8	16	1	3	9	27	81	1	4	16	64	256
1	0	0	0	0																						
1	1	1	1	1																						
1	2	4	8	16																						
1	3	9	27	81																						
1	4	16	64	256																						
(c)	<table border="1"><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1
1	0	0	0	0																						
0	1	0	0	0																						
0	0	1	0	0																						
0	0	0	1	0																						
0	0	0	0	1																						
(d)	<table border="1"><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	1	0	0	0	1	0	1	0	1	0	0	0	1	0	0	0	1	0	1	0	1	0	0	0	1
1	0	0	0	1																						
0	1	0	1	0																						
0	0	1	0	0																						
0	1	0	1	0																						
1	0	0	0	1																						

²`ita.array.make2d` 関数の使用に際しては、`ita.array.make1d` 関数と同様、事前に `import ita` を用いて、アルゴリズム入門の演習用ライブラリを読み込んでおく必要があります。

なお、Python のプログラムで 2 次元配列を出力する際は、`print` 関数ではなく `pprint.pprint` 関数を使うと見易くできます。但し、これを使うには、事前に `import pprint` を用いて、`pprint` 用ライブラリを読み込んでおく必要があります。`pprint.pprint` 関数は、`width` オプションで出力を改行するまでの文字数を指定できます。使い方は、こんな感じ:

```
>>> import pprint
>>> a = [[0,0,0,0,0], [0,1,2,3,4], [0,2,4,6,8], [0,3,6,9,12], [0,4,8,12,16]]
>>> print(a)
[[0, 0, 0, 0, 0], [0, 1, 2, 3, 4], [0, 2, 4, 6, 8], [0, 3, 6, 9, 12], [0, 4, 8, 12, 16]]
>>> pprint.pprint(a, width=20)
[[0, 0, 0, 0, 0],
 [0, 1, 2, 3, 4],
 [0, 2, 4, 6, 8],
 [0, 3, 6, 9, 12],
 [0, 4, 8, 12, 16]]
```

3 セル オートマトン

セル オートマトン (Cellular Automaton — 略称: CA) は、配列を利用して、自然現象や社会現象をコンピュータ上で簡潔にシミュレートする手法の一つです (教科書にあるライフ ゲームもセル オートマトンの一つです)。セルは“柀目”を意味し、オートマトンは“自動機械”を意味します。セル オートマトンは、次のように振る舞います。

- 各セルは状態 (取り敢えずここでは、値のようなものと考えておいて下さい) を持つ³。
- 各セルは、時刻 t における自セルおよび近傍 (要は、その周囲) セルの状態により、時刻 $t+1$ ⁴ の状態が決まる。

これだけ見るとシンプルですね (まあ逆に、シンプルだからこそ、様々な自然現象や社会現象に適用できるというわけですが)。セル オートマトンのミソは、上記 2 項目の状態を変化させる規則です。以下では、説明を簡素化するため、状態は 0 あるいは 1 のどちらか 2 通りしかないとします。

3.1 1 次元セル オートマトン

セル オートマトンの中で、一番簡単な例は、セルが横一列に並んだ 1 次元セル オートマトンです。図 2 のように、1 次元セル オートマトンを各時刻毎に並べることで様々な状態パターンが作成されます。

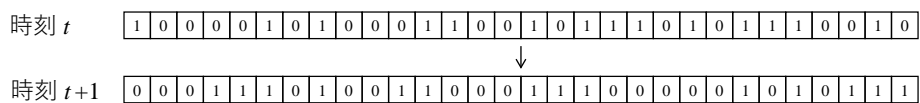


図 2: 1 次元セル オートマトンの例

このセル オートマトンは 1 次元なので、時刻 $t \rightarrow$ 時刻 $t+1$ での状態変化に関与するセルは、自セルとその両脇セルの計 3 セルです。これら 3 セルの状態が取り得る場合の数は、図 3 (次ページ) で示すように 8 通りです。この図では、変化後の (時刻 $t+1$) の状態は“?” になっていますが、実際にはここが 0 あるいは 1 になります。8 個の 0 あるいは 1 からなる場合の数は $2^8 = 256$ 通りなので、時刻 $t \rightarrow$ 時刻 $t+1$ の状態変化規則は 256 種類存在します。

³正確に言えば、“状態”とは、例えば駅の発券窓口にいる係員を例にすると、1) お客さんが来るまでは待っている状態、お客さんが来た後は、2) 要望を聞いている状態、3) 要望に合った切符を検索している状態、切符が取れた後は、4) 料金を受け取る状態などがありますが、これら各状態に値を割り当て、その値で表すことも (例えば、“3” の状態とか “5” の状態とか) 可能なので、ここではあまり難しく考えず、値のようなものと考えておきましょう、ということです。

⁴これは、実世界での時間の流れのような連続的な時間経過ではなく、1 回目、2 回目といった非連続的な時間経過を意味しています。

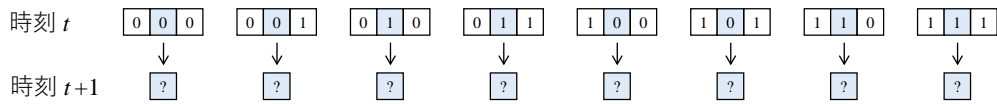


図 3: 1次元セル オートマトンの状態変化規則

これまでに、全 256 種類の状態変化規則から生成される状態パターンが調べられています。その結果、各状態パターンは、以下の四つに場合分けできることが分かりました。

- クラス 1: どのような初期状態から始めても、やがて全てのセルが均質な状態になるパターン
- クラス 2: セルが周期的な状態変化を繰り返すパターン
- クラス 3: カオス的な (乱雑な) パターン (乱雑の中に一部、自己相似的な局所構造が現れることもある)
- クラス 4: クラス 1・2 の秩序状態と、クラス 3 のカオス状態が共存するようなパターン

以下に、適当な初期状態から開始した各クラスの状態パターン例を示しておきます。これらの図は、各時間における 1次元セル オートマトンの状態を、時間の経過に伴い上から下へ並べたものです。黄緑色のセルは状態 1 を、白色 (というか、もう灰色だね) のセルは状態 0 を表しています。必要に応じて、拡大して見てみて下さい。

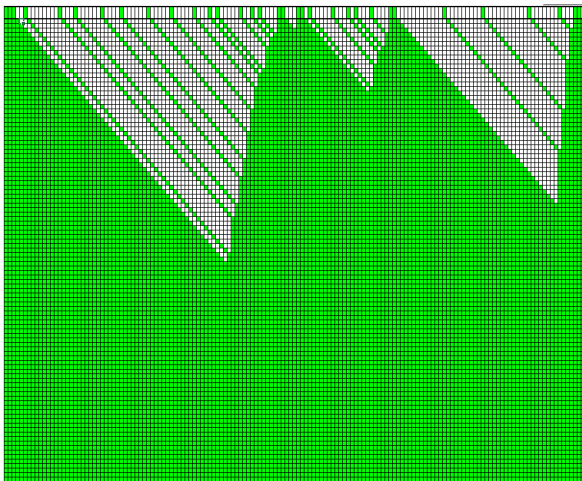


図 4: クラス 1 の状態パターン例

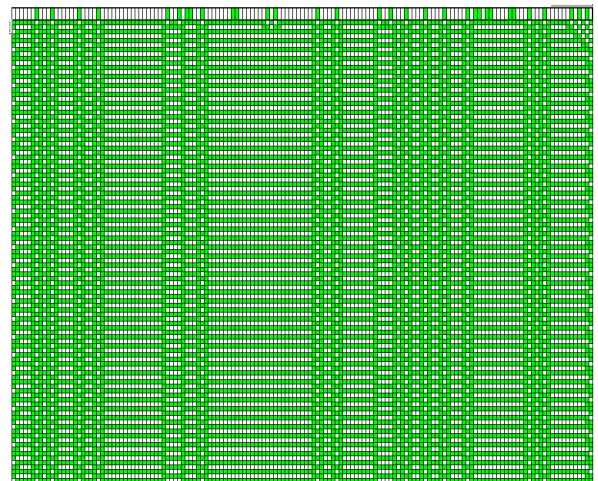


図 5: クラス 2 の状態パターン例

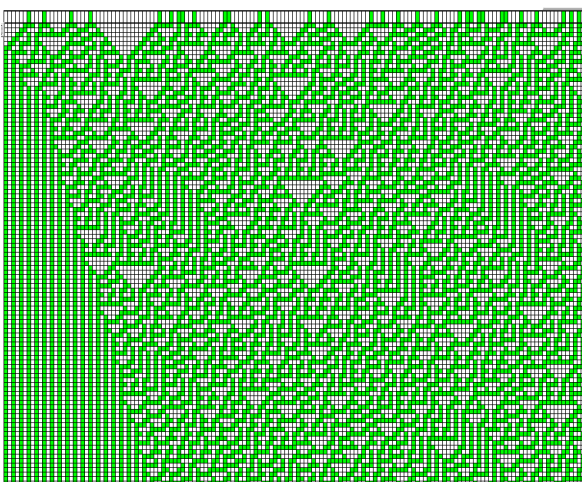


図 6: クラス 3 の状態パターン例

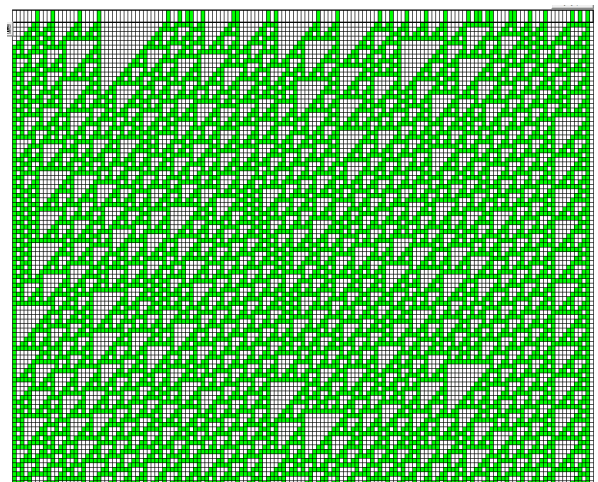


図 7: クラス 4 の状態パターン例

これらを見る限り、はあそうですか、という感じしかしないのですが、実はこれまでに、カオスに分類されない新たなクラス 4 の状態から、大変興味深い事実が導かれています。図 7 を見てみると、様々な大きさの三角形が並んでいます。向きは揃っているのですが、大きさはバラバラだし、その出現パターンについても一見規則性があるようには見えませ

ん。これのどこが興味深いかというと、実は、このパターンはプログラム言語になっているのではないかと考えられたからです。何やら唐突な話で、ちょっと戸惑いますね。でも知りたいという人もいると思うので(私としては、いて欲しいのですが…)、簡単に紹介しておきます。

1次元セルオートマトンは、0・1のパターンを渡され、それを元にまた0・1のパターンを生成するものです。実は、クラス4の規則では、渡されたビットパターンの一部(以後、これをビットパターンと呼ぶことにします)に対して、下記の特徴を備えたビットパターンを(どこかの時刻で)生成することができるのです。

- 渡されたビットパターンと同じものが、同じ列に生成される。
- 渡されたビットパターンと同じものが、(規則性を持って)別の列に生成される。
- 渡された二つのビットパターンが、入れ替わる。
- 渡された二つのビットパターンから、別のパターンが生成される(生成されたパターンは規則性があり、また以後の時間経過において変化しない)。

これがプログラム言語とどのような関係があるかを考えるために、まずは、プログラムを表現するのに必要な機能とは何か、という観点から少し掘り下げてみましょう(要は、何があれば制御構造や各種演算を表せるか、ということです)。天下りのですが、下記の動作をする `subleq(a, b, X)` 命令を考えます。

1. `b`にある値から、`a`にある値を引き、その結果を `b` にしまう。
2. ステップ1の引き算の結果が0以下であった場合、次は場所 `X`にある `subleq` 命令を実行する。
3. 同、結果が0より大きい場合 or `X`の指定がない場合は、単に次行の `subleq` 命令を実行する。

実は、この `subleq(a, b, X)` 命令が一つあれば、プログラムに必要な機能をほぼ全て用意することができるのです。例えば、次のような `if` 文は、

```
(条件分岐前の処理)
if v == 0:
    (条件成立時の処理)
else:
    (条件非成立時の処理)
(条件分岐後の処理)
```

以下のように作れます(しかしながら、相当複雑ですね…)。

```
(条件分岐前の処理)
subleq(a,a)          # aは作業領域, a-a → aより a=0
subleq(b,b)          # bも作業領域, 同様に b=0
subleq(v,a,Label-1)  # a(=0)-v → a, a=(-v) ≤ 0 (つまり v ≥ 0)の時は Label-1へ移動
subleq(a,a,Label-F)  # ここに来るのは -v > 0 (つまり v ≠ 0)の時, 条件非成立時の処理へ移動
Label-1:
subleq(a,b,Label-2)  # a=(-v), b(=0)-a → b, b=(-a)=v ≤ 0の時は Label-2へ移動
subleq(a,a,Label-F)  # ここに来るのは v > 0 (つまり v ≠ 0)の時, 条件非成立時の処理へ移動
Label-2:
subleq(a,a,Label-T)  # ここに来るのは v ≥ 0かつ v ≤ 0 (つまり v = 0)の時, 条件成立時の処理へ移動
Label-T:
(条件成立時の処理)
subleq(a,a,Label-E)  # 条件成立時の処理を終えたので、条件分岐後の処理へ移動
Label-F:
(条件非成立時の処理)
subleq(a,a,Label-E)  # 条件非成立時の処理を終えたので、条件分岐後の処理へ移動
Label-E:
(条件分岐後の処理)
```

上の処理で、`subleq(a, a, Label-xx)` となっている部分は、必ず `Label-xx` へ移動したいため、単に同じ値を引き算しているだけです (結果が 0 以上なので、`Label-xx` へ移動するという意)。また、`Label-2` の存在は冗長で、実は `Label-T` へ直接移動できるのですが、分かり易さのために入れました。

上で示した `if` 文だけではなく、同様に適宜 `Label` を導入することで、`subleq` 命令だけで `while` 文も作ることができます。よって、第 3 回で説明した構造化定理により、プログラムに必要な全ての制御構造は `subleq` 命令だけで作ることができるわけです。さらに演算について言えば、例えば比較演算は、上の `if` 文における `v` の判定と同様な書き方をすれば作成できます。算術計算を行なう演算については、皆さんがこれまでに扱った関数は全て、1) 定数関数、2) 後者関数 (引数の次の値を返す)、3) 射影関数 (引数群から指定された引数を返す)、および a) 関数の合成、b) 原始帰納法により作成できることが示されており⁵、これらも `subleq` 命令だけで作ることができます。比較・算術以外の演算も同様です。つまり、全てのプログラムは、`subleq` 命令だけで作ることができます。以上より、一見多機能なプログラミング言語も、極めてシンプルな命令に分解することができるというわけです。

このような考え方のもと、10 ページで挙げたクラス 4 による出現ビット パターンの特徴をうまく組み合わせてみたら、実はプログラミング (つまり、手順を定義して実行すること) の原始となるシステムを作り出すことができた⁶、というわけなのです (但し、`subleq` 命令の例で見たように、理論上は利用可能であるという意味で、例えば本講義で取り上げたプログラムなどを自在に作成することは現実的ではありません)。これが突破口となり、以後、様々な思わぬものがプログラミング言語として利用できるようになってきました (これも、理論上は利用可能であるという意味で、実際にプログラムを作成するという点では、やはり現実的ではありません)。例えば、Wii U 用のゲーム ソフトであるスーパーマリオ メーカーでは、マリオの通過を邪魔する障害物の出現パターンを設定できます。そこで、0 を障害物 A、1 を障害物 B で表わすことにして、これらの出現パターンをクラス 4 と同等のパターンにする設定を作れるようです (他にも、Minecraft やポケット モンスター、マジック ザ ギャザリングなども話題になっています/まあ、これらの信憑性はよく分かりません…)

⁵これは、クルト・ゲーデル (Kurt Gödel) の偉業です。上記 1) ~ 3) および a), b) により作成できる関数を原始帰納的関数と呼び、多項式関数・有利関数 (分数関数)・三角関数・指数関数・対数関数ほか、初等関数は全て原始帰納的関数に含まれます。但し、最近では原始帰納的関数ではない関数も発見されています。

⁶ここまで来てこんな説明では納得しない or さらに興味を深めたい人のために、もう少しだけ説明しておきます。例えば、 $m+n$ を計算する “簡潔な” システムを考えます。まずは配列 a を用意し、数値 m は m 個の要素に連続して 1 が入ったものとしましょう。(今回は、 $a[0] \sim a[m-1]=1$, $a[m]=0$, $a[m+1] \sim a[m+n]=1$ です)。ここで、 $a[m]$ を基準に、 $a[0] \sim a[m-1]$ を演算子の左項 (L)、 $a[m+1] \sim a[m+n]$ を演算子の右項 (R) とします。次に、現在参照している要素の添字番号をインデックス i とし、 $a[i]$ が L と R のどちらに存在するかを状態 s で表すことにします。そして、 $a[i]$ の値が 1 であれば、1) $a[i]=0$ に変える、2) 適当な場所 $a[X+i]$ に 1 をコピーする、3) i を一つ増やすことにします (これを規則と呼びましょう)。また、 $a[i]$ の値が 0 であれば処理を終了するのですが、但し、状態 s が L の場合は、1) 状態 s を R に変える、2) i を一つ増やすことにします。この時、 $i=0$, $s=L$ に対して上の規則を逐次適用すると、 $a[X]$ 以降の要素に $m+n$ 個の 1 が並び (それ以外は 0)、結果として $m+n$ の計算をしたこととなります。このシステムには、配列・インデックス・状態・規則しかありません (規則と言っても、A: 状態の変更, B: 要素に 0 or 1 を代入, C: インデックスの変更、の三つしか行ないません)。簡潔なシステムですが、実は、計算可能なアルゴリズムは全てこのシステム上でプログラミングできることが証明されています (このようなシステムを、考案者アラン・チューリング (Alan Turing) の名を取って、チューリング機械 (Turing Machine) と呼びます)。これを言い換えれば、チューリング機械の動作を真似できれば、プログラミング言語に必要な機能を備えていることになります。そして、前ページで挙げたクラス 4 の特徴 (4 種類) を用いることで、このチューリング機械を模倣できたというわけなのです。但し、プログラムと言っても、これまで見て来たものとは大きく異なります。1 次元セル オートマトンでは、時刻 t における $a[i]$ の値は、時刻 $t+1$ では $a[i \pm 1]$ に影響を与えます。つまり、ビット パターン (の影響) は、斜め方向に伝わって行くわけです。これより、プログラムに必要なビット パターンは、配列 a の遥か前方 ($a[\rightarrow -\infty]$ のどこか) or 後方 ($a[\rightarrow +\infty]$ のどこか) に入れておくという形態を取ります (念のため、この脚注内の配列は、1 次元セル オートマトンの世界におけるセルを意味しており、Python の配列とは異なります/ Python の配列では、負の添字番号はまた違った意味を持っています)。

3.2 セル オートマトンの応用

1次元セル オートマトンは、その生成規則から得られる複雑な現象について探ることで、前節で紹介した話題だけではなく、秩序とカオスの混在した複雑系 (Complex System) と呼ばれる新しい分野も切り開きました。しかしながら、自然現象や社会現象をコンピュータ上で簡潔にシミュレートするという観点からは、あまり使い勝手の良いものではありません。そこで、取り敢えずここは、計算とは何ぞや現象とは何ぞやという話題を少し脇に置いて (まあ、私はそちらの方が好きですけど)、セル オートマトンを利用したシミュレーションについて考えてみることにしましょう。

セル オートマトンは、自身および近傍の状態により、次の状態が決まるというものです。実世界も、周囲からの影響により様相を変えて行くので、セル オートマトンは実世界を簡潔に表現できるはずですが、但し、実世界では1次元という関係は少なく、大半は複数次元の関係を持っています⁷。そこで、まずは問題を考えるに当たり、セル オートマトンを何次元に拡張すればよいかを考えましょう。それには、対象となる現象を説明するモデルを考える必要があります。

では、次の問題をコンピュータ シミュレーションによる解析を通じて、検討してみましょう。

「現時点で緑地と砂漠が混在する対象地域 A は、砂漠化の進行が懸念されています。

A において、砂漠が今後どの程度まで進行するのか調査しなさい。」

この問題を調査するに当たり、まずは、砂漠が広がる理由 (or 仕組み) をモデル化する必要があります。これは、天候や人間の社会活動が影響していると考えられますが、これらをモデル化する方針として、因果関係を定量的 (or 解析的) に分析するミクロ的視点と、結果の定性的分析に基づくマクロ的視点の二つがあります。今回はマクロ的視点に立ち、シンプルに、

「対象地域 A を小さい矩形 a に分け、a の周囲に砂漠が多い場合は、a の降雨量も減ると想定されるので、a も砂漠化する。」

という方針を考えてみましょう。この方針では、天候が及ぶ影響については、周囲の環境が及ぼす影響に含まれると説明できるので、モデルの対象から外すことができます。次に、社会活動が及ぶ影響ですが、周囲が砂漠であれば、それに伴う天候からの影響により、社会活動に必要な水の確保が益々砂漠化を進めてしまう、と考えることができます。よってこれも、モデルの対象から外すことができます。以上より、今回は矩形領域 a の周囲にある土地の状況だけを考えることにします。これより、各矩形をセルとみなす2次元のセル オートマトンを利用できます。

次に、先ほどの方針に基づき、2次元のセル オートマトンを用いた砂漠化のモデルを定義します。今回は、砂漠化の進行を示す状態変化規則を、図8のように定義しました。

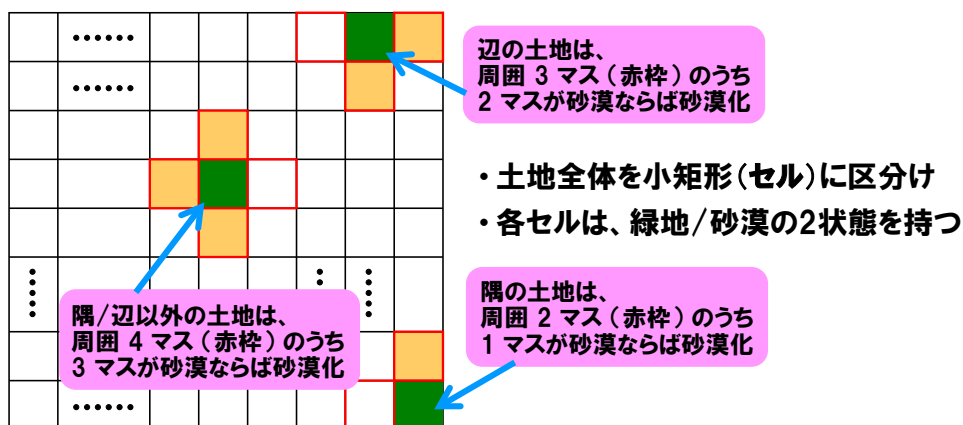


図 8: 砂漠化の進行を示す状態変化規則

⁷ここで言う次元の数は、影響を受ける相手の数に関係すると考えて下さい。この関係を、数学的な2, 3次元で表せるのか、あるいは4次元以上が必要なのかを考えることとなります

このモデルを元に、砂漠化の進行をシミュレートするプログラムを作ることになります。まずは、疑似コードを考えてみましょう。

- cell_auto() — 砂漠化の進行を調べる
 - 必要な変数を用意
 - (# 対象地域を分けした各区画を表す 2 次元配列, 各区画が調査済みであることを示すフラグなど)
 - 区画用データの準備
 - (# 土地データを読み込み、区画データ用の 2 次元配列に格納するなど)
 -
 - 緑地⇒砂漠と変化した区画が存在する間、繰り返し (# 変化した区画の有無はフラグで表す)
 - フラグを True に初期化
 - 左上セルから右下セルまで、繰り返し (# これが調査 1 回分・1 回の調査で変化させるセルは 1 個)
 - もし、調査対象区画が「既に砂漠」であったら、
 - 何もせず次の区画へ移動
 - (枝分かれ終わり)
 - もし、調査対象区画が四隅だったら、
 - 隣接する 2 区画を調べる (# どちらかが砂漠ならば砂漠化が進行)
 - 調査対象区画が砂漠に変わった場合は、フラグを False に変える
 - (枝分かれ終わり)
 - もし、調査対象区画が四辺だったら、
 - 隣接する 3 区画を調べる (# 三つのうち、2 区画以上が砂漠ならば砂漠化が進行)
 - 調査対象区画が砂漠に変わった場合は、フラグを False に変える
 - (枝分かれ終わり)
 - もし、調査対象区画が四隅・四辺でなかったら (# 周囲が囲まれているならば)、
 - 隣接する 4 区画を調べる (# 四つのうち、3 区画以上が砂漠ならば砂漠化が進行)
 - 調査対象区画が砂漠に変わった場合は、フラグを False に変える
 - (枝分かれ終わり)
 - (調査 1 回分の繰り返し終わり)
 - (全調査の繰り返し終わり)
 -
 - 結果を表示

最後は、上の疑似コードを元に Python のプログラムを作成します。緑地・砂漠の状態は、数字の 0・1 で表すことにします。また、各セルの位置関係ですが、2 次元配列を用いれば、図 9 のように扱うことができます⁸。

各セルを配列型データ (例えば dset) で管理

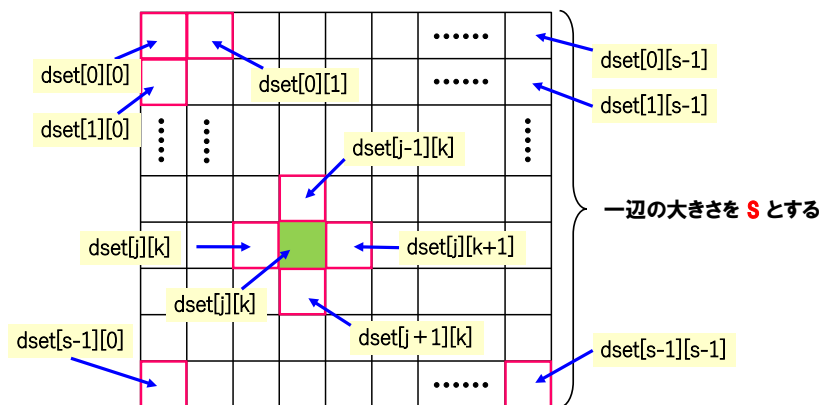


図 9: 2 次元配列を用いた各セル間の位置関係

⁸規模の大きいプログラムを作成する際は、(取り敢えずは簡単なものでも構わないので) 必ず疑似コードおよび、図 8 や図 9 といったモデル図を用意しましょう。これが設計図になります。

以上の検討・設計を踏まえ作成した Python プログラムの例を 17 ページに示します (プログラム内のコメントをしっかりと読んで下さい)。また、以下では、このプログラムを作成する際に留意点について挙げておきます。

このプログラムで利用する土地データは、“land.data”に入っています。また、このプログラムによるシミュレーションは、例えば cell.auto(“land.data”)と実行します⁹。図 10 に、シミュレーションの実行結果を示します。左側が land.data に入っている初期データです。また、右側が土地の最終的な状態 (シミュレーション結果) です。最終状態にある赤字部分は、緑地 → 砂漠に変わった区画を示しています (実際の出力は赤くありません)。このプログラムを実行すると、まずは初期データを出力し、次に最終的な状態を出力します。

001101110111	→	001100110000
011110111010		011110111000
111110101001		111110101000
100011111011		100011111000
100110111110		100000111110
101001101111		100000101110
111110111001		111100111000
001101101001		001101101000
101111111101		001111111100
001111111110		001111111100
100101000010		000101000000
011111111101		000111000000

土地の初期状態

土地の状態変化が終了した状態

図 10: シミュレーション結果の例

緑地 → 砂漠に変わった区画があるかどうかは、フラグ finish に記録されます。finish は、全ての調査用関数が利用するため、プログラムの冒頭で大域変数として定義しています。但し、Python では関数内で大域変数を利用する場合、再度 global 宣言をする必要があります。これは、大域変数と関数内で定義した変数との違いを明確にするためです。大域変数として定義した変数は、各関数へ引数として渡す手間が省けますが、どれかの関数で誤用されると、(特に並列処理においては) エラー箇所の発見が難しくなるため、注意深く利用する必要があります。このプログラムでは、区画データが入った配列 dset も全ての調査用関数が利用します。dset については今回、本体である cell.auto 関数内で定義しました。これは大域変数ではないので、各調査用関数では利用できません。この場合は、引数として渡すことで利用できるようになります (両者を使い分ける意味は、第 6 回資料の図 14 および p.12 を参照)。このような定義する場所に合った有効範囲を “スコープ” と呼びます。プログラムを正しく動作させるには、“スコープ” を正しく把握することが重要になります。

さて、17 ページ以降にあるプログラムですが、最終的に完成すると 250 行前後の規模になります。プログラムの規模としては、決して大きくはないですが、この程度の規模でも初めて作るとなると戸惑うことも多いので、以下に要領を整理しておきます (当然ですが、教科書の「幕間: テストとデバッグの基本」には目を通して下さい)。

- 全てを間違いなく一度に作成することは不可能なので、少しずつ作成して行きます。ある機能を作成した後は、その部分のテストを行ない、これがうまく行けば、次の機能に取り掛かるといわけです。作成した部分までを実行して途中で抜きたい場合は、例えば、一時的にその箇所へ return 関数を入れるという方法があります。テストにより作成した機能が正しく動作することが確認できた後は、もちろんこれ (一時的な return 関数) の削除 (or コメントアウト) を忘れないで下さい。
- 機能単体のテストを行なう場合は、分かりやすいテスト データを用意する必要があります。今回の例で言えば、例えば四隅の調査機能をテストする場合、四隅だけが 1 で残りが 0 である試験用土地データを作成して、テストします (land.data を参考に作ればよいですね)。

⁹cell.auto 関数に渡す引数となる土地データのあったファイルについては、そのパス名に注意して下さい。Google Colaboratory は、データ ファイルの扱いが少々面倒なので、PC にインストールした IDLE or Jupyter Notebook 上でプログラムを作成した方が、次の説明に沿った扱いとなるため見通しは良いです。土地のデータ ファイルと作成した Python プログラムが同じフォルダにある場合は、cell.auto(“land.data”)と実行できますが、違うフォルダにある場合 (例えば、土地のデータ ファイルが data フォルダにある場合) は、cell.auto(“data/land.data”)となります。

- 教科書にある `print` 法を実践するために、今回は `print_data` 関数を用意してあります (適宜、必要な箇所に埋め込んで利用して下さい)。また、特定の処理が正しく実行されているかどうかは、
 - その前後に `print` 関数でメッセージや該当データを出力させてみる。
 - デバッグ用の `if` 文を用意して、メッセージや該当データを出力させてみる。といった手法が有効です (特に、後者を使いこなせるようになれば一人前です)。

最後に

前の方でも述べましたが、セルオートマトンを利用することで、様々な自然現象や社会現象を対象としたシミュレーションを実行することができます。今回は、図 8 (13 ページ) のような状態変化規則を定義しましたが、例えば、規則に適合した場合に必ずしも状態を変化させるのではなく確率的に変化させるとか、隣接セルだけではなく 2 セル以上離れたセルも距離に応じた影響を及ぼすなど、対象に合わせて様々な規則を導入することで、シミュレーションの適合範囲を広げることができます。皆さんも、実践的な技法の一つとして、是非覚えておいて下さい。

参考: セル オートマトンを用いて砂漠化の進行をシミュレートする Python プログラム

```
import ita      # アルゴリズム入門の演習用ライブラリを読み込む。

SIDE = 12      # 地域全体の大きさ (一辺の大きさ)
finish = False # 緑地 => 砂漠へと変化した区画が存在するかどうかを表わすフラグ
              # cell_auto() の起動時に while ループの中へ入れるように、
              # 便宜上、初期値は False にしてある。
              # while ループへ入った後は、改めて初期値を設定する。

# 区画用データの表示 (確認のため、適宜プログラム内に挿入/コメントアウトする)
def print_data(dset):

    # 各変数の宣言
    j = 0; k = 0

    # 動作確認用に、2 次元配列の要素を一つずつ表示
    # dset[j][k] は、(j 行, k 列) の要素 (XY 座標系とは逆なので要注意)
    # print(..., end="") は "改行なし出力", print("") は "改行一つ" を表わす。
    for j in range(SIDE):
        for k in range(SIDE):
            print(dset[j][k], end="")
            print("")

    # 返す値は特にないが、関数定義の終わりを示すために return 関数を入れておく。
    return(None)

# 区画用データの準備
def init_data(fname, dset):

    # 各変数の宣言
    cnt = 0          # 読み込んだデータ数 (区画データの総数)
    j = 0; k = 0

    # 区画データのファイルを読み込む (open したファイルの各情報は、変数 fd に入る)
    fd = open(fname, "r")
    raw_data = fd.read() # ファイルより生データを読み込み、配列 raw_data にしまう。
    fd.close()

    # 生データから区画データ (2 次元配列) に変換
    # 添字番号 (index) の関係および境界処理に注意のこと
    for i in range(len(raw_data)):

        # 改行文字は読み飛ばす。
        if raw_data[i] == "\n":
            continue

        # 便宜上、以後の処理では、区画データを文字データではなく数値データとして利用するため、
        # int 関数で "文字 → 数値" の変換をしておく。
        dset[j][k] = int(raw_data[i])
        cnt = cnt + 1
        k = k + 1
        if k == SIDE:
            k = 0
            j = j + 1

    # 初期データを 2 次元配列全体として表示 (確認用/確認後はコメントアウト)
    print_data(dset)

    # 区画データの総数を返して終了
    return(cnt)
```

```

# 調査対象区画が四辺のどれかであった場合の処理
def check_four_sides(dset, j, k):

    # 各変数の宣言
    global finish
    dp = 0          # 砂漠化した区画の数

    # 調査対象区画が上辺だった場合
    if (j == 0) and ((1 <= k) and (k <= SIDE - 2)):

        # 調査対象区画の左区画/右区画/下区画を調べる。
        # 3 区画を全て必ず調べたいので、elif ではなく if を使う (要注意)。
        # もし砂漠だったら、dp にその区画数をしまっておく。
        if dset[j][k - 1] == 0:
            dp = dp + 1
        if dset[j][k + 1] == 0:
            dp = dp + 1
        if dset[j + 1][k] == 0:
            dp = dp + 1

        # 隣接する 3 区画のうち、2 区画以上が砂漠ならば、その土地も砂漠化
        if dp >= 2:
            finish = False
            dset[j][k] = 0

    # 同、左辺だった場合
    elif ((1 <= j) and (j <= SIDE - 2)) and (k == 0):
        +-----+
        | この部分は未完です。調査対象区画が上辺だった場合を参考に作成して下さい。 |
        +-----+

    # 同、右辺だった場合
    elif ((1 <= j) and (j <= SIDE - 2)) and (k == SIDE - 1):
        +-----+
        | この部分は未完です。調査対象区画が上辺だった場合を参考に作成して下さい。 |
        +-----+

    # 同、下辺だった場合
    elif (j == SIDE - 1) and ((1 <= k) and (k <= SIDE - 2)):
        +-----+
        | この部分は未完です。調査対象区画が上辺だった場合を参考に作成して下さい。 |
        +-----+

    # 調査対象区画が四辺のどれでもなかった場合は、何もしない。

    # 返す値は特にないが、関数定義の終わりを示すために return 関数を入れておく。
    return(None)

```

```

# 調査対象区画が四隅のどれかであった場合の処理
def check_four_corners(dset, j, k):

    # 各変数の宣言
    global finish

    # 調査対象区画が左上端だった場合
    + -----+
    | この部分は未完です。check_four_sides 関数を参考に作成して下さい。|
    +-----+

    # 同、右上端だった場合
    + -----+
    | この部分は未完です。check_four_sides 関数を参考に作成して下さい。|
    +-----+

    # 同、左下端だった場合
    + -----+
    | この部分は未完です。check_four_sides 関数を参考に作成して下さい。|
    +-----+

    # 同、右下端だった場合
    + -----+
    | この部分は未完です。check_four_sides 関数を参考に作成して下さい。|
    +-----+

    # 調査対象区画が四隅のどれでもなかった場合は、何もしない。

    # 返す値は特にはないが、関数定義の終わりを示すために return 関数を入れておく。
    return(None)

# 調査対象区画が周囲を囲まれている場合の処理
def check_other_cells(dset, j, k):

    # 各変数の宣言
    global finish
    dp = 0          # 砂漠化した区画の数

    # 調査対象区画が四隅/四辺でない場合（周囲が囲まれている場合）
    + -----+
    | この部分は未完です。check_four_sides 関数を参考に作成して下さい。|
    +-----+

    # 調査対象区画の周囲が囲まれていない場合は、何もしない。

    # 返す値は特にはないが、関数定義の終わりを示すために return 関数を入れておく。
    return(None)

```

```

# プログラム本体
def cell_auto(fname):

    # 各変数の宣言
    global finish
    cnt = 0          # 区画データの総数 (今回は使っていない)

    # 区画データ用の領域を確保 (今回は一辺が SIDE となる正方形です)
    # 区画データの意味:
    #   砂漠 = 0 (文字の "0" でなく、数値の 0 として扱う点に注意)
    #   緑地 = 1 (文字の "1" でなく、数値の 1 として扱う点に注意)
    dset = ita.array.make2d(SIDE, SIDE)

    # 区画用データの準備
    cnt = init_data(fname, dset)

    # 緑地 => 砂漠と変化した区画が存在しなくなるまで調査を続ける。
    #   finish == True: 緑地 => 砂漠と変化した区画が存在しない。
    #   finish == False: 緑地 => 砂漠と変化した区画が存在する。
    while not finish:

        # 終了フラグを初期化
        + -----+
        | この部分は未完です。finish に正しい値を設定して下さい。|
        + -----+

        # 調査 1 回分:
        #   左上から右下まで、(j 行, k 列) にある区画を一つずつ調べ終わると、
        #   1 回分の調査が終了する。
        + -----+
        | この部分は未完です。|
        | 2次元配列を左上から右下まで一つずつ順に巡る for ループを作成して下さい。|
        | for ループのブロックでは、各区画を dset[j][k] として参照します。|
        + -----+

        # 調査対象区画が既に砂漠であったら、何もせず次の区画へ移動
        if dset[j][k] == 0:
            continue

        # 調査対象区画が四隅の場合は、
        # 隣接する 2 区画のうち、どちらかが砂漠ならば砂漠化
        check_four_corners(dset, j, k)

        # 調査対象区画が四辺の場合 (四隅でなかった場合) は、
        # 隣接する 3 区画のうち、2 区画以上が砂漠ならば砂漠化
        check_four_sides(dset, j, k)

        # 調査対象区画の周囲が囲まれている場合 (四隅/四辺でなかった場合) は、
        # 隣接する 4 区画のうち、3 区画以上が砂漠ならば砂漠化
        check_other_cells(dset, j, k)

    # 結果を表示
    print("")
    print_data(dset)
    return(None)

```