アルゴリズム入門　＃9

地引　昌弘

2024.12.05

はじめに

コンピュータを用いてシミュレーションを行なう場合、必要なデータを全て事前に実世界より取得できるとは限りません。例えば、大規模施設における人流 (人の行き来) の安全な誘導方法を検証する人流シミュレーションを考えてみましょう。施設内にいる人々が、それぞれどこに行こうとしているかを事前に調べることは難しいため (例えば、時間や曜日、年齢や性別などにより様々ですよね)、これらは適当な分布を割り当てざるを得ません。このような場合に、適当な分布を与えてくれる仕組みとして乱数があります。そこで今回、まずは乱数について取り上げます。

その後は、これまで学んだ基本的な制御構造 · データ型 · 数値解析手法をもとに、より複雑なアルゴリズムのための制御構造やプログラムについて考えて行きます。特に、今回取り上げる「再帰」構造を理解できるようになると、複雑な問題を簡潔に表せるようになります。

1　前回の演習問題の解説

1.1　演習 8-2a — p 値を用いた仮説検定

仮説検定では一般に、母集団の確率分布が不明な場合 (かつ正規分布と想定しても、大きな乖離がない場合) には、着目するパラメータに応じて、前回紹介した t 検定を始めとする様々な検定方法が存在します。これに対し、母集団の確率分布が分かっている場合 (かつ簡単な計算で求められる場合) は、実際に確率を求めることで、だいぶ見通し良く仮説検定を行なうことができます。この演習で用いる観測データ (広告を流す前後でユーザに対して実施した知名度に関するアンケート結果) は、下記の通りでした:

	商品 a を知っている	商品 a を知らない
広告前	17 人	21 人
広告後	35 人	23 人

この結果より、各ユーザは、広告を流さずとも 17/(17+21=38) の確率で商品 a を知っています。よって、帰無仮説 (知名度が上がったように見えるのは偶然である) の立場から、広告後に 17/38 の確率で (つまり、広告後も確率に変化はないとして) 58 人中 35 人以上のユーザが商品 a を知っている確率 (即ち p 値) は、次の数式で表わされます:

$$\sum_{i=35}^{58}(\frac{17}{38})^i(1-\frac{17}{38})^{58-i}\cdot {}_{58}C_i \tag{1}$$

式 (1) に従って p 値を計算する Python プログラムは、以下の通りです (組合せ数を計算する方法については、第 5 回資料の 4 ページで説明してあります):

```
def comb(n, r):
  result = 1
  for i in range(r):
    result = result*((i + 1) + (n - r))/(i + 1)
  return(result)
```

```
  # user_y1: 広告前から既に知っていたユーザの数
  # user_n1: 広告前は未だ知らなかったユーザの数
  # user_y2: 広告後に知っていたユーザの数
  # user_n2: 広告後も知らなかったユーザの数
  def h_test1(user_y1, user_n1, user_y2, user_n2):

    # n2: 広告後にアンケートを受けたユーザ数
    n2 = user_y2 + user_n2

    # p_y1: 広告前のユーザが知っている確率， p_n1: 同、知らない確率
    p_y1 = user_y1/(user_y1 + user_n1)
    p_n1 = 1 - p_y1

    # さらに極端な結果も含む p 値の計算（式 (1) の Σ 部分）
    # 極端な場合の確率は値が小さいので、情報落ちを防ぐために足し算の順番を変える。
    val = 0
    for i in range(n2, user_y2 - 1, -1):    # カウンタの終値に注意
#     print(i)
      val = val + (p_y1**i) * (p_n1**(n2 - i)) * comb(n2, i)
    result(val)
```

帰無仮説では極端な結果も想定するため、“58 人中 35 人が商品 a を知っている” 確率から、“58 人全員が商品 a を知っている” 確率まで足し算する必要があります。データ総数が大きい場合、極端な結果の確率は値が極めて小さい可能性があり、val に大きな値が入っていると情報落ちが発生する恐れがあります。よって、このプログラムでは、小さい順から (極端な場合から) 足し算をするようにしています。その際、range 関数におけるカウンタの終値の扱い (第 6 回資料の 8 ∼ 9 ページ参照) に注意して下さい。実行結果 (p 値) は、以下の通り:

```
  >>> h_test1(17, 21, 35, 23)
  0.012132687797913232
```

1.2 演習 8-2b — 十分な仮説検定に必要なデータ数 (自由度)

次は、アンケートを実施するユーザ数を 500 人に増やし、このうち何人のユーザが広告後に商品 a を知っていれば、有意水準 0.05 で広告に効果があったと言えるか、該当人数を求めるという演習でした。帰無仮説の立場から、広告後に 500 人中 k 人以上のユーザが商品 a を知っている確率は、次の式で表わされます。

$$\sum_{i=k}^{500}(\frac{17}{38})^i(1-\frac{17}{38})^{n-i}\cdot {}_{500}C_i < 0.05$$

この数式を満たす最大の k を求めるわけですが、これを解析的にいきなり求める方法はないので、k を 0 から一つずつ増やして行き (つまり、広告後の商品 a を知っているユーザ数を 0 人から増やして行き[1])、$\sum$ の結果が 0.05 未満となる k を求めます。Python によるプログラムは、以下の通りです (ここでは、演習 8-2a で作成した h_test1 関数を利用しています):

```
  # n:     調査対象のユーザ総数
  # s_lv: 有意水準の値
  def h_test2a(user_y1, user_n1, n, s_lv):
    for k in range(0, n + 1):    # カウンタの終値に注意
      val = h_test1(user_y1, user_n1, k, n - k)
      if val < s_lv:
        break
    return(k)
```

[1]帰無仮説では極端な結果も想定するため、例えば $k = 0$ とは、“n 人中 0 人が商品 a を知っている場合” から、“n 人全員が商品 a を知っている場合” までを表わすことに注意して下さい (つまり、p 値は 1)。よって、k が増えるにつれ、p 値は 1 から順に減って行きます。

実行結果は、以下の通り:

```
>>> h_test2a(17, 21, 500, 0.05)
243
```

各ユーザは、広告を流さずとも 17/38 の確率で商品 a を知っており、これを適用すると 500 人では 224 人が知っていると想定されるので、ここから 20 人程度増えただけで広告に効果があったと言えるのは、少々不思議な感じがします。これを理解するために、1/2 の確率で表 · 裏が出るコインを投げる場合を考えてみましょう。表が r 回出る確率は二項分布に従いますが、実は、二項分布は非常に収束の速い分布であり、n が大きくなるにつれ、r は平均値 $n/2$ から離れた値をほとんど取らなくなります。

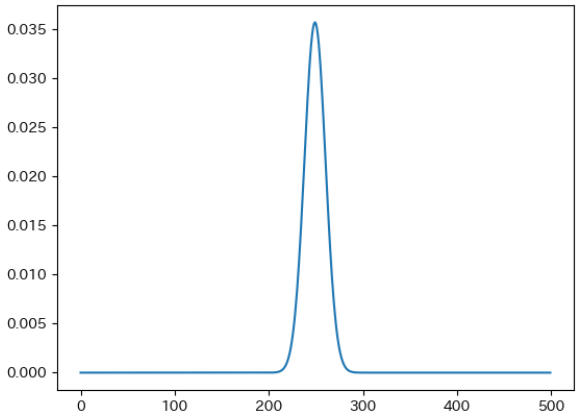

図 1: コインを 500 回投げた時に表が出る確率

図 1 を見てみると、250 ± 25 回ぐらいの範囲にほとんどの場合が入っており、表が 220 回出ることなどほぼありません (確率: 220 回＝0.001, 225 回＝0.004, 230 回＝0.008)。つまり、二項分布に従う 500 個のデータが平均値より 20 個分偏ることは、だいぶ少ないと言えるわけです(各回の確率値ではなく、累積確率であることに注意)。

ところで、上のプログラムは、演習 8-2a で作成した `h_test1` 関数を利用しており、分かり易いですね。これを、`h_test1` 関数を使わずに全て一つの関数で記述する場合は、次のようになります:

```
def h_test2b(user_y1, user_n1, n, s_lv):
  p_y1 = user_y1/(user_y1 + user_n1)
  p_n1 = 1 - p_y1
  for k in range(0, n + 1):
    val = 0

    # 商品 a を知っているユーザ数が k の場合における p 値を計算する。
    for j in range(n, k - 1, -1):
      val = val + (p_y1**j) * (p_n1**(n - j)) * comb(n, j)
    if val < s_lv:
      break
  return(k)
```

`h_test2b` 関数では、k を増やすループ (L1) と帰無仮説で極端な場合を想定するループ (L2) の、二重ループを用意する必要があります。この時、p 値の計算における情報落ちを防ぐため、L1 と L2 では、カウンタの増やし方 (ループ処理の順番) を逆にする必要があります。また、ループ処理の範囲を規定する `range` 関数の引数を、L1 · L2 間で整合させることも必要になります (L2 の設定は誤り易いですね)。加えて、各 k 毎に L2 を実行する (p 値を計算する) 前に、p 値をしまう `val` の初期化も必要となります。これらを考慮して作成する `h_test2b` 関数は、`h_test2a` 関数に比べて少々複雑な構造になっています。これを最初から正確に作成することは、難しそうですね。`h_test2a` 関数のように、プログラム内にて独立した手順として切り出せる部分を別の関数として定義し、見通し良く利用する手法を**モジュール化** (Modularization) と呼びます。モジュール化は、プログラムの開発効率や信頼性を向上させる重要な手法なので、皆さんもプログラムを作成する際はモジュール化を必ず意識して下さい。

2　乱数とランダム アルゴリズム

2.1　乱数とは

一般に**乱数**とは、「ある分布に従う、互いに独立な事象を表す確率変数の実現値」を指します。また、その並びを**乱数列** (Randam Sequence) と呼びます。例えば、図 2 にある区間 $[0, 1)$ の**一様分布** (Uniform Distribution) であれば、乱数の範囲は 0 以上 1 未満で、その区間にあるどの数も同じくらいの確率で出現します。これを**一様乱数** (Uniform Random Number) と言います。偏りのないサイコロを振って出る目の数は 1 以上 6 以下の整数値ですが、どの数も同じ確率で出現するため、これも一様乱数です。この他によく使われる乱数としては、前回の付録で取り上げた**正規分布** (Normal Distribution) に従う**正規乱数** (Normal Random Numbers) があります。

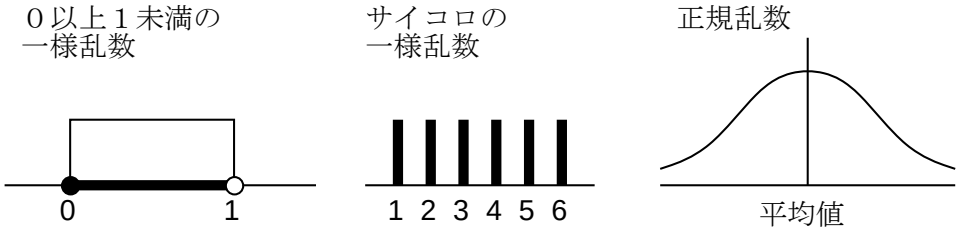

図 2: 乱数と分布

2.2　擬似乱数

擬似乱数 (Pseudorandom Number) とは、プログラムで順次計算を行なうことにより生成される数値の列で、乱数列のように見えるものを言います。しかしながら、プログラムの動作は完全に決定的なので、「でたらめな」数を生成するのは思いのほか難しいものです。擬似乱数を生成する代表的なアルゴリズムとして、主に下記のアルゴリズムがあります:

- **自乗採中法** (Middle-Square Method) — w ビットの数を 2 乗すると $2w$ ビットになるので、そこから中央付近の w ビットを取り出して「次の数」とする。これを繰り返すことで、w ビットの乱数列を生成する[2]。
- **線形合同法** (LCM: Linear Congruential Method) —剰余計算を入れた漸化式 $x_{i+1} = (x_i \times a + c)\ mod\ m$ により、次々に値を生成して行く[3]。
- **メルセンヌ ツイスター** (MT: Mersenne Twister) — 1997 年に松本眞、西村拓士が開発した乱数アルゴリズム[4]。

どのようなアルゴリズムでも計算方法が決まっている以上、値を順次生成して行く過程で既出の値が再度現れてしまうと、それ以降の数列は前と同じものの繰り返しになります。これを**周期** (Period) と呼び、もちろん周期の長い乱数が望まれます。MT は、$2^{19937} - 1$ という長い周期を備えた画期的な乱数生成法です。**Python では、`random.random` 関数 (引数なし) で区間 $[0, 1)$ の実数一様乱数が得られます**[5]。`random.random` 関数の中では、乱数アルゴリズムとして MT が使われています。

この他、**オペレーティング システム** (OS: Operating System — コンピュータ上で常に稼働し、資源管理などを行う基本ソフトウェア) が、乱数機能を提供する場合があります。OS の乱数機能は、ユーザのキーボード入力やディスクの動作など、外部からの操作などに基づいた「ランダムさ」を活用するので、擬似乱数のような周期の問題がありません。さらに最近では、CPU チップ自体に物理的な乱数発生装置を持つものもあります。

---

[2]自乗採中法は古くからあるアルゴリズムですが、あまり良い乱数列を生成できないことが知られています。

[3]線形合同法は、MT の発明以前は主流のアルゴリズムでした。但し、良い擬似乱数とするためには、パラメタ $a,\ c,\ m$ の選定に注意が必要となります。

[4]興味を持った方のため、MT の原理を少しだけ紹介しておきます。LCM は剰余を使うことから (見通しを良くするため、以下では漸化式を $x_{i+1} = x_i\ mod\ m$ と簡略化します)、乱数値を m 個生成すると同じ値が必ず入ってしまいます。その理由は、$x_{i+1} = \underline{x_i}\ mod\ m = \underline{x_j}(= x_i + m)\ mod\ m$ となるからです (これを周期 m と呼ぶことにします)。ここで、下線部分が意味することを (即ち x_i と x_j の関係について) 注意深く考えてみましょう。この式は、x_i の値だけに応じて x_j が決まることを示しています。これは、漸化式にて x_i と x_{i+1} が一対一に対応しているからです。そこで、この一対一対応を緩和するため、漸化式を $x_{i+n} = \underline{(x_{i+(n-1)} + x_{i+(n-2)} + \cdots + x_i)}\ mod\ m$ と拡張してみることにします。この漸化式にある下線部分を X と置くと、$x_{i+n} = X_{i+(n-1)}\ mod\ m$ より、$i = j + m$ の時、$X_{i+(n-1)} = X_{j+(n-1)}$ となります。しかし、$X_{i+(n-1)}$ を構成する要素 $\{x_{i+(n-1)},\ x_{i+(n-2)},\ \ldots,\ x_i\}$ と、$X_{j+(n-1)}$ を構成する要素 $\{x_{j+(n-1)},\ x_{j+(n-2)},\ \ldots,\ x_j\}$ に、$x_{i+(n-t)} = x_{j+(n-t)}$ (但し、$i = j + m$) という関係はありません。簡略化のため $m = 2$ とした場合、$x_{i+(n-t)} = \{0, 1\}$ より $X_{i+(n-1)} = a$ となる要素のパターンは ${}_{n-1}Ca$ 通り存在し、周期もこれに準じて大きくなります。MT は、この考え方を拡張しています。

[5]`random.random` 関数を利用する場合は、random モジュールを事前に読み込んでおく (`import random` しておく) 必要があります。

2.3 ランダム アルゴリズム

ランダム アルゴリズム (Randomized Algorithm) とは、(擬似) 乱数を活用して、ランダムな振舞いを持たせたアルゴリズムを言います。これに対し、通常の決定的な動作を行なうアルゴリズムは、**決定的アルゴリズム** (Deterministic Algorithm) と呼ばれます。

ランダム アルゴリズムは、対象とする問題の性質に応じて、実は決定的アルゴリズムより効率的な場合があります。例えば、1 億個の要素を持つ配列 a があるとして、下記のどちらかの状態になっているとします。

- その半分に該当する 5 千万個の要素には値 x が入っているが、それがどこかは分からない。
- 値 x が全く入っていない。

以下では、配列 a がどちらであるかを判断する場合を考えてみましょう。もちろん、絶対に間違いがないように調べなければならない場合は、1 億個全ての要素を調べる必要があります。しかし、実用上困らない範囲で調べればよいとなれば、乱数を用いて 1 億個の要素からランダムに幾つかを選び、その値を調べる方法が効率的です。例えば、10,000 個ほど調べた結果、1 回も値 x に遭遇しなければ、「値 x はない」と判断してまず問題ありません。何故ならば、この判断が間違っている確率は $1/2^{10000}$ であり、それはこの計算をするコンピュータが故障する確率よりはるかに小さいからです。

このような、微小だが 0 ではない「間違う可能性」を持ったアルゴリズムを**モンテカルロ アルゴリズム** (Monte Carlo Algorithm) と言います。これに対し、たとえ確率的な (乱数的な) 方法を使いながらも、「間違うことがない」アルゴリズムを**ラスベガス アルゴリズム** (Las Vegas Algorithm) と言います[6]。ラスベガス アルゴリズムは、(絶対に間違いが起こらない) 決定的アルゴリズムを用いるため、状況によっては処理時間が実用的でなくなる等、答えを出さない場合があります。(少し分かりにくいかも知れませんが) ラスベガス アルゴリズムの例として、データを大きい順に並べるアルゴリズムを考えてみましょう。データの組から乱数により一つを取り出し、これと残りのデータを比較して行く場合 (この比較を繰り返すことで、必ず正しい順番に整列できることは明らかですよね)、取り出す値が常に最悪の場合は (このような確率は 0 ではありません)、最終的な処理時間が実用的な範囲に収まらないことがあります (つまり、整列アルゴリズムが答えを返さないように見える)。

2.4 モンテカルロ法

モンテカルロ法 (Monte Carlo Method) とは、**シミュレーション** (Simulation) などにおいて乱数を活用する手法を言います。例えば、交通の流れを実際に観察する代わりに、乱数を用いてランダムに車を (プログラム内で) 走らせ、それらの車がどのように流れて行くかを見ることで、交通制御の様々な方式による交通状況を簡単に試すことができます。

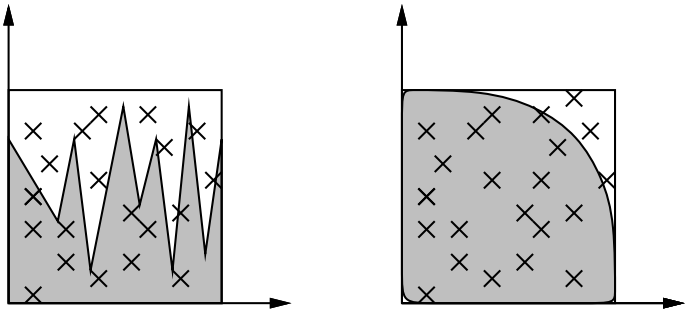

図 3: モンテカルロ法による数値積分

また、モンテカルロ法は数値積分にも使うことができます (図 3)。具体的には、積分範囲にある最大の関数値による長方形の領域を考え、その範囲内に乱数で多数の点を打ちます。そして、関数内部に含まれる点と、打った点の総数との比率から、積分値を近似的に求めるわけです。以前 (#4, #5) に説明したように、数値的に積分を求める場合は、シンプソンのアルゴリズムを用いることで速く正しい結果を得られます。しかし、(積分した結果の関数ではなく) 被積分関数自体の計算が難しい場合や、連続かつ微分可能 (滑らか) でない場合などは、シンプソンのアルゴリズムをうまく利用できません。このような場合はモンテカルロ法が有力な手法の一つとなるのです。

[6] ラスベガスは米国にあるカジノで有名な都市の名前ですね。実は、モンテカルロもヨーロッパにあるカジノで有名な都市の名前です。

では、モンテカルロ法による数値積分の例として、半径 2 の 1/4 円の面積を求め、π の近似値を計算してみましょう[7]。以下のプログラムでは、`x=2.0*random.random(); y=2.0*random.random()` より、(x, y) は 2.0 × 2.0 の正方形内へ無作為に打たれますが、これらの点は `count/n` の確率 (つまり割合) で 1/4 円に入っています。よって、`(count/n)*4.0` が、1/4 円の面積に該当するというわけです(全体となる正方形の面積 4.0 の内、`(count/n)*4.0` の割合が 1/4 円の面積)。この考え方は、モンテカルロ法による数値計算の基本的な考え方なので、よく理解しておいて下さい。

```
import random

def pirandom(n):
  count = 0
  for i in range(n):
    x = 2.0*random.random(); y = 2.0*random.random()
    if x**2 + y**2 < 4.0:
      count = count + 1
  return((count/n) * 4.0)    # 4.0 は 2.0 × 2.0 正方形の面積であることに注意
```

実行結果は次の通り:

```
>>> pirandom(10000)          ← 1 万回
3.1452
>>> pirandom(100000)         ← 10 万回
3.14244
>>> pirandom(1000000)        ← 100 万回
3.140732
>>> pirandom(10000000)       ← 1000 万回
3.1416192
>>> pirandom(100000000)      ← 1 億回
3.1414702
>>> pirandom(1000000000)     ← 10 億回
3.141548644
```

有効数字が 5 桁前後では使えない、と思いますか? 実際には、5 桁の有効数字が得られれば十分な場合は結構あります。例えば、有効数字 5 桁の円周率 (3.1415) と NASA が宇宙開発に用いる有効数字 15 桁の円周率 (3.141592653589793) を使って、それぞれ地球の赤道面における円周を計算してみると、その差は 1.182km です。地球 1 周 40,000km に対して誤差が 1km 程度ということであれば、通常の社会活動においてはあまり大きな影響はないですよね (とは言え、宇宙空間のような極めて大きな対象を扱う場合は、やはりそれなりの有効数字が必要になります)。

2.5 モンテカルロ法の誤差

モンテカルロ法による誤差について、もう少し調べてみましょう。以下では、関数 $y = x$ の区間 $[0, 1)$ における積分を求めてみます。答えは、両辺が 1 の直角 2 等辺三角形の面積ですから、0.5 であることはすぐ分かりますね。モンテカルロ法によるプログラムは次の通り:

```
def integrandom(n):
  count = 0
  for i in range(n):
    x = random.random(); y = random.random()
    if y > x:
      count = count + 1
  return((count/n) * 1.0)
```

では実行させてみます:

[7]もちろん円周は十分連続かつ微分可能ですが、それは置いておいて。

```
>>> integrandom(100)
0.46                        ← 誤差 0.04
>>> integrandom(1000)
0.533                       ← 誤差 0.033
>>> integrandom(10000)
0.4967                      ← 誤差 0.0033
>>> integrandom(100000)
0.49768                     ← 誤差 0.00232
>>> integrandom(1000000)
0.50068                     ← 誤差 0.00068
>>>
```

これを見る限り、どうも試行数が 100 倍になると、誤差が $\frac{1}{10}$ に減って行くように見えます。その理由について考えてみましょう。このプログラムでは、1 回の**試行** (Trial) において「打った点が関数 f の上か下か」つまり「0 か 1 か」を調べています。もし上であれば `count` は増やさず (つまり 0 を足し)、下であれば `count` を 1 増やして、最後に試行回数 N で割っています。これは、「0 か 1」を取る確率変数の期待値を求めていることと同じです。そして、この確率変数が 1 となる確率は関数の面積と等しいため、N を増やしていけば**大数の法則** (Law of Large Number) により、観測される期待値 (確率変数が 1 になった回数) が理論的期待値 (この場合は関数の面積) に近付いて行くわけです。では、どれくらいの速さで近付くのでしょうか。それは**中心極限定理** (Central Limit Theorem) が教えてくれます。真の分布における平均が μ、その分散が σ^2 の時、観測される平均値を $\bar{X}$ とすると、その誤差 $\bar{X}-\mu$ は平均 0、分散 $(\frac{\sigma}{\sqrt{N}})^2$ の正規分布に収束します[8]。これより、試行回数を N 倍にすると誤差の散らばり具合 (標準偏差 = 分散の平方根) は $\frac{1}{\sqrt{N}}$ 倍になるわけです。これは確かに上の結果と合致していますね。

最後に、乱数に基づいて点を打つのでなく、一定の規則により (例えば格子点上に) 点を打った場合との違いを見てみましょう。上と同じ積分を、格子上の点により計算してみます。簡単のため、縦横の分割数を同じ値とし、プログラムへはこの値 N を与えるようにしました:

```
def integgrid(n):
  count = 0; d = 1.0/n
  for i in range(n):
    y = i*d           # 本来ならば x = x+h, y = f(x) とすべきだが、今回は y = x なので省略
    for j in range(n):
      if y >= j*d:    # y = f(x) = x(=i*d) より下の領域にある格子点を数える。
        count = count + 1
  return((count/(n**2)) * 1.0)
```

動かしてみた結果は次の通り:

```
>>> integgrid(10)
0.55
>>> integgrid(100)
0.505
>>> integgrid(1000)
0.5005
```

実際の格子点数は 2 乗で増えて行くので、100、10,000、1,000,000 点となります。しかし、これを見る限り、何やら「規則的」な変化に見えますね。

そこで今度は、区間 $(\frac{0}{100},\frac{1}{100})$、$(\frac{2}{100},\frac{3}{100})$、$\cdots$、$(\frac{98}{100},\frac{99}{100})$ で 1、それ以外で 0 となるちょっと意地悪な関数を考えてみましょう (図 4)。x の定義域は $0 \le x \le 1$ で、区間の両端は含まれて**いない**ことに注意して下さい。

[8]実は真の分布に種類の制限はありません!! 平均と分散を持つ全ての分布が対象です。誤差が正規分布に従う事情については、前回の付録で説明しました。ここではどちらも誤差に言及していることから、正規分布と中心極限定理は同じことを言っているように誤解し易いので少し整理しておきます。正規分布は、ある真の値 X を n 回観測して得られた誤差の分布です。これに対し中心極限定理は、分布全体を n 回観測して得られた平均値の分布です。中心極限定理の厳密な証明は少々難しいです (新しい数学的知識が必要になります)。

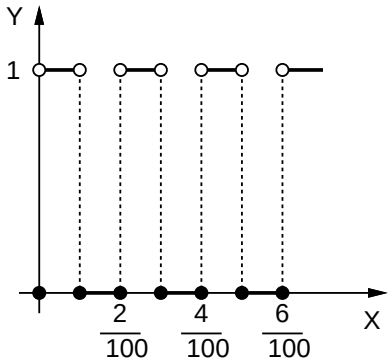

図 4: 意地悪な関数の例

この関数値を計算する oddfunc 関数と、それを格子点で積分する関数を用意しました:

```
def oddfunc(x):
  if (x*100 == int(x*100)): return(0)     # 区間の両端→ 0 を返す
  else: t = int(x*100) % 2                # 区間数の計上/意味は本文の説明 (次ページ) を参照
  if t == 0: return(1)                    # t の反対を返す
  else: return(0)

def integoddgrid(n):
  count = 0; d = 1.0/n
  for i in range(n):
    for j in range(n):
      if oddfunc(i*d) >= j*d:     # 厳密には 0 <= i*d < 1 だが、取り敢えずここでは無視
        count = count + 1         # y = f(x) = oddfunc(x(=i*d)) より下の領域にある格子点を数える。
  return((count/(n**2)) * 1.0)
```

動かした結果は次の通り:

```
>>> integoddgrid(10)
0.28
>>> integoddgrid(100)
0.0496
>>> integoddgrid(1000)
0.454546
```

ほとんど「滅茶苦茶」ですね…(正しくは 0.5 になるはずです)。では、モンテカルロ法はどうでしょうか:

```
def integoddrandom(n):
  count = 0
  for i in range(n):
    x = random.random(); y = random.random()
    if y >= oddfunc(x):
       count = count + 1
  return((count/n) * 1.0)
```

こちらの実行結果は次の通り:

```
>>> integoddrandom(100)
0.35
>>> integoddrandom(10000)
0.4876
>>> integoddrandom(1000000)
0.499372
```

関数 `oddfunc(x)` の動作について、少し説明しておきます。このプログラムにある `int(x*100)` は、$0 \sim 1$ の範囲にある `x` を $0 \sim 100$ の整数に変えることで、`x` が原点から数えて何個目の関数区間に入っているかを示します (関数区間 $(\frac{i}{100}, \frac{i+1}{100})$ が、整数区間 $(i, i+1)$ に該当)。よって、`t = int(x*100) % 2` より区間数の偶奇を求めています。

モンテカルロ法では、他の関数の場合と変わらず、試行回数を N 倍にすると誤差は $\frac{1}{\sqrt{N}}$ に減っています。その理由ですが、格子点のような一定規則に基づく点を利用すると、先の意地悪な関数のように、関数に周期性がある場合は点を打つ部分がそれと同期してしまい (例えば、x 座標が $\frac{0}{100}, \frac{1}{100}, \frac{2}{100}, \frac{3}{100}, \ldots, \frac{98}{100}, \frac{99}{100}$ となる位置で点を打ってしまうとか)、常に偏った点を打ってしまいます。

以上、「おかしな」関数でもそれなりに計算できるというモンテカルロ法の利点が、お分かりいただけたかと思います。

演習 9-1 モンテカルロ法で、π の近似値を計算するプログラムを作成せよ。

参考:

`oddfunc()` は意地悪な関数という設定なので、少々動作の追い辛い超 (?) 技巧的な版も紹介しておきます。

```
def oddfunc(x):
  t = x*100 % 2
  if 0 < t and t < 1: return(1)
  else: return(0)
```

上記のプログラムにある "`t = □ % 2`" の演算では、$2n < \square < 2n+1$ の時に $0 < t < 1$ を、$2n-1 < \square < 2n$ の時に $1 < t < 2$ を返します (これは、知っていないと (コードを見ただけでは) 思い付かないですよね)。そこで、`x*100` により関数区間 $(\frac{i}{100}, \frac{i+1}{100})$ 内にある x を整数区間 $(i, i+1)$ 内にある X へ変換した後、`X % 2` を計算します。その結果、$0 < t < 1$ であれば X、即ち x が原点から数えて奇数番目の区間に入っていると分かるわけです (`if 0 < t and t < 1: return(1)`)。それ以外 (x が偶数番目の区間にある, $x = 2n$, $x = 2n+1$) は、全て 0 になります (`else: return(0)`)。

3　再帰呼び出し

3.1　再帰手続き · 再帰関数の考え方

関数や手続きの興味深い用法として、ある関数の中から直接または間接に自分自身を呼び出す、というものがあります。これを**再帰** (Recursion) と呼びます。例えば、正の整数 a、b について、その最大公約数を $gcd_sub(a, b)$ とした場合、以下の関係が成り立ちます[9]:

$$gcd_sub(a,\ b) = \begin{cases} a & (a = b) \\ gcd_sub(a-b,\ b) & (a > b) \\ gcd_sub(a,\ b-a) & (a < b) \end{cases}$$

この関係式に従い、`a = b` が成立するまで単純にループを回す Python のプログラムは、以下のようになります:

```
def gcd_sub1(a, b):
  while a != b:
    if a > b:
      a = a - b
    else:
      b = b - a
  return(a)
```

これに対し、パラメータを変えながら自分自身を呼び出す再帰を用いた Python プログラムは、以下のようになります:

```
def gcd_sub2(a, b):
  if a == b:
    return(a)
  elif a > b:
    return(gcd_sub2(a-b, b))
  else:
    return(gcdsub2(a, b-a))
```

ループ版に比べて再帰版の方が、関係式と直接対応しているので、プログラムそのものは大変分かり易いのですが、なぜ「堂々巡り」にならずに計算が終わるのでしょうか。それは、図 5 を見れば分かります。

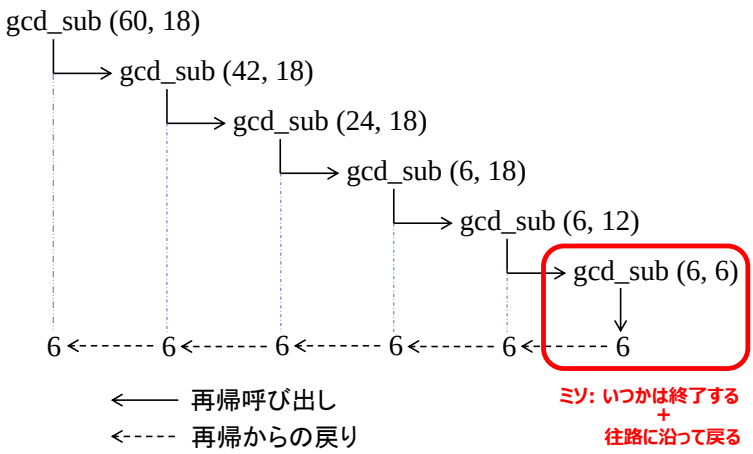

図 5: 再帰関数による最大公約数の計算

[9]参考のため、$gcd_sub(a, b) = gcd_sub(a-b,\ b)$ $(a > b)$ が成立する理由を示しておきます。
略証: a と b の最大公約数を G とします。a と b は G の整数倍なので $(a = pG, b = qG)$、$a - b$ もまた G の整数倍です。これより、G は $a - b$ と b の公約数です (最大かどうかはまだ分かりません)。ここで、G より大きい $a - b$ と b の公約数 $H(> G)$ が存在すると仮定します。H は、$a - b$ と b の約数なので $(a - b = rH, b = sH)$、$(a - b) + (b) = a$ の約数になります。これより、H は G より大きい a と b の公約数になります。これは、G の定義に矛盾します。よって、G は $a - b$ と b の最大公約数になります。

図 5 (前ページ) を注意深く見てみると、再帰関数 (再帰手続き) の動作は、次の原則に従っていることが分かります:

- 問題を「それ以上簡単化 (or 単純化 · 細分化) できない場合 (取り敢えずここでは、これを最小粒度と呼ぶことにします)」は、結果を返す (上の例では $a = b$ の場合)。
- それ以外は、問題を「少し簡単な問題に変形した上で」自分自身を呼び出す (上の例では、少し小さい数の最大公約数問題に変形)。

扱う問題に最小粒度が存在する場合 (例えば、実数を半分にして行くような問題は、最小粒度が存在しません)、上の原則に従って問題を順々に簡単化して行くと、いつかは最小粒度に至るため、堂々巡りとはならずに正しく実行できる (必ず結果が返る) わけです。

3.2 再帰呼び出しの興味深い特性

再帰呼び出しの興味深い特性として、「現在実行している者 (コード) と、再帰的に呼び出した自分とは、動作は同一だが (同じプログラムだから当然 !)、人物としては別人」という関係があります。例えば、${}_nC_r$ の計算の様子を図 6 に示します。${}_5C_3$ を計算する場合、「私」は「${}_4C_2$ を計算する人」と「${}_4C_3$ を計算する人」に作業を依頼します。これらの「人」は、データ (n とか r) が「私」とは違っているので別人ですが、動作は「私」と同じです。

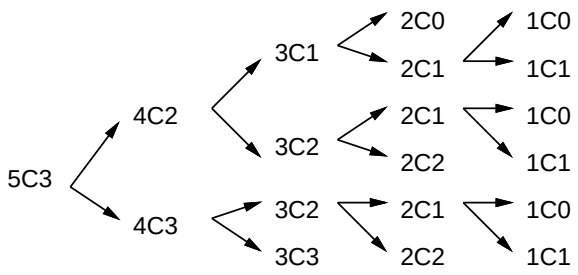

図 6: 再帰関数による組合せ数の計算

このような一種の別人格性を利用すると、興味深い処理が可能になります。例えば、1 ∼ 3 の数字を一つ打ち出す場合は、次のように一重ループを使えばできますね:

```
for i in range(1, 4):
  print(i)
```

「1 ∼ 3 が二つ並んだ全ての組み合わせ」を打ち出す場合はどうでしょう。2 次元配列の各要素を取り扱った時を思い出せば、これもループを二重にすればできることが分かりますね:

```
for i in range(1, 4):
  for j in range(1, 4):
    print(i, j)
```

では、「N 個」だとどうでしょう。最初から N が分かっていれば N 重のループを用意できそうですが、例えば、外部から呼び出された時のパラメータとして N を指定された場合 (つまり、N の具体的な数値が事前に判明していない場合) など、「1 ∼ 3 が N 個並んだ全ての組み合わせ」を作れるでしょうか?

このような場合は、次のような再帰呼び出しを用いて求めることができます (n には並びの個数 N を指定する):

```
def nest3(n, s):                # 呼び出し方: nest3(5, "") ← 空文字列を渡す
  if n <= 0:
    print(s)
  else:
    for i in range(1, 4):
      nest3(n-1, s + str(i))    # 文字列 s に、数値 i を str 関数で文字に変換した後、連結 (+) する。
  return
```

ここでは、「自分」が「親」より、担当部分 n と数字が並んだ文字列 s を渡されています。担当部分が 0 の場合は、「文字列を打ち出す」ことが自分の仕事です。担当部分が 0 以外の場合は、ループを用いて「親」から渡された文字列 s に 1 ～ 3 の数字 (str(i)) を一つずつ追加 (+) し、担当部分を一つ小さくして (n - 1)、3 人の「子」を呼び出します。各呼び出しの様子を図 7 に示します。

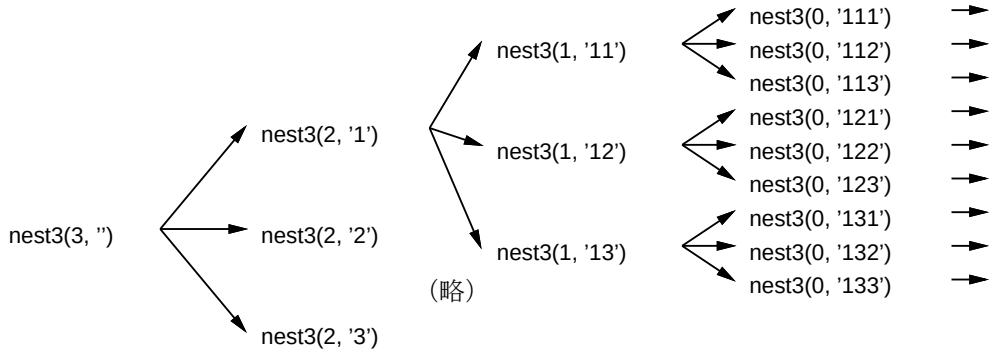

図 7: 1 ～ 3 が三つ並んだ全ての場合を出力

参考までに、実行の様子も示しておきます:

```
>>> nest3(5, "")     ← 1～3 が 5 個並んだ全ての組み合わせを表示する
11111
11112
 (略)
33332
33333
```

3.3 再帰呼び出しによる枝分かれ

前節で取り上げた例のように、再帰手続きにおいて自分自身を 2 回以上呼び出す場合は、「複数の状態に枝分かれし、それぞれ同様な処理をする」というアルゴリズムになります。これをもう少し見てみましょう。

例えば、「減少列の打ち出し」という問題を考えてみます。「5 から始まり、1 ずつ小さくなる整数の列」(但し、0 にはならない) は、もちろん「5, 4, 3, 2, 1」ですね。では「1 または 2 ずつ小さくなる列 (1～2 減少列)」だと、どうでしょうか。上に加えて「5, 4, 3, 2」、「5, 4, 3, 1」、「5, 4, 2, 1」等が含まれることになります。それらを「全部」打ち出すには、どうしたらいいでしょうか。この例では前節同様、作業用の配列 b を一つ用意し、そこに減少列を作成して打ち出すことを考えます。その作業を行う再帰手続きのアルゴリズム (疑似コード) を示します:

- decr1(n, b) --- 配列 b の末尾に n から始まる 1～2 減少列を追加して打ち出す
- もし n>1 ならば、
- b の末尾に n を追加
- b の後ろに n-1 から始まる 1～2 減少列を追加して打ち出す
- b の後ろに n-2 から始まる 1～2 減少列を追加して打ち出す
- b の末尾を取り除く --- (*)
- そうでなくて n>0 ならば、
- b の末尾に n を追加
- b の後ろに n-1 から始まる 1～2 減少列を追加して打ち出す
- b の末尾を取り除く --- (*)
- そうでなければ、
- b の内容を打ち出す
- 呼び出し元へ戻る

このアルゴリズムを、例えば “decr1(5, [])” のように呼び出した場合、まずは 5 を配列に追加して [5] とします。その後、“decr1(4, [5])” と “decr1(3, [5])” をそれぞれ呼び出します。これにより、前者の呼び出しは「5, 4, …」で

始まる 1 ～ 2 減少列を、後者の呼び出しは「5, 3, …」で始まる 1 ～ 2 減少列を (作り出して打ち出すことを) 担当します。このように、「1 減らす場合」と「2 減らす場合」の 2 通りの枝分かれを、自分自身を 2 回呼び出すことでそれぞれ処理しているわけです。この呼び出しの関係を図 8 に示します。

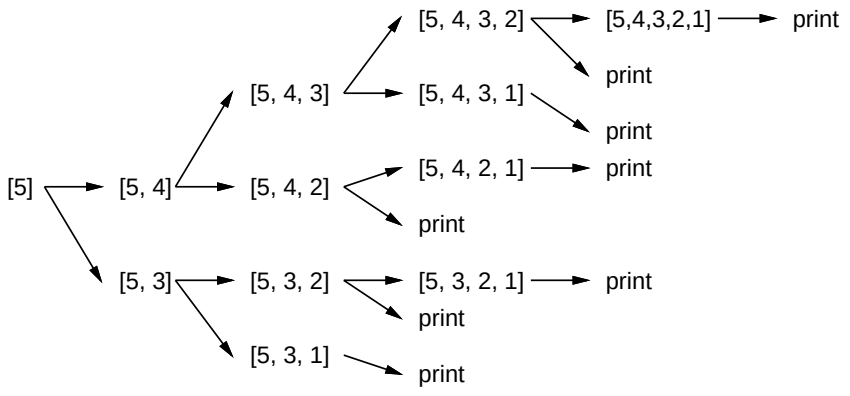

図 8: 再帰関数による 1〜2 減少列の生成

なお、n が 1 の時は、二つ減らしたらマイナスになってしまい、処理を誤るので、一つ減らしたものしか呼び出しません。そして n が 0 の時は、もう追加するものはないので、追加する代わりに配列を打ち出します。

ところで、上記のアルゴリズムには、なぜ (*) を付けた処理が入っているのか分かりますか。再帰の呼び出し元に、自分が b の末尾に追加した n (再帰の呼び出し元では n-1 or n-2) が渡されてしまい、それが “副作用” となってしまうからです。図 8 の一番上の部分を見てみると、[5, 4, 3, 2, 1] を扱う再帰呼び出し (これを RC としましょう) は、[5, 4, 3, 2] を受け取った後に 1 を追加して、次の再帰処理 (そこでは print で [5, 4, 3, 2, 1] を打ち出す) へ渡しています。その後、RC が終わって RC の呼び出し元へ戻った後は、次に [5, 4, 3, 2] を打ち出したいのですが、RC で 1 が追加されているため、そのままだと [5, 4, 3, 2, 1] が打ち出されてしまいます。これを防ぐため (再帰の呼び出し元へ副作用を及ぼさないため) に、(*) のような処理が必要なのです。再帰呼び出しにより、どのような副作用が生じるか、それをどう防ぐかは慎重に考える必要があります。

では Python のコードを見てみましょう (pop 関数は、配列から指定された添字番号の要素を削除する関数です):

```
def decr1(n, b):
  if n > 1:                        # n は 2 以上なので、1〜2 減少列を作成できる
    b.insert(len(b), n)            # 自分の担当する数字を追加
    decr1(n-1, b); decr1(n-2, b)   # 1 減少列，2 減少列の処理を再帰呼び出し
    b.pop(len(b)-1)                # 自分が追加した数は、再帰の呼び出し元で副作用になるので削除
  elif n > 0:                      # n は 1 なので、1 減少列しか作成できない
    b.insert(len(b), n)            # 自分の担当する数字を追加
    decr1(n-1, b)                  # 1 減少列の処理だけを再帰呼び出し
    b.pop(len(b)-1)                # 自分が追加した数は、再帰の呼び出し元で副作用になるので削除
  else:                            # 一つも減少できない → 1〜2 減少列が完成しているはず
    print(b)
  return
```

実行の様子も示します:

```
>>> decr1(5, [])
[5, 4, 3, 2, 1]
[5, 4, 3, 2]
[5, 4, 3, 1]
[5, 4, 2, 1]
[5, 4, 2]
[5, 3, 2, 1]
[5, 3, 2]
[5, 3, 1]
```

なお、ここでは自分自身を呼び出す回数は、「2 回 or 1 回 or 0 回 (再帰を止める場合)」だったので、それぞれを `if` 文で枝分かれさせました。しかし、「N 回」呼び出す場合も考えられます。このような場合は、ループを使って繰り返し呼び出すことになります (次節の例題がそうです)。

さて次に、前述した副作用を排除するコードを入れなかった場合を少し見てみましょう。例えば、次のような Python プログラムを作ってしまったとします (`decr1` に比べ、副作用を排除する `b.pop` 関数が抜けていますね):

```
def decr2(n, b):
  if n > 1:
    b.insert(len(b), n)
    decr2(n-1, b); decr2(n-2, b)
  elif n > 0:
    b.insert(len(b), n)
    decr2(n-1, b)
  else:
    print(b)
  return
```

実行の様子も示しましょう:

```
>>> decr2(5, [])
[5, 4, 3, 2, 1]
[5, 4, 3, 2, 1]
[5, 4, 3, 2, 1, 1]
[5, 4, 3, 2, 1, 1, 2, 1]
[5, 4, 3, 2, 1, 1, 2, 1]
[5, 4, 3, 2, 1, 1, 2, 1, 3, 2, 1]
[5, 4, 3, 2, 1, 1, 2, 1, 3, 2, 1]
[5, 4, 3, 2, 1, 1, 2, 1, 3, 2, 1, 1]
```

これは、明らかに予期した結果ではないので、プログラムの誤りを見つけ出し、修正することになります (この作業を**デバッグ** (Debug) と呼びます)。基本的な制御構造だけで作られたプログラムに比べ、再帰呼び出しは、振舞いが直感的に分かりにくいため、そのデバッグはだいぶ難しく思えます。

再帰呼び出しのデバッグにおいて、最も重要な情報は、図 8 のような再帰関係の概略図です。まずは、このような図を作れるかどうかが鍵となります。一般的な再帰関係図の特徴は、左側から右側へ末広がりになっていることです。このような関係図は、一番左側を**根** (Root)、一番右側を**葉** (Leaf)、その間にある分岐部分を**節** (Node) と見立てた**樹形図**と呼ばれます。一見複雑そうに見える再帰関係の樹形図ですが、“プログラムの処理の流れ (これを**制御フロー**と呼ぶことにします) は一つ” であることを考慮すると、うまく読み取ることができます。つまり、図 9 のように、`root` から始まり、全ての `node`, `leaf` を通過する 1 本の制御フローが、再帰呼び出しの順番になるわけです。

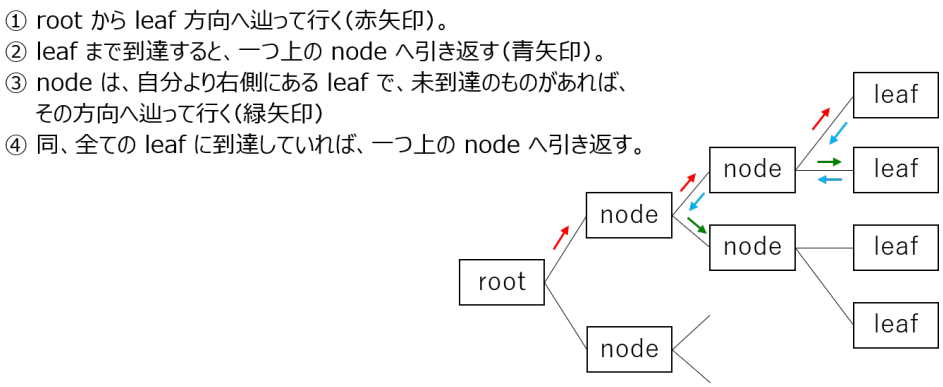

図 9: 再帰呼び出しとその戻りの関係 (樹形図内の制御フローによるイメージ)

この後は、(再帰呼び出しがこの順番通りに行われるとして) 呼び出し時に渡されるデータが図 8 のようになっているかどうかを調べることになります。先ほど作った decr2 関数へ、再帰呼び出しに渡されたデータを調べるコードを追加してみます (関数名を変えた場合は、再帰呼び出し部分の関数名も忘れずに変えましょう):

```
def decr2a(n, b):
  print("Debug: n=%d, " % (n), end=""); print(b)
  if n > 1:
    b.insert(len(b), n)
    decr2a(n-1, b); decr2a(n-2, b)
  elif n > 0:
    b.insert(len(b), n)
    decr2a(n-1, b)
  else:
    print(b)
  return
```

これを実行してみましょう:

```
>>> decr2a(5, [])
Debug: n=5, []
Debug: n=4, [5]
Debug: n=3, [5, 4]
Debug: n=2, [5, 4, 3]
Debug: n=1, [5, 4, 3, 2]
Debug: n=0, [5, 4, 3, 2, 1]    ← ここまでの動作は想定通り (図 8 右上の [5, 4, 3, 2, 1])
[5, 4, 3, 2, 1]
Debug: n=0, [5, 4, 3, 2, 1]    ← これはおかしい ([5, 4, 3, 2, 1] から [5, 4, 3, 2] へ戻るはず)
[5, 4, 3, 2, 1]
...
```

上の結果を見てみると、図 8 右上の node: “[5, 4, 3, 2, 1]” 以降の処理を終えて、再帰が node: “[5, 4, 3, 2]” へ戻る際、配列 b に入っている値が [5, 4, 3, 2, 1] から [5, 4, 3, 2] へ一つ減らないといけないのに、減っていませんね。これより、再帰が戻る際に b.pop 関数が必要だと分かります[10]。

[10] 第 7 回の演習課題 (砂漠化シミュレーション) に取り組む際でも述べましたが、少し複雑な処理・アルゴリズムのプログラムを作成する場合は、全て頭の中だけで (or その場で) 考えながら手を動かすのではなく、このように、動作のイメージ図を描く → 全体を見通したアルゴリズムの設計 → プログラムの仕様定義 → プログラムの作成、といった作業手順が重要です。

3.4 再帰呼び出しによる順列の列挙

アルゴリズムとしてよく求められるものの一つに、「与えた列の全ての順列を生成する」というものがあります。例えば、「123」という列を与えたら、「123」「132」「213」「231」「312」「321」の 6 通りを生成するわけです。

これも先の例題と似たアルゴリズムで扱うことができます。つまり「途中まで作った並び」の後に「残っているものから一つを取って追加」し、その先については、自分自身を呼び出してやらせるわけです。この時、「残っているもの」をどうやって扱うかが問題になりますが、配列を利用すれば、例えばその要素を取った時に目印として None (何もないという印) と置き換えておくことができます。Python のプログラムは下記のようになります:

```
def perm1(a, b):
  if len(a) == len(b):
    print(b)
  else:
    for i in range(len(a)):
      if a[i] != None:
        x = a[i]
        a[i] = None
        b.insert(len(b), x)
        perm1(a, b)
        a[i] = x
        b.pop(len(b)-1)
  return
```

上の例では、結果の列 b が元の列と同じ長さならば、全部並べ終わったと見なして出力しています。そうでない場合は、a の全要素について、None でない (まだ残っている) ものを取り出し、b の末尾に追加します。その後、自分自身を呼び出して残りを処理させ、終わったら自分が取ったものを元に戻します (そして別のものを取り出して、次の処理を行ないます)。これら再帰呼び出しの関係を示す樹形図を図 10 (次ページ) に示します。このコードによる実行結果は下記の通り:

```
>>> perm1([1,2,3], [])
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
```

ところで、コーディング スタイルに関する話題ですが、このようにインデントが深くなる場合は、「この処理をしたら終わり」という処理を入れて、インデントが深くならないようにすると見通しが良くなる場合があります。例えば、次のように書きます:

```
def perm2(a, b):
  if len(a) == len(b):
    print(b)
    return
  for i in range(len(a)):
    if a[i] == None:
      continue
    x = a[i]
    a[i] = None
    b.insert(len(b), x)
    perm2(a, b)
    a[i] = x
    b.pop(len(b)-1)
  return
```

この例では、「最後まで来たら、順列を表示して、この再帰呼び出しは終わり (`retrun`)」と「`a[i]` が `None` なら、この周回は終わって、次の周回へ飛ぶ (`continue`)」という処理を入れています。動作はどちらも同じですが、どちらが読み易いでしょうか。

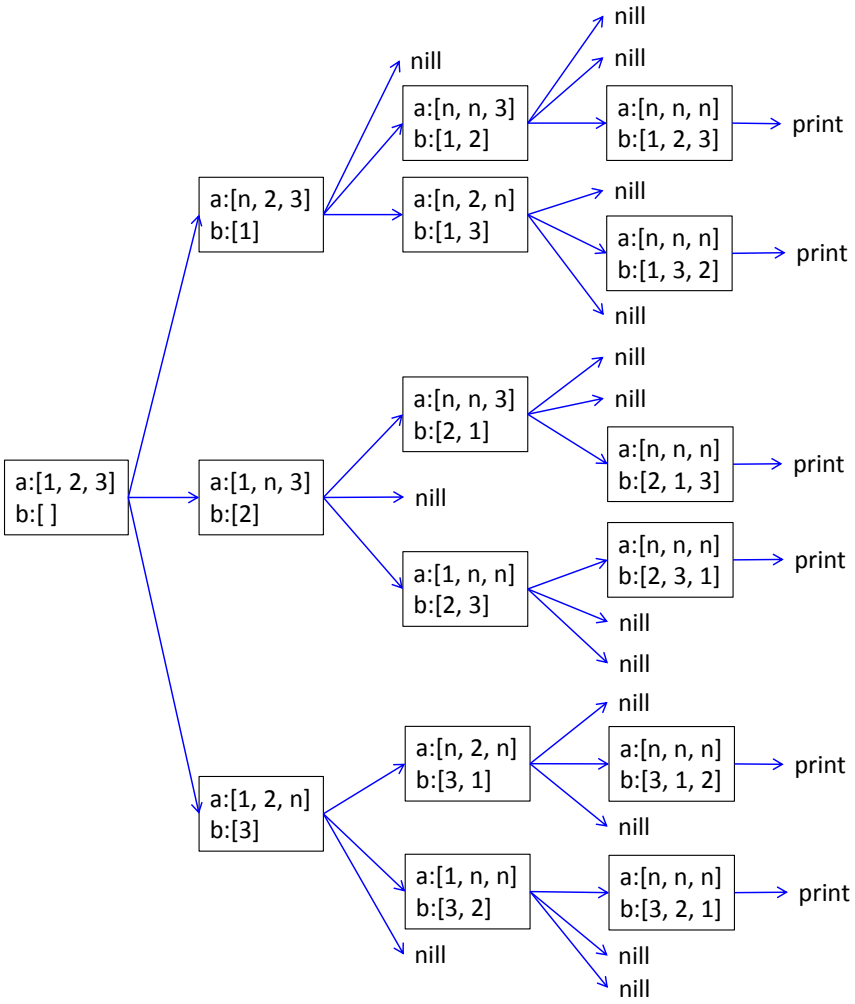

図 10: 再帰関数による順列の生成

演習 9-2 次の再帰的定義に従った計算を再帰関数として作成し、動かせ。

a. 階乗の計算:

$$fact(n) = \begin{cases} 1 & (n = 0) \\ n \times fact(n-1) & (\text{otherwise}) \end{cases}$$

b. フィボナッチ数:

$$fib(n) = \begin{cases} 0 & (n = 0) \\ 1 & (n = 1) \\ fib(n-1) + fib(n-2) & (\text{otherwise}) \end{cases}$$

c. 組合せ数の計算:

$$comb(n, r) = \begin{cases} 0 & (n < r) \\ 1 & (n = r \ \texttt{or} \ r = 0) \\ comb(n-1,\ r) + comb(n-1,\ r-1) & (\text{otherwise}) \end{cases}$$

d. 正の整数 n の二進表現[11]:

$$binary(n) = \begin{cases} \text{“0”} & (n = 0) \\ \text{“1”} & (n = 1) \\ binary(n \div 2) + \text{“0”} & (n \text{ が 2 以上の偶数}) \\ binary(n \div 2) + \text{“1”} & (n \text{ が 2 以上の奇数}) \end{cases}$$

演習 9-3 1 ∼ 2 減少列の例題を参考に、次のような数列を表示する Python プログラムを作れ。

a. N から始まる単なる減少列 (1 ∼ N までの幾つ減ってもよい) を全て表示する。

b. 配列として渡した値に対し、重複を許して L 個並べた列を全て表示する。例えば、「`['a','a','b']`, 2」を渡した場合、`['a','a']` `['a','a']` `['a','b']` `['a','a']` `['a','a']` `['a','b']` `['b','a']` `['b','b]` を表示する。

c. 1 ∼ N までの値を L 個並べた全ての列を表示する (全部で N^L 個あるはず)。例えば、「$N = 3$, $L = 2$」であれば、`[1,1]` `[1,2]` `[1,3]` `[2,1]` `[2,2]` `[2,3]` `[3,1]` `[3,2]` `[3,3]` の 9 通りを表示する。

＊ 再帰処理の仕組みや、再帰処理とループ処理の特徴 · 違いは、これから取り上げる様々なテーマの基礎となるので、教科書を含め理解を深めておいて下さい。

[11] この問題では、関数の返す値が文字列であることと、“+” は文字列の連結演算を表していることに注意して下さい。Python では、整数同士の割り算でも余りがある場合は、計算結果が実数値になります。つまり、単純に `n/2` とやるだけでは、0 or 1 以外の値が得られてしまうわけです。2 以上の偶数 · 奇数の場合分けをうまく取り扱って、対処して下さい。