アルゴリズム入門　＃11

地引　昌弘

2024.12.19

はじめに

今回は、まず最初に各整列アルゴリズムの手順を分解しながら、アルゴリズム解析において重要な「計算量」の概念について学びます。続いて、様々なアルゴリズムを対象に、計算量を見積もる方法について検討してみます。これらの検討を通じ、ループ処理や再帰処理などの制御構造において、アルゴリズムの速さを決めるポイントについて考えます。

1　前回の演習問題の解説

1.1　演習 10-1: — 各整列アルゴリズムの所要時間

この演習は、各整列アルゴリズムに対して、データ量を変えながら実行し、その時間計測を通じてデータ量と所要時間の関係を分析するというものでした。これは、アルゴリズムの性能を調べる第一歩に当たります。取りあえず、手元のマシンで計測した結果を表 1 に示しておきます:

表 1: 様々な整列アルゴリズムの所要時間 (msec)

データ数	5,000	10,000	15,000	30,000	60,000	90,000	120,000	150,000
単純選択法	563	2,281	5,625	-	-	-	-	-
単純挿入法	969	3,953	9,141	-	-	-	-	-
バブル ソート	3,313	13,719	30,828	-	-	-	-	-
マージ ソート	31	63	94	219	438	703	953	1,234
クイック ソート	16	31	47	94	203	344	484	563

これを見ると、バブル ソートが圧倒的に遅く (一見、最も技巧的な手順のように見えるのですが…)、また単純選択法と単純挿入法には 2 倍ほどの差がある、ということが分かります (単純選択法と単純挿入法との差異については、4 ページで説明します)。その理由ですが、バブル ソートはデータを並び替える際、“毎回データ全体を対象に” 隣接データの交換があるかどうかを調べるのに対し (`for` ループが常に N 回実行されます)、単純選択法 · 単純挿入法では、「データを一つ選んでは、それを適切な位置へ “直接” 置く」ことを繰り返すので (`for` ループの実行回数が次第に減少して行くので)、バブル ソートに比べて処理の回数が少なくなるからです。取り敢えず当面は、整列アルゴリズムの速さに影響する処理 (ここでは比較 · 移動) の回数を、**計算量**と呼ぶことにしましょう。これらに比べると、マージ ソートやクイック ソートはとても速いことがわかります。その理由は、何故でしょうか。N 個のデータを降順 (大きい順) に並び替える場合を例にして、少し形式的に考えてみましょう。

1.1.1　単純選択法の計算量

単純選択法では、データの列から最大値を取り出すために、先頭から最後まで各値を比較して行きます。一番大きいデータを取り出すために比較する回数は N 回です。次に、2 番目に大きいデータを取り出す際も、残ったデータの列を先頭から最後まで比較して行きます。この回数は $N-1$ 回です。これを繰り返し、最終的にデータを降順に並び替えるのに必要な比較回数 (つまり、計算量) は、$N+(N-1)+(N-2)\cdots$ より、N の 2 乗の式で表わされます。

1.1.2　マージ ソートの計算量

次に、マージ ソートの動作を見てみましょう。マージとは、二つの整列した列を「併合」し、1 本の整列した列を作ることでしたね。ここで鍵になるのは、“二つの整列した列をどうやって作るか” です。これを作るための手順が、「自分の担当範囲を前半と後半に分け、自分自身を (再帰的に) 呼び出して行く」というものでした。参考までに、マージ ソートの概要図および Python よるマージ ソートのプログラムを再掲しておきます。

図 1: マージ ソートによる整列のイメージ

```
def merge_sort(a, i, j):        # 配列 a の添字番号 i から j までの要素を整列する。
  if j <= i:
    pass                        # 要素数 1 の副列なので何もしない (既に整列済みという扱い) で戻る。
  else:
    k = int((i + j)/2)          # Python では int 関数を忘れないように
    merge_sort(a, i, k)         # 各再帰呼び出しの担当範囲に注意
    merge_sort(a, k+1, j)       # (よく分からない場合は、merge_sort() の先頭で表示させてみよう)

    # 上の再帰呼び出し二つから戻って来た時点で、a[i]～a[k] および a[k+1]～a[j] は整列済みのはず。
    b = merge(a, i, k, a, k+1, j)
    for l in range(len(b)):
      a[i+l] = b[l]
  return

def merge(a1, i1, j1, a2, i2, j2):    # a1[i1, ..., j1] と a2[i2, ..., j2] を併合
  b = []
  while (i1 <= j1) or (i2 <= j2):
    if (i1 > j1) or (i2 <= j2 and a1[i1] > a2[i2]):
      b.insert(len(b), a2[i2]); i2 = i2 + 1
    else:
      b.insert(len(b), a1[i1]); i1 = i1 + 1
  return(b)
```

再帰が正しく動くには、自分自身に対し、より簡易な処理をさせて行く必要があります。今回は、「自分の担当範囲を半分にした “副列” を渡す」ことで、処理を簡易化しています。最後は、副列の長さが 1 になり、長さが 1 ならば、そのままで整列した列として扱えるので、マージ ソートでは、素朴な整列アルゴリズムのような直接的な並び替えは発生しません。

では、長さ 1 の整列した副列を作るまでの計算量、およびこれらをマージして行く計算量を見積もってみましょう。図 1 を見てみると、「長さ N の配列を 1 個」整列するには、1 回目の再帰で「長さ $\frac{N}{2}$ の配列を 2 個」整列し、2 回目では「長さ $\frac{N}{4}$ の配列を 4 個」整列していることが分かります (“第 i 世代の呼び出し/第 i 世代からの戻り” が対になります)。Python のコードで確認してみると、最初の merge_sort 関数では、配列を i ～ k と k+1 ～ j の半分に分け、1 回目の再

帰として二つの merge_sort 関数を呼び出した後、半分にした配列を merge 関数でマージ (要は比較) しています (合計: 半分にした配列 2 個 × 一つのマージ)。次に、再帰呼び出しされた "二つ" の merge_sort 関数では、"それぞれ" が同様に配列を半分にし (当初比 1/4)、2 回目の再帰としてそれぞれが二つの merge_sort 関数 (計四つ) を呼び出した後、それぞれが merge 関数でマージ (要は比較) しています (合計: 当初比 1/4 の配列 2 個 × 二つのマージ)。これらは、それぞれの段 or 世代 (再帰の深さ: "その" 再帰呼び出しに至るまでに再帰を繰り返した回数) において、自分自身の再帰呼び出しを除くと、合計で長さ N のデータを整列 (つまり、N 回比較) していることになります。

では、再帰は何段起こるでしょうか? この再帰呼び出しは、副列の長さを半分ずつにして行き、長さが 1 になったら終了します。段数を L とすると、この再帰は "$2^L \geq N$ を満たす最小の L 回" 呼び出されます。ここで、およそ $2^L \approx N$ と見積もると、$L = \log_2 N$ 回呼び出されることになります。以上より、N 回の比較が $\log_2 N$ 段あるので、最終的にデータを降順に並び替えるのに必要な比較回数は、$N \log_2 N$ の式で表わされます。

これより、マージ ソートの比較回数は、単純選択法の比較回数 (N の 2 乗で表わされる) に比べて少ないことが分かります。特に、N が大きくなるにつれ、その傾向は顕著になります。この見積もりが合っているかどうか、データ量を 2 倍、3 倍にして、その所要時間の変化を見てみましょう。単純選択法 · 単純挿入法 · バブルソートでは、所要時間がほぼ 4 倍、9 倍になっているように見えます。$4 = 2^2$、$9 = 3^2$ なので、これらのアルゴリズムでは「所要時間はデータ量の 2 乗に比例している」、つまり計算量の見積もりは合っていると言ってよいでしょう。マージ ソートについては、データ量が k 倍になった場合を考えると、

$$N \log_2 N \to (k \cdot N) \log_2 (k \cdot N) = \underline{k \cdot (N \log_2 N) + (k \cdot N) \log_2 k}$$

より、通常は $k < N$ なので、

$$k \cdot (N \log_2 N) < \underline{k \cdot (N \log_2 N) + (k \cdot N) \log_2 k} < k \cdot (N \log_2 N) + (k \cdot N) \log_2 N = 2k \cdot (N \log_2 N)$$

が成り立ります[1]。これより、データ量が k 倍になった時の計算量は、k 倍より大きいが、$2k$ 倍より小さいことが分かります。実測した結果を見てみると、この計算量の見積もりもほぼ合っていると言えそうです。

ところで、この結果を見る限り、データ数が 1,000,000 (100 倍) になった時の所要時間は、単純選択法で $2.3 \times 100^2 = 23,000$ 秒 $= 6$ 時間以上 (!) となってしまい、終わるまで待つのはあまり嬉しいものではないことが分かりますね。一方、マージ ソートでは、k と N に $k \ll N$ といった関係がある場合 (データ量の大きさに比べて、その増加量が緩やかな場合)、$\log_2 (k \cdot N) \approx \log_2 N$ と見積もることができるので (ここにも対数関数の特徴が出ています)、データ量が k 倍になった時の計算量もほぼ k 倍と見積もることができます。これは、規模の増加に対する計算量の増加を考える上で、理想的な状況になっていると言えます。

1.1.3 クイック ソートの計算量

最後は、クイック ソートです。クイック ソートでは、まず始めに列の中からピボットと呼ばれる値を一つ決めて、それを中心にデータ列を二つの副列に分けます。次に、左半分 · 右半分の各副列に対して、それぞれ自分自身を再帰呼び出しすることで整列させます (左右の副列が整列した時点で、ピボットと併せて全体の整列が完了している)。参考までに、クイック ソートの概要図および Python よるクイック ソートのプログラムを再掲しておきます (swap 関数は省略しています)。

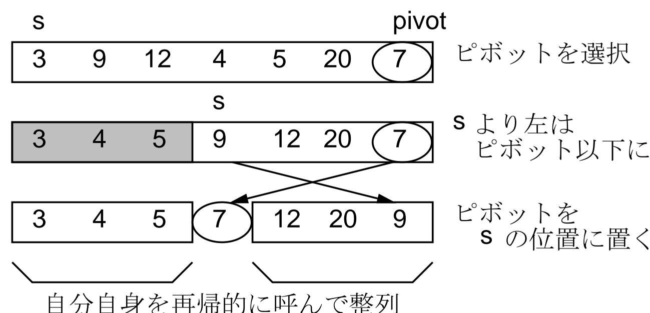

図 2: クイック ソートによる整列

[1] ここでは、対数関数が単調増加関数であることを利用しています。計算量だけではなく、数値の概算を見積もる or 比較する場合に、単調増加関数の性質は大変よく使われます。

```
def quick_sort(a, i, j):      # 配列 a の添字番号 i から j までの要素を整列する。
  if j <= i:
    pass                      # 要素数 1 の副列なので何もしない (既に整列済みという扱い)。
  else:
    pivot = a[j]              # 取り敢えず、配列の最後 (数列の右端) をピボットにする。
    s = i                     # s は左副列と右副列の境界を示す。
    for k in range(i, j):     # 終値=j-1
      if a[k] <= pivot:
        swap(a, s, k)
        s = s + 1
    swap(a, j, s)
    quick_sort(a, i, s-1)
    quick_sort(a, s+1, j)
  return
```

では、クイック ソートの計算量について考えてみましょう。ピボットを中心に列を半分に分けて行く再帰処理の計算量は、再帰の樹形図 (図 3) をもとに考えます。クイック ソートでは、最初に「長さ N の配列を 1 個」整列する形で呼び出すと、1 回目の再帰では「長さ $\frac{N}{2}$ の配列を 2 個」整列し、2 回目では「長さ $\frac{N}{4}$ の配列を 4 個」整列しています。つまり、再帰の各段 (深さが同じ再帰処理) では、(自分自身の再帰呼び出しを除くと) 合計で長さ N のデータを整列 (もう少し正確に言うと、ビボットの値で分類) して元の配列に書き戻しているわけです。これより、マージ ソートと同様、各段の計算量は N となります。

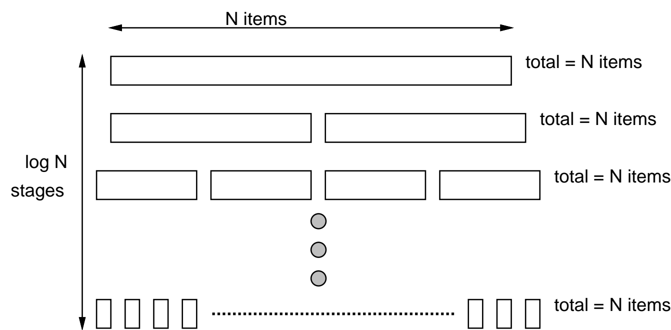

図 3: クイック ソートのコード実行回数の検討

問題は再帰の段数です。再帰の段数については、ピボットの選択が完璧なら $\log_2 N$ 回ですが、これを任意に選んだとして、その定数倍 (例えば k 倍) だと考えることにします。ここで、N が十分に大きい場合 (データ数が十分に大きい場合)、定数 k は N に対して影響が小さい (つまり、無視できる) と割り切れば (その妥当性については前節で述べました/詳細は 2.1 節で説明します)、最終的な計算量は $N \log_2 N$ になります。しかし、極めて運が悪い場合、つまり列の最大値か最小値を毎回ピボットにしてしまうと、再帰の段数が N になってしまうため、最悪の計算量は N^2 ということになります。渡されたデータが多ければ、そんな運の悪いことはないだろうと思うかも知れませんが、何かのはずみで、既に整列済みのデータ列を渡されてしまうと、まさに最悪な運に出会ってしまいます (これが演習 10-2 です)。

ところで、クイック ソートとマージ ソートは、どちらの計算量も $N \log_2 N$ と見積もれるのに対し、表 1 の計測結果では、クイック ソートの方がマージ ソートより 2 倍ほど速いように見えます。この理由ですが、マージ ソートでは二つの副列から一つの副列へマージする際、merge 関数内で一時的な配列 b を毎回用意して、まずは二つの副列からそこへマージした後、merge_sort 関数内で b から元の (起動時に渡された) 配列 a へコピーするという作業を行なっています。これに対し、クイック ソートでは、swap 関数により必要なデータを入れ替えているだけです。つまり、余分なコピーをしていません。この違いが、微妙な所要時間の差となって現れているわけです (特に Python では、配列に対する処理が遅いですよね)。単純選択法と単純挿入法の差についても、理由は同じです (単純挿入法は毎回、要素を一つずらす、つまりコピーをしていますね)。大きなプログラムになると、定量的・形式的な計算量の差だけではなく、このようなデータ コピーの有無といった実装の影響も顕著になって来ます。これを解消する一つの手段として、各関数で同じデータを使い回すことが考えられますが、逆に副作用が増えるという弊害も生じます (データに誤りが見つかった場合、どの関数で誤ったかを調べるのに手間が掛かる)。

1.2 演習 10-2 — クイック ソートの弱点

クイック ソートが整列済みの配列に対しては遅いという弱点を実際に示すには、次のように整列を連続して 2 回行なうだけでよいでしょう。但し、Python では、そのままでは再帰呼び出しの段数に制限があるため、(*) のコードにより、段数の制限を緩和しています。

```
import sys
sys.setrecursionlimit(5000)         # 許容される再帰の段数を 5,000 回まで増やす (*)

def measure_time(n, r):
  a = rand_array(n, r)
  start = time.process_time()
  quick_sort(a, 0, len(a)-1)        # 通常のクイック ソート
  finish = time.process_time()
  print("%.10g" % (finish - start))
  start = time.process_time()
  quick_sort(a, 0, len(a)-1)        # 整列済みデータに対するクイック ソート
  finish = time.process_time()
  print("%.10g" % (finish - start))
```

これを実行してみると、結果は次のようになりました:

データ数	1,000	2,000	3,000
未整列 (msec)	0	0	16
整列済み (msec)	109	484	1,094

データ数が 1000, 2000 個では、データが未整列だとクイック ソートの所要時間は計測限界以下でした。これに対して、データが整列済みの場合は相当に遅く、計算量は N^2 に近くなっているように見えます。これを改良するにはどうしたら良いでしょうか? 今回の問題は、ピボット値を常に一番端から選んでいたことが原因でした。これを解消するには、ピボット値をランダムに選ぶという方法があります:

```
def quick_sort(a, i, j):
  if j <= i:
    pass
  else:
    p = i + int((j-i+1)*random.random()); swap(a, p, j)     # ピボット値をランダムに選択する
    pivot = a[j]                                            # a[j] は a[p] と入れ替わり済み
    s = i
    for k in range(i, j):
      if a[k] <= pivot:
        swap(a, s, k)
        s = s + 1
    swap(a, j, s)
    quick_sort(a, i, s-1)
    quick_sort(a, s+1, j)
  return
```

プログラムの修正は、コメントを付けた 1 行を挿入しただけです。つまり、i ～ j の範囲にある整数 p を一つランダムに選び[2]、a[j] と a[p] を交換してから、後はこれまでと同様に処理しています。(実行例は省略しますが) これを実行してみると、上の表とは異なり、データが未整列でも整列済みでもほぼ同じ時間で整列が終わるようになっています。

[2]もう少し詳しく言うと、i ～ j の範囲には整数が i - j + 1 個あるので、その個数に対応する乱数を int((j-i+1)*random.random()) より得ます。次に、得られた乱数を配列の添字番号として使いたいので、範囲の (つまり配列の) 先頭を示す i を加えています。

2　時間計算量

2.1　時間計算量の考え方

これまで様々な整列アルゴリズムを実現するプログラムの所要時間を計測し、その速さについて話題にして来ましたが、いよいよ本節では、アルゴリズムの**性能** (Performance) を評価する指針の一つである**計算の複雑さ** (Computational Complexity) ないし**計算量** (Complexity) について、形式的[3]な議論を行ないます。complexity だと日本語は「複雑さ」になりそうですが、「複雑さ」という日本語では一般的すぎて何のことか分かりにくいので、日本語では「計算量」と呼ぶわけです。なお、計算量には「どれくらいメモリが必要になるか」を表わす**領域計算量** (Space Complexity) などもありますが、ここではとりあえず「所要時間」に着目する**時間計算量** (Time Complexity) を取り上げます。

まずは、単純選択法 (`selection_sort`) を例題にして、これがどれくらいの時間を要するかを見積もってみましょう。議論を始めるにあたり、次の前提を置きます:

「コンピュータでは、ある一つの決まった動作は、その動作に応じた決まった時間で実行される。」

これは、データのサイズを変えながら各整列アルゴリズムの実行時間を計測した際、結果がそう大きく変動していないことから、無理な前提ではないことが分かると思います。

次に、単純選択法のプログラムを「実行回数によって区分した」ものを、図 4 に示します。

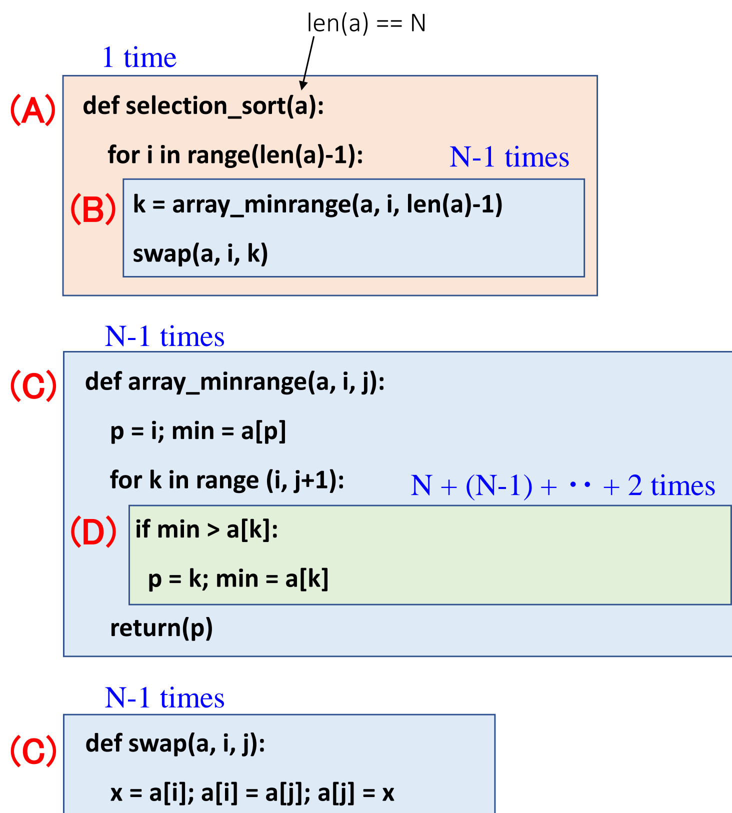

図 4: 単純選択法のコード実行回数の検討

ここでは、渡された (整列する) 配列の長さを N とします。図 4 より、実行回数について次のことが分かります。

(A) 関数 `selection_sort` の本体部分は「1 回」実行される。

(B) `for` ループの内側 (ブロック) については、「$N-1$ 回」実行される。

(C) 従って、`array_minrange` や `swap` も「$N-1$ 回」実行される。

(D) `array_minrange` にある `for` ループのブロックは、最初は N 回、次は $N-1$ 回、その次は $N-2$ 回⋯、実行され、合計で $\dfrac{N(N+1)}{2}-1$ 回実行される[4]。

[3] 数理的な定式化という意味です。初回 (#1) でも述べましたが、この用語はよく使われるので、覚えておきましょう。

[4] 普通に考えると、最後は 1 回実行されることになりますが、最後に残った数値は比較をせずとも最小値であることが明らかなので、このブロックは実行しません。つまり、2 回実行される時が最後の実行となるので、総実行回数は $N+(N-1)+(N-2)+\cdots+2=\frac{N(N+1)}{2}-1$ 回になります。

ここで、(A) の部分 ((B) および (C) の部分を除いた (A) だけの部分、以下同様) の実行に掛かる時間を C_a とし、(B) と (C) の部分を 1 回実行するのに掛かる時間を C_{b+c} とし、(D) の部分を 1 回実行するのに掛かる時間を C_d とすると、合計実行時間は次のようになります:

$$T = C_a + (N-1)C_{b+c} + \left\{\frac{N(N+1)}{2} - 1\right\}C_d$$

これを展開して N について整理すると次のようになります:

$$T = \frac{C_d}{2}N^2 + \left(\frac{C_d}{2} + C_{b+c}\right)N + \left(C_a - C_{b+c} - \frac{C_d}{2}\right)$$

この式より、たとえ C_a や C_{b+c} の所要時間が C_d の 100 倍だったとしても (確かに、(B) や (C) のコード量は (D) に比べて多いので、その実行時間 C_{b+c} も C_d より多そうに見えますが)、N が十分増えれば、N の 2 次の項に比べて N の 1 次以下の項は無視できる値になるので、T を次のように近似してもほぼ問題はないと言えます:

$$T \approx \frac{C_d}{2}N^2$$

よって、時間計算量は、計算量を示す数式のうち「最高次の項だけを対象」にすればよく、これを O(“N の最高次の項”) と表わします。例えば、単純選択法の時間計算量は $O(N^2)$ となります。ところで、N の最高次の項にある係数についてですが、「同じ計算量のプログラム同士」であれば係数を考慮することに意味はあります。しかし、計算量が違うプログラムであれば、係数に大小があっても N が十分増えた場合を考えると、結局は N の最高次の項が計算量に最も影響を及ぼすので、無視しても構わないという扱いにされています。同様に、例えばマージ ソートおよびクイック ソートの計算量について、1.1.2 節や 1.1.3 節の説明では、それぞれ $O(N \log_2 N)$ としましたが、計算量の議論では log の底が何かも省略するのが通例なので、両者の計算量は、正しくは $O(N \log N)$ と表わされます。

一般に N 個のデータを入力するようなプログラムでは、そのデータの読み込みに $O(N)$ は最低必要です。このような、$O(N)$ のアルゴリズムのことを (N に比例するわけですから) **線形時間** (Linear Time Complexity) と呼びます。例えば、最大や最小を求める問題は、データを読みながら一巡すれば結果が求まるので、線形時間のアルゴリズムで扱えます。このような問題はコンピュータで簡単に処理できると言えます。少し込み入ったアルゴリズムは、$O(N^2)$ や $O(N^3)$ などの計算量になります。これを**多項式時間** (Polynominal Time Complexity) と呼びます。さらに時間計算量の大きなものとしては、$O(C^N)$ すなわち**指数時間** (Exponential Time Complexity) となる場合があり、このような問題をコンピュータで実用的に扱えるのは、小さい N に限られてしまいます。

2.2 時間計算量の例

さて、これまでは整列アルゴリズムを題材に時間計算量を考えて来ましたが、以後は、アルゴリズムの効率性 (or 性能) を議論をするために、他のアルゴリズムについても時間計算量を見積もって行く必要があります。時間計算量の求め方を非常に簡単にまとめると、次のようになります:

「入力の値 n に対して、プログラム中の “最も多く実行される箇所” の実行回数を求め、
これを n の式で表わして $O(f(n))$ の形で記す。」

極端な例ですが、n が出て来なければ $O(1)$ (定数計算量)、つまり n の値に関わらず一定時間で終わることを意味します。例として、多くのアルゴリズムが属する典型的な時間計算量を、速い方から順に挙げておきます:

- $O(1)$ ———— 定数計算量
- $O(\log n)$ ——— 対数計算量
- $O(\sqrt{n})$ ———— 平方根計算量
- $O(n)$ ———— 線形計算量 (n に比例)
- $O(n \log n)$ ——— 良い整列アルゴリズム
- $O(n^2)$, $O(n^3)$ — 一般に多項式計算量と呼ぶ
- $O(2^n)$, $O(3^n)$ — 指数計算量
- $O(n!)$ ———— 階乗計算量

アルゴリズムの速さを評価する目安として、$O(n)$ までは「すごく速い」、$O(n \log n)$ は「まあまあ速い」、$O(n^2)$ は「遅い」、$O(2^n)$ 以降は「ひどく遅いので、小さい n にしか役に立たない」と考えたら良いでしょう。

3 時間計算量の見積もり方

本章では、これまでに出て来た様々なアルゴリズムを対象に、計算量を見積もるための考え方を見て行きましょう。

3.1 整列アルゴリズムの時間計算量

最初の例として、まだ計算量を見積もっていない残り二つの整列アルゴリズムについて、その時間計算量を見積もってみましょう。まずは、単純挿入法の時間計算量です。単純挿入法では、外側のループで i を $1 \sim N$ まで変えながら、その番号の要素を適切な位置に挿入して行きます。挿入位置を探索するのに、平均して $\frac{i}{2}$ 個の要素を比較するとした場合、合計で $\frac{1}{2} + \frac{2}{2} + \cdots + \frac{N-1}{2} \approx \frac{N^2}{4}$ 回の比較が生じます。挿入位置が見つかったら、これも平均して $\frac{i}{2}$ 個の要素を後ろにずらすことになりますが、同様に考えると $\frac{N^2}{4}$ 回程度の処理となり、(前章で説明したように N の係数は無視して) 最終的な計算量は $O(N^2)$ になります。

次に、バブル ソートの時間計算量です。内側のループでは毎回必ず $N-1$ 回の比較を行ないます。**最善の場合** (Ideal Case)、つまり最初から全部並んでいる場合は、内側のループ全体を 1 回実行すれば完成です。これは $O(N)$ と言えます。しかし、**最悪の場合** (Worst Case)、つまり完全に逆順に並んでいる場合は、内側ループ全体の 1 回目は最も大きい要素を最後の位置に持って来るだけで終わってしまい (ループ内の動作は、それまでに発見した未整列かつ一番大きい要素だけが動いて行くイメージ)、2 回目は 2 番目に大きい要素を最後から 2 番目の位置に…というわけで、内側ループ全体が N 回実行されます。これは $O(N^2)$ ですね。**平均の場合** (Average Case) には内側ループ全体が $\frac{N}{p}$ 回実行されると考えれば[5]、合計で $(N-1) \cdot \frac{N}{p} \approx \frac{N^2}{p}$ 回の比較が生じます。後はデータの交換回数ですが、内側ループ全体の実行 1 回で $\frac{N}{q}$ 回生じると想定すれば (比較の度に交換されるわけではない)、最終的な計算量は $\frac{N}{q} \cdot \frac{N}{p}$ より上と同様に $O(N^2)$ となります。

3.2 最大公約数

2 番目は、最大公約数を求めるプログラムです。二つの数 M, N ($M > N$ とする) の最大公約数を求める一番素朴な方法は、下記のアルゴリズムでしょう:

- gcd_pri(x, y) — x と y の最大公約数を求める
-     i を $min(x, y)$ から 1 まで 1 ずつ減らしながら繰り返し、
-         x も y も i で割り切れるなら、i を返して終了
-     (繰り返し終わり)

(小さい方の) N を初項としたカウンタを 1 ずつ減らしながら、その値で両者が割り切れるかどうかを調べるわけです。Python のコードは以下になります:

```
def gcd_pri(x, y):
  min = x
  if min > y: min = y                # x, y の小さい方を min に代入
  for i in range(min - 1, 0, -1):    # 終値=1 (増分が負数なので注意)
    if (x % i == 0) and (y % i == 0):
      return(i)
```

この計算量は、x と y がどちらも i で割り切れるかどうかを N 回調べる (ループが N 回だけ回る) ことになるので、当然 $O(N)$ になると考えられます。

では、x と y の引き算を用いる方法はどうでしょうか (このアルゴリズムの詳細については、#9 にある “3.1: 再帰手続き · 再帰関数の考え方” を参照のこと):

```
def gcd_sub(x, y):
  while x != y:
    if x > y: x = x - y    # (*1)
    else:     y = y - x    # (*2)
  return(x)
```

[5] 直感的に $p \approx 2$ となりそうですが、実はそう簡単ではありません。例えば $a > b > c$ という関係にある 3 データが $[b, a, c]$ の順で並んでいた場合、$a > c$ より内側のループ全体を 1 回実行することで a と c の順序は入れ替わるものの、$[b, c, a]$ かつ $b > c$ より、新たに b と c の入れ替えが生じます。この時、for ループのカウンタは先に進んでいるので、これを入れ替えるには内側ループ全体をもう 1 回実行しなければなりません。$[b, a, \ldots]$ のような部分的に正しい順に並んでいる所に出会うと、そこから先は、内側ループ全体の実行 1 回で一つ隣りにしか移動できないわけです。実際に計測してみると、データが増えるにつれ $p \approx 1$ に近付いて行きます。この差が、バブル ソートと単純挿入法との差と言えます。

このプログラムにあるループは計数ループ (for ループ) ではないので、ループの回数を直接数えることができません。再帰呼び出しの段数を見積もる時は、各再帰呼び出し毎に担当範囲 (扱うデータのサイズ) がどのくらい小さくなるかを考えました。この場合も同じように考えてみましょう。

上のプログラムにおいて、計算量が最善の場合とは二つの数 M, N が $M = N$ となる時で、これはすぐ終わります ($O(1)$)。最悪の場合は $N = 1$ (or $M - N = 1$) の時で、ほぼ M 回 (or N 回) の引き算が発生し、その計算量は $O(M)$ (or $O(N)$) です。ここで、最悪の場合をもう少し詳しく見てみると、どちらの場合でも上のプログラムにある引き算 (*1) と (*2) の片方だけが、ほぼ M 回 (or N 回) 連続して行なわれることが分かります。これより、どうも両方の引き算がバランス良く行なわれると早く終了しそうな予感がしますね[6]。

では、両方の引き算がバランス良く行なわれる場合の計算量を見積もってみましょう。i 回目の引き算をする場合を考えます。この引き算をする前は、x, y に $x_p > y_q$ という関係があるとし、i 回目の引き算の結果を $x_{p+1} = x_p - y_q$ とします。次に、$i+1$ 回目の引き算をする前は、$x_{p+1} < y_q$ という関係になったとし、その結果を $y_{q+1} = y_q - x_{p+1}$ とします。以下同様に、各引き算の前は $x > y$ と $x < y$ が交互に続くとします。ここで、x_{p+2} について考えると、

$$x_{p+2} = x_{p+1} - y_{q+1} = x_{p+1} - (y_q - x_{p+1}) = 2x_{p+1} - (x_p - x_{p+1}) = 3x_{p+1} - x_p$$

が成り立ちます。さらに、$x_{p+2} < x_{p+1} < x_p$ であることから、$x_{p+2} = 3x_{p+1} - x_p < x_{p+1}$ より、

$$x_{p+1} < \frac{x_p}{2}$$

となります。よって、各引き算により x, y の値が半減して行くので、その計算量は $O(\log N)$ と見積もることができます。

3.3 フィボナッチ数

次は、フィボナッチ数を求めるアルゴリズムです。再帰処理を用いたアルゴリズムは、下記の通りでした:

```
def fib(n):
  if   n < 1:  return(0)     # 不等号により、n に負数を渡された場合まで対応
  elif n == 1: return(1)
  else:        return(fib(n-1) + fib(n-2))
```

大変綺麗ですね。では、その計算量を見積もってみましょう。

この再帰処理では、fib(n) の計算に fib($n-1$) と fib($n-2$) を呼び出します。次に、fib($n-1$) では fib($n-2$) と fib($n-3$) を呼び出し、fib($n-2$) では fib($n-3$) と fib($n-4$) を呼び出します。以下、同様に考えると、再帰の各段毎に二つの再帰呼び出しが発生するため、n 段目の再帰処理は 2^n 個になります。マージ ソートやクイック ソートでも n 段目の再帰処理は 2^n 個ありました。これらは、整列アルゴリズムの中では速いアルゴリズムだったので、再帰処理を用いたフィボナッチ数の計算も早く終了すると予想できます。ちょっと見てましょう。fib 関数の実行時間を計測する関数は、以下の通り:

```
def measure_fib(n):
  start = time.process_time()
  fib(n)
  finish = time.process_time()
  print("%.10g" % (finish - start))
```

その実行結果は:

```
>>> measure_fib(25)
0.03125
>>> measure_fib(30)     ← まずは、5 だけ増やしてみる (20%増)
0.328125                ← えっ?
>>> measure_fib(35)     ← もう 1 回、5 だけ増やしてみる (当初比 40%増)
3.515625                ← ええっ!!
```

[6] クイック ソートでも、ピボットの値がデータ列の中央値ぐらいとなり、ピボットより大きい · 小さい副列を整列させる各再帰呼び出しの段数がほぼ同じ深さに揃う時が、一番効率が良かったですよね。

$O(n \log n)$ どころの比ではありません。これでは $O(n^2)$ より遅く、もはや $O(2^n)$ に近いレベルです。いったい何が起きているのでしょか。

もう一度、フィボナッチ数列の再帰アルゴリズムをよく見てみましょう。fib(n) の計算では、fib($n-1$) と fib($n-2$) を呼び出します。次の fib($n-1$) では、fib($n-2$) と fib($n-3$) を呼び出します‥。よく見ないと気付かないのですが、fib($n-2$) が 2 回出て来ました (皆さんは、気付きましたか?)。つまり、同じ計算を 2 回しているわけです。どうも無駄がありそうですね。その影響を見積もってみましょう。マージ ソートやクイック ソートの再帰アルゴリズムでは、再帰処理の重複はなく、段数が増えるにつれて処理対象のデータも少なくなるため (要は副列が小さくなるため)、それに伴い演算量も減りました。ところが、フィボナッチ数列の再帰アルゴリズムでは、再帰が深くなっても演算量は減っていません (毎回、比較 2 回と足し算 1 回ずつ)。つまり、1 段目から n 段目にある再帰処理、合計 $\sum_{i=1}^{n} 2^i$ 個の再帰処理が、全て同じ量の演算を行なうわけです。これは遅い。。全体の計算量は、少なくとも $O(2^n)$ です。

では、フィボナッチ数列を計算する速いアルゴリズムは存在しないのでしょうか。フィボナッチ数列を手動で素直に計算する場合は、fib(1) と fib(0) を足し算して fib(2) を求め、この新たに得られた fib(2) と既に得られている fib(1) を足し算して、さらに fib(3) を求めて行きます。この手順をもう少し形式的に考えると、フィボナッチ数列の入った変数 x0 と x1 を用意し、まずは x0 + x1 を計算して一時的に避難させておき、x1 を x0 に移してから、一時的に避難させていた x0 + x1 を x1 に移すことを繰り返しているわけです(x0 へは fib(n-1) が、x1 へは fib(n) が入ります)。この手順は、ループを用いた Python のプログラムとして、次にように作ることができます:

```
def fib_loop(n):
  x0 = 1; x1 = 0     # x1 が fib(n) になるように、x0 の初期値を調整していることに注意
  for i in range(n):
    t = x0 + x1
    x0 = x1
    x1 = t
  return(x1)
```

x0, x1 は、それぞれ図 5 のように振る舞います (念のため: fib(0) ≠ x0, fib(0) = x1, fib(1) = x1, fib(2) = x1, ...)。

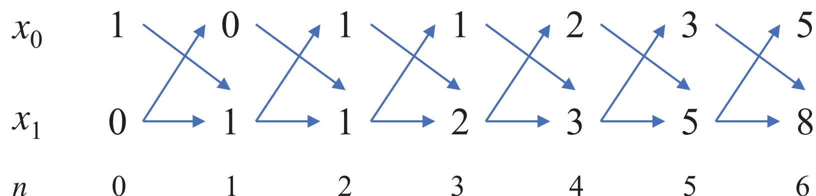

図 5: ループによるフィボナッチ数の計算 (矢印 1 本=代入, 矢印 2 本=加算)

そしてこの計算量は、単に n 回のループを回すだけなので、$O(n)$ になっています。念のため、fib 関数の代わりに fib_loop 関数を呼び出すように改造した measure_fib 関数を用いて、所要時間を計測してみましょう:

```
>>> measure_fib(10000)      ← n=10,000 程度であれば、計測限界以下
0
>>> measure_fib(100000)
0.109375
>>> measure_fib(200000)
0.375
>>> measure_fib(300000)
0.796875
```

上の結果を見る限りでは、「n が k 倍になれば時間も k 倍」から外れているように見えますが、これは fib_loop(50) くらいから、途中の計算に出て来る整数の桁数が一定値を超えてしまうため、Python の処理系が (相当) 余分な時間を使ってしまうからです。因みに、計測限界以下であった fib_loop(10000) の値は、次ページのようなります。

33644764876431783266621612005107543310302148460680063906564769974680081442166662368155595513633734025582065332680836159373734790483865268263040892463056431887354544369559827491606602099884183933864652731300088830269235673613135117579297437854413752130520504347701602264758318906527890855154366159582987279682987510631200575428783453215515103870818298969791613127856265033195487140214287532698187962046936097879900350962302291026368131493195275630227837628441540360584402572114334961180023091208287046088923962328835461505776583271252546093591128203925285393434620904245248929403901706233888991085841065183173360437470737908552631764325733993712871937587746897479926305837065742830161637408969178426378624212835258112820516370298089332099905707920064367426202389783111470054074998459250360633560933883831923386783056136435351892133279732908133732642652633989763922723407882928177953580570993691049175470808931841056146322338217465637321248226383092103297701648054726243842374862411453093812206564914032751086643394517512161526545361333111314042436854805106765843493523836959653428071768775328348234345557366719731392746273629108210679280784718035329131176778924659089938635459327894523777674406192240337638674004021330343297496902028328145933418826817683893072003634795623117103101291953169794607632737589253530772552375943788434504067715555779056450443016640119462580972216729758615026968443146952034614932291105970676243268515992834709891284706740862008587135016260312071903172086094081298321581077282076353186624611278245537208532365305775956430072517744315051539600905168603220349163222640885248852433158051534849622434848299380905070483482449327453732624567755879089187190803662058009594743150052402532709746995318770724376825907419939632265984147498193609285223945039707165443156421328157688908058783183404917434556270520223564846495196112460268313970975069382648706613264507665074611512677522748621598642530711298441182622661057163515069260029861704945425047491378115154139941550671256271197133252763631939606902895650288268608362241082050562430701794976171121233066073310059947366875

驚きの数値ですね。この桁数を処理する影響を考えると、ほぼ $O(n)$ に近い値が出ているように見えますね。

3.4 組合せ数

組合せ数を計算する場合も、再帰的定義をそのままプログラムにしてしまうと非常に遅くなります。この場合、フィボナッチ数と同様、再帰の各段毎に二つの再帰呼び出しが発生し、再帰が深くなっても演算量は減らないため (毎回、比較と足し算が 1 回ずつ)、$\sum_{i=1}^{n} 2^i$ 個の再帰処理が、全て同じ量の演算を行ないます。よって、全体の計算量は、少なくとも $O(2^n)$ になってしまいます。これに対し、組合せ数を高速に計算する方法としては、以前 (#5) に説明した掛け算・割り算を交互に行なう方法があります (以下に再掲します):

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

このコードでは、ループの回数が r 回であることが分かります。これより、$r \approx N$ として、その計算量は $O(N)$ と見積もれます。

また、これ以外にも**パスカルの三角形** (Pascal's Triangle) を作る方法があります (図 6):

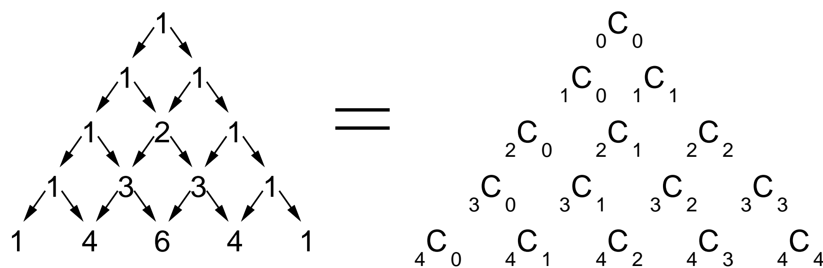

図 6: パスカルの三角形

パスカルの三角形を樹形図と考え (図 6 の左側)、これを用いて組合せ数 ${}_nC_r$ を計算するアルゴリズムを少し整理してみましょう。樹形図内の `node` が持つ数値は、自身に矢印を向けている上側 `node` が持つ数値を全て足したものになっています。そして、上から n 段目、左から r 番目にある `node` の持つ数値が、${}_nC_r$ に該当しています (通例ですが、n, r は 0 から始まります)。このアルゴリズムを擬似コードで表わすと、次のようになります:

- comb_array(n, r) — $_nC_r$ を計算する
- 各 node が持つ数値を保持しておく 2 次元配列 a を用意 (初期値は 0)
- 始まりの値として $_0C_0$ の値 1 を a[0][0] へ代入
- i を 0 から n まで変化させながら以下を繰り返し: # 樹形図を上から下へ
- j を 0 から i まで変化させながら以下を繰り返し: # 各段毎の処理
- a[i - 1][j - 1] と a[i - 1][j] を加算した結果を a[i][j] へ代入
- (繰り返し終わり)
- (繰り返し終わり)
- 2 次元配列 a を表示

1.1
これを用いて組合せ数を求める Python プログラムは、次のようになります:

```
def comb_array(n, r):
  a = ita.array.make2d(n+1, n+1)
  for i in range(n+1):
    for j in range(n+1):
      a[i][j] = 0

  # ここからがパスカルの三角形による組合せ数の計算 (ここより上は 2 次元配列の準備)
  a[0][0] = 1
  for i in range(1, n+1):
    for j in range(0, i+1):
      a[i][j] = a[i-1][j-1] + a[i-1][j]
  pprint.pprint(a, width=25)
  return(None)
```

では、ちょっと動かしてみましょう:

```
>>> comb_array(5,3)
[[1, 0, 0, 0, 0, 0],
 [1, 1, 0, 0, 0, 0],
 [1, 2, 1, 0, 0, 0],
 [1, 3, 3, 1, 0, 0],
 [1, 4, 6, 4, 1, 0],
 [1, 5, 10, 10, 5, 1]]
```

成長の様子 (?) が確認できますね。

最後に、このアルゴリズムの計算量を見積もってみましょう。上のプログラムでは、外側のループが k 回目の時、内側のループが回る回数は $k+1$ 回です。n 段までパスカルの三角形を作るには、k を $1\ldots n$ まで変化させるので、その合計は、$2+3+4+\ldots+(n+1) = \frac{(n+1)(n+2)}{2} - 1$ になります。よって、計算量は $O(n^2)$ と見積もれます[7]。

[7]残念ながら、パスカルの三角形を用いる方法は、掛け算・割り算を交互に行なう方法 (計算量 $O(N)$) に比べて遅いと言えます。では、パスカルの三角形は全くの無駄かと言えば、その限りではありません。例えば、同じプログラム中で様々な値に対する組合せ数を複数回使う場合は、その都度計算するのではなく、事前に計算しておき、状況に応じて必要な部分を取り出す方が効率的です。この考え方は、#12 以降で取り上げる動的計画法に活かされています。

演習 11-1 以下に出て来る Python プログラムへ、様々な n を与えた場合の時間計算量を見積もりなさい。見積もり後は、その見積もりが正しいかどうかを、下記の measur_time 関数により、実際の所要時間を計測して確認しなさい。

```
def measure_time(n, c):
  start = time.process_time()
  for i in range(c):
    square1(n)
  finish = time.process_time()
  print("%.10g" % (finish - start))
```

注意!

measur_time の計測値はあまり時間が短いと誤差が大きいので、少なくとも 0.1 秒よりは大きくなるように回数 c を増やして計測すること (言わずもがなですが、1 回当たりの実所要時間は、表示された時間を回数 c で割れば求められますね)。

a. n^2 を計算する関数 (その 1)

```
def square1(n):
  return(n * n)
```

b. n^2 を計算する関数 (その 2)

```
def square2(n):
  result = 0
  for i in range(n):
    result = result + n
  return(result)
```

c. n^2 を計算する関数 (その 3)

```
def square3(n):
  result = 0
  for i in range(n):
    for j in range(n):
      result = result + 1
  return(result)
```

d. 1.0000000001^n を計算する関数 (その 1)

```
def near1pow1(n):
  result = 1.0
  for i in range(n):
    result = result * 1.0000000001
  return(result)
```

e. 1.0000000001^n を計算する関数 (その 2)[8]

```
def near1pow2(n):
  return(math.exp(n * math.log(1.0000000001)))
```

[8] math.log は $\ln x$、math.exp は e^x を計算する関数です。

f. 1.0000000001^n を計算する関数 (その 3)[9]

```
def near1pow3(n):
  if n == 0:
    return(1.0)
  elif n == 1:
    return(1.0000000001)
  elif n % 2 > 0:
    return(near1pow3(n-1) * 1.0000000001)
  else:
    return(near1pow3(n/2) ** 2)
```

g. 1〜3 の値が n 個並んだ全組み合わせを生成する (関数の詳細については#9 を参照のこと)

```
def nest3(n, s):    # 呼び出し方: nest3(5, "") ←空文字列を渡す。
  if n <= 0:
#   print(s)        # 画面への表示はとても遅いので、時間計測時はコメント アウト
    pass
  else:
    for i in range(1, 4):
      nest3(n-1, s + str(i))
  return
```

h. 1〜n の値による全順列を生成する (関数の詳細については#9 を参照のこと)

```
def perm(n):
  a = ita.array.make1d(n)
  for i in range(len(a)):
    a[i] = i+1
  perm1(a, [])

def perm1(a, b):        # 呼び出し方: nest3([1,2,3], []) ←空配列を渡す。
  if len(a) == len(b):
#   print(b)            # 画面への表示はとても遅いので、時間計測時はコメント アウト
    pass
  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
```

[9] このアルゴリズムは、n 乗を高速に計算する際の基本的な方法です。もし、n が偶数 ($n = 2m$) であれば、$X^n = X^{2m} = \left(X^m\right)^2 = \left(X^{\frac{n}{2}}\right)^2$ より、指数を半分にできるので、掛け算の回数を減らせます。但し、n が奇数 ($n = 2m + 1$) の時は、$\left(X^{m+\frac{1}{2}}\right)^2$ となり、X の平方根を求める必要が生じるため、この方法はあまり嬉しくありません。この場合は、near1pow3 関数にあるように、素直に計算 (掛け算) した方が良さそうです。

演習 11-2 引き算の代わりに剰余演算を用いた**ユークリッドの互除法** (Euclid's Algorithm) により最大公約数を求める関数 `gcd_div(a, b)` を作成し、所要時間の測定および計算量を見積もりなさい。

ユークリッドの互除法により最大公約数を求める手順を以下に載せておきます。

1. 二つの数 a, b を比較し、大きい方 (例えば a) を小さい方 (b) で割り算する。
2. 手順 1 で割り切れた場合、最大公約数は b となる。
3. 同、割り切れなかった場合は、a を b とし (a $\leftarrow$ b)、また得られた余りを新たな b として手順 1 へ戻る。

参考のため、$a = bq + r$ $(a > b)$ の時、$gcd_div(a, b) = gcd_div(b, r)$ が成立する理由を以下に示しておきます (#9 で説明した引き算を用いて最大公約数を求める方法の略証と似ています)。

略証:
a と b の最大公約数を G とする (*1)。$a = Ga',\ b = Gb',\ a = bq + r$ より、$Ga' = Gb'q + r \rightarrow r = G(a' - b'q)$ なので、G は r の約数になる (*2: 但し最大かどうかは不明)。ここで、b と r の公約数として G より大きい H $(> G)$ があると仮定する。この時、$b = Hb$",$\ r = Hr'$ より $a = Hb$" $+ Hr'$ なので、H は a の約数となる。仮定より、H は b の約数であり、a と b には G より大きい公約数が存在することになる。これは G の定義に矛盾する。よって、H は存在せず、*1 · *2 より、$a,\ b$ の最大公約数と $b,\ r$ の最大公約数は等しいと言える。

＊ 再帰処理 · ループ処理 · その他における計算量の見積もりは、とても重要な分野なので、教科書を含め理解を深めておいて下さい。

おまけ: ゲームの作成

最後に、お楽しみの話題として、ゲームの作成をしてみましょう (皆さん、好きですよね)。ゲームの中には、将棋や囲碁のように (先手 · 後手を決める以外は) 厳密な最適解の探索によるものや、サイコロやカードのシャッフルなどを通じてランダム性を採り入れたものがあります[10]。以下では、ランダム性を採り入れた簡単なゲームとして、次の規則に従う「数当て」を作ってみましょう:

- プログラムは外部より、秘密の数字の桁数 n を指定される。
- プログラムは内部で n 桁の秘密の数字を作成する (n 桁の中に重複はない/また 0 もない)。
- プレーヤはその n 桁の数字を当てることを目指し、自分も n 桁の数字を入力する。
- プログラムは二つの数字を照合して、「同じ位置に同じ数字がある (これをヒットと呼ぶ)」個数と、「同じ数字はあるが、但し違う位置にある (これをブローと呼ぶ)」個数とを数えて知らせる。
- プレーヤはその情報を見て再度チャレンジする。
- 2*n 回以内のチャレンジで当たればプレーヤの勝ち、さもなければプレーヤの負けとする。

このルールに従う Python プログラムを次ページに示します。このプログラムでは、プレーヤと「やり取り」をするために、キーボードからの入力を 1 行取り出す `input` 関数を使っています。`n` 桁のランダムな数を作る方法がちょっと分かりづらいかも知れません。まずは、`a` を空配列 (要素数は引数として渡された `n` 個) とし、`b` を 1 ～ 9 の数字が一つずつ入った配列とします。次に、「`b` の中からランダムに一つの要素を選んで、そこにある数字を順番に `a` へ代入し、選ばれた `b` の要素は 0 に置き換える」ことを `n` 回やっています。秘密の数字は、各桁の数字が重複しないという規則なので、ランダムに選んだ `b` の要素が 0 だった場合は、その数字は既に秘密の数字のどこかに使われているとして、再度選び直します (言い忘れましたが、`n` が 10 以上だと必ず重複してしまうので、これは一番最初に調べて弾いています)。最後に、プログラム本体では、プレーヤが入力した数字とプログラムが作成した秘密の数字との全組合せを照合し、ヒットとブローの数を数えています。

改造: 数当てゲームに対し、上手 · 下手に応じた手加減を指定できるように拡張せよ。

例えば、

- 試行回数を増やす。
- ヒットしたら、その数字を教える。
- 同じ数字を 2 回ブローしたら、その数字を教える。
- ブローが 2 個あったら、片方の数字を教える。

など。

[10]ランダム性を入れる理由は、ゲームの「場面」が毎回違ったものになり新鮮さが保たれることだけではありません。例えば、複数プレーヤで行なう場合に、「上手 · 下手」以外の要因が入ることで下手な人にも勝つチャンスが生まれ、勝負の行方をそう簡単に予想できないという点もあります。

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

def make_secret_figure(a, b):

  # b = [1, 2, 3, 4, 5, 6, 7, 8, 9]
  for i in range(len(b)): b[i] = i + 1
  i = 0

  # a には、同じ数字を入れない。
  while i < len(a):
    j = int(len(b) * random.random())
    if b[j] != 0:
      a[i] = b[j]; b[j] = 0;
      i = i + 1

  return


def kazuate(n):

  # 本来ならば、n が文字や実数の場合を弾く必要があるが、今回はこのくらいで。
  if n <= 0 or n >= 10:
    return("error: 1 <= n <= 9, but given n is %d" % (n))
  a = ita.array.make1d(n)
  b = ita.array.make1d(9)
  make_secret_figure(a, b)

  # プログラム本体
  count = 0
  while True:
    print("your guess?")
    s = input()

    # 秘密の数字と入力された数字を 1 文字ずつ突き合わせる (二重のループに注意)。
    count += 1; hit = 0; blow = 0
    for i in range(len(a)):
      for j in range(len(a)):
        if int(s[i]) == a[j]:
          if i == j: hit += 1
          else: blow += 1

    # 以下は判定処理
    if hit == len(a):
      print("you win! answer = ", end=""); print(a)
      return(None)
    elif count > (2*len(a) - 1):
      print("you lose! answer = ", end=""); print(a)
      return(None)
    else:
      print("hit = %d, blow = %d." % ((hit), (blow)))

  return
```