アルゴリズム入門　＃10

地引　昌弘

2021.12.16

はじめに

今回からは、再帰処理の利用による、具体的な問題解決に向けた様々なアルゴリズム手法やその考え方を取り上げて行きます。最初は「整列」の問題です。「整列」を行なうアルゴリズムは、実は一つではありません。今回は、基本的な「整列」のアルゴリズムを学ぶとともに、アルゴリズム解析において重要な「計算量」への導入として、各アルゴリズムの所要時間を計測してみます。

- 基本的な整列アルゴリズム
- 整列アルゴリズムの所要時間

再帰処理の考え方

再帰処理は、自分自身を異なるパラメータで順々に呼び出して行く処理なので、基本的に**漸化式** (Recurrence Relation — 再帰関係式) として定義できます。例えば、前回取り上げた "1 ～ 2 減少列" は、下記のように表わせます。減少列では、自分の担当範囲を数値 n で表わせるため、漸化式も簡単になります。

$$decr1\,(\underline{n}\,,\,b) = \begin{cases} decr1\,(\underline{n-1}\,,\,b),\ decr1(\underline{n-2}\,,\,b) & (n \geq 2) \\ decr1\,(\underline{n-1}\,,\,b) & (n = 1) \\ \texttt{print}(\texttt{b}) & (n = 0) \end{cases}$$

因みに上の式では、Python のプログラムに合わせて作業用の配列 b も追記していますが、漸化式の意味を考えると、重要なのは下線を引いた部分です (漸化式の右辺と左辺で変化する部分)。また、これはあくまで呼び出し関係を表わす漸化式なので、必ずしも演習 9-2 のような "漸化式 ≡ 再帰処理" という関係にはなっていません。以後の漸化式も基本的に同じ扱いです。

1 ～ 2 減少列に対し、順列の表示では、減少列と異なり自分の担当範囲を一つの数値で表わすことが難しいため (言い換えれば、担当範囲が単純な法則に従った番号になっていないため)、配列 a と目印 None を導入しています。この場合の漸化式は、例えば下記のように表わせますが (← None は "None を代入する" の意)、だいぶ変則的ですね。

$$perm1\,([a_0,\,\ldots,\,a_i \neq \texttt{None},\,\ldots,\,a_{n-1}],\ b) = \begin{cases} perm1\,([a_0,\,\ldots,\,a_i \leftarrow \texttt{None},\,\ldots,\,a_{n-1}],\ b) & (\exists a_i \neq \texttt{None}) \\ \texttt{print}(\texttt{b}) & (\forall a_i = \texttt{None}) \end{cases}$$

再帰処理の動作を理解するには、このような漸化式や再帰関係を示す樹形図を作成することが、最初の 1 歩になります (これを怠り、頭の中だけで考えてもうまく行きません)。

再帰処理における副作用について

次に、再帰処理のプログラムを作成するには、副作用の有無を判断し、再帰処理により副作用が生じる場合は、それを修正する (つまり、再帰呼び出しを終了する際に該当データを原状復帰する) 必要があります。

例えば、下記のような再帰呼び出しがあったとします。

```
def r_call(..., a, ...):
    .....
    .....                  # 再帰呼び出しの直前 (*1)
    r_call(..., a, ...)
    .....                  # 再帰呼び出しの直後 (*2)
    .....
    return                 # 自身の終了時 (*3)
```

この場合、再帰呼び出しで利用するデータ a について、(*1) と (*2) における差異を考慮する必要があります。両者の差異については、次の 3 通りがありますが、

- 差異があってはならない
- 差異がなければならない
- どちらでもよい

どれに該当するかは、再帰処理の設計図となる呼び出し関係の樹形図を見て判断します (これが鍵です)。樹形図では、再帰処理の呼び出し元において、呼び出し先から帰って来た後のデータの扱い (or 想定しているデータ) に着目します。ここで、“差異があってはならない” と判断できれば、上記 (*3) の前に該当データを原状復帰した後、(自身の) 再帰呼び出しを終了します。

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。

1 前回の演習問題の解説

1.1 演習 9-2: — 再帰関数

これらは、定義通りに再帰関数を作ればできます。以下にコードを示しておきます:

```
def fact(n):
    if   n < 0:  return(0)                        # n に負数を渡された場合の対応を追加
    elif n == 0: return(1)
    else:        return(n * fact(n-1))

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))

def comb(n, r):
    if   (n < 0) or (r < 0) or (n < r): return(0)  # n,r に負数を渡された場合の対応を追加
    elif (r == 0) or (r == n):          return(1)
    else:                               return(comb(n-1, r) + comb(n-1, r-1))


def binary(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")
    elif n%2 == 0: return(binary(n/2) + "0")
    else:           return(binary((n-1)/2) + "1")
```

これらの例は単純な計算なので、再帰呼び出しによる副作用を消すコードは不要ですね。実行例も一応示しておきます:

```
>>> fact(4)
24
>>> fact(5)
120
>>> fib(9)
34
>>> fib(10)
55
>>> fib(11)
89
>>> comb(5, 2)
10
>>> comb(6, 2)
15
>>> binary(5)
'101'
>>> binary(7)
'111'
>>> binary(8)
'1000'
```

1.2 演習 9-3a: — 単純減少列の列挙

単純減少列を漸化式で表わすと、下記のようになります (1 ページの漸化式では、減少の大きさに応じて複数の行に分けましたが、ここでは全てまとめて 1 行にしてあります):

$$decr_all\,(n,\ b) = \begin{cases} decr_all\,(n-1,\ b),\ decr_all(n-2,\ b),\ \ldots,\ decr_all(n-i,\ b) & (n \geq i \geq 1) \\ \texttt{print(b)} & (n = 0) \end{cases}$$

この漸化式の 1 行目にある減少の種類は、ループを用いることで簡単に生成できます。但し、for ループにおけるカウンタ i の範囲に注意して下さい。直前の insert 関数により、この再帰呼び出しが担当する数字を配列 b に入れてあるため、以降の再帰呼び出しでは、1 以上減らした数字 n を渡す必要があります。また、再帰呼び出しに伴う副作用に注意しましょう:

```
def decr_all(n, b):
  if n == 0:
    print(b)
  else:
    b.insert(len(b), n)
    for i in range(1, n+1):     # i の範囲は 1 <= i <= n (終値に注意)
      decr_all(n-i, b)
    b.pop(len(b)-1)             # 副作用の解消
  return
```

この関数を呼び出す際は、これも “decr_all(n, [ ])” のように空の配列を渡します:

```
>>> decr_all(5, [])
[5, 4, 3, 2, 1]
[5, 4, 3, 2]
[5, 4, 3, 1]
[5, 4, 3]
(途中略)
[5, 1]
[5]
```

1.3 演習 9-3b: — 重複を許す順列の列挙 1

次は、「配列として渡した値を L 個を並べた全ての場合」ですが、これは重複を許してよいため、実は順列より簡単で、各再帰呼び出しの担当範囲は単純に L で表わすことができます。漸化式は下記の通り:

$$comb_all\,(a,\ l,\ b) = \begin{cases} comb_all\,(a,\ l-1,\ b) & (l \geq 1) \\ \texttt{print(b)} & (l = 0) \end{cases}$$

漸化式同様、Python のプログラムも簡単ですが、副作用に注意して下さい:

```
def comb_all(a, l, b):
  if l == 0:
    print(b)
  else:
    for i in range(len(a)):
      b.insert(len(b), a[i])
      comb_all(a, l-1, b)
      b.pop(len(b)-1)
  return
```

実行例も示しておきます:

```
>>> comb_all(['a','b'], 3, [])
['a', 'a', 'a']
['a', 'a', 'b']
['a', 'b', 'a']
['a', 'b', 'b']
['b', 'a', 'a']
['b', 'a', 'b']
['b', 'b', 'a']
['b', 'b', 'b']
```

ところで、`decr_all` 関数と `comb_all` 関数について、各再帰呼び出しが、自身の担当する部分を配列 `b` に入れている箇所を、少し見比べてみましょう。処理の流れはどちらも、

1. 自分の担当部分を記録
2. 自分の担当部分以降 (or 以外) を処理するため、再帰呼び出し
3. 自分が呼び出した再帰先から戻った後は、自身を呼び出した再帰元に戻るため、副作用を解消

というものです。これより、`decr_all` 関数を、(`comb_all` 関数に似せた) 次のような `decr_all1` 関数として作成しても、同じ結果を得ることができます (両者を見比べてみて下さい/ `for` 文のブロックに違いがあります):

```
def decr_all1(n, b):
  if n == 0:
    print(b)
  else:
    for i in range(1, n+1):     # i の範囲は 1 <= i <= n (終値に注意)
      b.insert(len(b), n)
      decr_all1(n-i, b)
      b.pop(len(b)-1)           # 副作用の解消
  return
```

念のため、実行例も示しておきます:

```
>>> decr_all1(5, [])
[5, 4, 3, 2, 1]
[5, 4, 3, 2]
[5, 4, 3, 1]
[5, 4, 3]
(途中略)
[5, 1]
[5]
```

しかし、comb_all 関数を、decr_all 関数と似せた作りにしようとしても、うまく行きません。その理由は、comb_all 関数では、各再帰呼び出しの担当部分、つまり配列 a の要素を参照する際に必要な添字番号として、for ループのカウンタ i を利用しているため、自分の担当部分を配列 b に記録する命令を、for ループのブロックとして記述せざるを得ないからです (decr_all 関数では、渡された数 n がそのまま各再帰呼び出しの担当部分になるので、自分の担当部分を配列 b に記録する命令を、for ループのブロック外に記述できるわけです)。また、“担当部分の記録” と “副作用の解消” は対になっているので、これらを記述する箇所を誤ると、当然プログラムは正しく動きません。再帰呼び出しを利用する場合は、このような部分に注意を向けるようにして下さい。

1.4 演習 9-3c: — 重複を許す順列の列挙 2

最後は、配列の要素の代わりに数字を列挙する問題です。演習 9-3b にある配列の要素を取り出す部分を、順番に数字を取り出すように変えるだけでできます (漸化式は同じなので、省略します)。但し、for ループにおけるカウンタ i の範囲に注意して下さい。各再帰呼び出しが担当する数字は、0 ∼ ではなく 1 ∼ なので、カウンタ i も 1 から始まるようにします:

```
def comb_all1(n, l, b):
  if l == 0:
    print(b)
  else:
    for i in range(1, n+1):    # i の範囲は 1 <= i <= n (終値に注意)
      b.insert(len(b), i)
      comb_all1(n, l-1, b)
      b.pop(len(b)-1)
  return
```

再帰処理の動作を理解するには、漸化式や再帰関係を示す樹形図を作成することが重要です。処理の種類によっては、漸化式 (or 樹形図) を作りにくい場合もありますが、そのような場合は、樹形図 (or 漸化式) の作成を試みましょう。

2 整列アルゴリズム

再帰呼び出しを用いる具体的アルゴリズムとして、まずは、数値の並んだ配列を受け取り、**昇順** (Ascending Order — 小さい値が先に来るような順番のこと)[1]に並べ換えることを考えましょう。このような、ある順番に従った並び替えを**整列** (Sorting) と言います。

2.1 単純選択法

整列アルゴリズムとして、まず最初に多くの人が思いつく整列アルゴリズムは、次のようなアルゴリズムではないでしょうか。

> 「数の並びから最小値を見つけ出し、それを取り出す動作を繰り返して行く。
> 数値を取り出した順に並べると昇順の整列結果になっている。」

この方法は、単純に小さい値をその都度選ぶことから、**単純選択法** (Selection Sort) と呼ばれます。

単純選択法のアルゴリズムでは、「配列 `a` の `i` 番目から `j` 番目までの間で、最も小さい要素が何番目にあるかを返す」操作が必要になります。その操作を下請け関数として用意したアルゴリズムは、次のようになります:

- selection_sort(a): 配列 a を単純選択法で整列
-     i を 0 から len(a) - 2 まで変化させながら繰り返し、
-         k ← a の i 番から len(a) - 1 番までの最小要素の番号
-         `a[i]` と `a[k]` の内容を交換

上の擬似コードでは、i の上限を len(a) - 2 にしています。その理由は次の通りです。今回採用した単純選択法のアルゴリズムでは、「配列 `a` の `i` 番目から `j` 番目まで」を探しますが、配列の添字番号は 0 ∼ len(a) - 1 なので、一番最後は「配列 `a` の len(a) - 2 番目から len(a) - 1 番目まで」を探すことになるからです。ところで、上では、選んだ最小の要素を取り出す代わりに、"今回調べた配列" の先頭 (添字の "`i`" に注意) にある要素と、最小の要素とを交換しています (図 1)。直感的には、別の配列 `b` を用意し、`a` から選んだ最小の要素を `b` に追加して行く方が分かり易いですね。しかしながら、整列アルゴリズムには、順番を入れ替える処理が多く含まれるので、それに慣れるため、ここでは敢えて交換という手段を利用しました (今後もそうします)。

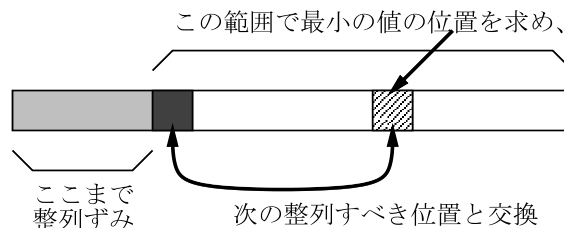

図 1: 単純選択法による整列

Python のプログラムを次に示します。`array_minrange` 関数は、最小値そのものでなく、その位置を返します。また、配列の要素を交換する `swap` 関数も用意しました。各 `for` ループの終値に注意して下さい:

```
def array_minrange(a, i, j):
  p = i; min = a[p]
  for k in range (i, j+1):    # 終値=j (len(a)-1 まで調べる)
    if min > a[k]:
      p = k; min = a[k]
  return(p)
```

[1]逆に大きい値が先に来るような順番の場合は、**降順** (Descending Order) と呼びます。

```
def swap(a, i, j):
  x = a[i]; a[i] = a[j]; a[j] = x

def selection_sort(a):
  for i in range(len(a)-1):    # 終値=len(a)-2
    k = array_minrange(a, i, len(a)-1)
    swap(a, i, k)
  return
```

2.2 単純挿入法

単純選択法は、数を「取り出す時」に正しい順にするという方法でした。これとは逆に、数を「入れる時」に正しい位置へ入れるという方法があります。

> 「数の並びから (何もせず) 順に数を取り出し、それを新しい列に加えて行く。
> 但し、新しい列に入れる時は、"順番として正しい" 位置に挿入する (挿入した数の後ろにある要素は、その分ずらす必要があることに注意)。」

この方法は、単純に各要素を次々とあるべき位置へ挿入して行くことから、**単純挿入法** (Insertion Sort) と呼ばれます。

単純挿入法のアルゴリズムでは、「配列 a の i 番目から j 番目までを一つ後ろにずらす」操作が必要になります。その操作を下請け関数として用意したアルゴリズムは、次のようになります:

- insertion_sort(a): 配列 a を単純挿入法で整列
- i を 1 から len(a) - 1 まで変化させながら繰り返し、
- x ← a[i]
- k ← 0
- k < i かつ a[k] ≤ x である間、k ← k+1 を繰り返す。
- a の k 番目から i - 1 番目までを一つ後ろにずらす。
- a[k] ← x

上の疑似コードでは、i の始値を 1 にしています。これは、先頭の要素は既に整列済みで、2 番目 (以降) の要素を、先頭の要素の前に挿入するか後ろに挿入するかを調べればよいからです。また、この疑似コードも、取り出した要素を入れる新しい配列を用意する代わりに、"今回調べた要素" より前の部分 (添字の "i" に注意) を利用しています (図 2) [2]。なお、配列を「後ろにずらす」時は、後ろから順にずらさないと問題が生じることに注意して下さい (図 3) [3]。

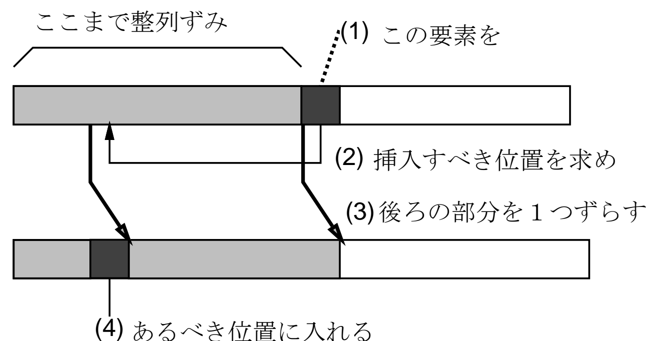

図 2: 単純挿入法による整列

[2] 単純選択・挿入法で、配列を一つしか用意せず、"交換" を利用する理由は、整列アルゴリズムに慣れるだけではありません。実質的な利点として、メモリ領域を節約できることが挙げられます。例えば、携帯端末など、ハードウェア資源に制約がある場合は、その制約に束縛されたアルゴリズムが必要になる場合があります。

[3] 後ろから順にずらしたとしても、最後の要素 (図 3 で "3" の右側にある要素) は上書きされるではないか、という疑問が湧いて来ます。単純挿入法は、(上の括弧書きにあるように) 未整列の要素から何もせずに先頭の要素を取り出すので (図 2 の黒い部分)、"上書きされる要素" = "取り出された要素" として事前に覚えておくことにより、これを回避します (次ページの Python プログラムでは、"x" = "未整列の先頭要素" として退避)。

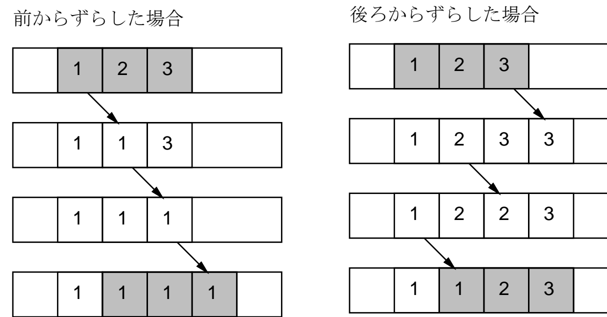

図 3: 配列をずらす

Python のプログラムは次の通りです。今回は、range(start, stop, step) 関数における増分 (step) が負数なので、終値 (stop) の扱いに注意して下さい:

```
def array_shiftrange(a, i, j):
  for k in range(j, i-1, -1):    # 終値=i (増分が負数なので特に注意)
    a[k+1] = a[k]

def insertion_sort(a):
  for i in range(1, len(a)):     # 終値=len(a)-1
    x = a[i]; k = 0
    while (k < i) and (a[k] <= x):
      k = k + 1
    array_shiftrange(a, k, i-1)
    a[k] = x
  return
```

実行例は下記の通り。insertion_sort 関数は、整列した配列を返すのではなく、渡された配列 a の中身を書き換えて昇順に整列するだけです:

```
>>> a = [1, 9, 5, 4, 2, 3]
>>> a
[1, 9, 5, 4, 2, 3]
>>> insertion_sort(a)
>>> a
[1, 2, 3, 4, 5, 9]
```

ここで改めて、range(start, stop, step) 関数における始値 (start), 終値 (stop), 増分 (step) と、カウンタとの関係を整理しておきます。

```
for i in range(start, stop, step):
  ...
```

において、カウンタ i は、start から始まり、stop の直前までの値を、step ずつ取ります (start は含みますが、stop は含みません)。start > stop の場合や step < 0 の場合も、この原則は変わりません[4]。よって、step = ±1 の場合、i を stop まで変化させるには、下記のどちらかとなります。

- start < stop, step = 1: range(start, stop + 1, 1)
- start > stop, step = -1: range(start, stop - 1, -1)

但し、|step| > 1 の時は注意が必要です。この場合、i は step ずつ増えて行くため (即ち、i = step の倍数)、例えば for ループ内の処理で "step の倍数 ± x" を抜けなく扱うには、終値を stop ± 1 ではなく、stop ± y にする必要があります。その際、常に x = y と指定できれば良いのですが、このような指定により、逆に新たな抜け · 余分な周回が生じる場合も存在するため、for ループ内の処理に合わせて慎重に y を決める必要があるわけです (#7 で取り上げた素数発見プログラムにおいて、6 の倍数 ± 1 だけを調べる場合を参照のこと)。

2.3 バブル ソート

単純選択・挿入法より技巧的な整列アルゴリズムとして、**バブル ソート** (Bubble Sort) と呼ばれるものがあります。バブル ソートでは、各要素と隣り合う要素を互いに比較して行き、逆順になっている箇所があれば交換します。この処理を一度行なうと、大きい要素が右の方に移動します。処理が終わった後は、再度先頭から処理を行ないます[5]。これを繰り返して行くと、最後は全ての要素が昇順に並び、交換が起きなくなるはずです (図 4)。

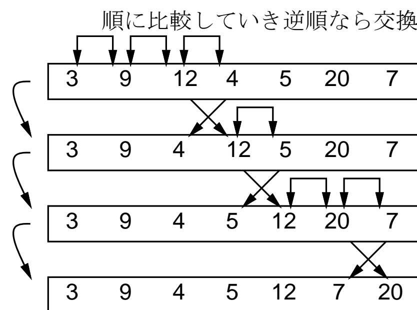

図 4: バブル ソートによる整列

以下に、バブル ソートの疑似コードを示します。

- bubble_sort(a): 配列 a を昇順に整列
-     done ← 偽。
-     done でない間繰り返し、
-         done ← 真
-         i を 0 から len(a) - 2 まで変えながら繰り返し、
-             もし `a[i]` と `a[i+1]` の順番が逆なら、
-                 `a[i]` と a[i+1] の値を交換
-                 done ← 偽
-             (枝分かれ終わり)
-         (繰り返し終わり)
-     (繰り返し終わり)

この疑似コードでは、内側のループで隣り合う要素を順に見て行きます。i の上限を len(a) - 2 とする理由は、隣との比較は右端にある要素 (添字番号: len(a) - 1) の一つ手前 (添字番号: len(a) - 2) まで行なえばよいからです。また、上のコードでは、繰り返しを終えてよいかどうかを判断するために、`done` (終了) というフラグを用意しています。まずはフラグを立ててから、比較+交換を行ないます。もし交換が生じたら、その事を示すためにフラグを降ろします。最後までフラグが立ったままならば、1 回も交換が発生しなかった、つまり 1 箇所も逆順の部分がなかったということなので、整列が完了したと判断できます。Python によるバブル ソートのコードは次のようになります (ここでは、配列の要素を交換する `swap` 関数を利用しています):

```
def swap(a, i, j):
  x = a[i]; a[i] = a[j]; a[j] = x

def bubble_sort(a):
  done = False
  while not done:
    done = True
    for i in range(len(a)-1):    # 終値=len(a)-2
      if a[i] > a[i+1]:
        swap(a, i, i+1)
        done = False
  return
```

[5] バブル ソートと呼ばれる由来は、各要素の移動する様子が、水中から泡が浮かんで来るのに似ているため、こう呼ばれるとされています。

2.4 マージ ソート

皆さんには意外かも知れませんが、実は整列アルゴリズムにおいて、「単純選択法」「単純挿入法」「バブル ソート」は、効率的な (速い) アルゴリズムではありません (振舞いは、直感的に分かり易いのですが)。では、整列処理において、どのようなアルゴリズムが速いのでしょうか。以下に、その一つとして、再帰処理を利用した**マージ ソート** (Merge Sort) と呼ばれるアルゴリズムを紹介します。ここで言う**マージ** (Merge) とは、**併合**とも呼ばれ、図 5 のように二つの整列済みの列を「併せて」一つの整列済みの列にすることを言います。

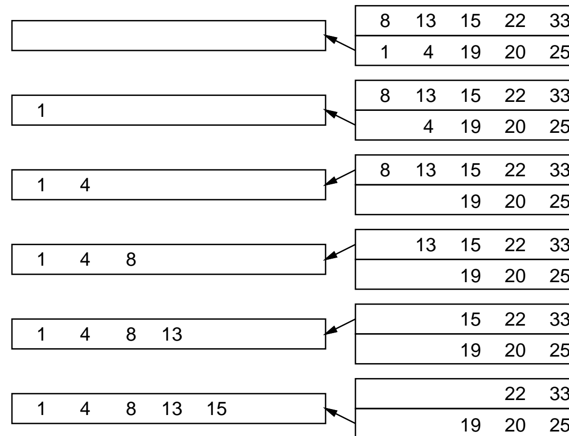

図 5: マージの処理

以下に、マージ ソートの擬似コードを示します:

- merge_sort(a, i, j) —- 配列 a の i 番から j 番の範囲を整列
- 　もし $j \leq i$ なら、　　# 副列は `a[i]` から `a[j]` まで, 副列のサイズ = j - i + 1
- 　　何もしない
- 　そうでなければ、
- 　　$k \leftarrow (i + j) / 2$
- 　　merge_sort(a, i, k), merge_sort(a, k+1, j)　　# 列を半分に分け、両者に再帰呼び出しを実施
- 　　b ← merge(a, i, k, a, k+1, j)　　# b は一時的な作業用配列
- 　　b の内容を a の位置 i ～ j にコピーして戻す

考え方としては、再帰呼び出しによって、列全体を半分ずつにした擬似的な複数の “副列” を作って行きます。ここでは、「配列のどこから (`i`)、どこまで (`j`) が、どの副列に属するか」を示す二つの添字番号 (`i, j`) が鍵になります。“副列の長さが 1 以下の時は (二つの添字番号の関係に注意)、「既に整列済み」の副列” と認識し、何もしないで帰ります。再帰から戻って来たら、二つの整列済みの副列をマージすることで、長い整列済みの副列にします (図 6)[6]。下請けとなるマージの擬似コードは次の通り:

- merge(a1, i1, j1, a2, i2, j2) —- `a1[i1, ..., j1]` と `a2[i2, ..., j2]` を併合
- 　b ← 空の配列
- 　i1, ..., j1 と i2, ..., j2 の少なくとも一方が空でない間、
- 　　もし i1, ..., j1 が空 または `a1[i1]` > `a2[i2]` なら、
- 　　　`a2[i2]` を b に追加し、i2 を 1 増やす
- 　　そうでなければ、
- 　　　`a1[i1]` を b に追加し、i1 を 1 増やす
- 　　(枝分かれ終わり)
- 　(繰り返し終わり)
- 　b を返す

[6]つまり、マージ ソートとは、まずは列を次々と単純に半分割して行き、副列の要素数が 1 に至れば、「要素数 1 ならば既に整列済み」として後はマージだけして行く (+マージ済みの副列は整列済みのはず)、というアルゴリズムです。

図 6: マージ ソートによる整列のイメージ

では、Python のコードを見てみましょう:

```
def merge_sort(a, i, j):
  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):
  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)
```

実際に動かした結果は、以下の通り:

```
>>> a = [3, 9, 12, 4, 5, 20, 7]
>>> a
[3, 9, 12, 4, 5, 20, 7]
>>> merge_sort(a, 0, 6)      ← 配列の添字番号は 0 から始まる点に注意
>>> a
[3, 4, 5, 7, 9, 12, 20]
```

マージ ソート (および再帰呼び出し) の理解を深めるため、図 6 を手掛かりに、Python コードの動作を少し追跡してみましょう。以下では、ユーザから呼び出された merge_sort 関数を $\mathrm{ms}^{(\mathbf{1})}(\ )$、第 i 世代の再帰呼び出しを $\mathrm{ms}^{(i)}(\ )$ と表わすことにします。

1. $\mathrm{ms}^{(\mathbf{1})}(\ )$ は、ユーザから渡された配列 [3, 9, 12, 4, 5, 20, 7] を [3, 9, 12, 4] と [5, 20, 7] の二つに分け、前半の [3, 9, 12, 4] を $\mathrm{ms}^{(\mathbf{2})}(\ )$ に渡す (`merge_sort(a,i,k)`)。
2. $\mathrm{ms}^{(\mathbf{2})}(\ )$ は、渡された配列 [3, 9, 12, 4] を [3, 9] と [12, 4] の二つに分け、前半の [3, 9] を $\mathrm{ms}^{(\mathbf{3})}(\ )$ に渡す (`merge_sort(a,i,k)`)。
3. $\mathrm{ms}^{(\mathbf{3})}(\ )$ は、渡された配列 [3, 9] を [3] と [9] の二つに分け、前半の [3] を $\mathrm{ms}^{(\mathbf{4})}(\ )$ に渡す (`merge_sort(a,i,k)`)。
4. $\mathrm{ms}^{(\mathbf{4})}(\ )$ は、渡された配列 [3] を (既に) 整列済みとして、$\mathrm{ms}^{(\mathbf{3})}(\ )$ に戻る (`pass`)。
5. $\mathrm{ms}^{(\mathbf{3})}(\ )$ は、配列 [3, 9] を二つに分けた後半の [9] を $\mathrm{ms}^{(\mathbf{4})}(\ )$ に渡す (`merge_sort(a,k+1,j)`)。
6. $\mathrm{ms}^{(\mathbf{4})}(\ )$ は、渡された配列 [9] を (既に) 整列済みとして、$\mathrm{ms}^{(\mathbf{3})}(\ )$ に戻る (`pass`)。
7. $\mathrm{ms}^{(\mathbf{3})}(\ )$ は、二つの配列 [3] と [9] をそれぞれ整列済みとしてマージし (`b = merge(a,i,k,a,k+1,j)` 以下)、$\mathrm{ms}^{(\mathbf{2})}(\ )$ へ戻る。
8. $\mathrm{ms}^{(2)}(\ )$ は、配列 [3, 9, 12, 4] を二つに分けた後半の [12, 4] を $\mathrm{ms}^{(3)}(\ )$ に渡す (`merge_sort(a,k+1,j)`)。
9. 以後、同様に繰り返し。

merge_sort 関数の動作が見えて来たでしょうか。前回も述べましたが (#9, 15 ページ)、merge_sort 関数の再帰関係を理解するには、その先頭で自身の担当範囲を表示させてみることが一つの方法です。その際、何世代目の再帰関数であるかも同時に表示させると、再帰関係はより見え易くなります。例えば、こんな感じ:

```
def merge_sort(a, i, j, n):        # 引数に、再帰の世代数を示す n を追加
  print(n, ": ", i, j)             # 世代数 (n) と自身の担当範囲 (i,j) を表示
  if j <= i:
    .....
  else:
    .....
    merge_sort(a, i, k, n+1)       # 再帰呼び出し時には、世代数を増やすことを忘れない
    merge_sort(a, k+1, j, n+1)     # 同上
    .....
```

実行する際は、ユーザからの呼び出しが 1 世代目である旨、指定を忘れないようにしましょう:

```
>>> merge_sort(a, 0, 6, 1)     ← 4 番目の引数で 1 世代目であることを指定
```

ところで、マージ ソートは、「バブル ソート」「単純選択法」「単純挿入法」に比べると速いアルゴリズムですが、これより速いアルゴリズムはまだ存在します (次に取り上げるクイック ソートが、それです)。しかし、マージ ソートの考え方には、データを端から順に処理して行けるという利点があります。このため、メモリに入り切らないような (ファイルに保管されているような) 大量のデータを処理する際によく使われます。その基本的な手順は、次のようなものです:

- データをファイルから読み込みながら、メモリに入る最大量ずつ高速なアルゴリズムで整列し、各整列毎に別のファイルへ書き出す。
- ファイル 1 とファイル 2 をマージしてファイル 1+2 を作り、ファイル 3 とファイル 4 をマージしてファイル 3+4 を作り、以下同様にファイルを対にしてマージして行く。
- これを繰り返し、最後に 1 本のファイルになったら完了。

このような、ディスクなど外部記憶の使用を前提とした整列のことを**外部整列** (External Sorting) と呼びます。これと対比して、本資料で扱っているようなメモリ上での整列のことを**内部整列** (Internal Sorting) と呼びます。

2.5 クイック ソート

最後に、**クイック ソート** (Quick Sort) と呼ばれる、いかにも速そうな名前が付いているアルゴリズムを紹介します。

クイック ソートは、マージ ソートと同じく、まずはデータの列を二つの副列に分けて行き、その後でマージする整列アルゴリズムです。但し、こちらは渡された数列の中から**ピボット** (Pivot) と呼ばれるある値 p を選び、「左側は p 以下、真ん中に p、右側は p より大きい」という副列に分けてから (分けるだけ/並び替えはしない)、左側および右側の各副列に対してそれぞれ自分自身を再帰的に呼び出します。再帰を続けることで、再帰呼び出しに渡される副列の長さが減って行き、最後は 1 になります。長さ 1 以下の副列は既に整列した列として扱い、マージ ソートと同様に再帰を終了します。その後、各段の再帰呼び出しは、自分が呼び出した二つ (左副列担当+右副列担当) の再帰呼び出しから戻った時点で、両者に渡した各副列がそれぞれ「p 以下の “整列された” 副列」と「p より大きい “整列された” 副列」になっているので、(何もしなくても) 自身の「p」と併せて整列が完了しているというわけです (図 7)。

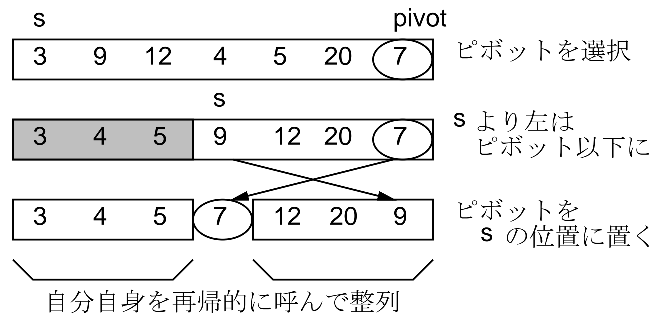

図 7: クイック ソートによる整列 (図中の s は下記コードの s に相当)

Python プログラムによるクイック ソートの手順は、以下のようになります (`swap` 関数は省略しています):

```
def quick_sort(a, i, j):
  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 (詳細は解説を参照)。
    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
```

一般に再帰呼び出しを利用する場合は、再帰の深さ (段数) が浅い方が全体として処理も少なそうなので、早く終わると予想できます。クイック ソートは再帰呼び出しを使っているので、直感的には、ピボット p が「ちょうど列を半分ずつに分ける値」になっていると、再帰の深さも浅くなり、整列を早く終えられそうです。しかし、通常はどんな値が該当するのか分からないため、取り敢えず上のプログラムでは右端 (添字番号 `j` の要素) の値を p にしています。変数 `s` は、「この添字番号の一つ手前までは p 以下の値を詰めてあるので、次に p 以下の値が見つかったらこの位置に入れる」添字番号を表しています。そこで (渡された配列の添字番号は `i` ～ `j` なので)、`k` を `i` (渡された配列の先頭) から `j-1` (同、最後の一つ前) まで左から順に調べて[7]、`a[k]` が p 以下なら、それを添字番号 `s` の要素と交換して `s` を一つ増やし、列を左側と右側に分けて行きます。分け終わったら、最後に添字番号 `j` の要素と `s` の要素を交換することで、保留してあったピボットの値をあるべき位置 (要は分け目) に置きます。以上の処理により、添字番号 `i` ～ `s-1` の要素にはピボット `p` 以下の値が、同 `s+1` ～ `j` にはピボット `p` 以上の値が入っているので、以後は、`i` ～ `s-1` (= `p` 以下の集合) と `s+1` ～ `j` (= `p` 以上の集合) の範囲についてそれぞれ再帰呼び出しを行ない、各副列を再帰的に整列させて行きます。

[7] 添字番号 `j` の要素はピボットが入っているので、調べる必要はありません。

2.6 整列アルゴリズムの拡張と検索

整列アルゴリズムに対するここまでの説明では、数値の列を整列させる場合を取り上げて来ましたが、当然ながら整列の対象は数値だけに限られることはありません。以下では、数値以外を整列させる例として、文字列の集合を整列させる場合を考えてみましょう。

整列させる対象が変わったとしても、整列アルゴリズムが根本的に変わるわけではありません。例えばクイック ソートを用いる場合、ピボットとなる文字列を基準に、配列の左側はピボット以下、右側はピボット以上の副列に分けてから、二つの副列に対して再帰処理を実施して行くというアルゴリズムの基本的な部分は変化しません。変わるのは、要素同士を比較する部分です。具体的には、`for` ループ内にある "`if a[k] <= pibot:`" の部分が変わります。文字列の大小は、一般的に下記の辞書式順序で決められます。つまり、`a[k] <= pibot` の代わりに、二つの文字列に対して下記の比較を行なう関数を用いることになります[8]。

1. 文字列を A: $a_1 a_2 \cdots a_n$, B: $b_1 b_2 \cdots b_m$ (但し $n < m$) とする。
2. a_1 と b_1 を比較し、$a_1 < b_1$ であれば、文字列 A $<$ 文字列 B とする。
   Python における大小関係は、1) 大文字 $<$ 小文字, 2) アルファベット順の優先順位で比較される。
   例: "cat" $<$ "dog", "Dog" $<$ "cat"
3. $a_1 = b_1$ であれば、a_2 と b_2 を比較する。以後、$a_i = b_i$ であれば、a_{i+1} と b_{i+1} を比較して行く。
4. a_n, b_n まで比較しても差がなかった場合、$n < m$ より (つまり文字列 A には、もう比較する文字 a がない) 文字列 A $<$ 文字列 B とする。

ところで、データを整列させる目的ですが、これも当然ながらデータを綺麗にしまっておくためではありません。データを整列させる利点の一つとして、データ検索の効率化が挙げられます。データの集合から、ある条件に合ったデータを検索したい場合、データが全くばらばらにしまわれていると、条件に合ったデータがどこにあるのか分からないので、全てのデータを調べなければなりません。しかし、データが何かの規則により整列されていれば、現在自分が調べている場所からどちらの方向へ探しに行けばよいのかが分かるため、効率的にデータを検索できます。

整列されたデータ集合から、必要なデータを効率的に検索する方法として、下記の**二分探索** (Binary Search) があります。

1. 整列されたデータ集合 $D^{(1)}$: $\{d_1, d_2, \cdots, d_n\}$ から、データ `d` を検索する。
2. $D^{(1)}$ の中央にあるデータ $d_{c^{(1)}}$ とデータ `d` を比較する。$d_{c^{(1)}} =$ `d` であれば検索終了。
3. $d_{c^{(1)}} >$ `d` ならば、$D^{(1)}$ の前半分となる $D^{(2)}$: $\{d_1, d_2, \cdots, d_{c^{(1)}}\}$ に対し、手順 2 と同じ比較を行なう。
4. $d_{c^{(1)}} <$ `d` ならば、$D^{(1)}$ の後ろ半分となる $\widehat{D}^{(2)}$: $\{d_{c^{(1)}+1}, \cdots, d_n\}$ に対し、手順 2 と同じ比較を行なう。
5. 以下、「$d_{c^{(k)}} =$ `d`」or 「$D^{(k)}$ または $\widehat{D}^{(k)}$ の要素数が 0」になるまで、手順 2 ～ 4 と同じ処理を繰り返す。

二分探索は、1 回の比較毎に検索対象となるデータ集合の大きさが半分になって行くため (別の言い方をすれば、1 回の比較毎に該当候補を半分に絞り込めるため)、高速に 1) 該当するデータが存在する場合は取り出す, 2) 存在しない場合はどのデータに一番近いかを調べる、ことができます。処理の速さに関する形式的な議論は、次回行ないます。

[8]Python では文字列同士を直接、比較演算子により辞書式順序で比較することができます (例えば、"Dog" $>$ "cat" とすれば False が返ります)。このような辞書式順序による比較演算機能がない言語では、比較用の関数を自前で作る必要があります。

3 整列アルゴリズムの計測

これまでの説明では、各整列アルゴリズム毎に速い/遅いなどと述べて来ましたが、本当に速い/遅いのかどうか、実際に「整列プログラムの処理時間」を計測してみましょう。これは、各アルゴリズム毎に、「数値を整列させるのに要する手間」を調べることを意味します。以下では、各整列アルゴリズムについて、データ量を変化させながら**時間計測**を行ないます。

まずは、整列させるための大量のデータが必要になりますが、次のコードにより、**乱数**を用いてランダムなデータ列を生成します:

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

def rand_array(n, r):
  a = ita.array.make1d(n)
  for i in range(len(a)):
    a[i] = int(r * random.random())
  return(a)
```

引数 n は生成するデータ列の要素数を、r は同じく乱数の種類を指定します。上のコードでは、n が r より大きいほど、同じ値が混ざり易くなって行きます。ちょっと試してみると、こんな感じ:

```
>>> rand_array(10, 1000)
[32, 24, 380, 507, 371, 643, 927, 531, 573, 160]
```

次に、時間計測の方法ですが、これは、以前 (#5, 演習 5-3) 素数を調べる際に利用したコードを利用できます。以下に、再掲しておきます:

```
import time
def measure_time(···):                     ← 所要時間を計りたい処理に応じた引数を用意
  start = time.process_time()              ← 開始時刻の取得
  ········                                 ← 所要時間を計りたい処理を、この間に挟む
  finish = time.process_time()             ← 終了時刻の取得
  print("%g" % (finish - start))           ← 両者の差 (所要時間) を秒単位で表示
```

前回 (#5) では詳しく述べませんでしたが、一般に、コンピュータで利用できる時計には、「現在の時刻を示す時計」と「CPU をどれだけ使ったかを示す時計」の二つがあります。コンピュータでは通常、非常に短い時間間隔で複数の処理を少しずつ切り替えて実行することにより、これらの処理を同時に実行しているように見せます。例えば、処理 P および Q が存在し、まずは P を少しだけ実行して (例えば、数ミリ秒)、次に Q を少しだけ実行して (その間は、P は停止)、また P を少しだけ実行して (その間は、Q は停止) という具合に、あたかも P と Q が同時に実行されているように見せているわけです。従って、P が時刻 t_i に始まり、t_{i+1} に終了したとしても、その間は P だけが実行されているわけではありません。これより、現在の時刻を示す時計しか存在しない場合は、実際に P だけが実行されている時間を計ることができないのです。よって、このような時間を計るために、P が CPU をどれだけ専有したかを示す時計が用意されています。上のプログラムで計測する時間は、time.process_time 関数で挟まれた処理が CPU をどれだけ専有したかを示す時間になっています。

以下に、これらの関数を利用し、例として単純選択法の実行時間を計測する measure_time 関数を示します:

```
import time

def measure_time(n, r):
  a = rand_array(n, r)
  start = time.process_time()
  selection_sort(a)    # これは単純選択法 (計測するアルゴリズムに応じて、関数を書き換えること)
  finish = time.process_time()
  print("%g" % (finish - start))
```

ちょっと使ってみると:

```
>>> measure_time(10000, 30000)
2.21875
```

単純選択法で 10,000 個のデータ列 (データの種類は 30,000 種類) を整列させてみると、2.22 秒ほど掛かることが分かります。

演習 10-1 処理が遅いと言われる整列アルゴリズム (単純選択法, 単純挿入法, バブル ソート) から一つを選び、同じく速いと言われる整列アルゴリズム (マージ ソート, クイック ソート) から一つを選び、それぞれデータ量を変えながら時間計測を行なうことで、データ量と所要時間の関係を分析せよ (グラフに描いて観察してみよう)。

演習 10-2 13 ページで示したクイック ソートのアルゴリズムには、実は非効率的な個所が存在する。この弱点を発見し、これを解消する工夫を実装せよ。