8. 整列と探索

Sort and Search

プログラミング・データサイエンス I

2024/5/30

1　今日の目的

今日の目的

- リストの整列
- リストの探索

前回までで、Python の基本を一通り学びました。そこで、今回は、アルゴリズムとデータ構造の関係の一部を見ていきます。アルゴリズムとは、問題解決の手順のことです。アルゴリズムには、それに適したデータ構造があります。

アルゴリズムとデータ構造の関係の例として、リストの要素を順番に並べ替える「整列」の方法を学ぶことにしましょう。また、リストから要素を探し出す「探索」も行います。

整列や探索は、Python に関数が用意されています。しかし、ここでは、整列や探索のプログラムを実際に作成し、その効率も調べましょう。

サンプルプログラムをダウンロードしてください。

`https://github.com/first-programming-saga/SortAndSearch`

2　整列: sort

2.1　泡立ち法

リストの中に、整数がでたらめな順序で入っているとしましょう。これを小さい順、あるいは大きい順に並べなおすことを整列 (sort) と云います。どんな方法が思い浮かびますか。

はじめに、泡立ち法 (bubble sort) という整列アルゴリズムを見てみましょう。その考え方は、以下のようなものです。

- 先頭から要素とその右隣の要素を比較し、右隣が小さい場合には、二つの要素を入れ替えます。
- これを右端まで繰り返すと、最も大きな要素が一番右に移動します。
- 再び、左端から同じ操作を繰り返します。すでに一番大きな要素が右端にありますから、右から 2 番目と 3 番目の比較までで十分です。
- この操作を、すべての要素が整列できるまで繰り返します。

Algorithm 1 泡立ち法

```
1: n は要素数
2: d_i は i 番目の要素
3: for j = n; j > 1; j−− do                 ▷ j は n から 2 まで 1 ずつ減る
4:     for i = 0; i < j − 1, i++ do          ▷ i は 0 から j − 2 まで 1 ずつ増える
5:         k = i + 1
6:         if d_i > d_k then                  ▷ 順序が逆の場合
7:             d_i と d_k を入れ替える
8:         end if
9:     end for
10: end for
```

泡立ち法をアルゴリズムとして整理しましょう (Algorithm 1)。要素の数を n、i 番目の要素を d_i とします。3 行目の `for` ループでは、ループインデクス j を n から初めて 2 まで、一つずつ減らします。この変数 j が、リストの右のどの位置まで、比較と入れ替えを行うかを定めています。4 行目の `for` ループでは、ループインデクス i を 0 から $j-2$ まで増やします。こちらは、先頭からの比較と入れ替えの処理に対応します。

図 1 を見ながら、データが小さい順に並べ代わる様子を見ていきましょう。はじめは $j = n = 7$ です。$i = 0$ から始めることで、左側から二つの要素を比較し、右側の要素が大きい場合に、入れ替えます。$i = 5$ まで行くと、$d_5 = 10$ と $d_6 = 3$ を比較します。この過程で、最大の要素が右端に移動します。

次に、j が一つ減って、$j = 6$ となります。j を減らす理由は、前述のように、最大の要素がすでに右端にあるためです。再び、左から二つの要素を比較して、必要な場合に要素入れ替えを繰り返すと、二番目に大きな要素が右から二番目に出てきます (図 2)。

それでは、要素の比較を何回行うかを考えましょう。$j = n$ の時の比較回数は $n-1$ 回です。$j = n-1$ の時の比較回数は、$n-2$ 回です。最後の比較回数は 1 回です。従って、

3	1	9	4	6	10	3
1	3	9	4	6	10	3
1	3	9	4	6	10	3
1	3	4	9	6	10	3
1	3	4	6	9	10	3
1	3	4	6	9	10	3
1	3	4	6	9	3	10

図 1　泡立ち法のイメージ。一番大きな要素が右端に移動した。

1	3	4	6	9	3	10
1	3	4	6	3	9	10
1	3	4	3	6	9	10
1	3	3	4	6	9	10

図 2　泡立ち法のイメージ。内側のループ終了時の配置

比較回数の総和 $C(n)$ は、1 から $n-1$ までの和です。つまり、以下のようになります。

$$C(n) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} \tag{2.1}$$

要素数 n が増えていくと、n が大きいときには $n^2 \gg n$ ですから、n の項は無視できて、n^2 に比例して比較の回数が増えることがわかります。これを以下のように表します。

$$C(n) \sim O(n^2) \tag{2.2}$$

泡立ち法でリストの要素を整列する python プログラムをソースコード 2.1 に示します。この関数は、リストの要素が一般のオブジェクトでも対応できるように定義しています。引数の `key` は、対象となるオブジェクトにおいて、何を比較して大小を決めるかを指定する関数です。指定しないと、オブジェクトそのもので大小を定めます。また、比較回数を数える `count` という変数を含んでいます。

ソースコード 2.1　泡立ち法

```
1  def bubbleSort(data_in:list, key:Callable=lambda x:x) -> tuple[list,
   ↪  int]:
2      data = list(data_in)
3      count:int = 0
4      for j in range(len(data), 1, -1):
5          for i in range(j - 1):
6              k = i + 1
7              count += 1
8              if key(data[i]) > key(data[k]):
9                  #i 番目と k 番目の要素を入れ替える
10                 data[i], data[k] = data[k], data[i]
11     return data, count
```

ソースコード 2.2　泡立ち法の実行する

```
1  n = 20
2  data_in = [(k, random.random()) for k in range(n)]
3  data, _ = bubbleSort(data_in, key:Callable = lambda x:x[1])
4  print(data)
```

整列リストを整列する場合について、ソースコード 2.2 に示します。リスト `data` の要素は、番号と $[0, 1)$ の範囲のでたらめな数のタプルです。`bubbleSort()` の `key` に指定しているのは、要素であるタプルの 1 番の要素を使って、大小関係を定めることを表しています。なお、3 行目で `bubbleSort()` の戻り値の二番目は使用しないために、`_`で表しています。もちろん、通常の変数に代入しても構いません。

2.2　クイックソート

泡立ち法では、要素の数 n に対して, 比較の回数が $O(n^2)$ で増えていきます。このことは、実行時間が $O(n^2)$ で長くなることを表しています。要素数が 10 倍になると、実行時間は 100 倍に、要素数が 100 倍になると実行時間は 10,000 倍になります。もっと速くできないでしょうか。できます。

もっと速く実行できるアルゴリズムの一つがクイックソート (quick sort) です。アルゴリズム 2 を見てください。始めに pivot という位置を定め、その要素の値を p とします。リストから p より小さい要素からなるリスト L、大きい要素からなるリスト R、同じ値か

Algorithm 2 クイックソート

```
1: procedure QuickSort(D)
2:     if |D| < 2 then
3:         return D
4:     end if
5:     pivot の値を p とする
6:     p より小さい値の要素からなるリスト L
7:     p より大きい値の要素からなるリスト R
8:     p と同じ値の要素からなるリスト C
9:     QuickSort(L)
10:    QuickSort(R)
11:    return L + C + R
12: end procedure
```

らなるリスト C を作ります。L と R を再帰的にクイックソートします。L と R が整列したら、間に C を挟んで、連結したリストを返します。再帰的に `QuickSort()` を呼び出していくと、いずれ引数のリストの長さは 1 以下となります。そのリストは整列する必要はありませんから、そのまま `return` で戻します。

3	1	9	4	6	10	3	4	2	5
3	1	4	3	4	2	5	9	6	10
1	2	3	4	3	4	5	9	6	10
1	2	3	3	4	4	5	6	9	10

図 3　クイックソート

次に、図 3 を見ながら整列されている要素を見ましょう。この例では、pivot はリストの最後の要素とします。図 3 の最初では、pivot の要素は 5 です。図の 2 段目では、5 より小さい要素と大きい要素に分割しています。5 より小さい要素のリストと大きい要素リストを独立したリストと考えて、それぞれに pivot を設定します。左側の pivot は 2、右側は 10 です。

5 より小さい要素のリストで、2 より小さい部分と大きい部分に分割します。5 より大

ソースコード 2.3　クイックソート

```
def quickSort(data:list, key:Callable=lambda x:x) -> tuple[list,int]:
    if len(data) < 2:
        return data, 0
    # pivot を選ぶ
    k:int = random.randint(0, len(data) - 1)
    p = data[k]

    left:list[Any] = list()# p より小さいの要素
    pivots:list[Any] = list()#P と同じ大きさの要素
    right:list[Any] = list()# p より大きい要素
    count:int = 0
    for v in data:
        count += 1
        if key(v) < key(p):
            left.append(v)
        elif key(v) == key(p):
            pivots.append(v)
        else:
            right.append(v)
    #再帰的にソートし、ソート済みのリストを連結して返す
    left, lc = quickSort(left, key)
    right, rc = quickSort(right, key)
    return left + pivots+right, lc + count + rc
```

きい要素リストも同様です。リストの長さが 1 になるまで繰り返します。

最初に、リスト中に要素がでたらめに並んでいれば、毎回、概ね半分にリストを分割するでしょう。そのため、比較の総数は以下のようになることが分かります。

$$C(n) \sim n \log_2 n \tag{2.3}$$

$\log_2 n$ は、底 2 の対数で、その整数部分は n を二進数で表現したときの桁数マイナス 1 です。従って、n に対して非常にゆっくりとしか増加しません。

一つ問題があります。リストが最初からほぼ小さい順に並んでいる場合です。その場合は、pivot を最後の要素とすると、pivot より大きい要素はほとんど存在せず、リストの分割は非常に非効率になります。そこで、pivot の位置をでたらめに選ぶことにします。実装をソースコード 2.3 に示します。関数のコメント部分を省略しています。このコードでも、比較の回数を数える変数 count があります。

図 4 に、データサイズ n に対する泡立ち法とクイックソートにおける比較の回数の変化を示します。縦軸横軸がそれぞれ対数で表される、両対数グラフになっていることに注

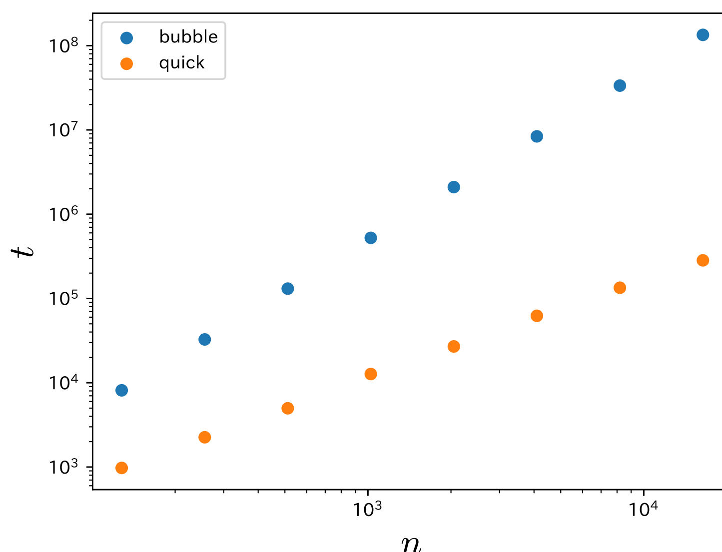

図 4　二つのソートにおける比較回数の要素数に対する変化

意してください。クイックソートでは、要素数が 2 桁増える間に、比較回数も 2 桁程度しか増えていません。一方、泡立ち法では、4 桁程度も増えています。

3　探索 (search)

3.1　二分木

次に、リストのような要素のコンテナ (container) に、目指す要素と同じものがあるかを判定することを考えましょう。要素をリストで保持している場合には、先頭から順に一致するかを判定することになります。一致する要素が無い場合、最後まで確認して、一致することが無いことを判定します。つまり、要素数 n に対して、$O(n)$ 回の比較が必要です。

二分木 (binary tree) というデータ構造を使うことで、もっと高速で検索することができます。図 5 を見て下さい。グラフの各頂点には (i, v) というラベルがついています。v という値が i 番目に、この二分木に追加されたことを表しています。

二分木は、各頂点が、一つの親 (上にある頂点) を持ち、二個以下の子 (下にある頂点) を持つ、木の形をしたグラフです。一番上の頂点を根 (root) と言います。根だけは、親を持ちません。

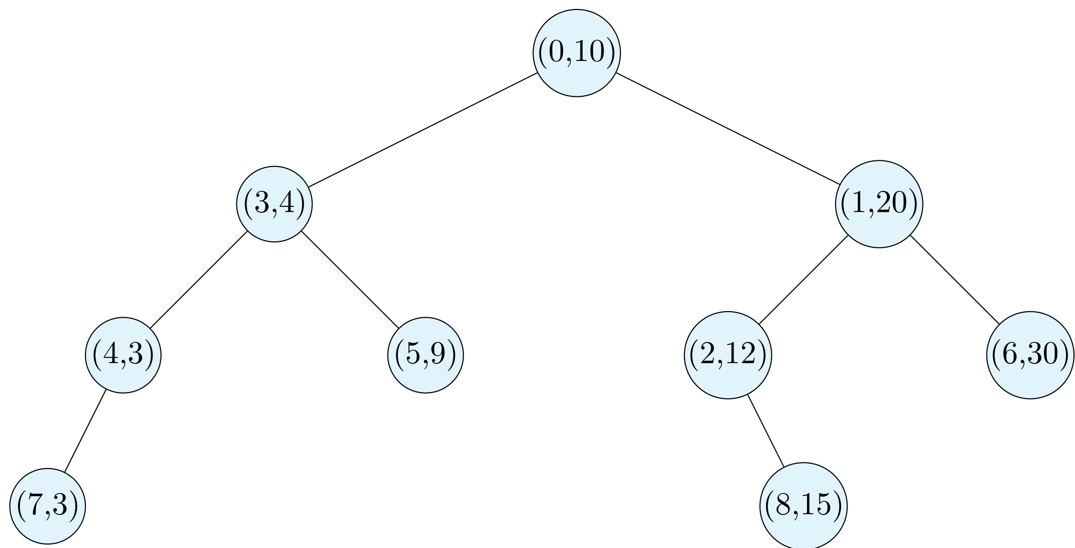

図 5　探索用二分木

図 5 では

```
[10,20,12,4,3,9,30,3,15]
```

というリストを順に、木に追加した結果です。

3.2　二分木の成長

図 6 を見ながら、探索用二分木が成長する様子を見ましょう。探索用二分木では、ある頂点にある値 v_{parent} に対して、左の頂点の値 $v_{\mathrm{left}} \leq v_{\mathrm{parent}}$、右の頂点の値 $v_{\mathrm{right}} > v_{\mathrm{parent}}$ となるように、要素を置くことにします。

はじめに一つだけ頂点 $(0, 10)$ があります。次の $(1, 20)$ は、値が 10 より大きいので、根の右側の子となります。その次の $(2, 12)$ は、値が 10 より大きいので右側の子を探し、20 より小さいので、頂点 $(1, 20)$ の左の子となります。4 番目の $(3, 4)$ は、値が 10 より小さいので、根の左側の子となります。

3.3　二分木を使った探索

それでは、二分木を使って探索をしましょう。図 5 において、値 $v = 12$ が含まれているかを探しましょう。はじめに、$10 < v$ であるため、根の右側を探します。次に頂点 $(1, 20)$ において、$20 > v$ であるため、その頂点の左側を探します。頂点 $(2, 12)$ に探索対

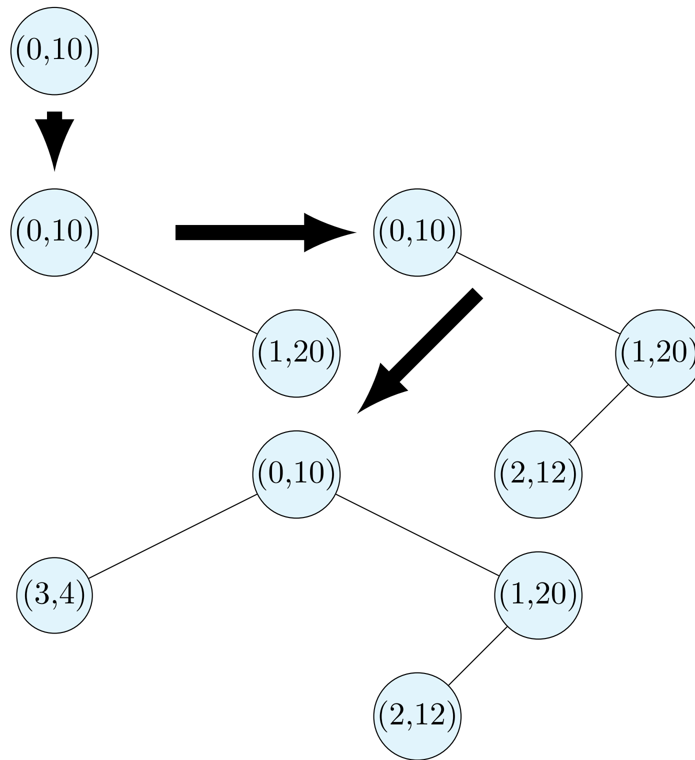

図 6　探索用二分木の成長

象を見つけました。

このように、根から各頂点が保持している値と探索対象を比較して、探索対象が頂点の値より大きければ右側へ、小さければ左へと探索していきます。データがでたらめな順番に入ってくれれば、根から一番下の頂点までの距離に大きなばらつきは無いと期待できます。そのため、根から一番下の頂点までの距離は $O(\log_2 n)$ となります。つなり、探索は $O(\log_2 n)$ で実行できます。

3.4　二分割探索

前述の二分木のなかで、要素は、ある意味で整列ずみであることが、高速な探索を可能としています。そこで、整列済みのリストから高速に探索する方法を、見ていきましょう。

二分割探索の考え方を説明します。リスト D の、ある範囲 $[l, r)$ に対象 t を探すことを考えます。中心の位置 $m = (l + r)/2$ として、$D_m = t$ であれば、m を返します。$D_m > t$ ならば、要素は m よりも左側に、$D_m < t$ ならば、要素は m よりも右側にあるはずです。その場合には、範囲を変更して、再帰的に探索します。手順を Algorithm 3 に示します。

Algorithm 3 二分割探索: D はリスト、t は探索対象

```
procedure BinarySearch(D, t, l, r)
    if l ≥ r then
        探索失敗
    end if
    if l = r − 1 then
        if D_l = t then
            return l
        end if
        探索失敗
    end if
    m = (l + r)/2
    if D_m = t then
        return m
    end if
    if D_m > t then                         ▷ 左側を探索
        return BinarySearch(D, t, l, m)
    end if
    return BinarySearch(D, t, m, r)         ▷ 右側を探索
end procedure
```

また、Python による実装をソースコード 3.1 に示します。

4 課題

`quiz.ipynb` を開き、選択ソートという別の方式を使って、要素数に対して比較の回数が増える様子を確認しなさい。

5 次回

エクセルファイルを python から操作する方法を扱います。

ソースコード 3.1　二分木探索の Python による実装

```python
def binarySearchSub(data:list, target, left:int, right:int,
↪   key:Callable=lambda x:x) -> tuple[int,Any]|None:
    """
    整列済みのリストから対象を探す、再帰的関数
    """
    if left >= right:
        return None
    if left == right - 1:
        if key(target) == key(data[left]):
            return left, data[left]
        return None
    m = (left + right) // 2
    if key(target) == key(data[m]):
        return m, data[m]
    if key(target) < key(data[m]):
        return binarySearchSub(data, target, left, m, key)
    return binarySearchSub(data, target, m, right, key)
```

付録 A　対数

対数関数 $y = \log_a x$ は、$a^y = x$ の逆関数です。その大きな特徴は、変数の積が対数の和になることです。a を底とした例を示します。

$$\log_a mn = \log_a m + \log_a n \quad \text{(付録 A.1)}$$

$$\log_a m^k = k \log_a m \quad \text{(付録 A.2)}$$

また、以下の性質があります。

$$\log_a 1 = 0 \quad \text{(付録 A.3)}$$

$$\log_a a = 1 \quad \text{(付録 A.4)}$$

対数の底として $a = 10$ としたものを、常用対数と呼びます。変数の積が対数の和になることから、以下のようなことがわかります。

$$\log_{10} 100 = \log_{10} 10 + \log_{10} 10 = 2 \quad \text{(付録 A.5)}$$

$$\log_{10} 10^n = n \log_{10} 10 = n \quad \text{(付録 A.6)}$$