② 線形探索と二分探索

もう少し別のアルゴリズムとして、リストの中に要素が含まれているかを判定することを考えてみます。

例題：リストからの探索

- リスト lst の中に、要素 x が含まれていれば True を、含まれていなければ False を返す関数 linear_search(lst, x) を定義しなさい
  - ※in演算子は使わずに書きましょう
  - これまでのPythonの知識があれば簡単に書けますよね？

以下のように書けましたか？

線形探索 Linear search

```
[4]: def linear_search(lst, x):
         for item in lst:
             if item == x:
                 return True
         return False
```

```
[6]: linear_search([1, 5, 4, 2], 4)
```

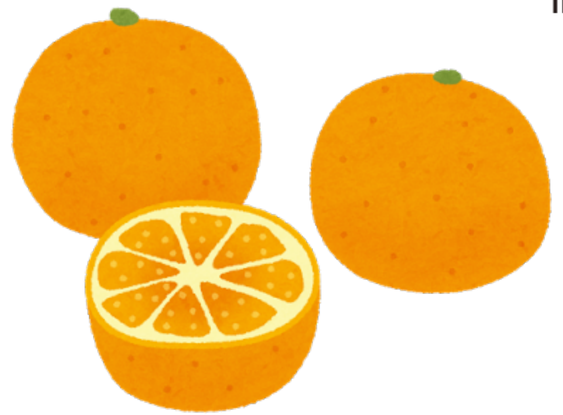
```
[6]: True
```

このように、リストからの値の検索をする際に、先頭から順にチェックする方法を「線形探索」と呼びます

例題：「ソートされたリスト」からの探索

- それでは、今度は予め昇順にソートされているリストから、要素を探索することを考えてみます
  - 昇順：値が小さい方から大きい方へ
  - 降順：値が大きい方から小さい方へ
- この場合も当然、線形探索で検索することは可能ですが、次のページに示すように、より効率的な探し方もあります

二分探索



- 調べる範囲を徐々に半分にしていきます
- 24を探してみると…

1. ソートされた数

5，8，15，22，22，24，26，32，42，69，71，81，92，96

2. 列中心にある数32と比べると、24は小さいので32より左の数に注目します

5，8，15，22，22，24，26，32，42，69，71，81，92，96

3. 列中心にある数22と比べると、24は22より大きいので22より右の数に注目します

5，8，15，22，22，24，26，32，42，69，71，81，92，96

4. 列中心にある数と比べると、24を発見しました → 3回の比較で発見しました！

他のリストでも試してみましょう

- 同じ手順で、以下のリストから 70 を探してみましょう
  - 対象とするリストの範囲が偶数の場合は、後半の先頭を「中心」とみなすこと

15, 23, 26, 41, 42, 45, 47, 55, 57, 64, 65, 67, 74, 78, 97, 99

15, 23, 26, 41, 42, 45, 47, 55, 57, 64, 65, 67, 74, 78, 97, 99

15, 23, 26, 41, 42, 45, 47, 55, 57, 64, 65, 67, 74, 78, 97, 99

15, 23, 26, 41, 42, 45, 47, 55, 57, 64, 65, 67, 74, 78, 97, 99

15, 23, 26, 41, 42, 45, 47, 55, 57, 64, 65, 67, 74, 78, 97, 99

→ 4回の比較で「無いこと」がわかりました

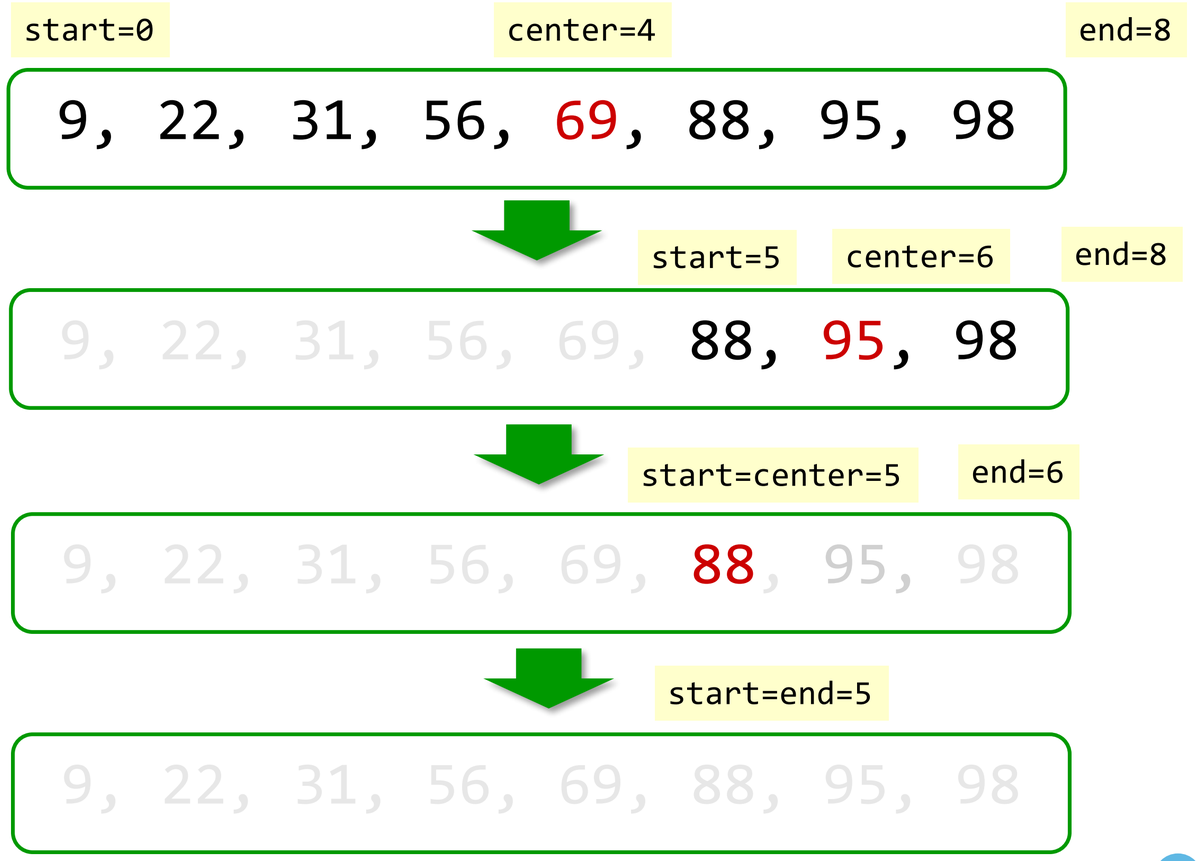
Python で記述すると…

- この手順を、Pythonで記述すると以下のようになります
  - 実際に「ソートされたリスト」から探索を試してみてください

二分探索 Binary search

```
[37]: def binary_search(lst, x):
          start = 0
          end = len(lst)
          while start < end:
              center = start + (end - start) // 2
              # print(lst[center])
              if lst[center] == x:
                  return True
              elif lst[center] > x:
                  end = center
              else:
                  start = center + 1
          return False
```

80を探す例だと…



線形探索 vs 二分探索

- 直感的に、二分探索の方が線形探索よりも効率的ですが、もう少しきちんと考えてみましょう
- 線形探索の場合…
  - 最悪の場合は、全部の要素をチェックしなければなりません
  - つまり、リストの長さに比例した時間がかかるということになります
- 二分探索の場合…
  - 毎回リストの長さが（概ね）半分になっていきますので、最悪の場合でも、概ね $\log_2(n)$ 回のチェックで済みます
  - つまり、リストの長さの対数に比例した時間がかかるということになります
- 長いリストになれば、二分探索の方が早く終わりそうですよね？

実際に時間を測ってみましょう

- randomモジュールを使って、0 以上 100000 未満の値からなる、長さ10のソートされたリストを作ってみます
  - random.sample 関数で、引数で与えた範囲から、パラメータkの数をランダムに選びます
  - sorted 関数は、昇順にソートします

[36]:
```
import random
```

[37]:
```
target = sorted(random.sample(range(100000), k=10))
target
```

[37]:
```
[3340, 12935, 37705, 47203, 48875, 61802, 67145, 85545, 90092, 93942]
```

実際に時間を測ってみましょう

- 線形探索と二分探索それぞれについて、実行時間を測ってみます
  - セルの先頭に、%%time を表示すると、実行時間が表示されます
- この例では、リストから100000を探索する処理を、10000回繰り返した時間を測っています
  - 1回だけだと、一瞬で終わってしまうためです
  - ※100000はリストに含まれていません

[ ]:
```
%%time

for _ in range(10000):
    linear_search(target, 100000)
```

[ ]:
```
%%time

for _ in range(10000):
    binary_search(target, 100000)
```

実際に時間を測ってみましょう

- リスト target の長さを、100, 1000, 10000 と変えて時間を測ってみましょう

```
[ ]: target = sorted(random.sample(range(100000), k=100))
```

```
[ ]: %%time

for _ in range(10000):
    linear_search(target, 100000)
```

```
[ ]: %%time

for _ in range(10000):
    binary_search(target, 100000)
```

リストが長くなるほど、二分探索の方が顕著に早く終わることが体感できるはずです！

まとめ

- ここでは、アルゴリズムの例として「線形探索」と「二分探索」を学習しました
  - 線形探索：リストの先頭から順にチェックする。リストの長さに比例する時間がかかる。
  - 二分探索：ソートされたリストに対する効率的な探索。リストの長さの対数に比例する時間がかかる。
- なお、このような計算時間の違いは、一般に次回学習する「オーダー記法」を用いて見積もります