教養としての
アルゴリズムとデータ構造
「探索」

小林浩二

kojikoba@mi.u-tokyo.ac.jp

授業の紹介

- 教養としてのアルゴリズムとデータ構造
  - アルゴリズムとは、「問題」を解くための手順
    - 我々の身の回りには様々な「問題」が存在する
    - 「問題」の例：
      - 数を昇順に並べ替える
      - データの集まりの中から、あるデータを探し出す
      - 長大な文字列の中から、特定の文字列を抜き出す
      - 地図上の2つの地点間の最短距離・経路を求める
      - 家系図の中から2者の関係を求める
      - コンビニで客をどのレジに並ぶか指示する
      - 1つのケーキを誰も不満を感じずに分け合う
      - 4人でタクシーに乗った時に1人が料金を立て替えた後、その料金を効率よく立て替えた人に支払う
  - この授業では、計算機上の「問題」を主に扱います

探索とは

- 探索とは、「データの集まり（集合）の中からあるデータを探し出す」こと
  - 探索の例：
    - 数の集合（例えば、Pythonならリスト、辞書、集合など）の中から特定の値（例えば、最小値、最大値、中間値など）を見つける
    - 文字列の中から特定の文字列を見つける

    など…
  - 「集合」：

    離散数学の用語。「もの」（数や文字など）の集まり。

    例：{3, 15, 37}、{a, b, x, y, z}、{1, 2, 3, …}
    - Pythonのデータ型である集合型とは別物

代表的な探索

- 線形探索(逐次探索、順探索)
- 二分探索
- ハッシュ

線形探索

- 線形探索：
  - 順序付けられたデータの集合に対して、最初から順に目的のデータを探す探索
    - （可変長ではない）順序付けられたデータの集合＝データ構造の用語的には配列と呼ぶ
      - Python上ではタプル・リストなどで表現
      - 先頭を0番目と呼びます

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
24	48	26	2	16	42	31	25	50	19	30	22	37	13	32

データを格納している配列

目的のデータ
（探したいデータ）　16

線形探索

- 線形探索：
  - 順序付けられたデータの集合に対して、最初から順に目的のデータを探す探索
    - （可変長ではない）順序付けられたデータの集合＝データ構造の用語的には配列と呼ぶ
      - Python上ではタプル・リストなどで表現
      - 先頭を0番目と呼びます

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
24	48	26	2	16	42	31	25	50	19	30	22	37	13	32

データを格納している配列

目的のデータ
（探したいデータ） 29

線形探索のアルゴリズム

- 線形探索：データ a を探索する（データ数は n ）
  - 1. i = 0
  - 2. i = n なら、a を発見できず （探索終了）
  - 3. 以下のいずれかを実行する
    - i 番目のデータが a の場合、発見 （探索終了）
    - i 番目のデータが a でない場合、i を1増加して2.へ戻る

線形探索の実装

- Pythonで実装する場合:
  - (データ構造)配列:リスト
  - (アルゴリズム)線形探索:演算子in
    - もしくはfor文+if文

```
list1 = [24, 48, 26, 2, 16, 42, 31, 25, 50, 19, 30, 22, 37, 13]
print(16 in list1)
print(29 in list1)
```

```
True
False
```

- 線形探索は性能はどのくらいなのか？
  - 実行にかかる時間は？
  - どれくらいの記憶領域を使う？

線形探索の性能評価

- アルゴリズムの性能は、「問題」を解くのに費やす時間で評価される
  - 線形探索の場合、配列の中から目的のデータを（もしくは、そのデータが配列の中に含まれないことを）発見するのにどれくらいの時間がかかるのか
- アルゴリズムの実行環境（PCなどの性能）に依存した評価は好ましくない
  - 例えば、スパコンの富岳と家庭用PC上で同じアルゴリズムを動かして、その実行時間を比較しても意味がない

線形探索の性能評価

- 一般に最悪計算量や平均計算量で評価する
  - 最悪計算量:
    - 「同じ規模」のあらゆる問題の中で、最も計算量がかかった場合を評価
    - 先の例だと、大きさ15のあらゆるリストとあらゆる目的のデータの組み合わせの中で最も計算量が大きくなるものを考える

  - データ数が n 個の場合:
    - n番目まで調べるので、計算量は n　→ O(n) と表す
      - n番目に目的のデータがある(データが見つからない)

用語解説

- O記法:
  - O(n) と書いて、nのオーダー（もしくは、オーダー n ）などと読む
  - 「 n に比例して大きくなる数」という様な意味
  - Oを用いる場合は、定数の係数を無視し、最も大きい次数の項のみで書くのが一般的
    - 例えば、n^2, $3n^2$, $100n^2$, $3n^2+2n+500$などは全て$O(n^2)$
  - 厳密には…

ある正定数 c と n’ が存在して、任意の $n \geqq n'$ に対して、$f(n) \leq c\, g(n)$ が成立するとき、$f(n) = O(g(n))$ という

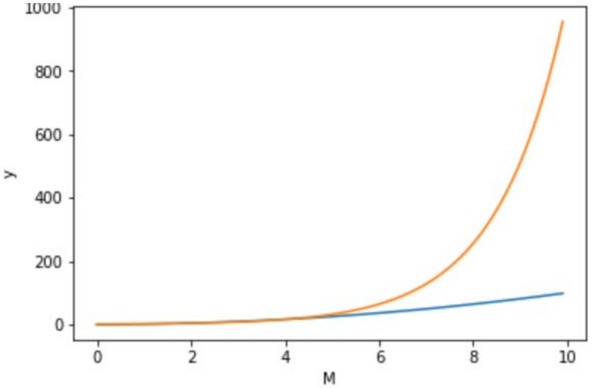
用語解説

- 計算量とO記法:
  - アルゴリズムとデータ構造の業界では計算量について次の様な世界観をもっています。
    - O(1) : 超いいね！
    - O(n) : いいね！
    - $O(n^2)$, $O(n^3)$, … : まあまあいいね！
      - ただし実用上はよろしくない
    - $O(2^n)$: あかん…

…定数時間

…多項式時間

…指数時間



線形探索の性能評価

- 一般に最悪計算量や平均計算量で評価する
  - 最悪計算量:
    - 「同じ規模」のあらゆる問題の中で、最も計算量がかかった場合を評価
    - 先の例だと、大きさ15のあらゆるリストとあらゆる目的のデータの組み合わせの中で最も計算量が大きくなるものを考える
  - データ数が n 個の場合:
    - n番目まで調べるので、計算量は n　→ O(n) と表す
      - n番目に目的のデータがある（データが見つからない）

線形探索の性能評価

- 一般に最悪計算量や平均計算量で評価する
  - 平均計算量:
    - 異なる入力に対して複数回問題を解いたときの平均的な計算量を評価
    - 下記の例だと、大きさ15のあらゆるリストに対して、リストに含まれる値に対する照会（探索）が等確率で行われる場合を考える
      - ただし、目的のデータは配列のどこかに含まれていると考える
  - データ数が n 個の場合:
    - 全てのデータに対する探索が等確率(1/n)で行われる
      - i 番目のデータまで調べると計算量はO(i)
    - 計算量は$1 \times 1/n + 2 \times 1/n + \ldots + n/n = (n+1)/2 \rightarrow O(n)$

線形探索の時間計算量

- データ数が n 個の場合:
  - 最悪計算量:
    - n番目まで調べるので、計算量はn → O(n)
      - n番目に目的のデータがある/データが見つからない
  - 平均計算量:
    - 全てのデータに対する探索が等確率(1/n)で行われるとする
      - i 番目のデータまで調べると計算量はO(i)
    - 計算量は1×1/n+2×1/n+…+n/n = (n+1)/2 → O(n)
  - 計算量を軽減する(高速化する)には?
    - データを小さい順に並べ替えておく

線形探索

- 線形探索：
  - データを昇順に並べ替えておくと（ソートしておくと）、途中で探索を打ち切ることが出来る
  - 最悪計算量・平均計算量は変化しない
    - 実用的には多少高速化が見込める

→より高速な計算量$O(\log_2 n)$の探索が存在します

2	13	16	19	22	24	25	26	30	31	32	37	42	48	50

データを格納している配列

目的のデータ(29)より大きい値になったので、
29はリストには含まれないことが分かる

目的のデータ
（探したいデータ）

29

代表的な探索

- 線形探索（逐次探索、順探索）
- 二分探索
- ハッシュ

二分探索

- 二分探索：
  - ソート済みの順序付けられたデータの集合に対する探索
  - 真ん中のデータ（データ数が偶数なら(n-2)/2 番目のデータ、奇数なら(n-1)/2番目のデータ）と目的のデータを比較する
    - 例では、データが15個なので7番目の値（26）と比較
    - 26 < 32 なので、32がデータとして存在する場合、8番目から14番目の間に存在する

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	13	16	19	22	24	25	26	30	31	32	37	42	48	50

データを格納している配列

目的のデータ
（探したいデータ） 32

二分探索

- 二分探索：
  - ソート済みの順序付けられたデータの集合に対する探索
  - 残ったデータの中で真ん中のデータと目的のデータを比較する
    - 例の場合、8番目から14番目の中で4番目（つまり、11番目の値=37）と比較
    - 37 > 32 なので、32がデータとして存在する場合、8番目から10番目の間に存在する

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	13	16	19	22	24	25	26	30	31	32	37	42	48	50

データを格納している配列

目的のデータ
（探したいデータ） 32

二分探索

- 二分探索:
  - ソート済みの順序付けられたデータの集合に対する探索
  - 残ったデータの中で真ん中のデータと目的のデータを比較する
    - 例の場合、8番目から10番目の中で2番目（つまり、9番目の値= 31）と比較
    - 31 < 32 なので、32がデータとして存在する場合、10番目に存在する

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	13	16	19	22	24	25	26	30	31	32	37	42	48	50

データを格納している配列

目的のデータ
（探したいデータ） 32

二分探索

- 二分探索：
  - ソート済みの順序付けられたデータの集合に対する探索
  - 残ったデータの中で真ん中のデータと目的のデータを比較する
    - 例の場合、10番目と比較
    - 10番目が32なので、探していたデータを発見して探索を終了

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	13	16	19	22	24	25	26	30	31	32	37	42	48	50

データを格納している配列

目的のデータ
（探したいデータ） 32

二分探索のアルゴリズム

- 二分探索:データ a を探索する(データ数は n)
  - 1. Left = 0, Right = n-1
  - 2. Left > Rightなら、a を発見できず （探索終了）
  - 3. Mid = [(Left+Right)/2]　([x] はx以下の最大の整数)
    - Mid番目のデータが a の場合、発見 （探索終了）
    - Mid番目のデータ < a の場合、Left = Mid+1として 2.へ戻る
    - Mid番目のデータ > a の場合、Right = Mid-1として 2.へ戻る

二分探索の実装

- Pythonで実装する場合:
  - （データ構造）配列：リスト
  - （アルゴリズム）二分探索：モジュールbisect
    - 使い方はnote_bisect.ipynbを参照のこと

二分探索の計算量

- データ数が n 個の場合:
  - 最悪計算量:
    - 探索対象となるデータの数が n → n/2 → n/4 → n/8 → … → $n/2^{k-1}$ → $n/2^k = 1$ となり、ほぼk回の操作で探索を終える
    - $O(k) = O(\log_2 n)$
      - $n/2^k = 1$ より、$k = \log_2 n$ が成立
  - 平均計算量:
    - 全てのデータを等確率(1/n)で探索する場合
    - i 回目の探索で発見可能なデータの数は 2^{i-1} 個＝（$1 \times 1 + 2 \times 2 + 3 \times 2^2 + \ldots + k \times 2^{k-1}$）$/n \leq k\,2^k / n = \log_2 n$ となるので、$O(\log_2 n)$
      - $1 \times 1 + 2 \times 2 + 3 \times 2^2 + \ldots + k \times 2^{k-1} = k\,2^k - 2^k + 1$

代表的な探索

- 線形探索（逐次探索、順探索）
- 二分探索
- ハッシュ（法）

ハッシュ

- ハッシュ（法）:
  - 順序付けられていないデータの集合に対して、目的のデータを探す探索に用いるデータ構造
    - 順序付けられていないデータの集合は、Python上では集合（セット）・辞書で表現

ハッシュ

- ハッシュ：
  - M = 実際に同時に使われる可能性のあるデータの数
  - 大きさ M の連続した記憶領域を確保
    - この記憶領域をハッシュ表という
      - ハッシュ表に格納（登録）するデータをキーとも呼ぶ
    - 例：M=19、データ：24, 48, 26, 2, 16, 42, 31, 25

ハッシュ表

0	1	2	3	4	5	6	7	8	9
		2		42	24	25	26		

10	11	12	13	14	15	16	17	18
48		31				16		

ハッシュ

- ハッシュ：
  - データ a はハッシュ表の「 a のハッシュ値」番目に格納（登録）する
    - ハッシュ値とは、データ a を与えると、0 から M-1 のいずれかの整数を返す関数の値のこと
    - この関数をハッシュ関数と呼ぶ
    - 例：ハッシュ関数＝キーの値を19で割った余りを求める関数

ハッシュ表

0	1	2	3	4	5	6	7	8	9
		2		42	24	25	26		

10	11	12	13	14	15	16	17	18
48		31				16		

ハッシュ

- ハッシュ表に登録するデータが全て異なるハッシュ値を取る様にし、「ハッシュ値を計算→ハッシュ表を参照」を実行して各データを利用(探索)します
  - 例：M=19の例
    - キー：24, 48, 26, 2, 16, 42, 31, 25
    - ハッシュ関数＝キーの値を19で割った余り
    - キー48を探索する場合、48のハッシュ値＝48%19=10 を計算
      →ハッシュ表の10番目の値を調べる
      →48を発見

ハッシュ表

0	1	2	3	4	5	6	7	8	9
		2		42	24	25	26		

10	11	12	13	14	15	16	17	18
48		31				16		

ハッシュ

- ハッシュ表に登録するデータが全て異なるハッシュ値を取る様にし、「ハッシュ値を計算→ハッシュ表を参照」を実行して各データを利用（探索）します
  - 例：M=19の例
    - キー：24, 48, 26, 2, 16, 42, 31, 25
    - ハッシュ関数＝キーの値を19で割った余り
    - キー49を探索する場合、49のハッシュ値＝49%19=11 を計算
      →ハッシュ表の11番目の値を調べる
      →空なので49は存在しない

ハッシュ表

0	1	2	3	4	5	6	7	8	9
		2		42	24	25	26		

10	11	12	13	14	15	16	17	18
48		31				16		

ハッシュ

- ハッシュ表に登録するデータが全て異なるハッシュ値を取る様にし、「ハッシュ値を計算→ハッシュ表を参照」を実行して各データを利用（探索）します
  - 例：M=19の例
    - キー：24, 48, 26, 2, 16, 42, 31, 25
    - ハッシュ関数＝キーの値を19で割った余り
    - キー50を探索する場合、50のハッシュ値＝50%19=12 を計算
      →ハッシュ表の12番目の値を調べる
      →値を発見したが50ではなかった　→50はどこに登録するの？？

ハッシュ表

0	1	2	3	4	5	6	7	8	9
		2		42	24	25	26		

10	11	12	13	14	15	16	17	18
48		31				16		

ハッシュ値の衝突

- 登録しているデータが増える程、異なるデータで同じハッシュ値を取るものが現れる（衝突する）可能性がある
  - 例では、ハッシュ表の a 番目の領域に格納する値は、a, a+19, a+38, …, と無数に衝突が発生し得る
  - 通常、衝突をできるだけ回避する為に十分に大きいMと巧妙なハッシュ関数を用意する
  - Pythonでは衝突回避の為に、登録するデータが増えると自動的にハッシュ表を大きくして、ハッシュ関数をそれに合わせて変更します

ハッシュの実装

- Pythonで実装する場合:
  - (データ構造)ハッシュ表:集合(セット)、辞書
    - 集合の使い方:https://utokyo-ipp.github.io/appendix/2-set.html
    - ただし、辞書はキー(dic.keys())の扱いがハッシュ
  - (アルゴリズム)ハッシュ:演算子in

```
set1 = {24, 48, 26, 2, 16, 42, 31, 25}
print(48 in set1)
print(49 in set1)
```

```
True
False
```

```
# キーに対応する値は何でも良い (Trueでなくて構わない)
dic1 = {24:True, 48:True, 26:True, 2:True, 16:True, 42:True, 31:True, 25:True}
print(48 in dic1)
print(49 in dic1)
```

```
True
False
```

ハッシュの計算量

- データ数が n 個の場合:
  - 最悪計算量・平均計算量:
    - 「ハッシュ値を計算→ハッシュ表を参照」＝O(1)

線形探索 vs. 二分探索

- n 個のデータの集合に対する探索:
  - 二分探索はデータ集合のソートが必須
    - ソートは$O(n \log_2 n)$の（最悪）計算量が必要
  - 事前にデータ集合がソートされている場合:
    - 二分探索: $O(\log_2 n)$ < 線形探索: $O(n)$
  - 事前にデータ集合がソートされていない場合:
    - ソートにかかる時間、探索回数に依存
    - 同じデータ集合に対して k 回探索を行う:
      - 二分探索:ソート+k回探索=$O(n \log_2 n + k \log_2 n)$
      - 線形探索:k回探索=$O(kn)$
      - k がかなり小さい ($k \ll n$):線形:$O(kn)$ < 二分:$O(n \log_2 n)$
      - k が大きい ($k > n$):線形:$O(kn) > O(n^2)$ > 二分:$O(k \log_2 n)$

線形･二分探索 vs. ハッシュ

- n 個のデータの集合に対するk回の探索：
  - ハッシュはハッシュ表の作成が必須
    - n 個のデータから作成する場合O(n)の計算量が必要
  - 事前にハッシュ表が作成されている場合：
    - ハッシュ：O(k) vs.
      - データがソート済み： 二分：O ($k \log_2 n$) vs. 線形：O(kn)
      - データが未ソート：二分：O ($n \log_2 n + k \log_2 n$) vs. 線形：O(kn)
  - 事前にハッシュ表が作成されていない場合：
    - ハッシュ：O(n+k) vs.
      - データがソート済み： 二分：O ($k \log_2 n$) vs. 線形：O(kn)
      - データが未ソート：二分：O ($n \log_2 n + k \log_2 n$) vs. 線形：O(kn)

線形･二分探索 vs. ハッシュ

- 範囲探索:
  - 2つのデータAとBを指定して、配列内のAとBの間に存在するデータを全て取り出す探索
  - 線形探索･二分探索は配列に対する探索なので、範囲探索を実行可能
    - ソートが必要な場合は二分探索は実行不可
  - ハッシュは範囲探索ができない
    - データの値に基づいて順序付けて管理していない
    - ハッシュ表内のAとBの間のデータを取り出しても意味がない

0	1	2	3	4	5	6	7	8	9
		2		42	24	25	26		

10	11	12	13	14	15	16	17	18
48		31				16		

線形・二分探索 vs. ハッシュ

- 領域計算量:
  - 計算機上で何らかの処理(つまり、計算)を行う際に、処理中のある時点で使用している記憶領域の最大量
    - 処理を行うのに最低限必要となる記憶領域の量
    - 処理を行うために使用した記憶領域の総量ではない
  - 線形探索・二分探索:O(n)
    - n 個のデータを格納する配列
  - ハッシュ:O(M)
    - M = 格納される可能性のあるデータ数 (M>n)

```
datasize = 1000000
list1 = list(range(datasize))
set1 = set(range(datasize))
print("リストの使用メモリ（バイト）:", total_size(list1), ", 集合の使用メモリ（バイト）:", total_size(set1))
```

```
リストの使用メモリ（バイト）: 37000108 , 集合の使用メモリ（バイト）: 61554652
```

ハッシュ値の衝突

- 登録しているデータが増える程、異なるデータで同じハッシュ値を取るものが現れる（衝突する）可能性がある
  - 例では、ハッシュ表の a 番目の領域に格納する値は、a, a+19, a+38, …, と無数に衝突が発生し得る
  - 通常、衝突をできるだけ回避する為に十分に大きいMと巧妙なハッシュ関数を用意する
  - Pythonでは衝突回避の為に、登録するデータが増えると自動的にハッシュ表を大きくして、ハッシュ関数をそれに合わせて変更します

```python
import sys
set1 = set()
for i in range(30):
    print("要素数:", len(set1), "使用メモリ（バイト）:", sys.getsizeof(set1))
    set1.add(i)
```

```
要素数: 0 使用メモリ（バイト）: 224
要素数: 1 使用メモリ（バイト）: 224
要素数: 2 使用メモリ（バイト）: 224
要素数: 3 使用メモリ（バイト）: 224
要素数: 4 使用メモリ（バイト）: 224
```

ハッシュ表再構成

```
要素数: 5 使用メモリ（バイト）: 736
要素数: 6 使用メモリ（バイト）: 736
要素数: 7 使用メモリ（バイト）: 736
要素数: 8 使用メモリ（バイト）: 736
要素数: 9 使用メモリ（バイト）: 736
要素数: 10 使用メモリ（バイト）: 736
要素数: 11 使用メモリ（バイト）: 736
要素数: 12 使用メモリ（バイト）: 736
要素数: 13 使用メモリ（バイト）: 736
要素数: 14 使用メモリ（バイト）: 736
要素数: 15 使用メモリ（バイト）: 736
要素数: 16 使用メモリ（バイト）: 736
要素数: 17 使用メモリ（バイト）: 736
要素数: 18 使用メモリ（バイト）: 736
```

ハッシュ表再構成

```
要素数: 19 使用メモリ（バイト）: 2272
要素数: 20 使用メモリ（バイト）: 2272
要素数: 21 使用メモリ（バイト）: 2272
要素数: 22 使用メモリ（バイト）: 2272
要素数: 23 使用メモリ（バイト）: 2272
要素数: 24 使用メモリ（バイト）: 2272
要素数: 25 使用メモリ（バイト）: 2272
要素数: 26 使用メモリ（バイト）: 2272
要素数: 27 使用メモリ（バイト）: 2272
要素数: 28 使用メモリ（バイト）: 2272
要素数: 29 使用メモリ（バイト）: 2272
```

課題

- 探索を体験しましょう
  - note_bisect.ipynb
    - （必修）二分探索が実行できるモジュール bisect について
  - note_complexity.ipynb
    - Pythonの基本操作の計算量に関するメモ
      - 興味のある方向け。必修ではありません
  - 集合型の使い方が分からない方
    - addとinだけ知っておいて下さい
    - https://utokyo-ipp.github.io/appendix/2-set.html
  - ex2.ipynb の前半
    - 基礎課題（18日締め切り）
  - ex2.ipynb の後半
    - 本課題（23日締め切り）