コンピュータサイエンス第２
ソート（整列）アルゴリズム：オーダー

南出 靖彦

第 5 回

準備: Windows

- コマンドプロンプトを実行，CS2 フォルダに移動.
  - `cd Documents`
  - `cd CS2`
- 講義のウェブページから `day5.zip` をダウンロードする.
  - Downloads（ダウンロード）フォルダに `day5.zip` を置かれる
  - ファイルを開く → すべてを展開
  - 展開先として，Documents¥CS2 を指定して，展開
- day5 フォルダに移動
  - `cd day5`

準備: Mac

- CS2 フォルダに移動.
  - `cd Documents`
  - `cd CS2`
- 講義のウェブページから `day5.zip` をダウンロードする.
- Terminal で day5 を CS2 に移動
  - `mv ~/Downloads/day5 ./`
  - `cd day5`

アルゴリズム（の実行時間）のオーダー

オーダー: 「十分大きな n に対して実行時間の増加がどのような関数の定数倍に比例するか，あるいはどのような関数の定数倍で抑えられるか」

- 入力が大きい時のアルゴリズムの振る舞い

実験をした四つのソートアルゴリズム

- データの大小比較をしてその結果に応じて適宜値の交換などを行う

データの比較回数が全体の実行時間を支配している最大要因になる。

$$\text{実行時間のオーダー} \approx \text{比較回数のオーダー}$$

各プログラム（アルゴリズム）について入力列の長さ n とした比較回数のオーダーを解析する。

定義：オーダー

関数 $f(n)$ のオーダーが $O(g(n))$ である

$\Leftrightarrow$ 定数 c と n_0 が存在して，すべての $n \geq n_0$ に対して

$$f(n) \leq cg(n)$$

例：

- 定数倍は無視される：$100n^2$ はオーダー $O(n^2)$ である
- 次数の低い項は無視される：$n^2 + n + 1$ はオーダー $O(n^2)$ である

$$n^2 + n + 1 \leq 3n^2 \quad (n \geq 1)$$

- $O(g(n))$ は，集合とも思える

$$O(g(n)) = \{f(n) \mid c \text{ と } n_0 \text{ が存在して，すべての } n \geq n_0 \text{ に対して } f(n) \leq cg(n)\}$$

- オーダーの大小

$$O(\log(n)) \subseteq O(n) \subseteq O(n\log(n)) \subseteq O(n^2) \subseteq \cdots \subseteq O(2^n)$$

  - アルゴリズムの計算時間のオーダー：小さい方が良い
    - 大きい入力に対する計算時間が短い

バブルソート

比較回数は

$$(n-1)+(n-2)+\cdots+1 \;=\; \frac{n(n-1)}{2}$$

で $O(n^2)$ である。

- 比較回数は長さのみに依存して，並び方（乱数・正順・逆順）による差はない。
- 交換回数は並び方に依存するので，プログラムの実行時間は同じ長さでも並び方で差がでる。

挿入ソート

挿入ソートは入力数列の並び方によって比較回数が変ってくる。このようなアルゴリズムに対しては，以下の場合のオーダを解析する．

- 最悪の場合
- 平均の場合

最悪の場合：「長さ n の数列のうちで，挿入ソートが最も苦手とする（最も比較回数が多い）数列を与えたときの比較回数」

平均の場合を考えるには，どんな列全体の中での平均か？　を定める必要があるが，

- 「値が異なる n 個の整数の $n!$ 通りの順列がすべて等確率で入力されると考えて挿入ソートを実行したときの平均比較回数」

実験する際には乱数列が平均の場合の代表と考えてよいが，ばらつきはある．

挿入ソート： 最悪

例えば：入力が逆順列の場合
挿入ソートで入力のサイズが n で最悪の場合の比較回数は

$$1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$$

であり（バブルソートと同じ），

$O(n^2)$ である。

挿入ソート：平均

すでに整列している $\langle a_1, \ldots, a_{i-1} \rangle$ に対して a_i の挿入場所を探すのに必要な比較回数

- 最小 1 回，最大 $(i-1)$ 回
- 平均では最大値のほぼ半分と考えてよい

トータルの比較回数は最悪の場合の約半分であり，
オーダーは最悪の場合と同じ $O(n^2)$ である。

なお正順列の場合は比較回数は $n-1$ であり，非常に少ない。

マージソート

簡単のために $n = 1, 2, 4, 8, \ldots, 2^x, 2^{x+1}, \ldots$ の場合だけを考える。

$\mathrm{merge}(2^x)$: 長さ 2^x の（ある）列をマージソートするのに必要な比較回数の総数

- 再帰呼び出しの中の比較の回数も含めた数
- 数列の長さ $n = 2^c$ が固定されていても並び方に依存して $\mathrm{merge}(2^c)$ の値は変動する。

どんな並び方であっても次の不等式が成り立つ。

$$2^{x-1} \cdot x \leq \mathrm{merge}(2^x) \leq 2^x \cdot x \tag{1}$$

これを x に関する数学的帰納法で証明する。

- （$x = 0$ の場合）$\mathrm{merge}(2^0) = \mathrm{merge}(1) = 0$ で (**??**) 式は成り立つ。

- （$x = k$ で成り立つと仮定して $x = k + 1$ の場合を示す）長さ 2^k の整列された２本の列をマージして長さ 2^{k+1} の列を作るのに必要な比較回数は
  - 最小 2^k 回，最大 $2^{k+1} - 1$ 回

したがって次が成り立つ。

$$\underbrace{2^k}_{\text{マージに必要な比較回数}} \overbrace{+ \operatorname{merge}(2^k) + \operatorname{merge}(2^k)}^{\text{再帰呼び出し中で必要な比較回数}} \quad \leq \quad \operatorname{merge}(2^{k+1})$$

$$\operatorname{merge}(2^{k+1}) \quad \leq \quad \underbrace{2^{k+1} - 1}_{\text{マージに必要な比較回数}} \overbrace{+ \operatorname{merge}(2^k) + \operatorname{merge}(2^k)}^{\text{再帰呼び出し中で必要な比較回数}}$$

この右側の不等式を用いると次が得られる。

$$\begin{aligned}
\operatorname{merge}(2^{k+1}) &\leq 2^{k+1} - 1 + \operatorname{merge}(2^k) + \operatorname{merge}(2^k) \\
&\leq 2^{k+1} - 1 + 2^k \cdot k + 2^k \cdot k \qquad \text{(帰納法の仮定より)} \\
&\leq 2^{k+1} + 2^k \cdot k + 2^k \cdot k \\
&= 2^{k+1}(k+1)
\end{aligned}$$

マージソート：オーダ

$$2^{x-1} \cdot x \leq \mathrm{merge}(2^x) \leq 2^x \cdot x$$

より

$$\frac{1}{2} n \log_2 n \leq \mathrm{merge}(n) \leq n \log_2 n$$

- 長さ n の数列をマージソートするのに必要な比較回数 $\mathrm{merge}(n)$ は $n \log_2 n$ の $\frac{1}{2}$ 倍から 1 倍の範囲

平均・最悪ともに $O(n \log n)$

- オーダーは定数倍に関してはどうでもよいので，log の底は書かなくてよい

$$\log_a n = \frac{\log_b n}{\log_b a}$$

クイックソート：平均

$\mathrm{q}(n)$: サイズ n で呼び出されたときに，必要な比較回数の総数

基準値によって分割される左右の長さをそれぞれ L, R とすると，(L, R) がとりうる値の組は

$$(0, n-1),\ (1, n-2),\ \ldots,\ (n-1, 0)$$

の n 通りである

- これらが起こる可能性はすべて等しいと考えてよい。

したがって，

$$\mathrm{q}(n) = \underbrace{(n-1)}_{\text{分割に必要な比較回数}} + \overbrace{\frac{1}{n}\Big((\mathrm{q}(0)+\mathrm{q}(n-1)) + (\mathrm{q}(1)+\mathrm{q}(n-2)) + \cdots + (\mathrm{q}(n-1)+\mathrm{q}(0))\Big)}^{\text{再帰呼び出し中で必要な比較回数}}$$

$$= n - 1 + \frac{2}{n}\sum_{i=0}^{n-1}\mathrm{q}(i) \quad (n \geq 2 \text{ のとき}),$$

$$\mathrm{q}(0) = \mathrm{q}(1) = 0$$

となる。

この漸化式を解くと（解き方は省略）

$$\mathrm{q}(n) = 2(n+1)\sum_{i=1}^{n}(\frac{1}{i+1}) - 2n$$

を得る。

- $\sum_{i=1}^{n}(\frac{1}{i+1})$ はオーダーの観点では $\log_e n$ と等しい
  - $\sum_{i=1}^{n}(\frac{1}{i+1}) \approx \frac{1}{n+1}$ の積分
- $\mathrm{q}(n)$ は $\mathrm{O}(n \log n)$

クイックソート：最悪

常に最小値か最大値のどちらかが基準値に選ばれて分割が偏ってしまう場合である（厳密にはこのことの証明も必要である）。
この場合のトータルの比較回数は

$$(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}$$

であり（バブルソート，挿入ソートの最悪時と同じ），$O(n^2)$ である。

クイックソート：最悪（続き）

実験で使用したプログラムは左端の値を基準値にしている.

- 正順列の分割結果 ：
  - $\vec{b}$: 空
  - $\vec{c}$: 長さ $n-1$, 正順列
- 逆順列の分割結果：
  - $\vec{b}$: 長さ $n-1$, 逆順列
  - $\vec{c}$: 空

よって正順や逆順の場合には比較回数が最悪になる。

プログラム実行時には再帰呼び出しの深さが $(n-1)$ になり，n が大きいと再帰呼び出しのために記憶する情報が多くなりすぎて実行が途中で打ち切られる。

各アルゴリズムのまとめ

	平均	最悪	備考
バブルソート	$O(n^2)$	$O(n^2)$	いつでも $\frac{n(n-1)}{2}$ 回。
挿入ソート	$O(n^2)$	$O(n^2)$	平均は最悪の半分程度。最良は $n-1$ 回。
マージソート	$O(n \log n)$	$O(n \log n)$	比較回数は安定して少ない。
クイックソート	$O(n \log n)$	$O(n^2)$	比較回数はマージソートより多い。

クイックソートは，平均的な場合に速いか？

- 頑張ったプログラムでは
  - 入力配列以外の配列を使用しない
  - プログラムがシンプルにできる

  のでマージソートより実行時間は短いことが多い。（配列が大きい場合）

```
>>> test_sort(quick_sort, random_array, 1000000)
実行時間 5.974787 秒
比較回数 25628489 回
>>> test_sort(merge_sort, random_array, 1000000)
実行時間 5.443224 秒
比較回数 18672653 回
>>> test_sort(quick_sort2, random_array, 1000000)
実行時間 4.844584 秒
比較回数 24496544 回
```

ソートアルゴリズム：オーダの下界

アルゴリズムを工夫すれば $O(n \log n)$ より小さいオーダーが達成できるか？

実は「それはできない」

- データ同士の比較に基づくソートアルゴリズムのオーダーは $O(n \log n)$ が最小であることが知られている。

アルゴリズムの設計手法

- 分割統治法（Devide & Conquer)：マージソート，クイックソート
  1. 問題をいくつかの子問題に分割
  2. 子問題を再帰的に解く
  3. 子問題の解を統合して全体の解を得る（統治）

- 動的計画法（DP: Dynamic Programming)
  1. 問題を部分問題に分割
  2. 必要な部分問題の計算結果を記録して，複数回計算するのを防ぐ

フィボナッチ数

$fib(n)$: n 番目のフィボナッチ数

$$\begin{aligned} fib(0) &= 0 \\ fib(1) &= 1 \\ fib(n+2) &= fib(n+1) + fib(n) \quad (n \geq 0) \end{aligned}$$

単純な再帰による計算法

- $fib(k)$ を何度も計算

$$fib(5) = fib(4) + fib(3) = (fib(3) + fib(2)) + fib(3) = \cdots$$

$\Rightarrow$ 指数関数時間

動的計画法：フィボナッチ数

$fib(n)$ の計算

- $fib(0), fib(1), \ldots, fib(n-1)$ を全て計算する必要がある.
- 小さい方から計算すれば良い.

```
def fib2(n):
    a = [0] * (n+1)  # a[i] = fib(i)
    a[0] = 0
    a[1] = 1
    for i in range(2,n+1):
        a[i] = a[i-1] + a[i-2]
    return a[n]
```

$\Rightarrow O(n)$

フィボナッチ数：実行例

```
% python -i fib.py
>>> fib(10)
55
>>> fib(20)
6765
>>> fib(30)
832040
>>> fib(40)
102334155
>>> fib2(20)
6765
>>> fib2(40)
102334155
>>> fib2(60)
1548008755920
>>> exit()
```

動的計画法：フィボナッチ数

$fib(n)$ の計算

- $fib(0), fib(1), \ldots, fib(n-1)$ を全て計算する必要がある.
- 小さい方から計算すれば良い.
- $f(n)$ の計算: $f(n-1)$, $f(n-2)$ のみが必要
  $\Rightarrow$ 直前の二つのみ覚えておけば良い

```
def fib3(n):
    k = 0
    f = 0           # f = fib(k)
    g = 1           # g = fib(k+1)
    while k < n:
        f, g = g, f + g
        k = k + 1
    return f
```

注意： `f, g = g, f + g` は, 変数 `f` と `g` を同時に代入する構文

最大部分和問題 (Maximum Segment Sum)

問題： 部分列の和の最大値を求める

- 入力： 数列 $\langle a_1, a_2, \ldots, a_n \rangle$
- 出力:

$$\mathrm{max\_seg\_sum}(\langle a_1, \ldots, a_n \rangle) = \max_{i,j} \sum_{k=i}^{j} a_k$$

例：$\langle -2, 1, 3, -2, 3, -3, -2, 4, -2 \rangle$

$$1 + 3 + -2 + 3 = 5$$

最大部分和問題

$$\mathrm{max\_seg\_sum}(\langle a_1, \ldots, a_n \rangle) = \max_{i,j} \sum_{k=i}^{j} a_k$$

単純なアルゴリズム

- すべての i, j に対して

$$\sum_{k=i}^{j} a_k$$

  を計算
- その最大値を求める

$\Rightarrow O(n^3)$

```
def max_seg_sum(a):
    n = len(a)
    max = 0
    for i in range(n):
        for j in range(n):
            s = sum(a,i,j) # a[i] +
                           #   ... + a[j]
            if s > max:
                max = s
    return max
```

実行例

```
% python -i max_seg_sum.py
>>> max_seg_sum([-2, 1, 3, -2, 3, -3, -2, 4, -2])
5
>>> a = random_array(10)
>>> a
[-8, -10, -6, -9, -10, 0, 9, 8, -3, 5]
>>> max_seg_sum(a)
19
```

配列の大きさが $n = 1000$ だとだいぶ待つ

```
>>> a = random_array(1000)
>>> max_seg_sum(a)
14065
```

最大部分和問題

$$\mathrm{max\_seg\_sum}(\langle a_1, \ldots, a_n \rangle) = \max_{i,j} \sum_{k=i}^{j} a_k$$

部分問題として

$$b_j = \max_{i \leq j} \sum_{k=i}^{j} a_k$$

考えると，

$$\mathrm{max\_seg\_sum}(\langle a_1, \ldots, a_n \rangle) = \max_j b_j$$

さらに，

$$\begin{aligned} b_j &= \begin{cases} b_{j-1} + a_j & b_{j-1} \geq 0 \\ a_j & \text{otherwise} \end{cases} \\ b_0 &= 0 \end{aligned}$$

$\Rightarrow O(n)$

課題４：動的計画法

- 締め切り： 1 月 18 日 22:00
- 提出するファイル: `max_seg_sum.py`
  - 問題 A: 必須
  - 問題 B: オプション

課題４：A

動的計画法に基づき，最大部分和を計算する関数 max_seg_sum2 を完成せよ．

```
def max_seg_sum2(a):
    n = len(a)
    b = [0] * (n+1)
    b[0] = 0
    for j in range(n):
        pass                          # pass は何もしない文（命令）
        # pass を消して，この部分を埋める
    return array_max(b)
```

インデックスがズレているので注意すること

- スライド

$$a_1, \ldots, a_n$$
$$b_0, b_1, \ldots, b_n$$

- Python のプログラム

$$a[0], \ldots, a[n-1]$$
$$b[0], b[1], \ldots, b[n]$$

課題４：B（オプション）

フィボナッチ数列の場合と同様に，b_j を格納する配列を用いなくても最大部分和を計算できる．この関数 max_seg_sum3 を完成せよ．

```
def max_seg_sum3(a):
    n = len(a)
    max = 0
    bj = 0
    for j in range(n):
        pass
        # pass を消して，この部分を埋める
    return max
```

実行例

```
>>> a = [-2, 1, 3, -2, 3, -3, -2, 4, -2]
>>> max_seg_sum(a)
5
>>> max_seg_sum2(a)
5
>>> max_seg_sum3(a)
5
```

配列の大きさが $n = 1000$ のとき

```
>>> a = random_array(1000)
>>> max_seg_sum(a)
14615
>>> max_seg_sum2(a)
14615
>>> max_seg_sum3(a)
14615
```