データ構造とアルゴリズム
（第8回）

モバイルコンピューティング研究室
柴田史久

1

1

本日の講義内容

- 整列（１）
  - 整列とは
  - 単純な整列アルゴリズム
  - バブルソート
  - 選択ソート
  - 挿入ソート

2

2

教科書 第11章（pp.285～293）

整列とは？

3

3

整列（sorting）という操作

- レコードの集まりをキー値の大小関係で並べ替え
  - 昇順（ascending order）：小から大へ
  - 降順（descending order）：大から小へ
- 同じキー値のレコードがあった場合は連続して並ぶ

4

4

人間ならどうやるか？

- 人間はコンピュータより有利
  - 全体を見渡せる
  - まとめて複数の人を入れ替えることができる
- コンピュータができるのは
  - ２つの項目を比較すること
  - ２つの項目を入れ替える，もしくは１つの項目をどこかへコピーすること

5

5

整列アルゴリズムの分類（1）

- 内部整列（internal sort）
  - 主記憶（メモリ）に置かれたデータを整列
  - 外部整列より高速
  - 主記憶には限りがある
  - どのレコードも一定時間でアクセス（ランダムアクセス）
- 外部整列（external sort）
  - 外部記憶に置かれたデータを整列
  - レコードの並びを順に読み込んで処理
    - Ex. 外部記憶として磁気テープを利用していた時代
  - 代表的なアルゴリズムはマージソート（併合ソート）

6

6

整列アルゴリズムの分類(2)

- 比較による整列
  - レコードのキーの大小を比較して位置を入れ替える
  - 一般には「整列」＝「比較による整列」
  - n個のレコードを整列するのに $O(n \log n)$ 必要：証明済
    - Ex. クイックソート（quick sort），ヒープソート（heap sort）
- 比較によらない整列
  - キーの値を特定の範囲に制限することで比較せずに整列
    - Ex. ビンソート（bin sort），分布数え上げソート（distribute counting sort），基数ソート（radix sort）
  - デジタルソート（digital sort）とも
  - 計算量は $O(n)$

7

7

整列アルゴリズムの種類

- 単純なアルゴリズム
  - バブルソート（bubble sort）
  - 選択ソート（selection sort）
  - 挿入ソート（insertion sort）
- 中間のアルゴリズム
  - シェルソート（Shell sort）
- 高速なアルゴリズム
  - クイックソート（quick sort）
  - ヒープソート（heap sort）
  - マージソート（merge sort）

8

8

整列の計算量

- 整列アルゴリズムの処理
  - 2つのレコードのキーの大小を判定（比較）
  - 必要ならそれらを入れ替える（交換）
- 同じ計算量のアルゴリズムでも比較回数と交換回数の比率は異なる
  - レコードが大きければ交換回数が少ないほうがよい
- 定数項部分にも注意
  - データ数が少ないなら高速で複雑なものより単純なもの

9

9

安定な整列

- 同じキーを持つデータ間で，整列前の位置関係が保たれるような整列アルゴリズム
- 実用面で重要な性質
- 一般には，単純な整列は安定で複雑な整列は安定でないという傾向がある
  - サブキーとしてレコード位置を記録すれば回避可能
    - キーが同じ場合には，サブキーで大小比較

10

10

教科書 第12章（pp.294～304）

単純な整列アルゴリズム

11

11

単純な整列アルゴリズムとは?

- バブルソート，選択ソート，挿入ソート
- 計算量は $O(n^2)$
- バブルソート，挿入ソートは安定なアルゴリズム

12

12

実装に関する補足(1)

- バブルソート：BubbleSortクラス
- 選択ソート：SelectionSortクラス
- 挿入ソート：InsertionSortクラス

- int型の配列を昇順に整列するsortメソッドを定義
  - public static void sort(int[] a);
  - a は整列対象のint型配列

- ユーティリティクラスとして定義

- 要素は「0番目」から「n-1番目」

13

13

実装に関する補足(2) Swap処理

- 配列中の２つの要素の値を入れ替える手続き
  - 整列処理ではSwap処理が頻繁に出てくる
- 実際には以下のようにテンポラリの変数を準備して入れ替えする

```
temp = a ;
a = b ;
b = temp ;
```

14

14

バブルソート(1)

- 配列の後ろから先頭に向かって以下を繰り返す
  - 隣接する２要素を比較
  - 右側（要素番号の大きい側）の値が大きくなるように比較結果によって値を交換
  - １度の手順で先頭に最小値がくることが保証されるので，スキャンする範囲を１ずつ狭めながら繰り返す

- 最小値が泡（バブル）のように浮き上がっていく

- 教科書によっては最大値が順に後ろにくる実装も

15

15

バブルソート(2)

詳細は教科書 p.296 List 12.1

```
public static void sort(int[] a)
{
  int n = a.length;   // 配列の要素数

  for (int i = 0; i < n - 1; i++) {   // 外側のループ
    for (int j = n - 1; j > i; j--) { // 配列の後ろから先頭へ
      if (a[j - 1] > a[j]) {          // 隣同士を比較して
        int temp = a[j];              // 後ろ側が大きくなるように
        a[j]     = a[j - 1];          // 必要ならば交換
        a[j - 1] = temp;
      }
    }
  }
}
```

16

16

選択ソート（1）

- 配列の先頭から後ろに向かって以下を繰り返す
  - 未整列部分から最小値を選択
  - 選択した最小値を未整列部分の先頭に置く
  - 一度の手順で先頭に最小値がくることが保証されるので，以上を未整列部分がなくなるまで繰り返す

17

17

選択ソート(2)

詳細は教科書 p.298 List 12.2

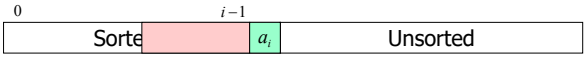
```
public static void sort(int[] a)
{
  int n = a.length;   // 配列の要素数

  for (int i = 0; i < n - 1; i++) {   // 先頭から順番に
    int lowest = i;
    int lowkey = a[i];
    for (int j = i + 1; j < n; j++) { // 未整列の中で最小値を探して
      if (a[j] < lowkey) {
        lowest = j;
        lowkey = a[j];
      }
    }
    int temp   = a[i];                // 先頭と最小値を交換
    a[i]       = a[lowest];
    a[lowest] = temp;
  }
}
```

18

18

挿入ソート（1）

- 手順
  - a_0 から a_{i-1} までがソート済みと仮定
  - a_i をしかるべき位置に挿入
  - 以上を i を増加させつつ繰り返す



19

19

挿入ソート（2）

詳細は教科書 p.301 List 12.3

```
public static void sort(int[] a)
{
  int n = a.length;   // 配列の要素数

  for (int i = 1; i < n; i++) {
    int j = i;
    while (j >= 1 && a[j - 1] > a[j]) {  // 隣同士を比べて
      int temp = a[j];                   // 適切な位置まで
      a[j]     = a[j - 1];               // 交換していく
      a[j - 1] = temp;
      j--;
    }
  }
}
```

20

20

挿入ソート（3）

一部改良

```
public static void sort(int[] a)
{
  int n = a.length;   // 配列の要素数

  for (int i = 1; i < n; i++) {
    int temp = a[i];  // 移動対象をいったん退避
    int j = i;
    while (j >= 1 && a[j - 1] >= temp) {  // 移動対象以上のものを
      a[j] = a[j - 1];                    // 1つずつずらしていく
      j--;
    }
    a[j] = temp;
  }
}
```

21

21

単純なアルゴリズムの性能

- 結果から計算量が $O(n^2)$ とわかる

教科書 p.303 Table 12.1 整列の実行時間（単位：ミリ秒）

配列の要素数 n	100	500	1000	5000	10000	50000
バブルソート	0.022	0.56	2.2	55	230	6100
選択ソート	0.010	0.17	0.64	14	59	1600
挿入ソート	0.0076	0.16	0.64	16	67	1800

22

22

選択ソートの計算量

- 比較回数： $O(n^2)$
- 詳しくみてみると…
  - 第 i 回目の繰り返しでは $n-i-1$ 回の比較が必要
  - 全体の比較回数は

$$\sum_{i=0}^{n-2}(n-i-1)=\sum_{i=0}^{n-2}n-\sum_{i=0}^{n-2}i-\sum_{i=0}^{n-2}1$$

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

- 入れ替え回数：$O(n)$

23

23

バブルソートの計算量

- 比較回数： $O(n^2)$
- 詳しくみてみると…
  - 第 i 回目の繰り返しでは $n-i-1$ 回の比較が必要
  - 比較回数は選択ソートと同様
- 多少の工夫として…
  - 整列状態を表すフラグを導入すれば少し高速化可能
- 入れ替え回数： $O(n^2)$

24

24

挿入ソートの計算量(1)

- 外側のforループは n-1 回ループ
- 内側のループの反復回数は $a_0 \leqq a_1 \leqq \cdots \leqq a_{i-1}$ $a_i \geqq a_{i-1}$ の場合に 0 回，$a_i < a_0$ の場合に i 回となる
- whileループ全体の反復回数は最良0回，最悪では

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

- 平均では内側のループの反復回数は $i/2$ 回と期待されるのでwhileループ全体では

$$\sum_{i=1}^{n-1} \frac{i}{2} = \frac{n(n-1)}{4}$$

25

25

挿入ソートの計算量(2)

- 計算量
  - 最良： $O(n)$
  - 最悪： $O(n^2)$
  - 平均： $O(n^2)$

26

26

まとめ

- 整列とは
- 単純な整列アルゴリズム
- バブルソート
- 選択ソート
- 挿入ソート
- 単純な整列アルゴリズムの計算量

27

27

参考文献

- 定本 Javaプログラマのための アルゴリズムとデータ構造（近藤嘉雪）
- 新・明解 Javaで学ぶ アルゴリズムとデータ構造（柴田望洋）
- 岩波講座ソフトウェア科学 3 アルゴリズムとデータ構造（石畑清）
- Javaで学ぶアルゴリズムとデータ構造 Robert Lafore（著）・岩谷 宏（翻訳）
- Java アルゴリズム+データ構造完全制覇 オングス (著)・杉山 貴章・後藤 大地 (監修)

28

28

選択ソートの計算量

Slide 23 画像版

- 比較回数： $O(n^2)$
- 詳しくみてみると…
  - 第 i 回目の繰り返しでは $n-i-1$ 回の比較が必要
  - 全体の比較回数は

$$\sum_{i=0}^{n-2}(n-i-1) = \sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1$$

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

- 入れ替え回数：$O(n)$

23

29

挿入ソートの計算量(1)

Slide 25 画像版

- 外側のforループは n-1 回ループ
- 内側のループの反復回数は $a_0 \leqq a_1 \leqq \cdots \leqq a_{i-1}$ $a_i \geqq a_{i-1}$ の場合に 0 回，$a_i < a_0$ の場合に i 回となる
- whileループ全体の反復回数は最良0回，最悪では

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

- 平均では内側のループの反復回数は $i/2$ 回と期待されるのでwhileループ全体では

$$\sum_{i=1}^{n-1} \frac{i}{2} = \frac{n(n-1)}{4}$$

25

30