

## データ構造とアルゴリズム (第12回)

モバイルコンピューティング研究室  
柴田史久



1

## 本日の講義内容

- 整列 (5)
  - 比較によらない整列アルゴリズム
  - ビンソート (bin sort)
  - 分布数え上げソート (distribution counting sort)
  - 基数ソート (radix sort)

2

教科書 第17章 (pp.379~407)

## 比較によらないソート

3

## 比較によらない整列アルゴリズム

- キーの大小比較による整列アルゴリズム
  - 計算量:  $O(n \log n)$  証明済み
- 大小比較をしなくても速くできる
  - 例
    - ビンソート (bin sort)
    - 分布数え上げソート (distribution counting sort)
    - 基数ソート (radix sort)
  - キーが特定の条件を満たす必要がある
  - $O(n)$  の領域計算量が必要

4

## ビンソート(bin sort)

- 前提
  - キーの取り得る値が決まっている
  - 要素の重複がない
- 手順
  - キーが取り得る値すべてに対して箱 (bin) を用意
  - すべてのデータを対応する箱に入れていく
  - 入れ終わったら小さい値の箱から順に中身を取り出す

5

## ビンソートの原理

- 1~10に対応する箱 (ビン) を準備  

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----
- データを対応する箱に入れる  

7	4	2	8	1					
---	---	---	---	---	--	--	--	--	--

  

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----
- 左から順に中身を取り出せば整列完了  

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

  

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

6

## ビンソートの実装(1)

詳細は教科書 pp.383~384 List 17.1

```
public class BinSortData {
    static final int M = 100 ; // キーは 0 ~ M = 100 までの整数
    private int    key ; // キー
    private Object data ; // その他の情報
    BinSortData(int key, Object data) {
        if (key < 0 || key > M) { // キーが範囲外なら例外を投げる
            throw new IllegalArgumentException() ;
        }
        this.key = key ;
        this.data = data ;
    }
    public int getKey() {
        return key ;
    }
    public Object getData() {
        return data ;
    }
}
```

7

## ビンソートの実装(2)

詳細は教科書 p.385 List 17.2

```
public class BinSort {
    public static void sort(BinSortData[] a) {
        final int N = a.length ; // 配列の要素数
        final int M = BinSortData.M ; // キーは 0 ~ M までの整数
        BinSortData[] bin = new BinSortData[M + 1] ; // ビンの初期化
        for (int i = 0 ; i < N ; i++) {
            int key = a[i].getKey() ;
            if (bin[key] != null) { // キーが重複したら例外を投げる
                throw new IllegalArgumentException() ;
            }
            bin[key] = a[i] ; // 要素をビンに入れる
        }
        int j = 0 ; // データをビンから昇順に取り出す
        for (int i = 0 ; i <= M ; i++) {
            if (bin[i] != null) {
                a[j++] = bin[i] ; // 配列 a に戻す
            }
        }
    }
}
```

null で初期化される

8

## ビンソートの実装(3)

詳細は教科書 p.386 List 17.3

```
public class BinSortMain {
    public static void main(String[] args) {
        int keys[] = {13, 24, 15, 5, 98, 44, 35, 96, 82, 73} ;
        BinSortData[] array = new BinSortData[keys.length] ;
        for (int i = 0 ; i < keys.length ; i++) {
            array[i] = new BinSortData(keys[i], "要素" + i) ;
        }

        BinSort.sort(array) ;

        for (int i = 0 ; i < array.length ; i++) {
            // 省略. 画面に出力する
        }
    }
}
```

9

## ビンソートの性質(1)

- 前提
  - データの個数を  $n$
  - ビンの個数を  $m$
- 計算量:  $O(n+m)$ 
  - ビンへの振り分け:  $O(n)$
  - ビンからの取り出し:  $O(m)$
- $m \gg n$  でなければ  $O(n)$  とみなせる
  - $n = 200, m = 1000$  なら  $O(n)$
  - $n = 10, m = 10000$  なら  $O(m)$
- 制限
  - キーの範囲が広すぎると使い物にならない

10

## ビンソートの性質(2)

- 作業用に  $m$  個のビンを使う
  - 作業領域:  $O(m) \asymp O(n)$  ( $m \asymp n$  ならば)
- データ量に比例した作業領域が必要
  - 分布数え上げソート, 基数ソートも同様
- ビンソートの弱点
  - キーの重複が許されない
  - キーはある範囲に収まる整数
- 別名
  - バケットソート (bucket sort)
    - バケツには複数のデータが入れられる場合も

11

## 分布数え上げソート

- Distribution counting sort
- 別名
  - 度数ソート
  - 計数ソート (counting sort)
- 手順
  - すべてのデータをスキャンしてキーの出現頻度を調べる
    - 出現回数を調べる
    - 出現回数の累計 (累積度数分布) を求める
  - 出現頻度をもとに要素を整理

12

## 分布数え上げソートの原理

● 対象となるデータの並び : 7, 1, 4, 2, 7, 8, 2

● 右の表を作成

- キーを数え上げる
- 累計 (累積度数分布) を求める

● 要素を整列

- キーkを持つ要素は配列の(b-a)番目から(b-1)番目に置く

キー k	出現回数 a	累計 b
0	0	0
1	1	1
2	2	3
3	0	3
4	1	4
5	0	4
6	0	4
7	2	6
8	1	7
9	0	7
10	0	7

0	1	2	3	4	5	6
1	2	2	4	7	7	8

13

## 分布数え上げソートの実装

詳細は教科書 p.391 List 17.4

```
public class DistributionCountingSort {
    static void sort(BinSortData[] a) {
        final int N = a.length; // 配列の要素数
        final int M = BinSortData.M; // キーは 0 ~ M までの整数
        int[] count = new int[M + 1]; // カウンタを初期化. 初期値は 0
        for (int i = 0; i < N; i++) { // キーを数え上げる
            count[a[i].getKey()]++;
        }
        for (int i = 0; i < M; i++) { // 累積度数分布を求める
            count[i + 1] += count[i]; // カウンタの内容は累積度数分布
        }
        BinSortData[] b = new BinSortData[N];
        // 累積度数分布に従って後ろから順にデータを配列aから作業用配列bへコピー
        for (int i = N - 1; i >= 0; i--) {
            b[--count[a[i].getKey()]] = a[i];
        }
        System.arraycopy(b, 0, a, 0, N); // 配列aに結果を書き戻す
    }
}
```

書き戻し処理は本来は不要  
インタフェースを他のソートに合わせるため

14

## 分布数え上げソートの動作(1)

```
for (int i = N - 1; i >= 0; i--) {
    b[--count[a[i].getKey()]] = a[i];
}
```

a[i].getKey() → 2

count[2] → 3 を先にデクリメントした 2

b[2] → a[i] を代入

配列 a	0	1	2	3	4	5	6
	7	1	4	2	7	8	2

  

配列 b	0	1	2	3	4	5	6
			2				

  

配列 count	キー k	count
	0	0
	1	1
	2	2
	3	3
	4	4
	5	4
	6	4
	7	6
	8	7
	9	7
	10	7

15

## 分布数え上げソートの動作(2)

```
for (int i = N - 1; i >= 0; i--) {
    b[--count[a[i].getKey()]] = a[i];
}
```

a[i].getKey() → 8

count[8] → 7 を先にデクリメントした 6

b[6] → a[i] を代入

配列 a	0	1	2	3	4	5	6
	7	1	4	2	7	8	2

  

配列 b	0	1	2	3	4	5	6
			2			8	

  

配列 count	キー k	count
	0	0
	1	1
	2	2
	3	3
	4	4
	5	4
	6	4
	7	6
	8	6
	9	7
	10	7

16

## 分布数え上げソートの動作(3)

```
for (int i = N - 1; i >= 0; i--) {
    b[--count[a[i].getKey()]] = a[i];
}
```

a[i].getKey() → 7

count[7] → 6 を先にデクリメントした 5

b[5] → a[i] を代入

配列 a	0	1	2	3	4	5	6
	7	1	4	2	7	8	2

  

配列 b	0	1	2	3	4	5	6
			2			7	8

  

配列 count	キー k	count
	0	0
	1	1
	2	2
	3	3
	4	4
	5	4
	6	4
	7	5
	8	6
	9	7
	10	7

17

## 分布数え上げソートの動作(4)

```
for (int i = N - 1; i >= 0; i--) {
    b[--count[a[i].getKey()]] = a[i];
}
```

a[i].getKey() → 2

count[2] → 2 を先にデクリメントした 1

b[1] → a[i] を代入

同じキーを持つデータの  
位置関係が保たれている

配列 a	0	1	2	3	4	5	6
	7	1	4	2	7	8	2

  

配列 b	0	1	2	3	4	5	6
		2	2			7	8

  

配列 count	キー k	count
	0	0
	1	1
	2	1
	3	3
	4	4
	5	4
	6	4
	7	5
	8	6
	9	7
	10	7

安定なソートになっている

18

## 分布数え上げソートの性質

- 処理の流れと計算量
  - カウンタ配列の初期化:  $O(n)$
  - キーを数え上げる:  $O(n)$
  - 累積度数分布を求める:  $O(m)$
  - 度数分布に従ってデータを配列aからbへコピー:  $O(n)$
  - 配列bのデータを配列aにコピー:  $O(n)$
- 全体の計算量:  $O(n)$ 
  - $m \gg n$  ならば  $O(m)$
- 領域計算量
  - $O(n)$  あるいは  $O(m)$

19

## 基数ソート(1)

- ビンソート・分布数え上げソートの弱点
  - キー  $m$  に比例した作業領域が必要
- 基数ソート (radix sort)
  - キーの種類が多い場合でも  $O(n)$  で整列可能
  - パンチカードの整列に利用

20

## 基数ソート(2)

- 前提条件
  - 対象: 位取り記数法 (N進法) で表現可能なもの
  - すべての入力データが決まった形式
    - Ex. 3桁の整数, 2文字のアルファベット
- 手順
  - 下の桁から順番に安定な整列を実行
  - 最後に最上位桁で整列すると全体の整列が終了
- 別の説明
  - キーをいくつかのサブキーに分割
  - 下位から上位の順でサブキーに対して安定な整列を実行

21

## 基数ソートの例: 10進数3桁の数字

	12	904	206	123	255	813	500	444
	381	999	99	768	3	111	640	888
第1回目 (1桁目)	第2回目 (2桁目)		第3回目 (3桁目)					
バケツ	データ		バケツ	データ		バケツ	データ	
0	500, 640		0	500, 3, 904, 206		0	3, 12, 99	
1	381, 111		1	111, 12, 813		1	111, 123	
2	12		2	123		2	206, 255	
3	123, 813, 3		3			3	381	
4	904, 444		4	640, 444		4	444	
5	255		5	255		5	500	
6	206		6	768		6	640	
7			7			7	768	
8	768, 888		8	381, 888		8	813, 888	
9	999, 99		9	999, 99		9	904, 999	

22

## 基数ソート(3)

- サブキーの整列アルゴリズム
  - 安定ならばなんでもよいが,  $O(n)$  の手法が必要
- 分布数え上げソートを利用するのが一般的

23

## 基数ソートの実装(1)

詳細は教科書 pp.400~401 List 17.5

```
public class RadixSortData {
    static final int KEY_MAX = 0xffff; // キーの最大値
    private int key; // キー
    private Object data; // その他の情報
    RadixSortData(int key, Object data) {
        if (key < 0 || key > KEY_MAX) { // キーが範囲外なら例外を投げる
            throw new IllegalArgumentException();
        }
        this.key = key;
        this.data = data;
    }
    public int getKey() {
        return key;
    }
    public Object getData() {
        return data;
    }
}
```

BinSortDataクラスとほぼ一緒

24

## 基数ソートの実装(2)

詳細は教科書 pp.402~403 List 17.6

```
public class RadixSort {
    static final int M = 15 ; // サブキーは 0 ~ 15(0xf) の整数
    static final int MASK = 0xf ; // サブキーを取り出すマスク
    static void sort(RadixSortData[] a) {
        final int N = a.length ; // 配列の要素数
        RadixSortData[] b = new RadixSortData[N] ; // 作業用配列
        int[] count = new int[M + 1] ; // 分布数え上げ用配列
        for (int bit = 0 ; bit < 16 ; bit += 4) { // 4ビットずつループ
            for (int i = 0 ; i <= M ; i++) { // カウンタを初期化
                count[i] = 0 ;
            }
            for (int i = 0 ; i < N ; i++) { // キーを数え上げる
                count[(a[i].getKey() >> bit) & MASK]++ ;
            }
            for (int i = 0 ; i < M ; i++) { // 累積度数分布を求める
                count[i + 1] += count[i] ;
            }
            for (int i = N - 1 ; i >= 0 ; i--) { // 後ろから順にコピー
                b[--count[(a[i].getKey() >> bit) & MASK]] = a[i] ;
            }
            System.arraycopy(b, 0, a, 0, N) ; // 作業用配列から戻す
        }
    }
}
```

25

## 基数ソートの実装(3)

詳細は教科書 p.406 List 17.7

```
public class RadixSortMain {
    public static void main(String[] args) {
        int data[] = {
            0x2f38, 0x2fb7, 0x9328, 0xa400, 0x000f,
            0x0002, 0x0844, 0xef85, 0x289a, 0x2f30, };
        RadixSortData[] array = new RadixSortData[data.length] ;
        for (int i = 0 ; i < data.length ; i++) {
            array[i] = new RadixSortData(data[i], "要素" + i) ;
        }

        RadixSort.sort(array) ;

        for (int i = 0 ; i < array.length ; i++) {
            // 省略. 画面に出力する
        }
    }
}
```

26

## 基数ソートの性質

- データの個数 :  $n$
- サブキーの数 :  $a$
- サブキーの種類 :  $m$  ( $<< n$ )
- 分布数え上げソートの計算量 :  $O(n)$
- 分布数え上げソートの回数 :  $a$
- 基数ソートの計算量 :  $O(an)$
- キーを分割してもキーの大小関係が保存される必要

27

## まとめ

- 比較によらない整列アルゴリズム
- ビンソート
- 分布数え上げソート
- 基数ソート

28

## 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)

29