第9回　並び替えアルゴリズム

〜さまざまなアルゴリズムを比較しよう〜

学習目標

- 適切なアルゴリズムを用いたプログラムが書ける
  - ■ 並び替え（ソート）アルゴリズムの性能について比較検討し、説明できる
  - ■ **Java** でソートのプログラムが書ける
  - ■ 問題が与えられたとき、適切なアルゴリズムでプログラムを書ける

9.1.　並び替えの性能比較プログラム

9.1.1.　今回扱うアルゴリズム

①.バイナリサーチをするためには

　前回はリニアサーチとバイナリサーチという二つの検索のアルゴリズムを紹介しました。しかしこのアルゴリズムはどんな状況でも使えるわけではありません。

リニアサーチ

- 要素が少ない時しか使えない（数十個程度）
- ソートされていないデータでも使える

バイナリサーチ

- ソート済み（順番に並んでいる）データでないと使えない

②.並び替えアルゴリズム

　そこで、今回はデータを順番に並びかえる（ソート）ためのアルゴリズム、を考えてみましょう。

　ソートは重要で時間のかかる処理なので、これまでにも多くのコンピュータ科学者が研究に取り組み、いくつもの高度な方法が発明されています。今回はその中で比較的簡単な三つの方法を紹介します。

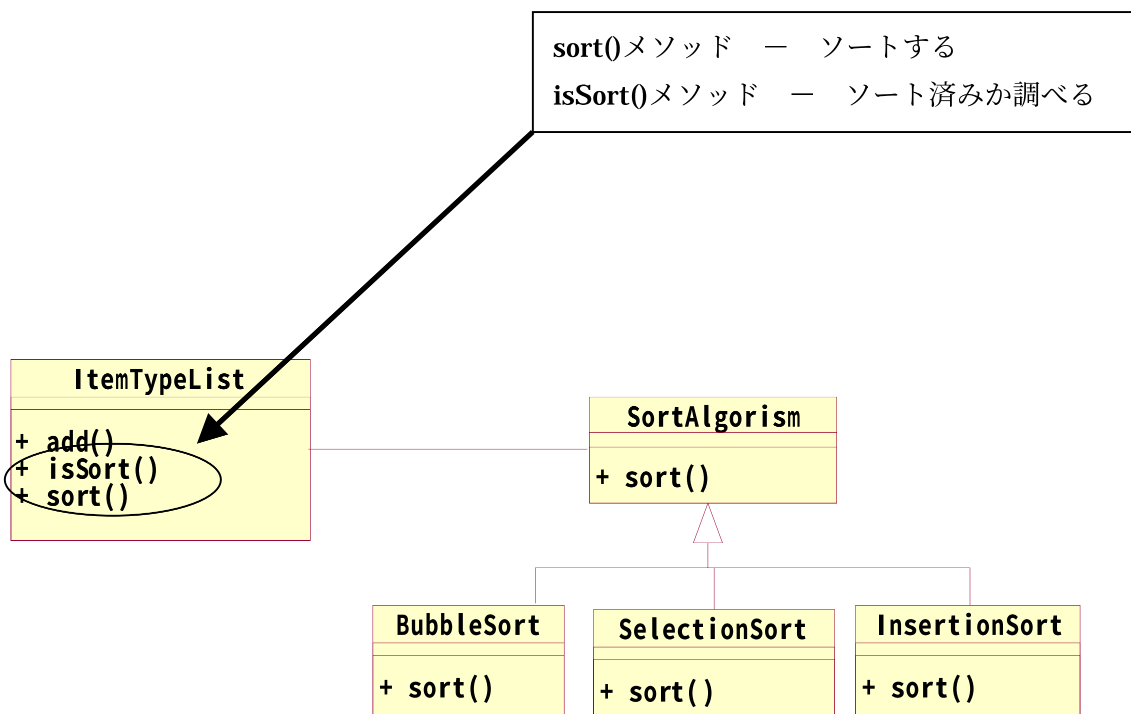
- バブルソート
- 選択ソート
- 挿入ソート

　これらのテクニックは基本的なアルゴリズムですが、単に分かりやすいだけでなく、データの並び方や数によって、これらの方法のほうが速い場合もあります。

9.1.2.　　実験環境の構築

①.ソート性能比較プログラムの設計

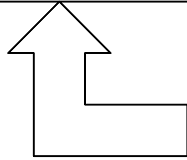
　まず最初にソートに必要なメソッドを配置して、クラス設計を行いましょう。
ソートをするためのメソッドと、ソート済みかを調べるメソッドを **ItemTypeList** に追加します。また、ソートのアルゴリズムをクラスにする方法で設計します。
　クラス図を以下に示します。



②.ランダムな数値を発生させる

今回はでたらめな数値を配列に入れて、それをソートします。そのために「ランダムな数値」を作るための仕掛けが必要です。Java では Math.random()メソッドを使うことでランダムな数値を得ることができます。

```
double randomNo = Math.random();
```



メソッドの返り値は double 型で、
0 以上～１未満の間のランダムな「小数」を返します。

このメソッドを使って、例えば 0～9 の間のランダムな整数を得るには以下のようにします。

```
//ランダムな数の範囲を決める。ここでは 0～9 の数値を得たいので 10 と入力。
int randomRange = 10;

//ランダムな数値を double 型で得る。
double randomNo = Math.random();

//ランダムな数値を１～１０の範囲内に変換する。
double randomDouble = randomNo * randomRange;

//ランダムな数値を int 型に変換する
int randomInt = (int)randomDouble;
```

上記のプログラムの具体的な例をあげましょう。Math.random()を実行すると 0～1 の範囲で小数が得られるので、例えば「0.351」という小数が randomNo に入ります。この小数を 0～9 の範囲に変換したいので、これに 10 をかけます。そうすると「3.51」が randomDouble に入ります。最後にこれを int 型に変換すると小数点以下が切り捨てられるので randomInt に３が入ります。
この計算を、ランダムな小数の値を変えてやって見ましょう。０～１の範囲の小数であれば必ず最後には 0～9 の範囲でランダムな数が randomInt に入ります。またランダム数の範囲を変えたい時は randomRange を変えてみましょう。randomRange に 10000 が入れば 0～9999 の範囲のランダム数が得られます。

上のプログラムを２行で書くと、以下のようになります。内容は全く同じです。

```
int randomRange = 10;
int randomInt = (int)( Math.random() * randomRange );
```

③.ソートの性能比較プログラム

　ソートアルゴリズムの性能比較プログラムを示します。各アルゴリズムのクラスは、次節で紹介するので、ここでは省略します。

例題 9-1：ソートアルゴリズムの比較(Example9_1.java)

```
1:      /**
2:      * オブジェクト指向哲学　入門編
3:      * 例題 9-1：ソートアルゴリズムの比較
4:      * 3 種類のソートアルゴリズムの性能を比較するプログラム
5:      *
6:      * メインクラス
7:      */
8:      public class Example9_1 {
9:
10:       /**
11:       * ソートの性能比較をするプログラム・メイン
12:       */
13:       public static void main(String[] args) {
14:
15:         int randomRange = 1000000;//ランダムの範囲（100 万なら 1～100 万の範囲のランダム
な数字を生成する）
16:         int itemTypeNum = 10000;//要素数
17:
18:         //3 種類のソートアルゴリズムを利用する商品種類リストを生成する
19:         ItemTypeList itemTypeListBubble = new ItemTypeList(new BubbleSort());
20:         ItemTypeList itemTypeListSelection = new ItemTypeList(new SelectionSort());
21:         ItemTypeList itemTypeListInsertion = new ItemTypeList(new InsertionSort());
22:
23:         //ランダムな数値を作り、商品種類として登録する
24:         for(int i=0;i<itemTypeNum;i++){
25:           double randomNo = Math.random();//ランダムな数値を生成する
26:           int randomInt = (int)(randomNo*randomRange);//ランダムな数値を指定された範囲
の整数に変換する
27:           //ランダムな商品番号を持つ商品種類を登録する
28:           itemTypeListBubble.add(new ItemType(randomInt, "cola"+randomInt,120));
29:           itemTypeListSelection.add(new ItemType(randomInt, "cola"+randomInt,120));
30:           itemTypeListInsertion.add(new ItemType(randomInt, "cola"+randomInt,120));
31:         }
32:
33:         //各ソートの時間を計る
34:         System.out.println("------------バブルソートの性能測定-----------------");
35:         sortTest(itemTypeListBubble);
36:
37:         System.out.println("------------選択ソートの性能測定-----------------");
38:         sortTest(itemTypeListSelection);
39:
40:         System.out.println("------------挿入ソートの性能測定-----------------");
41:         sortTest(itemTypeListInsertion);
```

```
42:        }
43:
44:        /**
45:        * 与えられた ItemTypeList のソートにかかる時間を計る
46:        */
47:        private static void sortTest(ItemTypeList itemTypeList){
48:          StopWatch sw = new StopWatch();//ストップウオッチをインスタンス化
49:
50:          //商品種類をソートする
51:          sw.start();//ストップウオッチをスタート
52:          itemTypeList.sort();//ソートする
53:          sw.stop();//ストップウオッチを止める
54:          long time = sw.getTime();//かかった時間
55:          System.out.println("ソートにかかった時間は"+time+"ミリ秒です");
56:
57:          //ソートができたかどうか表示する
58:          System.out.println("ソートされているか："+itemTypeList.isSort());
59:        }
60:
61:      }
```

例題 9-1：ソートアルゴリズムの比較(ItemTypeList.java)

```
1:      /**
2:      * オブジェクト指向哲学　入門編
3:      * 例題 9-1：ソートアルゴリズムの比較
4:      * 3 種類のソートアルゴリズムの性能を比較するプログラム
5:      *
6:      * 商品種類リストクラス
7:      * 指定されたアルゴリズムによって商品種類を商品番号順にソートすることができる
8:      */
9:      public class ItemTypeList {
10:
11:        private int ARRAY_SIZE = 10000;                              //用意する配列の大
きさ
12:
13:        private int size=0;                                          //現在配列に保存さ
れている要素数
14:
15:        private ItemType[] itemTypeArray = new ItemType[ARRAY_SIZE];  //商品種類を保存す
るための配列
16:        private SortAlgorithm sortAlgorithm;                        //ソートアルゴリズム
オブジェクト
17:
18:        /**
19:        * コンストラクタ
20:        */
21:        public ItemTypeList(SortAlgorithm newSortAlgorithm){
22:          sortAlgorithm = newSortAlgorithm;//ソートに使用するアルゴリズムを設定する
```

```
23:         }
24:         /**
25:         * 商品種類を追加する
26:         */
27:         public void add(ItemType addItemType){
28:           itemTypeArray[size] = addItemType;//空の箱から順に書き込む
29:           size++;//要素数を 1 つ増やす
30:         }
31:
32:         /**
33:         * 商品種類リストを商品番号でソートする
34:         */
35:         public void sort(){
36:           sortAlgorithm.sort(itemTypeArray,size);//設定されているソートアルゴリズムでソ
ートをする
37:         }
38:
39:         /**
40:         * ソート済みかどうか調べる
41:         */
42:         public boolean isSort(){
43:           for(int i=0;i<size-1;i++){
44:             if(itemTypeArray[i].getId() > itemTypeArray[i+1].getId()){
45:               return false;
46:             }
47:           }
48:           return true;
49:         }
50:
51:       }
```

例題 9-1：ソートアルゴリズムの比較(SortAlgorithm.java)

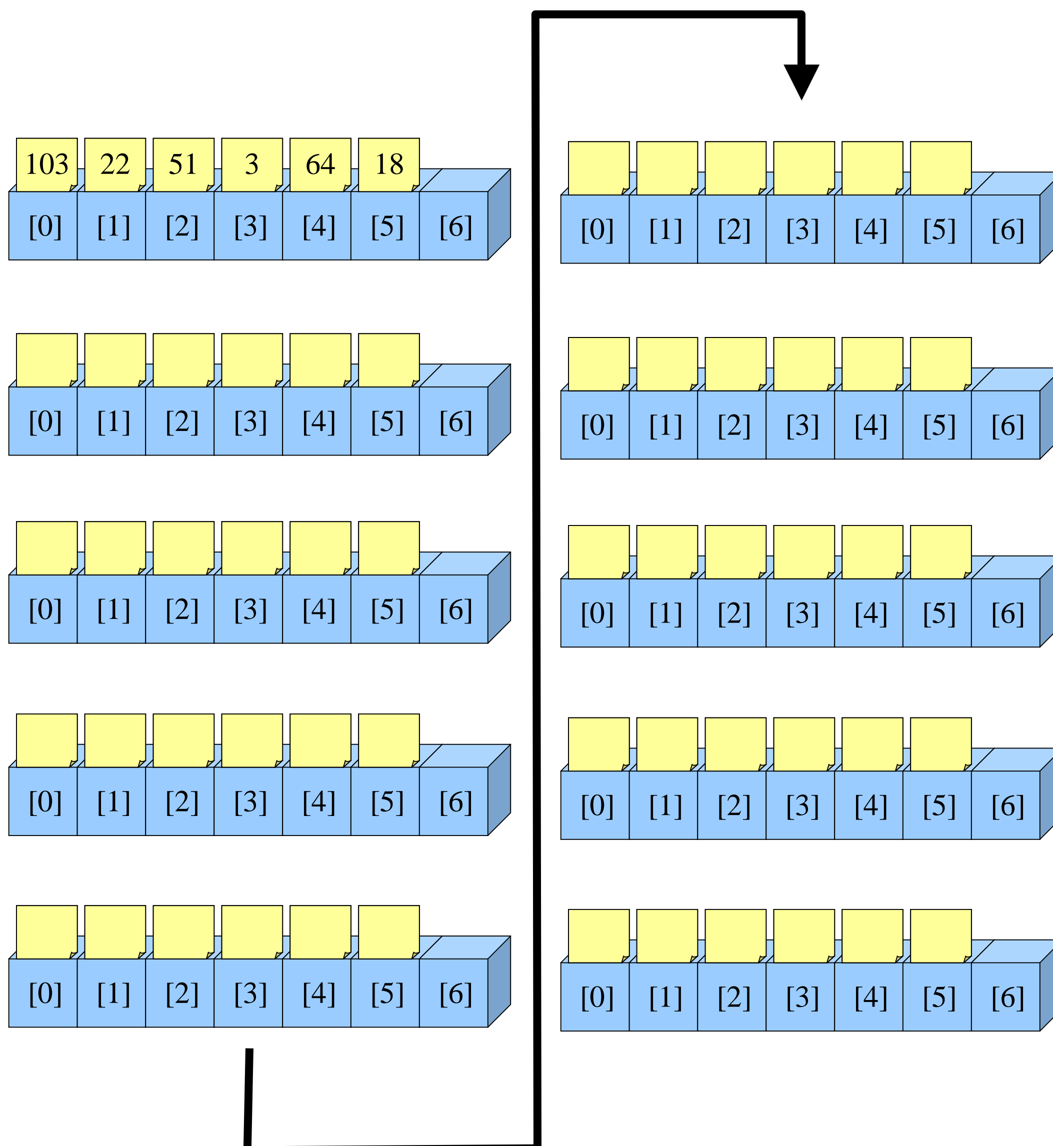
```
1:        /**
2:        * オブジェクト指向哲学　入門編
3:        * 例題 9-1：ソートアルゴリズムの比較
4:        * 3 種類のソートアルゴリズムの性能を比較するプログラム
5:        *
6:        * ソートアルゴリズムの抽象クラス
7:        */
8:        public abstract class SortAlgorithm {
9:
10:         /**
11:         * 並び替え（ソート）をする
12:         */
13:         public abstract void sort(ItemType[] itemTypeArray,int size);
14:
15:       }
```

9.2. ソートのアルゴリズム

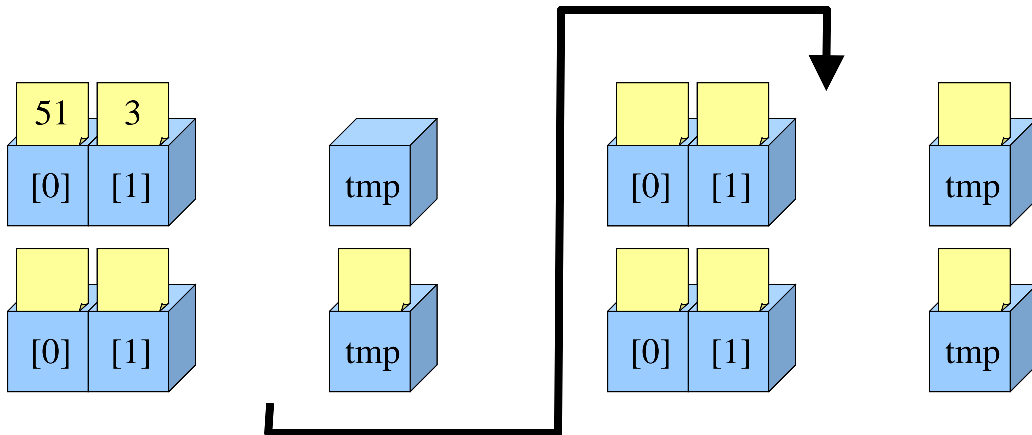
9.2.1. バブルソート

バブルソートの手順は、以下のようになります。

（１） ２つの商品番号を比較する
（２） 左側の方の番号が大きければ商品種類を入れ替える。
（３） 右へ一つ移動する。
（４） ソートの終わっていない部分（毎回小さくなる）に対して上のステップを繰り返す。



①.Swap



＜考えよう！＞バブルソートの効率

バブルソートの効率を考えてみましょう。

（1）比較の回数

	比較の回数
2 個の要素の時	□ 回
3 個の要素の時	□ ＋ □ 回
4 個の要素の時	□ ＋ □ ＋ □ 回
N個の要素の時	

（2）入れ替えの回数

確率的には、比較したうちの半分は入れ替えます。

よって…　□ 回

②. バブルソートの実装

バブルソートと **Swap** メソッドの実装例を示します。

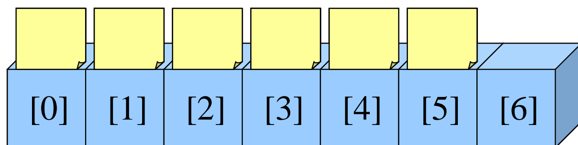
例題 9-1：ソートアルゴリズムの比較(BubbleSort.java)

```
1:     /**
2:     * オブジェクト指向哲学　入門編
3:     * 例題 9-1：ソートアルゴリズムの比較
4:     * 3 種類のソートアルゴリズムの性能を比較するプログラム
5:     *
6:     * バブルソートクラス
7:     */
8:     public class BubbleSort extends SortAlgorithm{
9:
10:      /**
11:      * 並び替え（ソート）をする
12:      */
13:      public void sort(ItemType[] itemTypeArray,int size){
14:        for(int i=size-1;i>1;i--){//要素数回始めから作業を繰り返す
15:          for(int j=0;j<i;j++){//ソート済みの所まで作業する
16:            if(itemTypeArray[j].getId() > itemTypeArray[j+1].getId()){
17:              swap(itemTypeArray,j,j+1);//もし右側のほうが大きかったら入れ替える
18:            }
19:          }
20:        }
21:      }
22:
23:      /**
24:      * 配列内の要素を入れ替える
25:      */
26:      protected void swap(ItemType[] itemTypeArray,int target1,int target2){
27:        ItemType temp;//temporary の箱を用意する
28:        temp = itemTypeArray[target1];
29:        itemTypeArray[target1] = itemTypeArray[target2];
30:        itemTypeArray[target2] = temp;
31:      }
32:
33:
34:    }
```

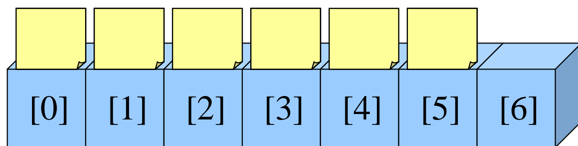
9.2.2.　　選択ソート

選択ソートの手順は、以下のようになります。

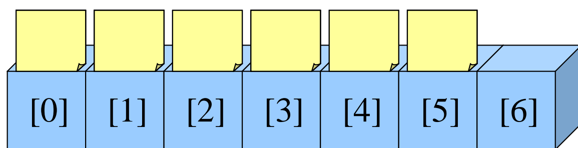
（１）全商品番号から、一番小さい番号を探す。
（２）見つかったものと、一番左の商品種類を入れ替える
（３）ソートの終わっていない部分（毎回小さくなる）に対して上のステップを繰り返す。



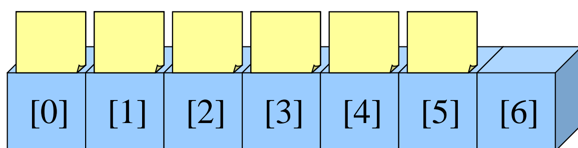
最小値の入っている
箱の番号



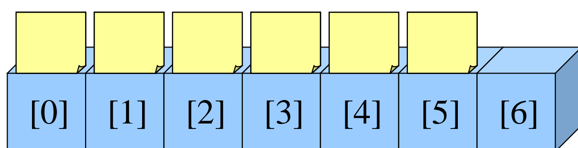
最小値の入っている
箱の番号



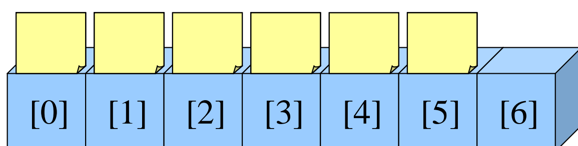
最小値の入っている
箱の番号



最小値の入っている
箱の番号



最小値の入っている
箱の番号



最小値の入っている
箱の番号

＜考えよう！＞選択ソートの効率

選択ソートの効率を考えてください。

（1）比較の回数

	比較の回数
2個の要素の時	□ 回
3個の要素の時	□ ＋ □ 回
4個の要素の時	□ ＋ □ ＋ □ 回
N個の要素の時	

（2）入れ替えの回数

	入れ替えの回数
2個の要素の時	□ 回
3個の要素の時	□ 回
4個の要素の時	□ 回
N個の要素の時	□ 回

①.選択ソートの実装

選択ソートの実装例を示します。

例題 9-1:ソートアルゴリズムの比較(SelectionSort.java)

```
1:      /**
2:      * オブジェクト指向哲学　入門編
3:      * 例題 9-1：ソートアルゴリズムの比較
4:      * 3 種類のソートアルゴリズムの性能を比較するプログラム
5:      *
6:      * 選択ソートクラス
7:      */
8:      public class SelectionSort extends SortAlgorithm{
9:
10:       /**
11:       * 並び替え（ソート）をする
12:       */
13:       public void sort(ItemType[] itemTypeArray,int size){
14:         for(int i=0;i<size-1;i++){//要素数回始めから作業を繰り返す
15:           int minimum = i;//最小値の配列番号を保存しておく
16:           for(int j=i+1;j<size;j++){//ソート済み以降を作業する
17:             if(itemTypeArray[j].getId() < itemTypeArray[minimum].getId()){
18:               minimum = j;//最小値より小さかったら、新しい最小値に
19:             }
20:           }
21:           swap(itemTypeArray,i,minimum);//一番左の要素(ソート済み除く)と最小と入れ替
える
22:         }
23:       }
24:
25:       /**
26:       * 配列内の要素を入れ替える
27:       */
28:       protected void swap(ItemType[] itemTypeArray,int target1,int target2){
29:         ItemType temp;//temporary の箱を用意する
30:         temp = itemTypeArray[target1];
31:         itemTypeArray[target1] = itemTypeArray[target2];
32:         itemTypeArray[target2] = temp;
33:       }
34:
35:
36:     }
```

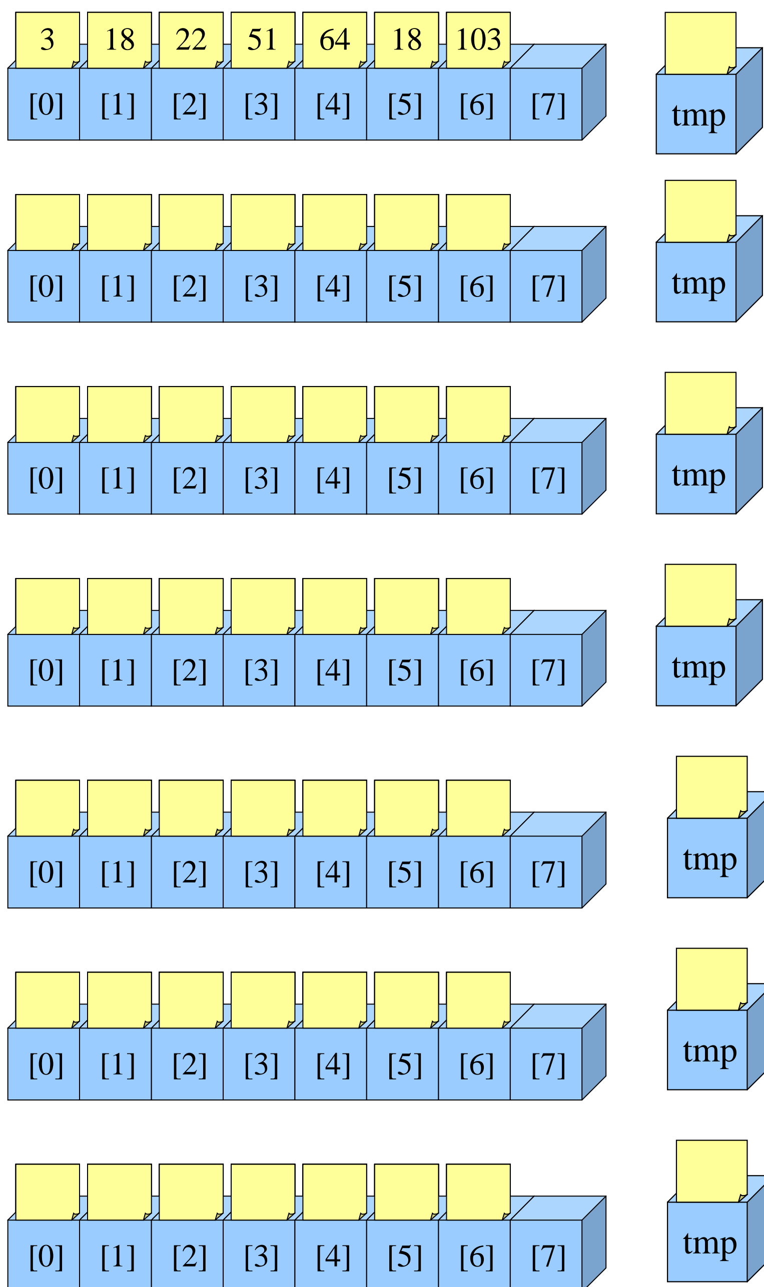
9.2.3. 挿入ソート

挿入ソートの手順は以下のようになります。

途中までソートが終わっているものとします。（その方が理解しやすい）

（１） ソートが終わっていない中で、一番左にある商品種類をソート済みの商品種類の中であるべき位置に挿入する。

（２） ソートの終わっていない部分（毎回小さくなる）に対して上のステップを繰り返す。



＜考えよう！＞挿入ソートの効率

挿入ソートの効率を考えてください。

（１）比較の回数

	比較の回数
２個の要素の時	□ 回
３個の要素の時	□ ＋ □ 回
４個の要素の時	□ ＋ □ ＋ □ 回
N個の要素の時	

（２）入れ替えの回数

	入れ替えの回数
２個の要素の時	□ 回
３個の要素の時	□ 回
４個の要素の時	□ 回
N個の要素の時	□ 回

＜考えよう！＞挿入ソートは場合によって効率が変わります。どんな時に効率が良くなるでしょうか。

-
-
-

①.挿入ソートの実装

挿入ソートの実装例を示します。

例題 9-1：ソートアルゴリズムの比較(InsertionSort.java)

```
1:      /**
2:      * オブジェクト指向哲学　入門編
3:      * 例題 9-1：ソートアルゴリズムの比較
4:      * 3 種類のソートアルゴリズムの性能を比較するプログラム
5:      *
6:      * 挿入ソートクラス
7:      */
8:      public class InsertionSort extends SortAlgorithm{
9:
10:       /**
11:       * 並び替え（ソート）をする
12:       */
13:       public void sort(ItemType[] itemTypeArray,int size){
14:         int target;//挿入する対象
15:         for(target = 1;target<size;target++){
16:           ItemType temp = itemTypeArray[target];//対象をコピーする
17:           int i=target;
18:           while(i>0 && itemTypeArray[i-1].getId() > temp.getId()){//挿入場所に出会うま
でシフトしつづける
19:             itemTypeArray[i] = itemTypeArray[i-1];//右へシフト
20:             i--;
21:           }
22:           itemTypeArray[i] = temp;//対象を挿入する
23:         }
24:       }
25:
26:     }
```

9.2.4.　挿入ソートの効率化

挿入ソートをあと少しだけ効率化することを考えましょう。前ページの挿入ソートのコードの中で、四角で囲った条件式のところに注目してみましょう。

```
while(i>0 && itemTypeArray[i-1].getId() > temp.getId())
```

この条件式はこれは、二つの条件式から構成されています

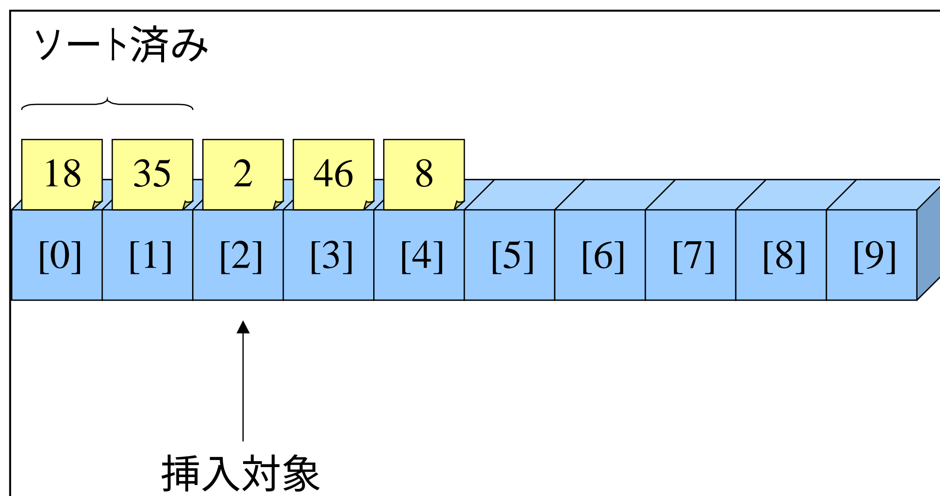
①「**i>0**」

i（走査している配列の番地）が **0** より大きい

②「**itemTypeArray[i-1].getId() > temp.getId()**」

配列の中の商品番号が対象商品番号より大きい

この①の条件は、ほとんど成り立ちません。にも関わらず N＾2／4 回だけ行われてしまいます。これは無駄ですね。

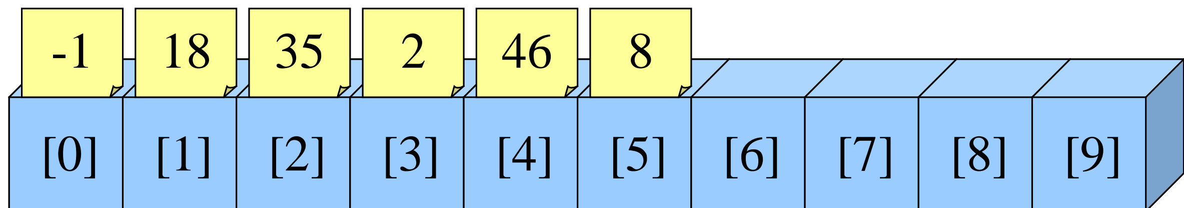


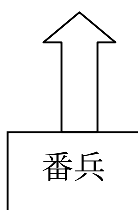
＜考えよう！＞①の条件を無くしたらどんな問題が起こるでしょう。

-
-
-
-

①.「番兵」の考え方

前ページの①の条件をなくしてもエラーをなくす方法があります。それが「番兵」という概念です。





このように配列の一番左に、「絶対に有り得ないような小さい商品番号」を入れておきます。例えばー１は絶対に有り得ない小さい商品番号なので、それを配列の０番目に入れます。こうすると①の条件を外してもエラーは発生しません。

番兵を利用した結果、**while** 文のアルゴリズムは以下のように変わります。

```
while(i>0 && itemTypeArray[i-1].id > temp.id)
```

```
while(itemTypeArray[i-1].id > temp.id)
```

番兵を使う時の注意点を示します。

(1)番兵を使うと、使える配列が一つ少なくなる

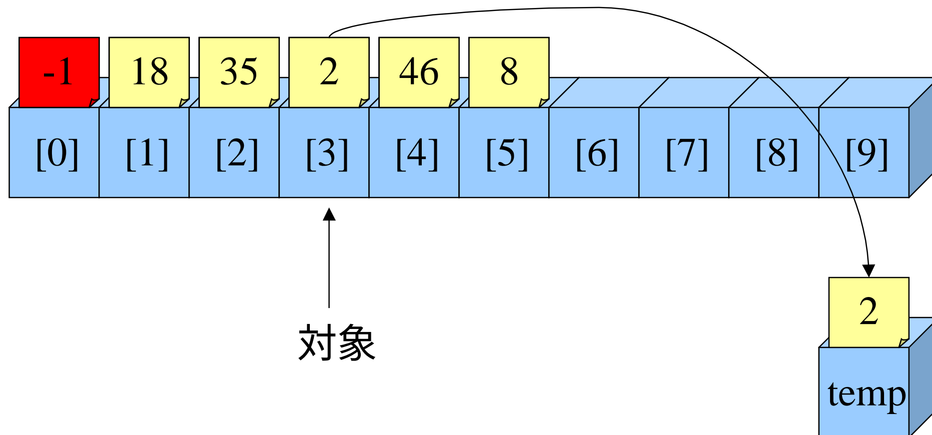
- 配列がいっぱいにならないようにするか、サイズを一つ増やした配列を用意するなどの対応を取るのが一般的です。

(2) ０番目は番兵用に空けるために、追加アルゴリズムの変更が必要です

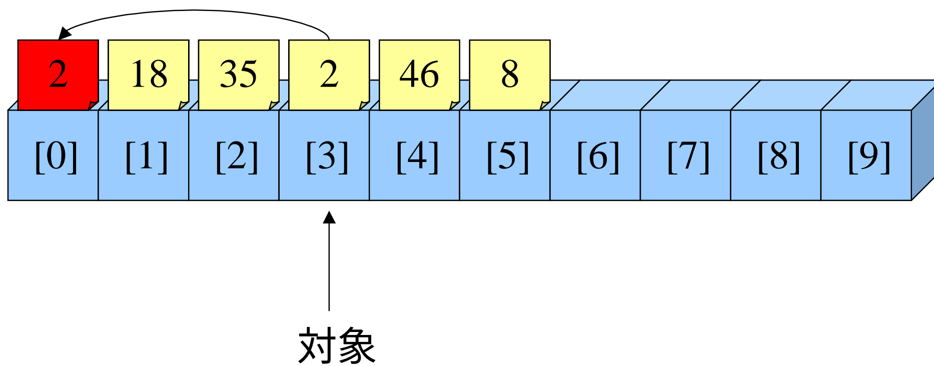
- ソートの時だけずらすやり方でまずはやってみましょう（ただし効率は少し落ちます）。
- 今回は配列の一番初めに番兵を立てますが、アルゴリズムによっては一番最後に番兵を立てる場合もあります。その場合は他のアルゴリズムの変更が不要になります。

②.さらにスマートなアルゴリズムに

さっきまでは以下のように **temp** の変数が必要でした。



ところがこの **temp** 変数を番兵用の場所に入れてもうまくいきます。このように **temp** 変数のメモリが必要なくなるために、少しだけ効率が上がります。



練習問題

＜記述問題＞

☆記述問題 9-1

　プログラム問題 **9-1** で作った性能比較プログラムの実行結果より、**3** 種類のソートの効率について考察せよ。

＜プログラム問題＞

☆プログラム問題 9-1

　例題 **9-1** を利用して、様々な条件で性能比較を行え。

☆プログラム問題 9-2

　例題 **9-1** を利用し、さらに番兵つきの挿入ソートができる **InsertionSortWithGuard** クラスを **SortAlgorithm** クラスのサブクラスとして実装し、性能を比較せよ。