第2節 バブルソート

バブルソート (bubble sort) は、隣り合う要素の大小を比較しながら整列させるソートのアルゴリズムである。最悪計算時間が $O(n^2)$ と遅いが、アルゴリズムが単純で実装が容易なため、また並列処理との親和性が高いことから、しばしば用いられる。安定な内部ソート。基本交換法、隣接交換法ともいう（単に交換法と言う場合もある)。バブルソートという名称は、最も大きい (あるいは小さい) 要素が列の中を一つ一つ順番に移動していく様子を、泡が浮かんでいく様子に例えたもの。

アルゴリズム （データ:『5 9 6 1 3 7 3 9 9』）

- $n = 9$ のデータを 9 番目 ～ 1 番目までを隣り合わせのデータどうしの大小を比較して一番小さなものを 1 番目に移動させる。
  1. 9 番目＜一番下＞のデータ「9」と 8 番目のデータ「9」を比較すると同じ値なので交換しない。
     処理後：5 9 6 1 3 7 3 9 9
  2. 8 番目のデータ「9」と 7 番目のデータ「3」を比較すると 7 番目のデータの方が小さいので交換しない。
     処理後：5 9 6 1 3 7 3 9 9
  3. 7 番目のデータ「3」と 6 番目のデータ「7」を比較すると 7 番目のデータの方大きいので、7 番目のデータ「3」と 6 番目のデータ「7」を交換する。
     処理後：5 9 6 1 3 3 7 9 9
  4. 6 番目のデータ「3」＜全ステップで交換＞と 5 番目のデータ「3」を比較すると同じ値なので交換しない。
     処理後：5 9 6 1 3 3 7 9 9

5. 5 番目のデータ「3」と 4 番目のデータ「1」を比較すると 4 番目のデータの方が小さいので交換しない。
   処理後：5 9 6 1 3 3 7 9 9
6. 4 番目のデータ「1」と 3 番目のデータ「6」を比較すると 3 番目のデータの方大きいので、4 番目のデータ「1」と 3 番目のデータ「6」を交換する。
   処理後：5 9 1 6 3 3 7 9 9
7. 3 番目のデータ「1」＜全ステップで交換＞と 2 番目のデータ「9」を比較すると 2 番目のデータの方大きいので、3 番目のデータ「1」と 2 番目のデータ「9」を交換する。
   処理後：5 1 9 6 3 3 7 9 9
8. 2 番目のデータ「1」＜全ステップで交換＞と 2 番目のデータ「5」を比較すると 1 番目のデータの方大きいので、2 番目のデータ「1」と 1 番目のデータ「5」を交換する。
   処理後：1 5 9 6 3 3 7 9 9
9. 1 番目のデータに達したので 1 回目の処理を終了する
   1 回目の処理結果：1 5 9 6 3 3 7 9 9

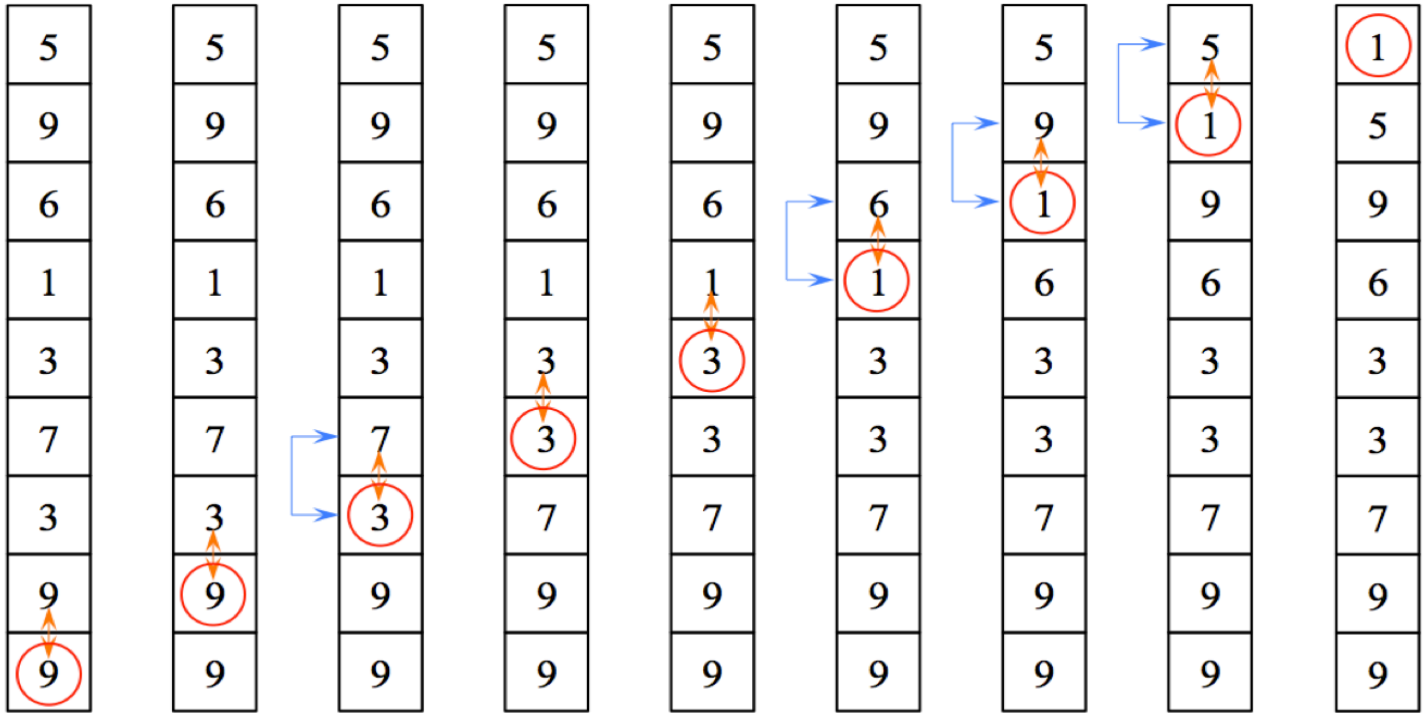

5 9 6 1 3 7 3 9 9
5 9 6 1 3 7 3 9 9
5 9 6 1 3 7 3 9 9
5 9 6 1 3 7 7 9 9
5 9 6 1 3 3 7 9 9
5 9 6 6 1 3 7 9 9
5 9 9 1 6 3 7 9 9
5 5 1 9 6 3 7 9 9
1 5 9 6 3 7 9 9

図 1.1　1 順目の整列

- 次に、9 番目 ～ 2 番目までを隣り合わせのデータどうしの大小を比較して一番小さなものを 2 番目に移動させる。

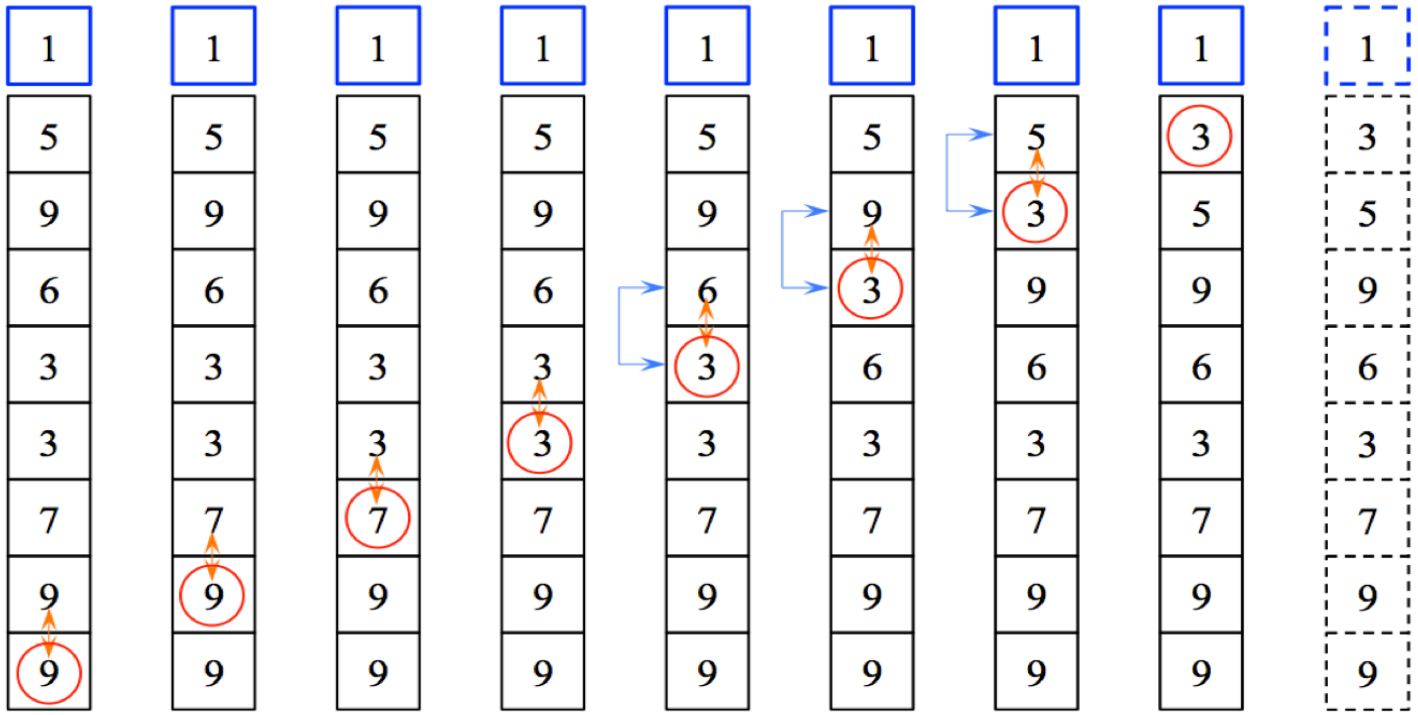


図 1.2　2 順目の整列

- 次に、9 番目 ～ 3 番目までを隣り合わせのデータどうしの大小を比較して一番小さなものを 3 番目に移動させる。

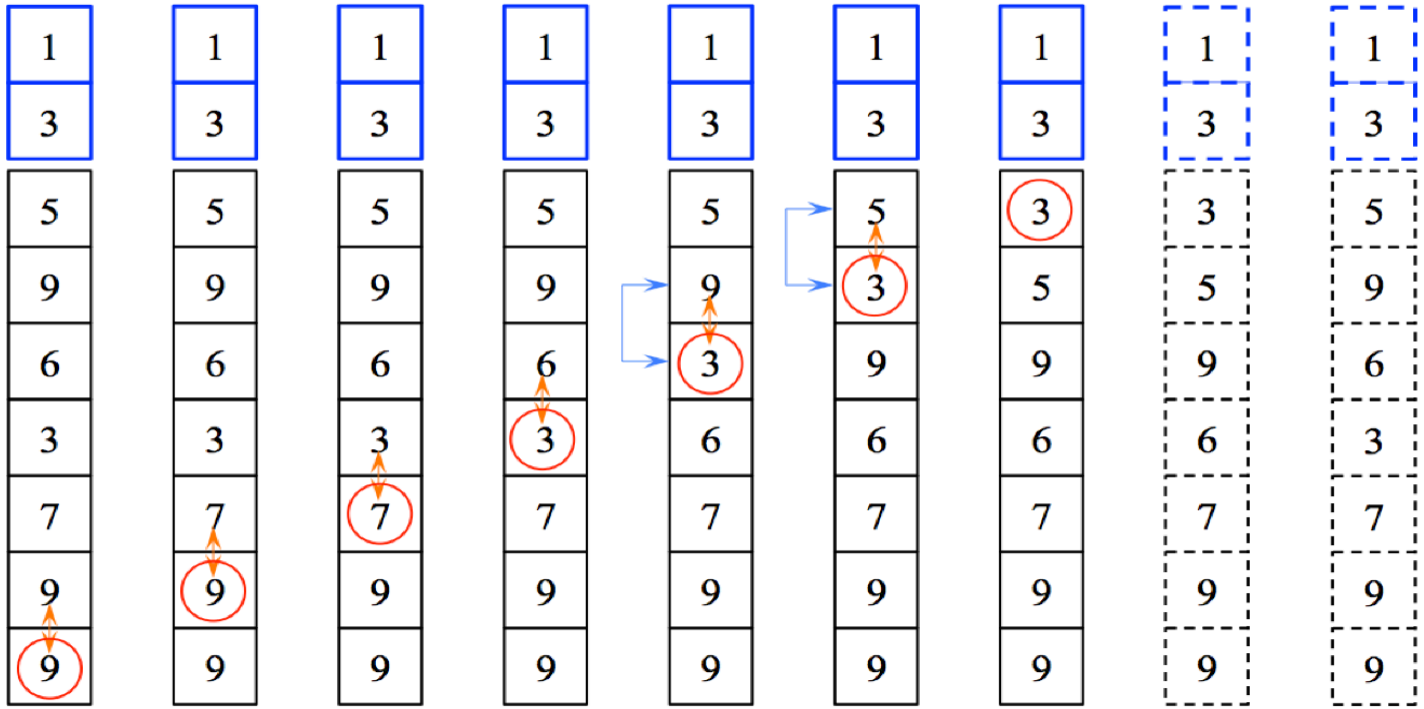


図 1.3　3 順目の整列

- ４回目の並び替え。９番目 ～４番目までを隣り合わせのデータどうしの大小を比較して一番小さなものを４番目に移動させる。

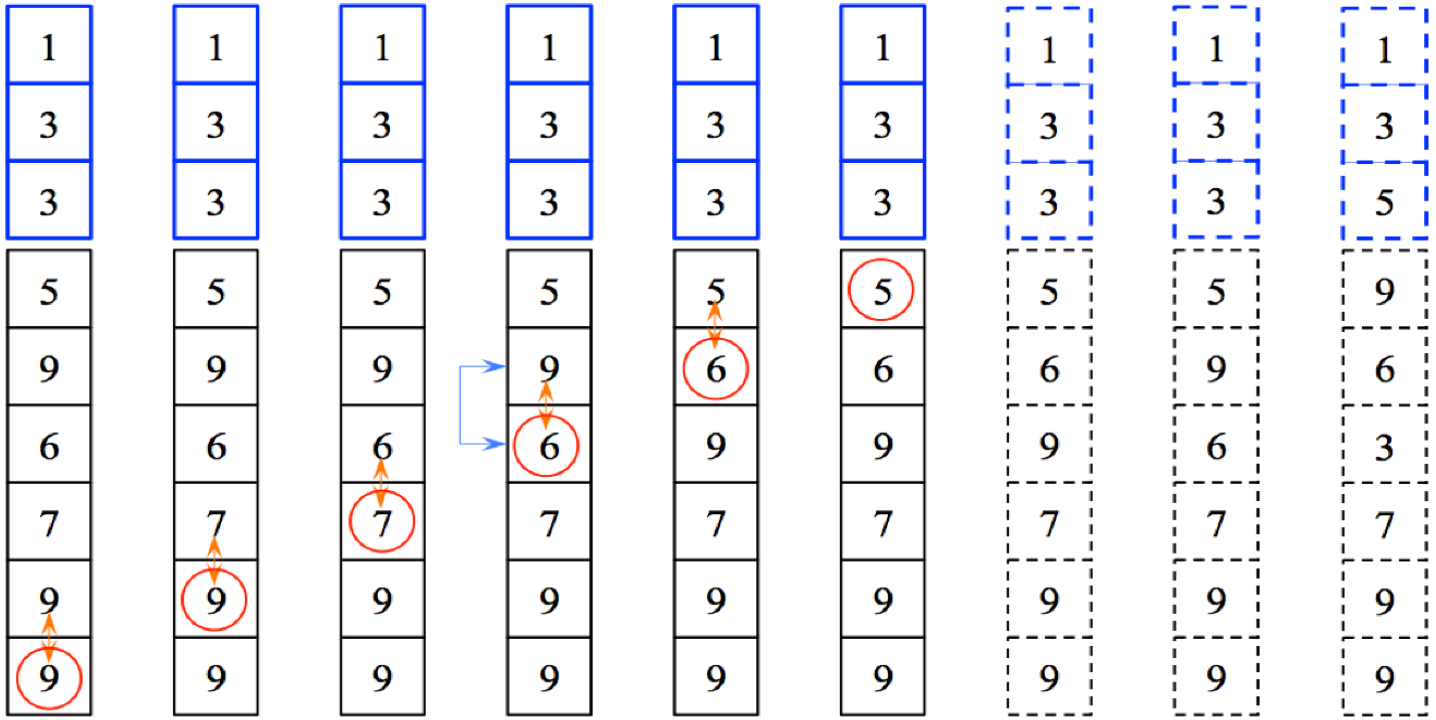


図 1.4　４順目の整列

- 5 回目の並び替え。9 番目 ～ 5 番目までを隣り合わせのデータどうしの大小を比較して一番小さなものを 5 番目に移動させる。

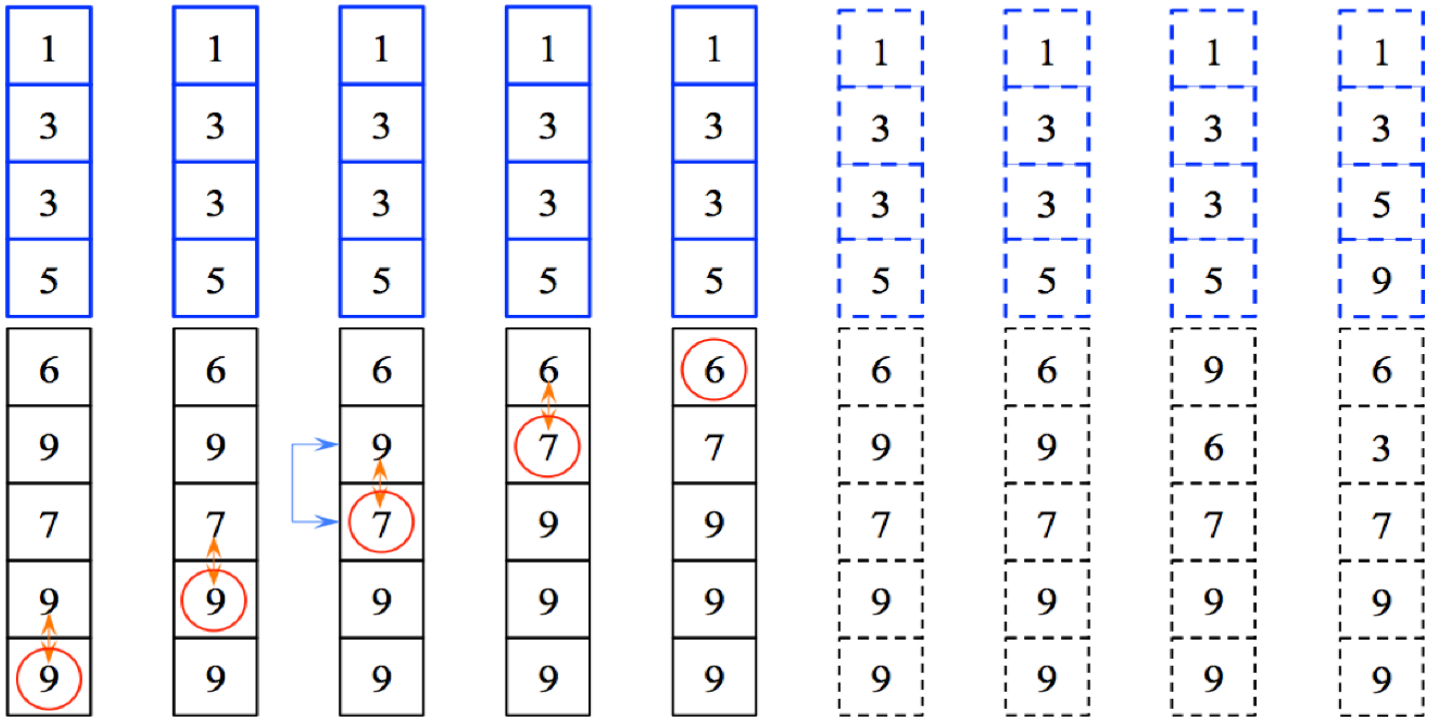


図 1.5　5 順目の整列

- 6 回目の並び替え。9 番目 ～ 6 番目までを隣り合わせのデータどうしの大小を比較して一番小さなものを 6 番目に移動させる。

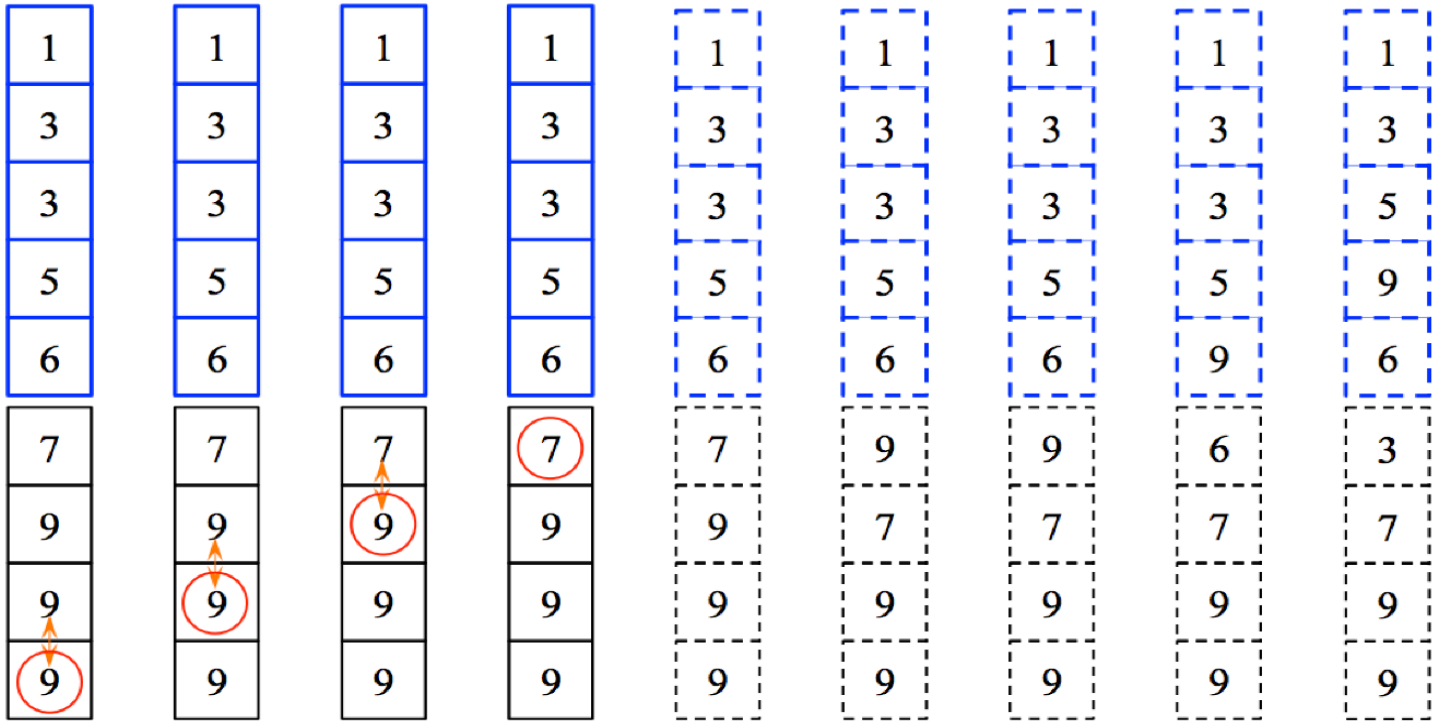


図 1.6　6 順目の整列

- 7 回目の並び替え。9 番目 ～ 7 番目までを隣り合わせのデータどうしの大小を比較して一番小さなものを 7 番目に移動させる。

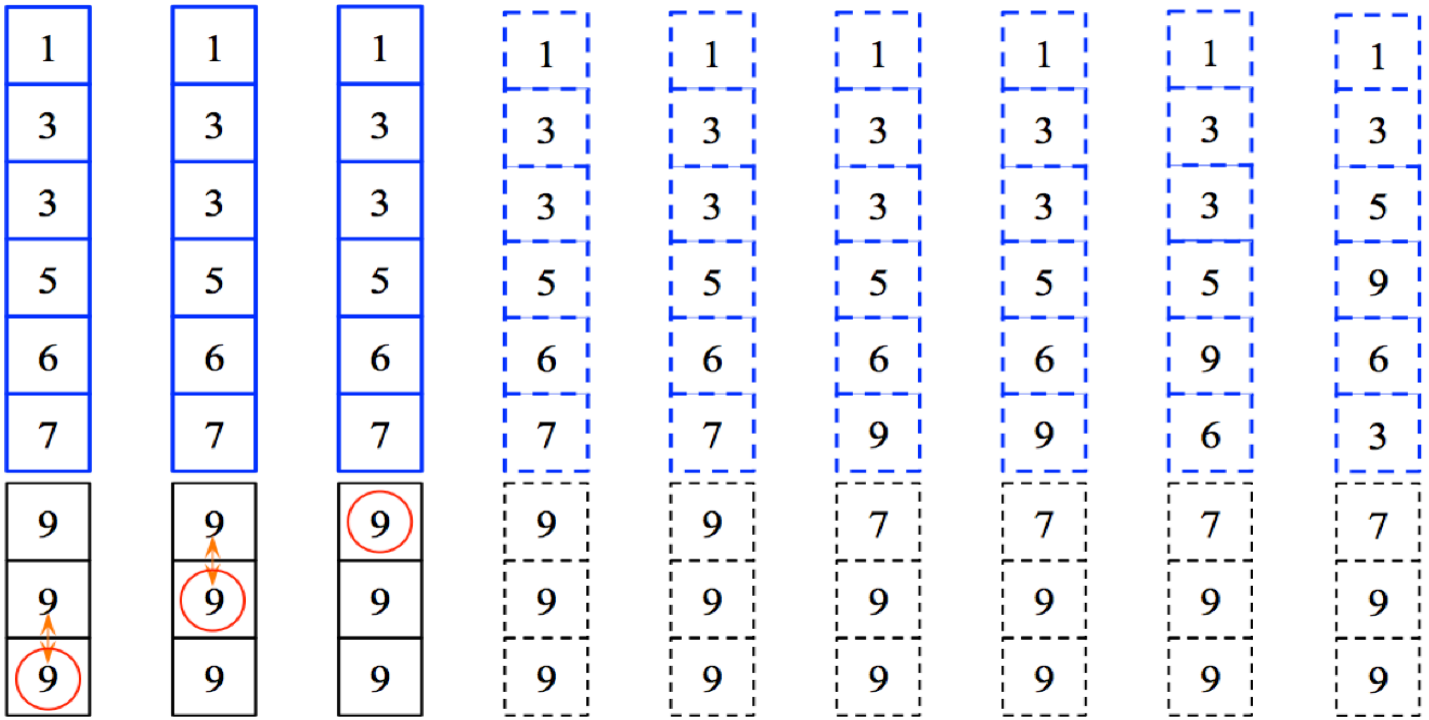


図 1.7　7 順目の整列

- 8 回目の並び替え．9 番目と 8 番目の隣り合わせのデータどうしの大小を比較して一番小さなものを 8 番目に移動させる。

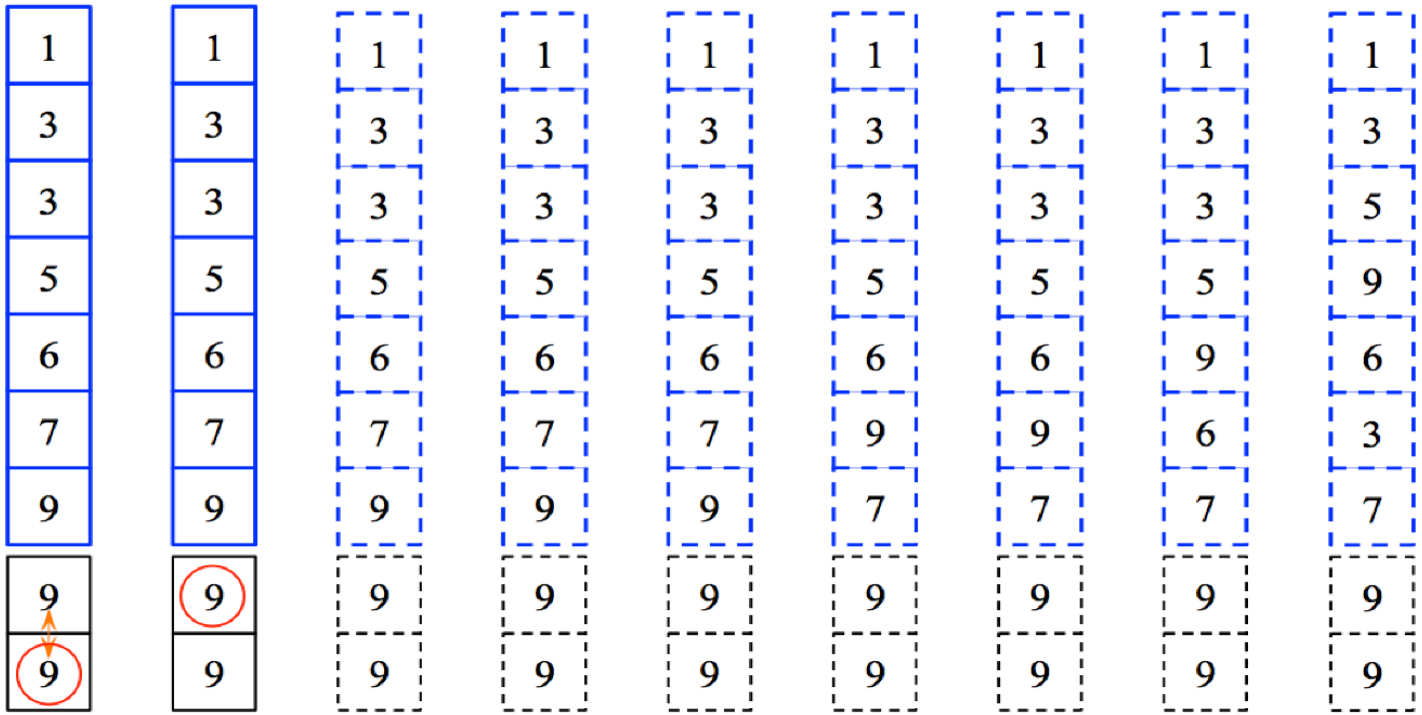


図 1.8　8 順目の整列

- 大小を比較するデータが無いので、処理を終了する

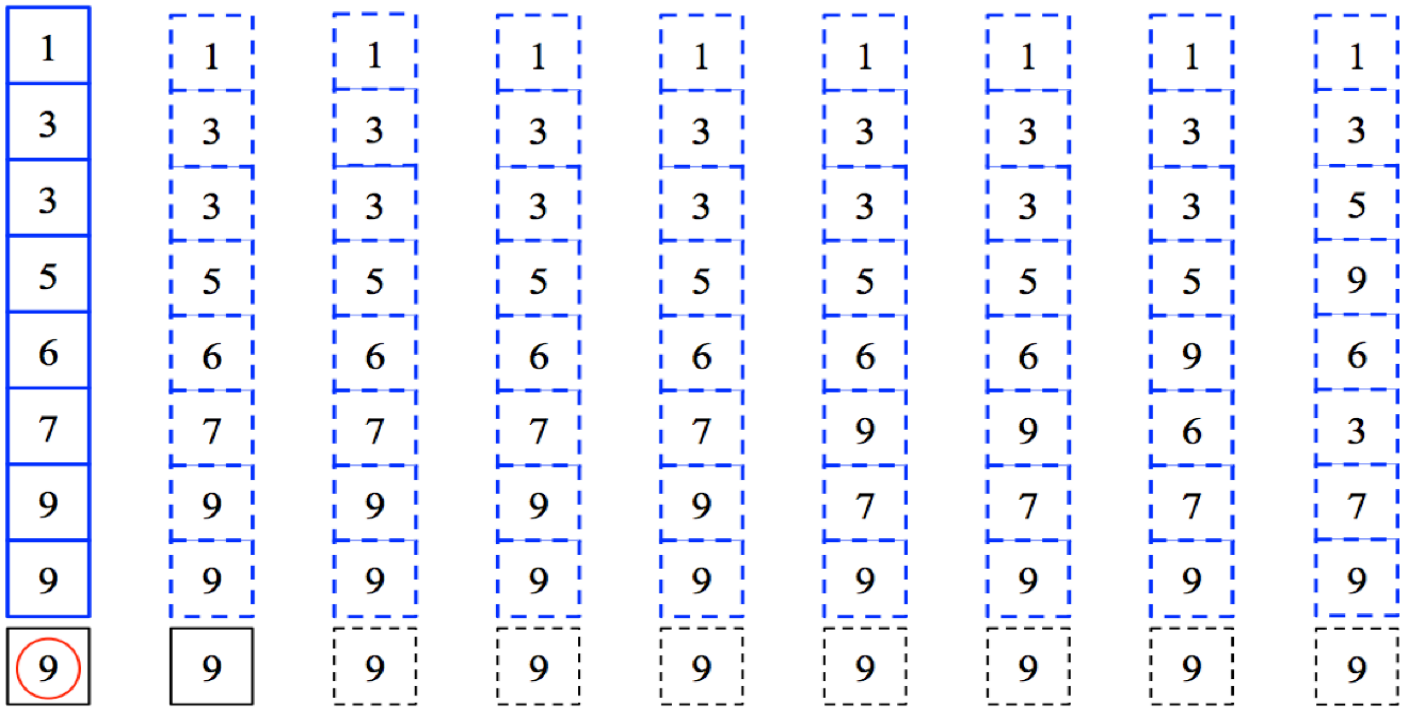


図 1.9　整列の終了

計算時間の評価

バブルソートの基本となっている処理は、2 つのデータの入れ替えなので、アルゴリズムでは比較し入れ替える操作である。

- n このデータでもっとも小さいデータを一番上まで運ぶのに、比較においては $n-1$ 回、入れ替えの場合においては最悪の時は $n-1$ 回ある。
- 2 番目に小さいデータを上から 2 番目まで運ぶのに、比較においては $n-2$ 回、入れ替えの場合においては最悪の時は $n-2$ 回ある。
- 3 番目に小さいデータを上から 2 番目まで運ぶのに、比較においては $n-3$ 回、入れ替えの場合においては最悪の時は $n-3$ 回ある。
- :
- :
- 最後から 2 番目に小さいデータを下から 2 番目に運ぶのに、比較においては 1 回、入れ替えの場合においては最悪の時は 1 回ある。

以上から、比較と入れ替えは合計以下と成る

$$(n-1)+(n-2)+(n-3)+\cdots+2+1=\sum_{i=1}^{n-1} i=\frac{n(n-1)}{2}=\mathrm{O}(n^2)$$

実装

```
int n=x.length;
for(int i=0;i<n-1;i++){
  for(int j=n-1;j>i;j--){
    if(x[j]<x[j-1]){
      int t=x[j];
      x[j]=x[j-1];
      x[j-1]=t;
    }
  }
}
```

図 1.10　バブルソートのフローチャート

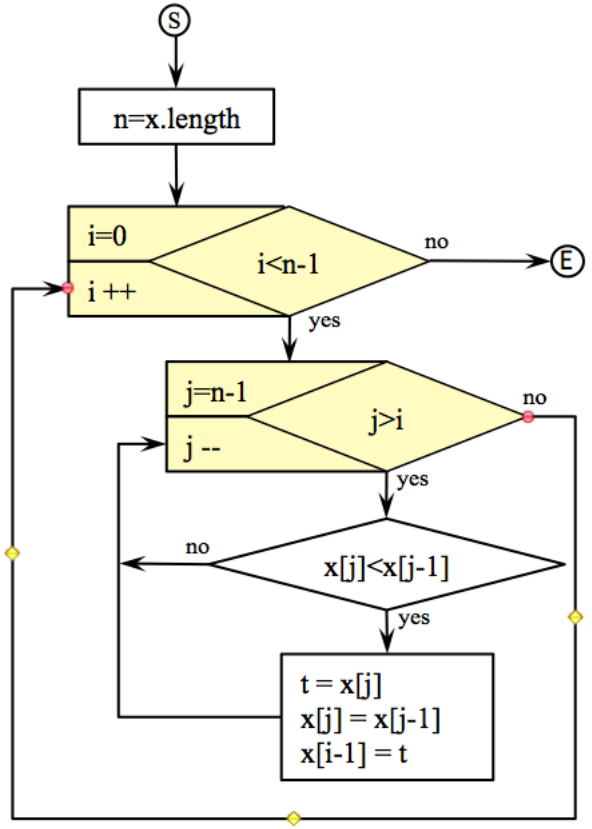
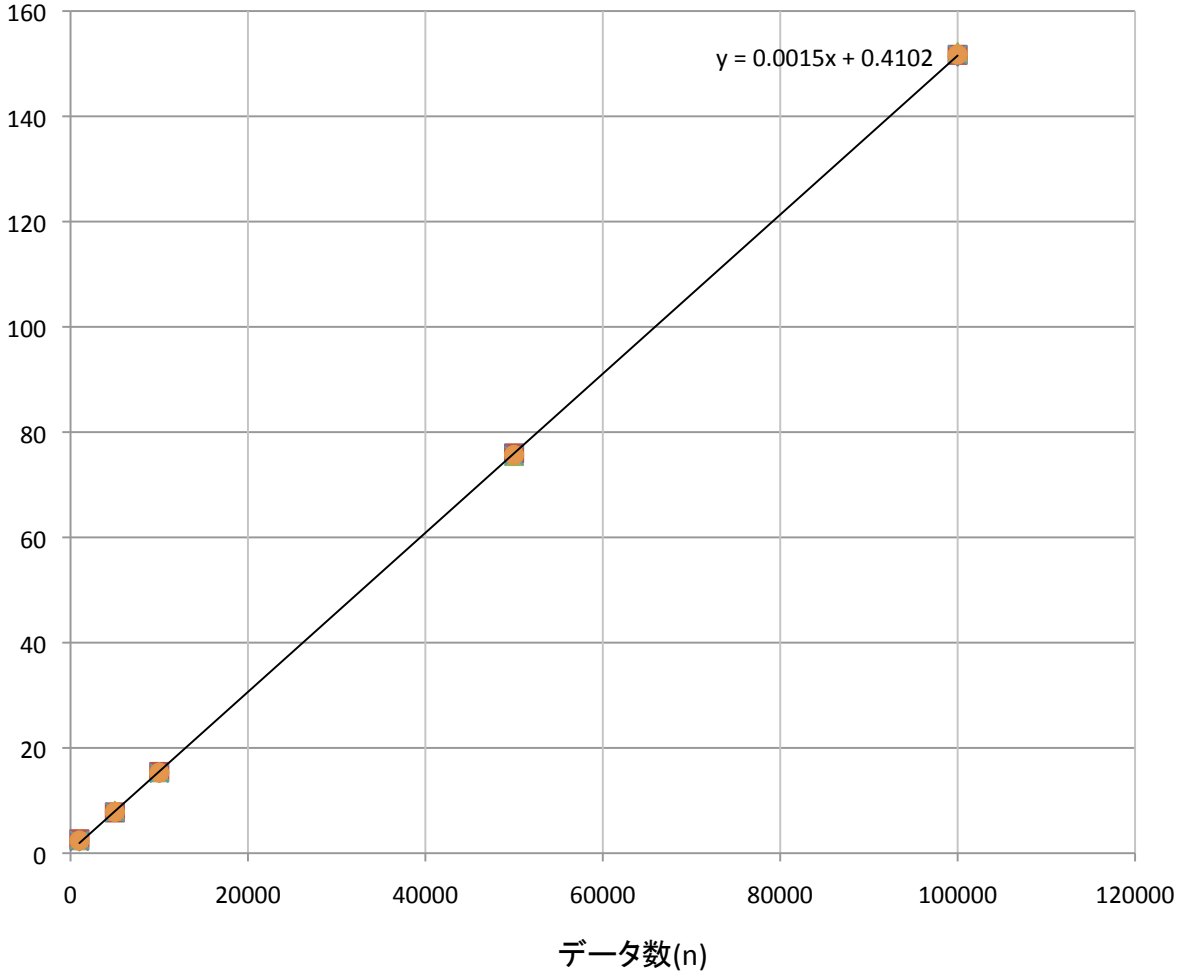


図 1.11　バブルソートのプログラム

計算時間の事後評価




```
import java.util.Random;
public class BubbleSort{
  public static void main(String[] args){
    int n= 100000;
    int x[] = new int[n];
    Random rnd = new Random();
    for(int i=0; i<n; i++){
       x[i] = rnd.nextInt(10000);
    }
    long start = System.currentTimeMillis();
    for(int i=0;i<n-1;i++){
      for(int j=n-1;j>i;j--){
        if(x[j]<x[j-1]){
          int t=x[j];
          x[j]=x[j-1];
          x[j-1]=t;
        }
      }
    }
    long stop = System.currentTimeMillis();
    System.out.println("cpu time = "+(stop-start)+" msec");
  }
}
```