BubbleSort

Prima di considerare questo algoritmo di ordinamento per array particolarmente importante definiamo in generale in che cosa consiste il problema di ordinamento di vettori:

Dato un vettore di elementi di un tipo ordinale (cioè, su cui è definita una relazione di ordine totale) si vogliono permutare gli elementi i suoi in modo che siano ordinati, per esempio in ordine crescente. Questo significa che, se il vettore S è di lunghezza N, avremo

$S[i] \leq S[i + 1]$ per $i = 1,...,N - 1$.

Esistono molti algoritmi per l'ordinamento di vettori (di elementi, come è ovvio, di tipo ordinale). Il metodo impiegato dall'algoritmo Bubble Sort si basa sull'idea seguente:

per ordinare la sequenza S = d c b a è sufficiente portare l'elemento massimo della sequenza all'ultimo posto ed ordinare la parte della sequenza compresa tra la prima e la penultima posizione. L'elemento massimo della sequenza viene determinato attraverso una serie di confronti ed eventuali scambi tra elementi adiacenti: se l' i-esimo elemento della sequenza è maggiore dell'i+1-esimo elemento, allora i due elementi vengono permutati.

Per esempio, se si vuole ordinare la sequenza S, si avranno i seguenti passaggi:

d c b a

c *d b* a

c b *d a*

c b a d (l'elemento massimo d è stato portato nella sua posizione definitiva, e si prosegue ordinando il (sotto)vettore c b a)

b *c a* d

b a c d (l'elemento c è stato portato nella sua posizione definitiva, e si prosegue ordinando il (sotto)vettore b a)

a b c d (l'elemento b è stato portato nella sua posizione definitiva e non resta piu` nulla da fare)

```
class ordinamento {

/* Un'implementazione dell'algoritmo di bubblesort in Java. */

	public static void main (String args[]) {

		int[] dati = new int[10];

		for (int i = 0;i < dati.length; i++) {
			dati[i] = SavitchIn.readLineInt();
		}
		for (int i = dati.length - 1; i >= 1; i--) {
			for (int j = 0; j < i; j++) {
```

```
                        if (dati[j] > dati[j+1]) {
                        int z;
                        z = dati[j];
                        dati[j] = dati[j+1];
                        dati[j+1] = z;
                        }
                  }
            }

            for (int i = 0; i < dati.length; i++) {
                  System.out.print (dati[i]);

            }
      } //fine main
}  //fine classe
```

Riconsideriamo ora l'implementazione dell'algoritmo di BubbleSort utilizzando i metodi, questa volta annotando il programma con le asserzioni che permettono di dimostrarne la correttezza:

```
class Ordinamento {

      public static int[] dati = new int[10];

      public static void main (String args[]) {

            System.out.println("Scrivi 10 numeri interi su righe diverse");
            leggi();
            ordina(dati);
            scrivi();
      } //fine main

      public static void bolli(int i) {
/**
Precondizione: 0 ≤ i < dati.length
Postcondizione: la posizione i del vettore dati contiene l'elemento massimo del
sottovettore di dati compreso tra le posizioni 0 ed i
*/

      for (int j = 0; j < i; j++)
            if (dati[j] > dati[j+1])
                        scambia(dati,j,j+1);
      }  //fine bolli

      public static void ordina(int[] dati){
/**
Precondizione: true
Postcondizione: il vettore dati è ordinato in ordine crescente
*/
      for (int i = dati.length - 1; i > 0; i--)
/**
Invariante: gli elementi di dati nelle posizioni  ≥ i e  < dati.length sono ordinati in
ordine crescente e sono ≥ di tutti gli elementi nelle posizioni ≥ 0 e < i
*/
                  bolli(i);
      }
```

```
	public static void leggi() {
		for (int i = 0;i < dati.length; i++) {
			dati[i] = SavitchIn.readLineInt();
		}  //fine for
	}  //fine leggi

	public static void scrivi() {
		for (int i = 0; i < dati.length; i++) {
			System.out.println (dati[i]);
		}  //fine for
	}  //fine scrivi

}  //fine classe
```

Questa è una versione di BubbleSort che fa uso di metodi ricorsivi:

```
static void bolli(int[] vet,int j,int i){
/**
Precondizione: 0 <= i,j < vet.length
Postcondizione: elemento massimo del sottovettore vet[j],…,vet[i] in posizione i
*/
	if (j < i) {
		if (vet[j] > vet[j+1])
			scambia(vet,j,j+1);
		bolli(vet,j+1,i);
	}
}

static void sort(int[] vet,int i){
/**
Precondizione:	0 <= i < vet.length
Postcondizione:	∀j,k in 0,…,i (j < k => vet[j] <= vet[k])
*/
	if (i >= 1) {
		bolli(vet,0,i);
		sort(vet,i-1);
	}
}
```

La correttezza del metodo sort equivale alla proposizione che sort(vet,i) soddisfa la postcondizione, per induzione su i:

Base: se i = 0, il metodo termina immediatamente, ma il sottovettore che contiene un solo elemento è ordinato in ordine crescente.

Passo induttivo: assumiamo (ipotesi induttiva) che la postcondizione del metodo valga per sottovettori di lunghezza < i, e consideriamo la chiamata sort(vet,i).
Allora la chiamata bolli(vet,0,i) porta il massimo del sottovettore vet[0],…,vet[i] in posizione i, per la postcondizione di bolli. Quindi vet[i] >= vet[j] per ogni j < i e per ipotesi induttiva la chiamata ricorsiva sort(vet,i–1) soddisfa la postcondizione, perciò per ogni j,k in 0,…,i (j < k => vet[j] <= vet[k]) da cui segue per ogni j,k in 0,…,i (j < k => vet[j] <= vet[k]).

Ora un paio di algoritmi di ordinamento (per array di numeri interi) che assumono che gli interi presenti nell'array siano compresi entro un intervallo noto a priori, poniamo 0,…,N. Allora si può procedere in questo modo: si crea un nuovo array di N+1 "secchi" in cui vengono contate le occorrenze di ciascun intero presente nell'array. Per ordinarlo, basta scorrere l'array dei secchi scrivendo l'intero corrispondente tante volte quanto è il valore contenuto nel secchio. Il primo algoritmo (Bucket Sort) implementa esattamente questa idea, il secondo (Counting Sort) la raffina osservando che per ordinare un vettore a non è necessario scorrere tutto l'array dei secchi: se in ogni secchio k viene scritto il numero di elementi che precedono l'elemento k nella permutazione ordinata dell'array a, basta scorrere a e, per ognuna delle sue posizioni i=0,…,a.length-1, scrivere l'elemento a[i] nella posizione corretta finale, che è quella calcolata usando il valore del secchio a[i].

Bucket Sort

```
class Bucket{
static final int max = 50;

static int[] secchi (int [] a){
/**
Precondizione: a array di interi nell'intervallo 0..max
Postcondizione: crea un array di lunghezza max+1 dove l'intero in posizione i
conta le occorrenze dell'intero i in a
*/
int[] s = new int[max+1];
for (int i = 0; i< s.length; i++)
       s[i] = 0;
for (int i = 0;i < a.length; i++)
       s[a[i]]++;
return s;
}

static void ordina(int[] s){
/**
Precondizione: s array di dimensione max+1
Postcondizione: scrive il contenuto di s a partire da 0 fino alla posizione s.length-1
in modo tale che, se s[i] = n, viene scritto n volte l'intero i
*/
for (int n = 0; n < s.length; n++)
       for (int k = 0; k < s[n]; k++)
              System.out.print(n + "   ");
System.out.println();
}

public static void scrivi(int[] a) {
              for (int i = 0; i < a.length; i++) {
                     System.out.println (a[i]);
              }
              System.out.println();
       }


public static void main(String[] args){
       System.out.print("Lunghezza array: ");
       int n = SavitchIn.readLineInt();
       int[] a = new int[n];
```

```
	for (int i = 0;i < a.length; i++) {
		System.out.print("Inserire intero: ");
		a[i] = SavitchIn.readLineInt();
		}
	scrivi(a);
	int [] s = secchi(a);
	ordina(s);

}
}
```

Una variante del Bucket Sort è la seguente:

Counting Sort

```
static int[] countingSort(int[] a, int l) {
	System.out.print("Valori compresi tra 0 e ");
	int k = SavitchIn.readLineInt();
	int[] c = new int[k];
// per contare quante volte compare ogni valore
	int[] b = new int[l]; // per il vettore ordinato
	for (int i = 0; i < c.length; i++)
		c[i] = 0; // inizializzazione di c
	for (int i = 0; i < a.length; i++)
		c[a[i]]++; // numero di elementi uguali ad a[i]
	for (int i = 1; i < c.length; i++)
		c[i] = c[i] + c[i - 1];
// numero di elementi minori o uguali ad i
	for (int i = a.length - 1; i >= 0; i--) {
		b[c[a[i]] - 1] = a[i];
		c[a[i]]--;
	}
	return b;
}
```

L'algoritmo seguente ordina un array (di interi) in ordine crescente scorrendolo dalla posizione 1 e cercando ad ogni posizione i di inserire l'elemento in posizione i nella posizione corretta j ≤ i. La dimostrazione di correttezza completa è contenuta in un file separato sulla pagina Moodle.

Insertion Sort

```
    public static void insertionSort(int[] dati) {
      int j,v;
      for (int i = 1; i < dati.length; i++) {
            v = dati[i];
            j = i;
            while (j > 0  && dati[j-1] > v) {
            /* Se viene usato & invece di &&, oppure se viene invertito
               l'ordine dei congiunti (anche con &&)  si ha un errore in esecuzione
               array out of bound */
                dati[j] = dati[j-1];
                j--;
            }
            dati[j] = v;
      }
    }
```

Ricerca Dicotomica – versione iterativa

```
public static int ricerca(int[] vet, int c) {
/**
Precondizione: l'array vet è ordinato in ordine crescente
Postcondizione: se c presente nell'array vet restituisce la sua posizione, altrimenti
restituisce -1
*/
       int sx = 0,dx = vet.length - 1,m;
       while (sx < dx) {
              m = (sx + dx) / 2;  // determina la posizione mediana del sottovettore
              if (vet[m] < c) // se c maggiore di quello in posizione mediana
                     sx = m + 1;  // cerca nella meta' di destra
              else if (vet[m] > c) // se c minore di quello in posizione mediana
                     dx = m - 1;  // cerca nella meta' di sinistra
              else // c, se presente, si trova nella posizione mediana dell'array
                     sx = dx = m;
       }
       if (vet[sx] == c)
              return sx;
       else
              return -1;
    } // fine metodo ricerca
```

Ricerca Dicotomica — versione ricorsiva

```
static int ricerca(int[] vet, int c, int sx, int dx){
/**
Precondizione: l'array vet è ordinato in ordine crescente; 0 ≤ sx,dx < vet.length
Postcondizione: se c presente in vet in una posizione compresa tra sx e dx restituisce
questa posizione, altrimenti restituisce -1
*/

              int m,r;
              if (sx < dx) {
                     m = (sx + dx) / 2;  // determina la posizione mediana dell'array
                     if (vet[m] < c) // se valore cercato maggiore di quello in posizione
mediana
                            r = ricerca(vet,c,m+1,dx);  // cerca nella meta' di destra
                     else  if  (vet[m]  >  c)  //  se  valore  cercato  minore  di  quello  in
posizione mediana
                            r = ricerca(vet,c,sx,m-1);  // cerca nella meta' di sinistra
                     else // l'elemento si trova nella posizione mediana dell'array
                            r = m;
              }
              else
              if (vet[sx] == c)
                     r = sx;
              else
                     r = -1;
              return r;
}
```

Variazioni sul tema della ricerca dicotomica (ricorsiva)

In questi esempi non è necessario assumere che l'array sia ordinato:

```
static int min(int[] vet, int sx, int dx){
```

```
/**
Precondizione: 0 ≤ sx,dx < vet.length
Postcondizione: min(vet,sx,dx) restituisce il minimo valore
                    del sottovettore di vet compreso tra le
                    posizioni sx e dx
*/
     int m;
       if (sx+1 < dx) {
              int m1,m2;
              int h = (sx + dx) / 2;
              m1 = min(vet,sx,h); // m1 minimo del sottovettore tra sx e h
              m2 = min(vet,h+1,dx); m2 minimo del sottovettore tra h+1 e dx
              if (m1 < m2)
                    m = m1;
              else
                    m = m2;
       }
       else
       if (sx == dx)
              m = vet[sx];
       else
              if (vet[sx] < vet[dx])
                    m = vet[sx];
              else
                    m = vet[dx];
       return m;
}


static int filter (int[] vet, int k, int sx, int dx){
/**
Precondizione: 0 ≤ sx,dx < vet.length
Postcondizione: filter(vet,k,sx,dx) restituisce il numero di elementi di vet > k nel
sottovettore compreso tra le posizioni sx e dx
*/
     if (sx < dx) {
       return filter(vet,k,sx,(sx+dx)/2) +
       filter(vet,k,(sx+dx)/2 + 1,dx);
     }
     else
     if (vet[sx] > k)
       return 1;
     else
       return 0;

}

static boolean filter1 (int[] vet, int k, int sx, int dx){
/**
Precondizione: 0 ≤ sx,dx < vet.length
Postcondizione: filter1(vet,k,sx,dx) determina se ci sono elementi di vet > k nel
sottovettore compreso tra le posizioni sx e dx
*/
     if (sx < dx) {
       return (filter1(vet,k,sx,(sx+dx)/2) ||
       filter1(vet,k,(sx+dx)/2 + 1,dx));
     }
     else
     if (vet[sx] > k)
```

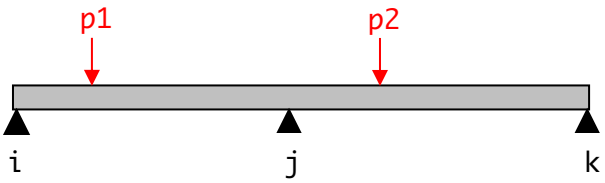
```
      return true;
    else
      return false;

}

static boolean filterAll (int[] vet, int k, int sx, int dx){
/**
Precondizione: 0 ≤ sx,dx < vet.length
Postcondizione: filterAll(vet,k,sx,dx) determina se tutti gli elementi di vet nel
sottovettore compreso tra le posizioni sx e dx sono > k
*/
    if (sx < dx) {
      return (filterAll(vet,k,sx,(sx+dx)/2) &&
      filterAll(vet,k,(sx+dx)/2 + 1,dx));
    }
    else
    if (vet[sx] > k)
      return true;
    else
      return false;

}


static int constant (int[] vet, int sx, int dx){
/**
Precondizione: 0 ≤ sx,dx < vet.length
Postcondizione: per un vettore di cifre binarie, determina se tutti gli elementi del
sottovettore compreso tra le posizioni sx e dx sono uguali restituendo -1 come valore
della chiamata ricorsiva quando la condizione non e' soddisfatta per il sottoarray su cui
avviene la chiamata, 0 se tutti gli elementi del sottoarray sono 0, 1 se tutti gli
elementi del sottoarray
   sono 1
*/
    int x = - 1;
    if (sx < dx) {
      int l = constant(vet,sx,(sx+dx)/2);
      int r = constant(vet,(sx+dx)/2+1,dx);
      if ((l == r) && (r >= 0)) x = l;
      else x = -1;
    }
    else
      x = vet[sx];
    return x;
}


static boolean increasing (int[] vet, int sx, int dx){
/**
Precondizione: 0 ≤ sx,dx < vet.length
Postcondizione: restituisce true se e solo se ∀i∀j (sx ≤ i,j ≤ dx → vet[i] ≤ vet[j])
*/
    if (sx < dx) {
      boolean l = increasing(vet,sx,(sx+dx)/2);
      boolean r = increasing(vet,(sx+dx)/2+1,dx);
      if (l && r && (vet[(sx+dx)/2] <= vet[(sx+dx)/2+1]))
            return true;
      else return false;
```

```
    }
    else
      return true;

}

static boolean crescente (int[] vet, int i){
/**
Precondizione: 0 ≤ i < vet.length
Postcondizione: restituisce true se e solo se
             ∀j∀k (i ≤ j,k < vet.length → vet[i] ≤ vet[j])
*/
    if (i < vet.length-1)
      return (vet[i] <= vet[i+1]) && crescente(vet,i+1);
    else
      return true;
}
```

L'algoritmo seguente ordina un vettore (di interi) dividendolo in due metà, ordinandole ricorsivamente, e ricomponendo le metà ordinate con un procedimento di fusione ordinata.

MergeSort

```
static void merge(int[] vet, int i, int j, int k) {
/**
Precondizione: 0 ≤ i,j,k < vet.length ∧
∀a∀b (i ≤ a ≤ b ≤ j → vet_out[a] ≤ vet_out[b]) ∧
∀a∀b (j+1 ≤ a ≤ b ≤ k → vet_out[a] ≤ vet_out[b])
Postcondizione:
∀a (i ≤ a ≤ k →
             (
               (∃b i ≤ b ≤ k ∧ vet_out[a] = vet_in[b])
             ∨ (∃b j+1 ≤ b ≤ k vet_out[a] = vet_in[b])
             )
    ∧
∀a∀b (i ≤ a,b ≤ k → vet_out[a] ≤ vet_out[b])
*/
  int[] aux = new int[vet.length];
  int p, p1 = i, p2 = j + 1;

  for (p = i; p <= k; p++)
/**
Invariante: p - i = (p1 – i) + (p2 – (j+1))
∧ ∀p > i (vet[p-1] ≤ vet[p1] ∧ vet[p-1] ≤ vet[p2])
*/
if (p1 > j) {
      aux[p] = vet[p2];
      p2++;
      }
    else if (p2 > k) {
      aux[p] = vet[p1];
      p1++;
      }
    else if (vet[p1] <= vet[p2]) {
      aux[p] = vet[p1];
      p1++;
      }
    else {
```



```
      aux[p] = vet[p2];
      p2++;
     }
for (p =  i; p <= k; p++)
      vet[p] = aux[p];
}

static void mergeSort(int[] vet, int sx, int dx) {
/**
Precondizione: 0 ≤ sx,dx < vet.length
Postcondizione: ∀a∀b (sx ≤ a ≤ b ≤ dx → vet[a] ≤ vet[b])
*/
  if (sx < dx) {
    mergeSort(vet, (sx+dx)/2 + 1, dx);
    mergeSort(vet, sx, (sx+dx)/2);
    merge(vet, sx, (sx+dx)/2, dx);
  }}
```

Semplici esercizi su metodi ricorsivi (su array)

Scrivere un metodo ricorsivo che stampa su stdout il contenuto di un array di interi:

```
static void stampa(int[] vet, int i){
       if (i < vet.length){
              System.out.print(vet[i]);
              stampa(vet,i+1);
       }
       else System.out.println();
}
```

Scrivere un metodo ricorsivo che restituisce il massimo elemento di un array di interi (di lunghezza > 0).

```
class Esercizio {
       public static int max(int[] a,int i){
              if (i < a.length-1) {
                     int m = max(a,i+1);
                     if (a[i] > m) return a[i];
                     else return m;
              }
              else return a[i];
       }
```

Scrivere un metodo ricorsivo che determina se tutti gli elementi di un array di interi sono pari.

```
public static boolean pari(int[] a,int i){
              if (i < a.length)
                     return (a[i] % 2 == 0) && pari (a,i+1);
              else return true;
       }
```

Scrivere un metodo per invertire il contenuto di un array di interi:

Versione iterativa

```
public static int[] reverse(int[] x){
int i = 0;
int aus;
while (i < s.length / 2) {
    aus = s[i];
    s[i] = s[s.length - (i+1)];
    s[s.length - (i+1)] = aux;
    i++;
    }
return s;
}
```

Versione ricorsiva

```
public static void reverse(int[] a,int i){
if (i < a.length/2) {
    int x = a[i];
    a[i] = a[(a.length - 1) - i];
    a[(a.length - 1) - i] = x;
    reverse(a,i+1);
    }
}
```

Cosa fanno questi metodi?

```
public static boolean EA(int [][] m){
boolean a=false,b;
for(int i = 0; i<m.length; i++){
       b = true;
       for (int j = 0; j < m[i].length; j++)
              b = b && m[i][j]%2==0;
       a = a || b;
       }
       return a;
}

public static boolean AE(int [][] m){
boolean a=true,b;
for(int i = 0; i<m.length; i++){
       b = false;
       for (int j = 0; j < m[i].length; j++)
              b = b || m[i][j]%2==0;
       a = a && b;
       }
       return a;
}
```