Università di Torino – Facoltà di Scienze MFN
Corso di Studi in Informatica

Programmazione I - corso B
a.a. 2009-10

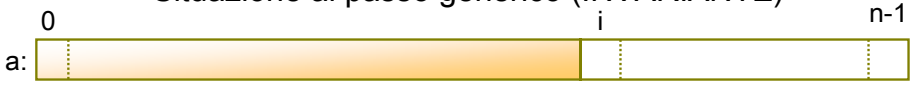
prof. Viviana Bono

Blocco 15 – Algoritmi su array: selection sort, insertion sort, fusione di due array ordinati

Algoritmo di ordinamento per selezione (*selection sort*)

Si cerca il minimo dell'array e lo si mette al primo posto (mettendo il primo al posto del minimo); poi si cerca il minimo nella parte di array dal secondo elemento alla fine, e lo si mette al secondo posto, e così via.

Situazione al passo generico (INVARIANTE)



a[0] <= a[1] <= ... <= a[i-1]

ovvero per 0 <= i <=n-1 si ha che:

la porzione dell'array a[0 .. i-1] e` ordinata

e

a[i-1] <= a[i], a[i-1] <= a[i+1], a[i-1] <= a[i+2], …, a[i-1] <= a[n-1]

ovvero che a[i-1] < di tutti gli el. della porzione a[i..n-1]

Selection sort

per mantenere l'invariante descritta nella slide precedente occorre mantenere anche la seguente (sotto)invariante:

a[i] <= a[i+1], a[i] < a[i+2], …, a[i]<a[j-1]

ovvero che per ogni i+1 <= j <= n-1 si ha che:

a[i] < di tutti gli elementi della porzione dell'array a[i+1.. j-1]

Questo corrisponde ad avere un ciclo annidato in cui se necessario avviene uno scambio di elementi (se a[i] > a[j]), per poter mantenere corrette la (sotto)invariante e l'invariante…

Selection sort

```
public static void selectionSort(int[] a){
  int n = a.length;
  for (int i=0; i<=n-1; i++)
     for (int j=i+1; j<=n-1; j++)
       if (a[i]>a[j]) scambia(a,i,j);

 }
```

NB nel metodo selectionSort il ciclo non controlla l'ultimo elemento, di indice n-1, poiché esso risulta automaticamente al posto giusto (essendo il massimo), come si può vedere anche dall'invariante ponendo i = n-1:

a[0] <= a[1] <= ... <= a[n-2]
a[n-2] <= a[n-1]

Tanto vale comunque non entrare neppure nel ciclo esterno per n-1, quindi ciclare per i<=n-2 nel ciclo esterno.

Un'altra ottimizzazione e` quella di fare solo uno scambio…

Selection sort ottimizzato

```
public static void selectionSort(int[] a) {
   int n = a.length;
   int posmin;
   for (int i=0; i<=n-2; i++) {
     posmin = i;
     for (int j=i+1; j<=n-1; j++)
       if (a[posmin]>a[j]) posmin = j;
     scambia(a,i,posmin);
   }
  }
```

Selection sort: predicato non INVARIANTE

Situazione al passo generico DOPO aver fatto almeno un passo (predicato su a[i] e non su a[i-1])

a: a[0] <= a[1] <= ... <= a[i]

Ovvero, con 0 <= i <=n-1, si ha che:

la porzione dell'array a[0 .. i] e` ordinata

e

a[i] <= a[i+1], a[i] <= a[i+2], …, a[i] <= a[n-1]

ovvero che a[i] < di tutti gli el. della porzione a[i+1..n-1]

Infatti questo non e` vero prima dell'esecuzione del ciclo: per i=0, cioe` prima di aver eseguito un passo di computazione, non si ha che a[0]< tutti gli elementi della porzione a[1..n-1]

Note sul predicato non INVARIANTE

- Scrivere un predicato che e` vero solo durante e dopo l'esecuzione del ciclo, MA non prima, almeno per il selectionSort non ci crea problemi, perche' per l'algoritmo non abbiamo previsto particolari inizializzazioni di variabili facenti parte del predicato di correttezza fuori dal ciclo
- Pero` e` sempre meglio cercare di scrivere una vera invariante, cioe` un predicato che e` vero prima, durante e dopo l'esecuzione del ciclo, in modo che si possa scrivere il programma corrispondente in modo corretto: l'esempio del max insegna!

Tempo di calcolo di selectionSort

Per cercare il minimo, il metodo deve percorrere interamente la parte di array dall'indice i all'indice n-1; la proc. di ordinamento effettua quindi n – i passi.

Il for piu' esterno viene eseguito la prima volta per i = 0, poi per i = 1, per i = 2, ecc., e per ogni i viene eseguito il ciclo interno; il numero totale di passi effettuato dall'algoritmo è quindi:

$$n + (n – 1) + (n – 2) + (n – 3) + ... + 3 + 2$$

cioè (formula di "Gauss bambino"):

$$n(n+1)/2 – 1 \text{ cioè } (n^2 + n)/2 – 1$$

Il tempo di calcolo cresce in modo quadratico rispetto alla lunghezza dell'array da ordinare, cioè è circa proporzionale al quadrato di tale lunghezza: se la lunghezza raddoppia, il tempo si quadruplica; se la lunghezza si triplica, il tempo si moltiplica per 9; e così via.

Algoritmo di ordinamento per inserimento (*insertion sort*)

Si considerano porzioni ordinate di array via via piu' grandi di un elemento e si cerca la posizione in cui *inserire* l'elemento che sta nella posizione che segue l'ultima posizione della sequenza ordinata, creando una sequenza ordinata piu' lunga di un elemento, fino all'esaurimento degli elementi

Es. 5 | 3 | 7 | 4 | 1

1 - La sequenza rappresentata dal solo elemento 5 e` ordinata

2 - Devo ordinare la sequenza 5, 3 (el da inserire = 3):

3 | 5 | 7 | 4 | 1

3 - Devo ordinare la sequenza 3, 5, 7 (el da inserire = 7): e` gia` ordinata

4 - Devo ordinare la sequenza 3, 5, 7, 4 (el da inserire = 4):

3 | 5 | 7 | 7 | 1 (ho spostato 7 indietro)

3 | 5 | 5 | 7 | 1 (ho spostato 5 indietro, trovando il posto giusto per 4)

3 | 4 | 5 | 7 | 1 (ho messo 4 nella posizione corretta)

5 – Devo ordinare la sequenza 3, 4, 5, 7, 1 (el da inserire = 1):

…

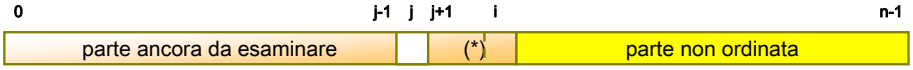
3 | 3 | 4 | 5 | 7 (ho spostato 7, 5, 4, 3 indietro, trovando il posto giusto per 1, che coincide con pos = 0)

1 | 3 | 4 | 5 | 7 (ho messo 1 nella posizione corretta)

Insertion sort

Situazione al passo generico (INVARIANTE)

`x = a[i]` è l'elemento da inserire in modo ordinato in `a[0..i-1]` e `a[0..i-1]` è ordinata



è `x` < `a[j+1]` ≦ `a[j+2]` ≦ `a[j+3]` ≦ ... ≦ `a[i]`

[(*): parte gia` spostata, `a[i]` non e` piu' `x` se ci sono stati spostamenti]

a[j] è il posto libero (che si è creato spostando a destra di uno tutti gli elementi successivi);

il ciclo termina se `j` = 0 oppure se (`j` > 0 e) `x` ≧ `a[j-1]` :

in entrambi i casi il posto libero ha raggiunto la posizione giusta, e il valore x va inserito in tale posto;

altrimenti l'elemento `a[j-1]` va spostato in `a[j]`, il che equivale a spostare a sinistra di uno il posto libero

L'invariante del ciclo piu' interno

`0 ≦ j ≦ i`

∧

`a[j+1], ... a[i]`

è uguale alla parte di array ordinato originariamente in

`a[j], ... a[i-1]`

∧

`x < a[j+1]`

Si esce dal ciclo per j = 0 oppure x >= a[j-1];
quindi, anche ricordando de Morgan,
si continua per j > 0 and x < a[j-1];
In conclusione:

```
while(j > 0 && x < a[j-1]) {
  a[j] = a[j-1];
  j--;
}
a[j] = x;
```

Inizializzazione: devo inserire l'elemento di posizione i fra quelli precedenti, quindi la posizione iniziale del posto libero è i:

```
j = i;
```


Insertion sort

```
public static void insertionSort(int[] a) {
   int n = a.length;
   for(int i = 0; i < n; i++) {
   int next = a[i];
   int j = i
   while( j > 0 && next < a[j-1]) {
     a[j] = a[j-1];
     j--;
   }
   a[j] = next;

   }
}
```

Tempo di calcolo di insertionSort

Il corpo della procedura è costituito da un ciclo for di n passi, senza uscite forzate. Tuttavia all'interno del ciclo principale viene eseguito un ciclo interno il cui tempo di calcolo non è costante, ma in generale dipende dal parametro i.

In media, il ciclo interno farà un numero di passi circa uguale a i/2, cioè 1/2 la prima volta, 2/2 la seconda, 3/2 la terza, ...;

approssimativamente, il numero totale di passi sarà allora:

$1/2\ (1 + 2 + 3 + ... + n) = 1/2\ (n(n+1)/2) = (n^2 + n)/4$

L'algoritmo e` in media quadratico.

Tempo di calcolo del ciclo interno di insertionSort

caso migliore: il valore da inserire è maggiore o uguale all'ultimo della sequenza gia` ordinata (e quindi a tutti);
il calcolo richiede un solo passo;

caso peggiore: il valore da inserire è minore di tutti i presenti,
quindi esso sarà confrontato con tutti gli elementi della sequenza gia` ordinata, i quali dovranno essere tutti spostati di uno;
il numero di passi di calcolo è proporzionale a n
(ossia lineare in n)

in media, come detto, il valore da inserire sarà a metà della sequenza ordinata, quindi il numero di passi sarà circa proporzionale a n/2, cioè ancora proporzionale a n (o lineare in n)

Tempo di calcolo di insertionSort

caso peggiore: l'array di partenza è ordinato inversamente; allora ogni ciclo interno esegue esattamente k passi, con k successiv. uguale a 1, 2, ...,n (verificare simulando a mano l'esecuzione); il numero totale di passi è quindi:

$1+2+3+ \ldots + n = n(n+1)/2 = (n^2 + n)/2$

quadratico

caso migliore: l'array di partenza è già ordinato; allora ogni inserimento fatto dal ciclo interno, inserendo un elemento che è maggiore o uguale dei precedenti, fa un solo passo; il numero totale di passi è quindi

1+1+1+ ... (n volte)

cioè proporzionale a n (o lineare in n)

Tempo di calcolo di insertionSort

Riassumendo:

- caso peggiore: tempo quadratico (rispetto alla lunghezza)
- caso medio: tempo quadratico
- caso migliore: tempo lineare

Note sui tempi di calcolo di insertionSort e selectionSort

Per il selectionSort, a differenza dell'insertionSort, non esistono un caso peggiore e un caso migliore: il numero di passi eseguiti non dipende dal modo in cui sono disposti i valori nell'array, ma è sempre rigorosamente lo stesso, ed è quadratico.

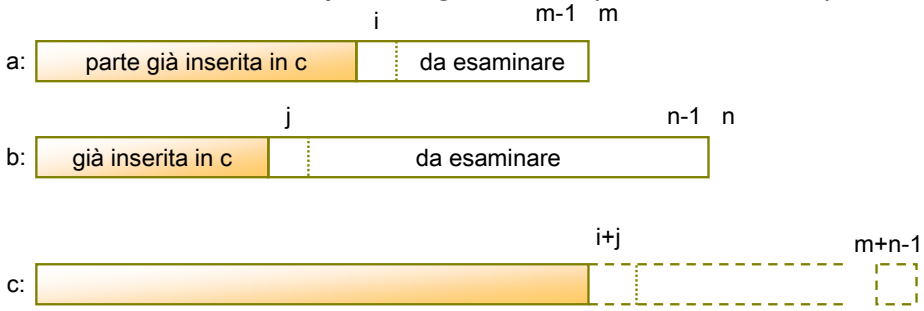
Il selection sort è quindi un algoritmo peggiore dell' insertion sort, che invece, come abbiamo visto, pur essendo anch'esso in media quadratico presenta un caso migliore in cui è lineare. Vedrete altri algoritmi di ordinamento migliori di entrambi (es. quick sort)

Fusione ordinata di due array ordinati (*merge*)

Definire una procedura (metodo statico) la quale, presi come argomenti due array pieni ordinati (che possono contenere elementi ripetuti), crea e restituisce un nuovo array pieno ordinato (eventualmente contenente elementi ripetuti) dato dalla fusione ordinata dei due array-argomenti (senza modificare tali array-argomenti).

Merge

Situazione al passo generico (INVARIANTE)



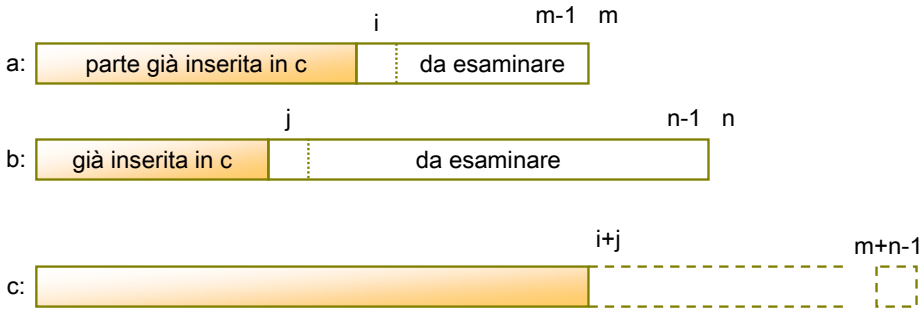
a[0] <= a[1] <= ... a[i] <= a[i+1] <= a[i+2] <=... <= a[m-1]

b[0] <= b[1] <= ... b[j] <= b[j+1] <= b[j+2] <=... <= b[n-1]

c[0] <= c[1] <= ...<= c[i+j-1] c[i+j-1] <= a[i] c[i+j-1] <= b[j]

la parte di array c fra gl'indici 0 e i+j-1 inclusi contiene tutti e soli gli elementi di a e di b fino agl'indici rispettivamente i-1 e j-1 (inclusi), fusi ordinatamente.

corpo del ciclo



caso 1: `a[i] <= b[j]` allora `a[i]` è <= di tutti gli elementi di b (e di a) ancora da esaminare, quindi può essere copiato in c;

caso 2: `a[i] > b[j]` allora `b[j]` è < di tutti gli elementi di a (e di b) ancora da esaminare, quindi deve essere copiato in c.

condizione

Il ciclo deve continuare per tutto il tempo in cui entrambi gli array a e b hanno ancora elementi da esaminare: `i < m && j < n`

All'uscita dal ciclo occorre ancora copiare in c la parte rimanente dell'array non completamente scandito.

A tal fine è inutile scrivere un if-else per sapere quale array ha ancora elementi da copiare: basta scrivere due while successivi (NON annidati!), e il test dei while, che comunque ci deve essere, al primo passo effettua "automaticamente" proprio il controllo che si farebbe con l'if.

```
while(i < m) {c[i+j] = a[i]; i++; }
while(j < n) {c[i+j] = b[j]; j++; }
```


Merge

```
public static int[] fondi(int[] a, int[] b) {
  int m = a.length; int n = b.length;
  int[] c = new int[m+n];
  int i = 0, j = 0;
  while(i < m && j < n) {
    if(a[i] <= b[j]) {
      c[i+j] = a[i];
      i++;
    }
    else {
      c[i+j] = b[j];
      j++;
    }
  }
  while(i < m) {c[i+j] = a[i]; i++; }
  while(j < n) {c[i+j] = b[j]; j++; }
  return c;
}
```

Complessità computazionale dell'algoritmo di fusione

L'algoritmo percorre interamente i due array, una volta sola; il numero di passi è pertanto proporzionale alla somma m+n delle lunghezze dei due array. È quindi un algoritmo lineare (se assumiamo che i due array abbiano approssimativamente la stessa lunghezza n, il numero di passi è 2n, quindi proporzionale a n).