Corso di Tecniche di Programmazione

Corsi di Laurea in Ingegneria Informatica ed Automatica

Anno Accademico 2003/2004

Proff. Giuseppe De Giacomo, Luca Iocchi, Domenico Lembo

Dispensa 2: Algoritmi di Ordinamento

Ordinamento di un array

Problema: Data una sequenza di elementi in ordine qualsiasi, ordinarla.

Questo è un problema fondamentale, che si presenta in moltissimi contesti, ed in diverse forme:

- ordinamento degli elementi di un array in memoria centrale
- ordinamento di una collezione in memoria centrale
- ordinamento di informazioni memorizzate in un file su memoria di massa (file ad accesso casuale su disco, file ad accesso sequenziale su disco o su nastro)

Noi consideriamo solo la variante base del problema dell'ordinamento.

Problema: Dato un vettore a di n elementi in memoria centrale non ordinato, ordinarne gli elementi in ordine crescente.

Ordinamento per selezione del minimo (selection sort)

Esempio: Ordinamento di una pila di carte:

- seleziono la carta più piccola e la metto da parte
- delle rimanenti seleziono la più piccola e la metto da parte
- ...
- mi fermo quando rimango con una sola carta

Quando ho un vettore:

- per selezionare l'elemento più piccolo tra quelli rimanenti uso un ciclo
- "mettere da parte" significa scambiare con l'elemento che si trova nella posizione che compete a quello selezionato

Ordinamento per selezione del minimo (selection sort)

Implementazione:

```
public static void selectionSort(int[] a) {
  int n = a.length;
  for (int i=0; i<n-1; i++) {
    // trova il piu' piccolo elemento da i a n-1
    int jmin = i;
    for (int j=i+1; j<n; j++) {
      if (a[j]<a[jmin])
        jmin = j;
    }
    // scambia gli elementi i e jmin
    int aux = a[jmin];
    a[jmin] = a[i];
    a[i] = aux;
  }
}
```

Complessità dell'ordinamento per selezione

Doppio ciclo $\Longrightarrow O(n^2)$

Più in dettaglio: Istruzione dominante è il confronto `(a[j] < a[jmin])`.

Viene eseguita $(n-1)+(n-2)+\cdots+2+1=\frac{n\cdot(n-1)}{2}$ volte.
$\Longrightarrow O(n^2)$ operazioni

La complessità è $O(n^2)$ in tutti i casi (anche se il vettore è già ordinato).

Numero di scambi:

- nel caso migliore: 0
- nel caso peggiore: $O(n)$

5	0	1	2	3	4
0	1	2	3	4	5

Ordinamento a bolle (bubble sort)

Idea: Si fanno "salire" gli elementi più piccoli ("più leggeri") verso l'inizio del vettore ("verso l'alto"), scambiandoli con quelli adiacenti.

L'ordinamento è suddiviso in $n-1$ fasi:

- fase 0: 0^o elemento (il più piccolo) in posizione 0
- fase 1: 1^o elemento in posizione 1
- $\cdots$
- fase $n-2$: $(n-2)^o$ elemento in posizione $n-2$, e quindi
  $(n-1)^o$ elemento in posizione $n-1$

Nella fase i: cominciamo a confrontare **dal basso** e portiamo l'elemento più piccolo (più leggero) in posizione i

Ordinamento a bolle (bubble sort)

Implementazione (semplificata):

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


Osservazione: Se durante una fase non avviene alcuno scambio, allora il vettore è ordinato. Se invece avviene almeno uno scambio, non sappiamo se il vettore è ordinato o meno.

$\Longrightarrow$ Possiamo interrompere l'ordinamento appena durante una fase non avviene alcuno scambio.

Implementazione:

```
public static void bubbleSort(int[] a) {
  int n = a.length; boolean ordinato = false;
  for (int i=0; i<n-1 && !ordinato; i++) {
    ordinato = true;
    for (int j=n-1; j>i; j--)
      if (a[j-1] > a[j]) {
        int aux = a[j-1];
        a[j-1] = a[j];
        a[j] = aux;
        ordinato=false;
      }
  }
}
```

Complessità della versione ottimizzata

- nel caso migliore (vettore già ordinato correttamente):

  $O(n)$ confronti, 0 scambi

- nel caso peggiore (vettore ordinato al contrario):

  $n-1$ fasi $\Longrightarrow$ $(n-1)+(n-2)+\cdots+2+1=\frac{n\cdot(n-1)}{2}$ confronti e scambi

  $O(n^2)$ confronti, $O(n^2)$ scambi

Ordinamento per inserimento (insertion sort)

- L'ordinamento per inserimento è un metodo di ordinamento che risulta essere tra i più efficienti metodi di ordinamento naive (cioè, intrinsecamente $O(n^2)$).
- L'idea di base dell'insertionsort è la seguente: consideriamo di aver già ordinato gli elementi tra 0 e i-1 di un array a, allora possiamo facilmente inserire l'elemento i-esimo nella posizione giusta scalando gli elementi piu grandi di lui.
- **Si noti, simile a come si ordina una carta nuova in una mano di carte di un tipico gioco di carte (scopone, bridge, ecc.)**

Esempio:

$a = 8\ 5\ 9\ 2\ 6\ 3$

$8\ 5\ 9\ 2\ 6\ 3$	elem 0-esimo ordinato (banalmente)
$5\ 8\ 9\ 2\ 6\ 3$	elem 0..1 ordinati
$5\ 8\ 9\ 2\ 6\ 3$	elem 0..2 ordinati
$2\ 5\ 8\ 9\ 6\ 3$	elem 0..3 ordinati
$2\ 5\ 6\ 8\ 9\ 3$	elem 0..4 ordinati
$2\ 3\ 5\ 6\ 8\ 9$	elem 0..5 ordinati (tutti)

Ordinamento per inserimento (insertion sort)

Implementazione:

```
public static void insertionSort(int[] a) {
  for (int i = 1; i < a.length; i++) {
    // gli elementi tra 0 e i-1 sono gia' ordinati

    // inserisci l'elemento i tra gli elementi ordinati
    //nella giusta posizione
    int tmp = a[i];
    int j;
    for (j = i; j > 0 && a[j-1] > tmp; j--)
      a[j] = a[j-1];
    a[j] = tmp;
  }
}
```


Ordinamento per inserimento (insertion sort)

Costo:

Nell'algoritmo di ordinamento insertion sort il caso peggiore si presenta quando l'array è ordinato all'inverso. In tal caso l'istruzione dominante (`tmp < a[j-1]`) viene ripututa per $1 + 2 + ...n - 2 + n - 1 = n * (n - 1)/2$ volte.

Quindi il suo costo è $O(n^2)$.

Si noti che il caso migliore si ha quando l'array è già ordinato. In tal caso l'istuzione dominante (`tmp < a[j-1]`) viene eseguita $n - 1$ volte.

Quindi il costo in tal caso è $O(n)$.

Ordinamento per fusione (merge sort)

Per ordinare 1000 elementi usando un algoritmo di complessità $O(n^2)$ (ordinamento per selezione o a bolle): sono necessarie 10^6 operazioni

Idea: Divido i 1000 elementi in due gruppi da 500 elementi ciascuno:

- ordino il primo gruppo in $O(n^2)$: $250\,000$ operazioni
- ordino il secondo gruppo in $O(n^2)$: $250\,000$ operazioni
- combino (fondo) i due gruppi ordinati: si può fare in $O(n) \implies 1000$ operazioni

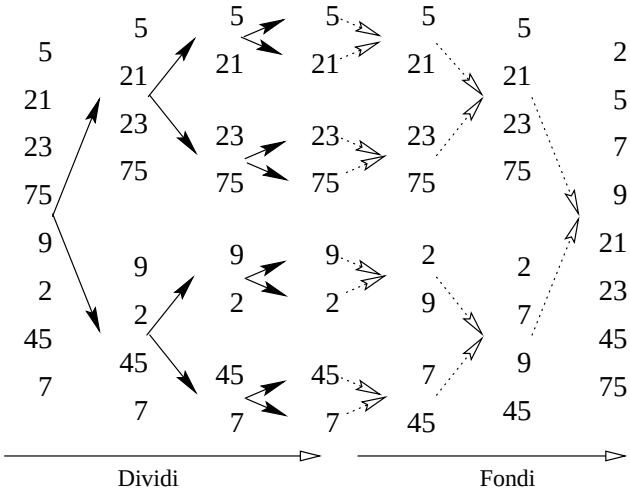
Totale: $501\,000$ operazioni (contro le 10^6)

Il procedimento può essere iterato: per ordinare le due metà non uso un algoritmo di complessità $O(n^2)$, ma applico lo stesso procedimento di divisione, ordinamento separato e fusione.

La suddivisione in due metà si ferma quando si arriva ad un gruppo costituito da un solo elemento (che è già ordinato).

algoritmo ordina per fusione gli elementi di *A* da *iniziale* a *finale*
  if *iniziale* $<$ *finale* (ovvero, c'è più di un elemento tra *iniziale* e *finale*, estremi inclusi)
  then *mediano* $\leftarrow$ (*iniziale* $+$ *finale*) $/2$
    ordina per fusione gli elementi di *A* da *iniziale* a *mediano*
    ordina per fusione gli elementi di *A* da *mediano* $+1$ *finale*
    fondi gli elementi di *A* da *iniziale* a *mediano* con
      gli elementi di *A* da *mediano* $+1$ a *finale*
      restituendo il risultato nel sottovettore di *A* da *iniziale* a *finale*

Esempio:



Per effettuare la **fusione** di due sottovettori ordinati e contigui ottenendo un unico sottovettore ordinato:

- si utilizzano un vettore di appoggio e due indici per scandire i due sottovettori
- il più piccolo tra i due elementi indicati dai due indici viene copiato nella prossima posizione del vettore di appoggio, e viene fatto avanzare l'indice corrispondente
- quando uno dei due sottovettori è terminato si copiano gli elementi rimanenti dell'altro nel vettore di appoggio
- alla fine si ricopia il vettore di appoggio nelle posizioni occupate dai due sottovettori

Ordinamento per fusione (merge sort)

Implementazione:

```
public class MergeSort {

  public static void sort(int v[]) {
    sort(v,0,v.length-1);
  }


  private static void mergesort(int[] v, int inf, int sup) {
    if (inf < sup) {
      int med = (inf+sup)/2;
      mergesort(v,inf,med);
      mergesort(v,med+1,sup);
      merge(v,inf,med,sup);
    }
  }
```

```
  private static void merge(int[] v, int inf, int med, int sup) {
    int[] a = new int[sup-inf+1];
    int i1 = inf;
    int i2 = med+1;
    int i = 0;
    while ((i1 <= med)&&(i2 <= sup)) { // entrambi i vettori
                                       //contengono elementi
       if (v[i1] <= v[i2]) {
         a[i] = v[i1];
         i1++;
         i++;
       }
       else {
         a[i] = v[i2];
         i2++;
         i++;
       }
    }
    if (i2 > sup) // e' finito prima il secondo pezzo del vettore
```



```
      for (int k = i1; k <= med; k++) {
        a[i] = v[k];
        i++;
      }
    else // e' finito prima il primo pezzo del vettore
      for (int k = i2; k <= sup; k++) {
        a[i] = v[k];
        i++;
      }
    // copiamo il vettore ausiliario nel vettore originario
    for(int k = 0; k < a.length; k++)
      v[inf+k] = a[k];
  }

}
```

- funzione `merge` effettua la fusione
- funzione `sort` effettua l'ordinamento di un sottovettore compreso tra due indici *iniziale* e *finale*
- funzione `mergesort` chiama `sort` sull'intervallo di indici da 0 a $n-1$

Complessità dell'ordinamento per fusione

Sia $T(n)$ il costo di ordinare per fusione un vettore di n elementi.

- se il vettore ha 0 o 1 elementi: $T(n) = c_1$
- se il vettore ha più di un elemento:

$$\begin{aligned} T(n) &= \text{costo dell'ordinamento della prima metà} \\ &\quad + \text{costo dell'ordinamento della seconda metà} \\ &\quad + \text{costo della fusione} \\ &= T(n/2) + T(n/2) + c_2 \cdot n \end{aligned}$$

Otteniamo un'equazione di ricorrenza la cui soluzione è la funzione di complessità cercata:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 1 \\ T(n/2) + T(n/2) + c_2 \cdot n & \text{se } n > 1 \end{cases}$$

La soluzione, ovvero la complessità dell'ordinamento per fusione è $T(n) = O(n \cdot \log_2 n)$.

Questo è quanto di meglio si possa fare. Si può infatti dimostrare che nel caso peggiore sono necessari $n \cdot \log_2 n$ confronti per ordinare un vettore di n elementi.

Ordinamento veloce (quick sort)

L'ordinamento veloce si basa sulla seguente idea: si prende in esame un elemento X dell'array (qualsiasi in generale) che viene detto *pivot*, e si partiziona (riorganizza) l'array in modo che:

- il pivot si trovi nella posizione corretta (sia essa k);
- tutti gli elementi più piccoli del pivot siano nelle posizioni precedenti k;
- tutti gli elementi più grandi del pivot siano nelle posizioni seguenti k.

Esempio: :

$a = \mathbf{6}\ \ 12\ \ 7\ \ 8\ \ 5\ \ 4\ \ 9\ \ 3\ \ 1$

$pivot = \mathbf{6} \rightarrow k = 4$

$a' = 5\ \ 4\ \ 3\ \ 1\ \ \mathbf{6}\ \ 12\ \ 7\ \ 8\ \ 9$

A questo punto i due vettori $a_1 = [5\ \ 4\ \ 3\ \ 1]$ e $a_2 = [12\ \ 7\ \ 8\ \ 9]$ possono essere ricosivamente ordinati allo stesso modo.

La scelta del pivot può essere effettuata secondo diversi criteri. Nell'implementazione che segue viene scelto semplicemente il primo elemento del vettore.

Ordinamento veloce (quick sort)

Implementazione:

```
public class QuickSort {

  public static void sort(int[] a) {
    quicksort(a,0,a.length-1);
  }

  public static void quickSort(int[] a,int inf, int sup) {
    if (inf >= sup) return;
    else {
      int posPivot scegliPivot(a,inf,sup);
      int k = partition(a, inf, sup, posPivot);
      quicksort(a,inf,k);
      quicksort(a,k+1,sup);
    }
  }
```



```
  private static int scegliPivot(int[] a, int inf, int sup) {
    //si consulti un libro di alg. e stutt. dati
    //per criteri di scelta avanzati.
    //Noi scegliamo semplicemente il primo elemento
    return inf;
  }

  private static int partition(int[] a, int inf, int sup, int posPivot) {
    int i = inf;
    int j = sup;
    int pivot = a[posPivot];
    while (i < j) {
      while (a[i] < pivot) i++;
      while (a[j] > pivot) j--;
      if (i < j) {
        // ora a[i] > pivot e a[j] < pivot
        // quindi scambio a[i] e a[j]
        int tmp = a[i];
        a[i] = a[j];
```

```
        a[j] = tmp;
      }
    }
    return i;
  }
}
```


Ordinamento veloce (quick sort)

Costo:

- Il quick sort funziona bene quando ad ogni passo si riesce a partizionare il vettore in due vettori di dimensione circa $n/2$. In questo caso la complessità è $O(n \cdot \log_2 n)$ - **l'analisi è molto simile a quella del merge sort**.
- Nel caso peggiore, che si verifica ad esempio se il vettore è già ordinato, si ha una complessità di $O(n^2)$, in quanto ad ogni passo si esamina un nuovo vettore di dimensione inferiore di uno rispetto a quella precedente.
- Pur avendo, nel caso peggiore, una complessità asintotica più grande del merge sort, il quick sort si comporta molto bene quando il vettore è mediamente disordinato, cioè la partizione del vettore avvinene in parti circa uguali. In questi casi in effetti è spesso preferito al merge sort in quanto non richiede l'allocazione di un array aggiunutivo per il merge.
- Il quick sort è l'algoritmo di ordinamento più studiato dal punto di vista delle ottimizzazioni. Tuttavia va tenuto presente che, specialmente introducendo ottimizzazioni, in codice del quick sort è molto sensibile a piccoli errori di programmazione che sono difficili da trovare e che ne degradano fortemente le prestazioni per particolari configurazioni dell'input.

Rissunto sulla complessità degli algoritmi di ordinamento

algoritmo	confronti	scambi	complessià totale
selection sort	$O(n^2)$ sempre	$O(n)$ 0 se già ordinato	$O(n^2)$ sempre
bubble sort (ottimizzato)	$O(n^2)$ $O(n)$ se già ordinato	$O(n^2)$ 0 se già ordinato	$O(n^2)$ $O(n)$ se già ordinato
insertion sort	$O(n^2)$ $O(n)$ se già ordinato	$O(n^2)$ 0 se già ordinato	$O(n^2)$ $O(n)$ se già ordinato
merge sort	$O(n \cdot \log_2 n)$ sempre	$O(n \cdot \log_2 n)$ sempre (serve vettore appoggio)	$O(n \cdot \log_2 n)$ sempre
quick sort			$O(n^2)$ caso peggiore $O(n \cdot \log_2 n)$ caso medio