Laboratorio di Algoritmi e Strutture Dati

Docente: Camillo Fiorentini

20 novembre 2007

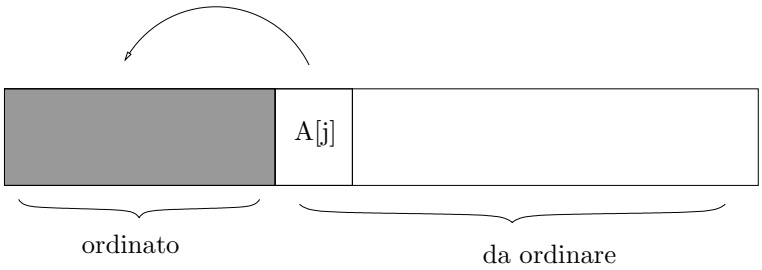
Lo scopo degli esercizi è quello di implementare due algoritmi di ordinamento e di confrontarne sperimentalmente l'efficienza.

Gli algoritmi sono descritti usando il formalismo di alto livello del libro di testo "*Introduzione agli algoritmi*", T. Cormen et al., seconda edizione. Alcune funzioni necessarie per svolgere l'esercizio si trovano nel file `sort.c`.

Esercizio 1: insertion sort

Uno degli algoritmo più semplici per ordinare una sequenza di n interi è l'*insertion sort*. L'idea dell'algoritmo è quella usata quando si pescano n carte da gioco: ogni volta che si prende una nuova carta, la si inserisce al posto giusto, in modo che le carte in mano siano ordinate. In questo modo, al termine della procedura le carte in mano sono ordinate.

In modo analogo, per ordinare una sequenza di n interi contenuta in un array A, per ogni $j = 1, \ldots, n-1$, si sistema il nuovo elemento $A[j]$ nel sottoarray $A[0\,..\,j-1]$, che è già ordinato (quando $j = 1$ la proprietà vale in quanto il sottoarray $A[0\,..\,0]$ cointiene solamente l'elemento $A[0]$). Per sistemare $A[j]$ occorre spostare a destra di una posizione gli elementi $A[i]$ di $A[0\,..\,j-1]$ tali che $A[i] > A[j]$, dove i parte da $j-1$ e ogni volta i viene decrementato di 1.



INSERTION-SORT(A)

```
1   for j ← 1 to length[A] − 1
2       do   key ← A[j]
3            ▷ Si inserisce A[j] nella sequenza ordinata A[0 .. j − 1]
4            i ← j − 1
5            while i ≥ 0 e A[i] > key
6                 do A[i + 1] ← A[i]
7                    i ← i − 1
8            A[i + 1] ← key
```

In C non è possibile calcolare $length[A]$, quindi il numero n di elementi da ordinare va passato come parametro. La funzione deve avere intestazione

```
/* Ordina  i primi n elementi dell'array a[] usando l'algoritmo  insertion sort */
```

```
void insertionSort(int a[], int n)
```

Per provare la funzione, definire una funzione `main()` in cui si legge una sequenza di di interi da standard input usando la funzione `read()` in un array

```
int v[MAX]
```

(`MAX` va definita nel programma e denota la lunghezza massima della sequenza da ordinare), si ordina la sequenza con la funzione `insertionSort()` e si stampa la sequenza ordinata usando la funzione `print()`.

Per provare il programma conviene scrivere la sequenza in un file `in.txt` e redirigere lo standard input da tale file:

```
a.exe < in.txt
```

Gli interi devono essere separati da *almeno* un carattere di spaziatura (spazio, a capo, tabulazioni, ecc.) e la sequenza è terminata automaticamente dall' "end-of-file".

Per quanto riguarda l'*efficienza* dell'algoritmo, si può dimostrare che per ordinare un array di n elementi:

- nel caso migliore la complessità è $\Theta(n)$;
- nel caso peggiore e nel caso medio la complessità è $\Theta(n^2)$.

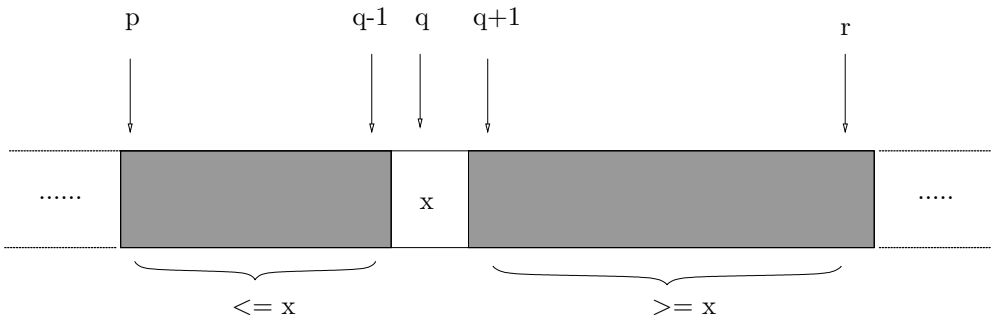
Quindi l'insertion sort *non* è un buon algoritmo (anche se con array di piccole dimensioni risulta essere efficiente).

Esercizio 2: quicksort

Un algoritmo più sofisticato, ma più efficiente, è il *quicksort* che adotta una strategia di tipo *divide-et-impera*. Per ordinare un array $A[p \,..\, r]$ (ossia, gli elementi dell'array A compresi fra gli indici p e r) si risistemano gli elementi di $A[p \,..\, r]$ in modo tale che i sottoarray $A[p \,..\, q-1]$ e $A[q+1 \,..\, r]$ verifichino le seguenti proprietà:

- Per ogni elemento $A[i]$ in $A[p \,..\, q-1]$, $A[i] \leq A[q]$;
- Per ogni elemento $A[j]$ in $A[q+1 \,..\, r]$, $A[j] \geq A[q]$.

L'elemento $A[q]$ è chiamato *perno* (o *pivot*).



Se $p = r$, il sottoarray $A[p \,..\, r]$ contiene un elemento e quindi è ordinato. Altrimenti,si possono ordinare ricorsivamente i due sottoarray $A[p \,..\, q-1]$ e $A[q+1 \,..\, r]$ (che contengono *meno* elementi di $A[p \,..\, r]$); al termine delle due chiamate ricorsive, $A[p \,..\, r]$ risulta essere ordinato.

Il partizionamento di $A[p \,..\, r]$ in $A[p \,..\, q-1]$ e $A[q+1 \,..\, r]$ viene effettuato chiamando PARTITION(A, p, r) che usa come perno il valore in $A[r]$ e restituisce l'indice q rispetto a cui l'array $A[p \,..\, r]$ risulta essere partizionato.

```
QUICKSORT(A, p, r)
1     if p < r
2       then q ← PARTITION(A, p, r)
3            QUICKSORT(A, p, q − 1)
4            QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1     x ← A[r]
2     i ← p − 1
3     for j ← p to r − 1
4       do if A[j] ≤ x
5            then i ← i + 1
6                 scambia A[i] ↔ A[j]
7     scambia A[i + 1] ↔ A[r]
8     return i + 1
```

Nel tradurre le linee 6 e 7 usare la funzione `swap()`.

Definire le funzioni `quicksort()` e `partition()` che traducono le corrispondenti funzioni di alto livello sopra descritte. Definire inoltre una funzione

```
/* Ordina i primi n elementi dell'array a[]  usando il quicksort   */

 void qksort(int a[], int n)
```

che è la funzione principale da usare per ordinare i primi n elementi di un array `a[]` (non si può chiamarla `qsort()` perché esiste già una funzione `qsort()` nella libreria standard). Per provare l'algoritmo, definire una funzione `main()` simile a quella usata nell'Esercizio 1.

Per quanto riguarda la complessità, per ordinare un array di n elementi il tempo richiesto è:

- $\Theta(n \log n)$ nel caso medio (si dimostra che tale tempo è *ottimale*).
- $\Theta(n^2)$ nel caso peggiore (quando si verifica?).

Esercizio 3: confronto sperimentale fra i due algoritmi

Per confrontare sperimentalmente l'efficienza dei due algoritmi, si caricano in un vettore dei valori generati casualmente e si ordina il vettore usando i due algoritmi (vedi `main3()` di `sort.c`).

In C per generare un numero intero in modo casuale si può utilizzare la funzione `rand()` il cui prototipo, contenuto in `<stdlib.h>` (da includere), è

```
int rand(void)
```

Il generatore va inizializzato ad ogni esecuzione del programma con `srand()`; occorre inizializzarlo ogni volta in maniera diversa, altrimenti viene sempre generata la stessa sequenza di numeri pseudocasuali. Il modo standard è quello di porre all'inizio di `main()` l'istruzione

```
srand(time(NULL));
```

Il valore iniziale dato al generatore è il valore restituito da `time(NULL)` (numero di secondi passati dall'1 gennaio 1970). Occorre includere `<time.h>`.

Per valutare sperimentalmente il tempo impiegato dal calcolatore per eseguire delle linee di codice si può usare la funzione `clock()` il cui prototipo è definito in `<time.h>`.

- La funzione `clock()` accede all'orologio interno della macchina e restituisce il numero di "unità di clock" impiegate dal calcolatore per eseguire il programma fino al punto in cui è chiamata. Il valore restituito da `clock()` è di tipo `clock_t` (non è un tipo dello standard ANSI C, ma è definito in `<time.h>`).
- Le durata di una unità di clock dipende dal sistema ed è definita dalla macro `CLOCKS_PER_SEC`.

Per valutare il tempo richiesto da `insertionSort()` per ordinare `n` elementi di `v[]` occorre fare:

```
clock_t  clock0, clock1;  // per cronometrare
double tempo;       // tempo di esecuzione

clock0 =  clock();     //  tempo iniziale
insertionSort(v, n);   //  funzione da cronometrare
clock1 =   clock();    //  tempo finale
tempo = ( (double)(clock1-clock0) ) / CLOCKS_PER_SEC;
printf("Tempo di esecuzione:  %4f sec.\n", tempo);
```

Per calcolare il tempo occorre fare la divisione *non intera*, per questo è necessario il cast a `double` di uno degli operandi (i valori di `clock0`, `clock1` e `CLOCKS_PER_SEC` sono interi).

L'istruzione

```
printf(" ... %.4f ...", tempo);
```

stampa il valore di `tempo` (di tipo `double`) usando 4 cifre decimali.

Si osserva che, per input grandi generati casualmente, il quicksort è molto più efficiente dell'insertion sort. Tuttavia la complessità del caso peggiore di quicksort è $O(n^2)$ (come per l'insertion sort). Il caso peggiore si verifica quando il partizionamento dell'array $A[p \,..\, r]$ produce sempre due sottoarray sbilanciati (uno grande e uno piccolo). Se si genera casualmente la sequenza di input, è poco probabile che capiti uno di questi casi limite. Scrivere una funzione `init()` che inizializza un array in modo tale che si verifichi il caso peggiore per quicksort.