Algoritmi di Ordinamento

Maurizio Palesi
Salvatore Serrano

Introduzione

- Ordinare una sequenza di informazioni significa effettuare una permutazione in modo da rispettare una relazione d'ordine tra gli elementi della sequenza (p.e. minore o uguale ovvero non decrescente)
- Sulla sequenza ordinata diventa più semplice effettuare ricerche

Importanza dell'ordinamento

- Sia il sistema operativo che i software applicativi devono continuamente effettuare ricerche su insiemi di informazioni e quindi l'efficienza degli algoritmi di ordinamenti usati è un fattore critico per garantire le buone prestazioni del sistema

Elementi

- Generalmente gli elementi che devono essere ordinati hanno la stessa struttura
- Tipicamente ogni elemento sarà costituito da un insieme di campi
- Esempio (Rubrica Telefonica)
  - ➔Nome
  - ➔Cognome
  - ➔Numero

Campi chiave

- Per effettuare l'ordinamento si deve scegliere almeno un campo, definito **chiave**, che sarà utilizzato per la ricerca nella sequenza
  - ➔E' comune anche la ricerca per più di un campo (**chiavi multiple**)
- La scelta della chiave viene effettuata per ridurre le possibili omonimie
- L'ordinamento potrà essere effettuato su più di un campo (chiave primaria, secondaria, …)

Operazioni elementari e complessità

- Per effettuare l'ordinamento è necessario riuscire ad effettuare le seguenti operazioni:
  - ➔Confronto
  - ➔Scambio
- E' importante poter confrontare i vari algoritmi di ordinamento in termini di complessità
  - ➔Numero di operazioni elementari necessarie in funzione del numero n di elementi della sequenza
- Gli algoritmi di ordinamento interno si dividono in
  - ➔Semplici Complessità $O(n^2)$
  - ➔Evoluti Complessità $O(n \cdot \log_2 n)$

Prestazioni di esecuzione

- Dipende dalla struttura dell'elemento informativo e dal numero di chiavi usate:
  - ➔Scambiare due interi è molto meno oneroso che scambiare due strutture complesse
  - ➔Confrontare solo una chiave è meno oneroso che confrontarne due

Indici delle sequenze

- Per migliorare le prestazioni di esecuzione è utile costruire un vettore ausiliare (**indice**) che contiene i puntatori agli elementi della sequenza
- Questo ci permette di ridurre il numero di operazioni necessarie per effettuare lo scambio anche in presenza di strutture complesse perché l'elemento da scambiare sarà sempre un indirizzo

Ipotesi semplificativa

- La sequenza di informazioni è sempre costituita da un vettore di n elementi nel quale ogni elemento è individuato da un indice variabile da **0** e **n-1**
- Gli elementi del vettore potranno essere tipi scalari o strutture
- Il vettore può essere allocato sia staticamente che dinamicamente

Bubble Sort

- Algoritmo semplice
- Si effettuano successivi spostamenti degli elementi dalla posizione di partenza a quella ordinata simile alla risalita delle bolle d'aria in un liquido
  - ➔Si consideri un vettore di n elementi da ordinare in ordine non decrescente
  - ➔Sono necessarie diverse scansioni del vettore (iterazioni)
  - ➔Si inizializza a 0 il *flag ordinato*
  - ➔Si scrive un ciclo **for** che controlla le **n-1** iterazioni necessarie ma anche se il *flag* ordinato è posto a 1

Bubble Sort

- La prima iterazione (i=0) si applica a tutto il vettore (n elementi) e posiziona correttamente il primo elemento
- La seconda iterazione si applica a tutto il vettore escluso il primo elemento (n-1 elementi)
- L'iterazione *i* si applica al sottovettore residuo, in quanto i primi *i* elementi sono già ordinati
- Se durante la generica *i-esima* iterazione c'è uno scambio settiamo a 0 il *flag ordinato* in modo che il **for** esterno sappia che il vettore non può ancora essere considerato ordinato

Bubble Sort

- Od ogni scansione, iniziamo puntando l'indice all'ultimo elemento del vettore e lo confrontiamo con il precedente
- Se l'ultimo elemento è minore dobbiamo scambiarlo con il penultimo
- Se c'è lo scambio, poniamo a 0 il *flag ordinato*
- Il **for** interno verifica se deve fare un successivo confronto

Esercizio

- Implementare l'algoritmo del **Bubble Sort**

Soluzione

```
void scambia (int ve[], int i, int j) {
  int tmp = ve[i];
  ve[i] = ve[j];
  ve[j] = tmp;
}
```

Soluzione

```
void BubbleSort(int v[], int ne) {
  int ordinato=0,i,j;
  for (j=0; j<ne-1 && !ordinato; j++) {
     ordinato = 1;
     for (i = ne-1; i>j; i--) {
          if (v[i]<v[i-1]) {
               scambia(v, i, i-1);
               ordinato = 0;
          }
     }
  }
}
```


Soluzione

```
#define N 10
void main() {
  int i,p[N];
  for (i=0; i<N; i++) {
     printf("Immetti il valore dell'elemento
          %d del vettore", i);
     scanf("%d", &p[i]);
  }
  BubbleSort(&p[0], N);
  for (i=0; i<N; i++)
     printf("V[%d]=%d\n", i, p[i]);
}
```

Complessità del BubbleSort

- L'algoritmo BubbleSort esegue alla prima iterazione **n-1** confronti e al più **n-1** scambi
- Alla seconda iterazione **n-2** confronti/scambi
- Le iterazioni sono in totale al più **n-1**
- Il numero totale di confronti/scambi è

  (N-1)+(N-2)+…+1

  quindi sarà

$$O(n) = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \approx n^2$$