Quicksort e qsort()

Alessio Orlandi

28 marzo 2010

Intro

- Quicksort è l'algoritmo di ordinamento più implementato, insieme con Mergesort.
- Tutte le librerie standard UNIX ne prevedono una implementazione.
- Sono quasi sempre ricorsive e scelgono il pivot come mediano tra tre posizioni casuali (randomized median-of-three).
- Purtroppo, non sono stabili.
- Trattandosi di ordinamento per confronto sono funzioni generiche.

Quicksort: ripasso

Quicksort: distribuisci rispetto ad un pivot, ricorri sulle due metà.

```
void distrib(int *A, int sx, int dx, int px) {
  if ( px != dx ) scambia(A, px, dx);
  i = sx; j = dx - 1;
  while (  i < j )  {
      while ( i < j && A[i] <= A[dx])
        i++;
      while ( i < j && A[j] >= A[dx])
        j--;
      if ( i < j ) scambia ( A, i, j );
  }
  if ( i != dx ) scambia ( A, i, dx );
  return i;
}
```

In parole: trova la prima coppia di discrepanze (da sx e da dx), scambia e riparti.

Quicksort per confronti

La procedura appena descritta richiede 2 confronti:

$$\texttt{A[i]} \leq \texttt{A[dx]} \text{ e } \texttt{A[j]} \geq \texttt{A[dx]}$$

e delle chiamate a `scambia`.
E se volessimo ordinare stringhe?

Quicksort per confronti

La procedura appena descritta richiede 2 confronti:

$$\texttt{A[i]} \leq \texttt{A[dx]} \text{ e } \texttt{A[j]} \geq \texttt{A[dx]}$$

e delle chiamate a `scambia`.
E se volessimo ordinare stringhe? Dovremmo cambiare il significato di $\leq$ e $\geq$, e (dentro scambia) $=$. $\Rightarrow$ Ordinamento per confronti è *astraibile* dalla tipizzazione.
Scopo delle prossime slide: arrivarci.

Quicksort astratto

1. Generalizzazione con *confrontatori*:

```
int comp ( TIPO a, TIPO b);
```

   restituisce -1 se $a > b$, 0 se $a = b$, $+1$, se $a < b$.
   Sostituiamo $A[i] < A[dx]$ con $comp(A[i], A[dx]) == -1$ e simili.
   $\Rightarrow$ Implementazione a carico dell'utilizzatore

2. Spostamento di aree di memoria:

```
s = sizeof(TIPO);
memcpy ( (byte *)A + i*s, (byte *)A + j*s, s );
```

Puntatori void *

Supponiamo di confrontare fornendo i puntatori agli elementi.

- Se ordino interi: `int comp ( int *a, int *b )`
- Se ordino reali: `int comp ( float *a, float *b )`
- Se ordino stringhe: `int comp ( char **a, char ** b )`

Mettiamoci nei panni di chi scrive la procedura di quicksort.
Come chiamare `comp` correttamente? Se sbaglio tipo?

Puntatori void *

Supponiamo di confrontare fornendo i puntatori agli elementi.

- Se ordino interi: `int comp ( int *a, int *b )`
- Se ordino reali: `int comp ( float *a, float *b )`
- Se ordino stringhe: `int comp ( char **a, char ** b )`

Mettiamoci nei panni di chi scrive la procedura di quicksort. Come chiamare `comp` correttamente? Se sbaglio tipo? Bisogna ricorrere a dei puntatori generici: `void *`.

Puntatori void * - II

`void *` è un puntatore che non ha significato se non dopo un cast: non può essere dereferenziato. Non si può accedere ad un array void *.
E' permesso *convertire* tramite casting i puntatori `void *` in qualsiasi altro tipo di puntatore:

```
void call(void * a, void * b )  {
    int *y1 = (int *) a;
    int **y2 = (int **) b;
    ...

}
..
  int * x = malloc(sizeof(int) * 1024);
  void *a = (void *)& x;
  call ( x,  a );
```

Dilemma risolto:

```
int comp ( void * a, void * b );
```

Scriviamo quicksort parlando di una generica comp, sarà chi la implementa a converitre i puntatori al tipo corretto. Quindi il nostro prototipo di quicksort *potrebbe* essere così:

```
void quicksort ( void *A, int n, int dim );
```

- *A*: array del tipo generico da ordinare.
- *n*: numero di elementi nell'array.
- *dim*: memcpy richiede la dimensione del tipo. `sizeof(...)`.

L'utilizzatore crea `comp` e chiama quicksort.

Esempio di chiamata

```
void quicksort ( void *A, int n, int dim );


int *array = malloc(sizeof(int)*n ) ;
for ( i = 0; i < n; i++ )
    array[i] = rand()%100;
quicksort ( array,  n, sizeof(int ) );
```

Puntatori a funzione

Ora supponiamo di voler fare quicksort due volte nel nostro programma, una volta su interi, una volta di stringhe. Come usare due confrontatori diversi?
La soluzione sarebbe aggiungere un parametro: la funzione di confronto da usare. Tramite puntatori a funzione.

Puntatori a funzione

Ora supponiamo di voler fare quicksort due volte nel nostro programma, una volta su interi, una volta di stringhe. Come usare due confrontatori diversi?
La soluzione sarebbe aggiungere un parametro: la funzione di confronto da usare. Tramite puntatori a funzione.
Scrittura di quicksort:

```
void quicksort(...,int (*compar)(void *,void *)){
    ...
    compar( a, b );
}
```

Utilizzo:

```
int mio_comp(void *a, void *b ) {
    return -1; }
quicksort(..., mio_comp, ... );
```

Puntatori a funzione - II

- Ogni *segnatura* di funzione definisce un tipo a se: tipo di ritorno, del primo argomento, etc..
- Una funzione con input `int` e restituisce `void` è del tipo

$$(\mathbf{void})(*)(\mathbf{int}, \mathbf{int})$$

- Dandogli un nome (notare le tonde intorno all'asterisco!)

$$(\mathbf{void})(*\text{nome})(\mathbf{int}, \mathbf{int})$$

- Chiunque voglia usare un puntatore a funzione compatibile, deve implementare una funzione che abbia esattamente la stessa segnatura: restituisce `void`, un solo parametro `int`.
- La funzione si passa tramite il suo nome (e avviene per riferimento).

Uso di qsort()

Il linguaggio *C* mette a disposizione una implementazione di quicksort:

```
void qsort( void *A, int n, int dim,
            int (*compar)(const void *,
                          const void *) );
```

- *A*: array da ordinare.
- *n*: numero di elementi nell'array
- *dim*: dimensione di un elemento dell'array (sizeof)
- *compar*: funzione di comparazione a cui vengono passati puntatori ad oggetti dell'array.

```
void qsort(void *A, int n, int dim, int (*cmp)(const
void*, const void * ) ;
```

Un esempio:

```
int fcomp(const void *a, const void *b )  {
    float *x = ( (float *) a );
    float *y = ( (float *) b );

    if ( *x < *y ) return 1;
    if ( *x > *y ) return -1;
    return 0;
}
...
    float *x; // array gia' pronto
    int n; // dimensione dell'array
    qsort( x, n, sizeof(float), fcomp );
```

Esercizio 1

Scrivere un programma che legga da tastiera un array A di stringhe e che utilizzi la funzione di libreria qsort per ordinare in ordine alfabetico crescente le stringhe in input. Stampare in outupt la sequenza di stringhe ordinata, una per riga. Linput è formattato nel seguente modo: sulla prima riga si trova il numero N di stringhe. Seguono N righe contenenti ognuna una una stringa dellinsieme da ordinare. Le stringhe contengono soltanto carat- teri alfanumerici (a - z minuscoli e maiuscoli o numeri, nessuno spazio o punteggiatura) e sono lunghe al più 1000 caratteri ciascuna.

Esercizio 2

Scrivere un programma che utilizzi la procedura qsort e ordini un vettore di interi non negativi (in input), in modo da ottenere il seguente effetto. Larray riordinato deve contenere prima tutti i numeri pari e, a seguire, i numeri dispari. I numeri pari devono essere ordinati in modo crescente fra di loro. I numeri dispari devono essere ordinati in modo decrescente fra di loro. La sequenza in input ‘e, come al solito, il numero N (non limitato) di interi seguiti da N valori interi non negativi. Stampare la sequenza riordinata in output su una sola riga. Nota: il numero zero ‘e, per convenzione, ritenuto sempre pari.

Esercizio 3

Risolvere il tema di esame del 28/05/2009 (ignorando le modalità di consegna).
Il testo (e relativa soluzione da guardare DOPO) sono disponibili all'indirizzo
`http://www.cli.di.unipi.it/doku/lib/exe/fetch.php/informatica/all-a/all09/lab20090528.zip`

Esercizio 4

Scrivere un programma che implementi la ricerca binaria (dicotomica) su un insieme di stringhe in input. Linput fornito ‘e formato da una prima riga contenente N il numero di stringhe tra cui effettuare la ricerca. Le successive N righe contengono stringhe ordinate in modo alfabetico crescente. Segue una sequenza di dimensione non conosciuta di coppie di righe. La prima riga di ogni coppia e il valore 1 o 0. Se il valore ‘e 0 il programma termina (non ci sono piu‘ richieste). Se il valore ‘e 1, sulla riga successiva trovate la stringa da cercare. Per tanto linput contiene le N + 1 righe dellinsieme e ogni successiva richiesta di ricerca viene preceduta da una riga con il numero 1. La procedura termina quando si trova uno 0. Le stringhe contengono solo caratteri alfanumerici e non sono piu‘ lunghe di 1000 caratteri. Loutput ‘e composto da una riga per ognuna delle richieste. Ogni riga contiene il risultato della ricerca binaria: se lelemento appare nellinsieme, stamparne la posizione (valore da 0 a N 1). Altrimenti, stampare -1.