Ricerca e Ordinamento

Ricerca di un elemento in un vettore

Problema: Effettuare le seguenti **operazioni** su un insieme di dati omogenei:

- **ricerca** di un elemento: dati un insieme *A* ed un elemento *elem*, vogliamo sapere se *elem* è o meno presente in *A*
- **inserimento** di un elemento: dati un insieme *A* ed un elemento *elem*, se *elem* non compare in *A*, vogliamo inserirlo
- **eliminazione** di un elemento: dati un insieme *A* ed un elemento *elem*, se *elem* compare in *A*, vogliamo eliminarlo

Struttura di dati utilizzata: vettore, con una componente per ogni elemento

Operazione fondamentale: cercare un elemento nel vettore (è necessaria anche per inserimento ed eliminazione)

Ricerca esaustiva (in un vettore qualsiasi)

Se il vettore è costituito da elementi qualsiasi, non si può far di meglio che cercare l'elemento tra tutti gli elementi del vettore. $\Longrightarrow$

Ricerca esaustiva: si scandisce il vettore, confrontando gli elementi con quello da cercare, fino a che

- si sono confrontati tutti gli elementi, oppure
- si è trovato l'elemento cercato.

Implementazione: file `riceordi/ricesau.c`

Complessità: lineare (ovvero $O(n)$, dove n è il numero di elementi del vettore)

Ricerca binaria (in un vettore ordinato)

Se il vettore è ordinato, possiamo sfruttare l'ordinamento dimezzando ad ogni passo l'intervallo di elementi ancora da considerare per la ricerca.

Per cercare un elemento *elem* in un vettore ordinato *A*:

- usiamo due indici *inf* e *sup* per delimitare l'intervallo di ricerca
- confrontiamo *elem* con l'elemento di *A* al centro dell'intervallo, ovvero con *A[med]*, dove *med* $=$ *(inf*$+$*sup)*$/2$.

  Distinguiamo 3 casi:
  1. *elem* $=$ *A[med]* $\Longrightarrow$ trovato
  2. *elem* $<$ *A[med]* $\Longrightarrow$ cerchiamo nell'intervallo con indici da *inf* a *med*-1
  3. *elem* $>$ *A[med]* $\Longrightarrow$ cerchiamo nell'intervallo con indici da *med*$+1$ a *sup*
- se non troviamo *elem* ci fermiamo quando l'intervallo di ricerca si è annullato (ovvero, *inf* $>$ *sup*)

Esempio: Ricerca di 24 nel seguente vettore ordinato.

0	1	2	3	4	5	6	7	8	9	10
12	14	19	24	31	33	40	45	52	60	68

$\uparrow$ *inf* $= 0$ $\quad$ $\uparrow$ *med* $= 5$ $\quad$ $\uparrow$ *sup* $= 10$

$\uparrow$ *inf* $\quad$ $\uparrow$ *med* $\quad$ $\uparrow$ *sup*

inf $=$ *med* $\uparrow$ $\quad$ $\uparrow$ *sup* $\quad$ $\Longrightarrow$ trovato 24

se *elem* $= 31$ $\quad$ $\uparrow$ *inf* $=$ *med* $=$ *sup* $\quad$ $\Longrightarrow$ trovato 31

se *elem* $= 28$ $\quad$ *sup* $\uparrow$ $\quad$ $\uparrow$ *inf* $\quad$ $\Longrightarrow$ non compare 28

algoritmo ricerca *elem* nella parte di *A* compresa tra *inf* e *sup*
  if *sup* $<$ *inf*
  then l'elemento non compare in *A*
  else *med* $\leftarrow$ *(inf*$+$*sup)*$/2$
    if *elem* $=$ *A[med]*
    then l'elemento è stato trovato
    else if *elem* $<$ *A[med]*
    then ricerca *elem* nella parte di *A* tra *inf* e *med-1*
    else ricerca *elem* nella parte di *A* tra *med+1* e *sup*

Implementazione: file `riceordi/ricbinri.c`
Usa una funzione ricorsiva che prende come parametri anche gli estremi inferiore e superiore dell'intervallo di ricerca.

Esercizio: Fornire l'algoritmo e l'implementazione iterativa della ricerca binaria.
Soluzione: file `riceordi/ricbinit.c`

Complessità: ad ogni confronto dimezziamo la dimensione dell'intervallo di ricerca:

$$n \rightarrow \frac{n}{2} \rightarrow \frac{n}{2^2} \rightarrow \cdots \rightarrow \frac{n}{2^m}$$

Nel caso peggiore (l'elemento non c'è) la ricerca termina dopo m confronti, con m tale che:

$$\frac{n}{2^m} \leq 1 \qquad \text{ovvero} \qquad n \leq 2^m \qquad \text{ovvero} \qquad m \geq \log_2 n$$

$\Longrightarrow$ La complessità della ricerca binaria è $O(\log_2 n)$.

Ordinamento di un vettore

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 vettore in memoria centrale
- ordinamento di una struttura collegata 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 più semplice del problema dell'ordinamento.

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

Usiamo le seguenti definizioni di tipo:

```
#define NumElementi 20
typedef int TipoElemVettore;
typedef TipoElemVettore TipoVettore[NumElementi];
```


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

Implementazione: file `riceordi/ordsel.c`

Complessità dell'ordinamento per selezione

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

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

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

Implementazione: file `riceordi/ordbub.c`

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: per **esercizio**: file `riceordi/ordbubot.c`

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 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 $O(n)$ $\Longrightarrow$ 1000 operazioni

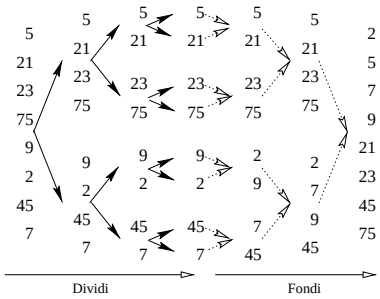
Totale: $501\,000$ operazioni (contro le 10^6)

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

Implementazione: file `riceordi/ordmerge.c`

- funzione `MergeVettore` effettua la fusione
- funzione `MergeRicorsivo` effettua l'ordinamento di un sottovettore compreso tra due indici *iniziale* e *finale*
- funzione `MergeSort` chiama `MergeRicorsivo` 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.

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

Quick sort è un algoritmo molto utilizzato in pratica

- si comporta bene in media, anche se ci sono vettori per cui ha costo $O(n^2)$
- in pratica è più efficiente e più semplice da realizzare di merge-sort
- non richiede l'utilizzo di un vettore di appoggio