ALGORITMI DI ORDINAMENTO

Consideriamo algoritmi di ordinamento interni (elementi in memoria centrale).

Vettore di elementi di un certo tipo, sul quale è definita una ***relazione d'ordine totale*** (ad esempio, tipo float o int)

```
#define MAX 7
#define true 1
#define false 0
typedef float vector [MAX];
```

L'obiettivo è quello di riposizionare (eventualmente) i valori nel vettore, in modo tale da rispettare un ordinamento tra i valori.

Esempio:

Dal vettore:

5	8	11	20	30	-3	5

Si vuole ottenere il vettore ordinato:

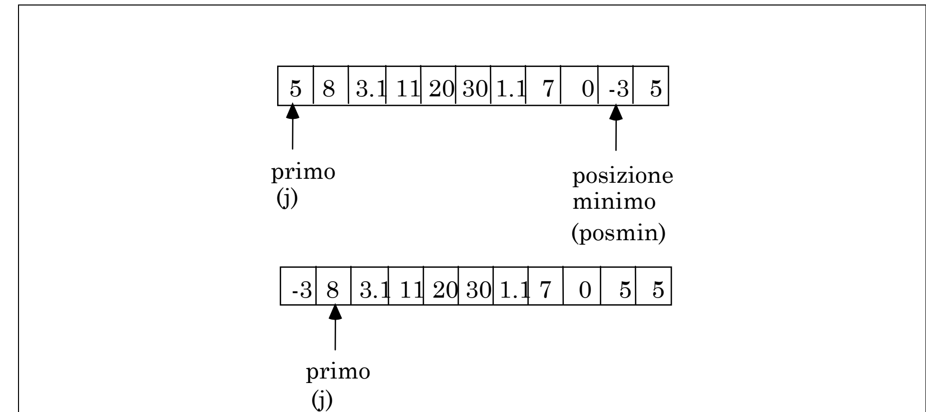
-3	5	5	8	11	20	30

In questo caso, ***ordinamento non decrescente***.

∀ i,j: i<j V[i]<= V[j]

Naive sort (o selection sort):

A ogni passo seleziona il minimo nel vettore e lo pone nella prima posizione, richiamandosi ed escludendo dal vettore il primo elemento.



Algoritmo:

<u>while</u> (<il vettore ha più di una componente>)
{ <trova posizione del minimo nel vettore corrente>
<scambia se necessario il primo elemento del vettore corrente con A[posmin]>
<considera come vettore corrente quello precedente tolto il primo elemento> }

```
/* NAIVE SORT */
#include <stdio.h>
#define N 7
typedef int vettore[N];

void main (void)
{void naive_sort (vettore vet);

 int i;
 vettore a;

 printf ("Scrivi %d numeri interi\n", N);
 for (i = 0; i < N; i++)
           { scanf ("%d", &a[i]); }
 naive_sort(a);
 puts ("Il vettore ordinato è: ");
 for (i = 0; i<N; i++)
       { printf(" %d",a[i]); }
}

void  naive_sort (vettore vet)
{int  j, i, posmin, min;
 for  (j=0; j < N-1; j++)
   {posmin=j;
    for (min=vet[j],i=j+1; i<N; i++)
        {if (vet[i]<min) /* istr. dom. */
             {min=vet[i];
               posmin=i;}  }
    if (posmin != j)      /* scambia */
       {
         vet[posmin]=vet[j];
         vet[j]=min;
       }   }  }
```


Complessità dell'algoritmo di naive sort:

Vettore di dimensione n

La ricerca del minimo si ripete n-1 volte (ciclo for esterno).

Per trovare il minimo, l'istruzione di confronto (dominante) viene eseguita:

n-1	n-2	...	3	2	1	volte
0	1	...	n-4	n-3	n-2	j

$$\sum_{(i=1..n-1)} i = n*(n-1)/2 = O(n^2)$$

Questo risultato è ***indipendente dai valori di ingresso***.

Comunque si eseguono $O(n^2)$ confronti anche se il vettore è già ordinato.

Esempio (naive sort):
Si consideri il seguente vettore di sette elementi:

5	8	11	20	30	-3	5

Dopo il primo ciclo (j=0) ovvero la prima scansione il vettore risulta così modificato:

-3	8	11	20	30	5	5

La seconda iterazione (j=1) produce:

-3	5	11	20	30	8	5

La terza iterazione (j=2) modifica ulteriormente il vettore:

-3	5	5	20	30	8	11

La quarta iterazione (j=3) produce:

-3	5	5	8	30	20	11

Dopo la quinta iterazione (j=4) otteniamo:

-3	5	5	8	11	20	30

La sesta iterazione (j=5) non produce scambi e quindi non modifica il vettore:

-3	5	5	8	11	20	30

A questo punto il vettore è completamente ordinato.

Bubble sort (ordinamento a bolla):

L'algoritmo di ordinamento a bolla effettua l'ordinamento del vettore A attraverso una serie di (al più) N-1 scansioni del vettore stesso.

In ogni fase di scansione si effettuano confronti tra le coppie di elementi in posizione adiacente: dati due elementi adiacenti A[i] e A[i+1], se non rispettano l'ordinamento, essi vengono scambiati di posizione tra loro.

Algoritmo:
La struttura dell'algoritmo, ad alto livello, è la seguente:

```
do
   per tutte le coppie di elementi adiacenti del
   vettore A esegui:
      se  A[i] > A[i+1] allora scambiali;
while   il vettore A non e' ordinato.
```

Il vettore è ordinato quando non ci sono più scambi da effettuare: l'algoritmo, quindi, si arresta al termine della scansione in cui non è stato effettuato alcun scambio, oppure al termine della (N-1)-sima scansione.

Esempio (bubble sort):

Si consideri il seguente vettore di sette elementi:

5	8	11	20	30	-3	5

Dopo la prima scansione il vettore risulta così modificato:

5	8	11	20	-3	5	30

La seconda scansione produce:

5	8	11	-3	5	20	30

La terza scansione modifica ulteriormente il vettore:

5	8	-3	5	11	20	30

La quarta scansione produce:

5	-3	5	8	11	20	30

Dopo la quinta scansione otteniamo:

-3	5	5	8	11	20	30

La sesta ed ultima scansione non produce scambi e quindi non modifica il vettore:

-3	5	5	8	11	20	30

A questo punto il vettore è completamente ordinato.


```
/* BUBBLE SORT */
#define true 1
#define false 0
#define N 10
typedef int vettore[N];

void bubblesort (vettore v, int iniz, int
fine)
{   int SCAMBIO, I;
    int temp;
do  {  SCAMBIO = false;
       for (I = iniz; I < fine; I++)
       {   if (v[I] > v[I+1])
           {   SCAMBIO = true;
               temp = v[I];
               v[I] = v[I+1];
               v[I+1] = temp;
           }
       }
    } while (SCAMBIO);
}
```

`bubblesort` ha tre parametri: il vettore da ordinare, l'indice del primo elemento del vettore (di solito: **`0`**) e l'indice dell'ultimo elemento del vettore (di solito: **`N-1`**):

```
bubblesort(a,0,N-1);
```

Si chiama ***ordinamento a bolla*** perchè dopo la prima scansione del vettore, l'elemento massimo si porta in ultima posizione (gli elementi più piccoli "salgono" verso le posizioni iniziali del vettore).

Ogni scansione ha come effetto la collocazione di un elemento nella sua posizione definitiva (parti grigie nelle figure precedenti).

✐ Esercizio 5.1:

Scrivere una versione ottimizzata dell'algoritmo bubble sort in cui ad ogni scansione si limiti l'analisi alla parte del vettore (sottovettore) non ancora ordinata, migliorando così l'efficienza dell'algoritmo.

Soluzione: Bubble sort ottimizzato

```
void bubble_opt(vettore v,int iniz,int fine)
{   int I, limit, last;
    int temp;

    limit=fine;  /*posiz. ultimo scambio */
    while (limit>iniz)
     { last=iniz;
       for (I = iniz; I <limit; I++)
        {if (v[I] > v[I+1])
           { temp = v[I];
             v[I] = v[I+1];
             v[I+1] = temp;
             last = I;
           }
         }
       limit=last;
      }
}
```

Utilizza una variabile ausiliaria (**`limit`**) per tenere traccia della posizione in cui è stato effettuato l'ultimo scambio.

Se non ci sono stati scambi, **`limit==iniz`** alla fine dell'esecuzione del corpo del ciclo.

Esempio (bubble sort):

5 8 11 20 30 -3 5
iniz fine

5 8 11 20 -3 5 30
iniz last fine

5 8 11 -3 5 20 30
iniz last fine

5 8 -3 5 11 20 30
iniz last fine

5 -3 5 8 11 20 30
iniz last fine

-3 5 5 8 11 20 30
iniz last fine

Complessità dell'algoritmo bubble sort ottimizzato:

Non sono sempre necessarie n-1 iterazioni. Se non avviene alcuno scambio, l'algoritmo termina dopo la prima iterazione.

La complessità ***dipende dai valori dei dati di ingresso***.

Caso migliore:
Vettore già ordinato.
Una sola iterazione, con (n-1) confronti e nessuno scambio: O(n)

Caso peggiore:
Vettore ordinato in senso decrescente.
Al passo di iterazione i, (n-i) confronti e (n-i) scambi.

n-1	n-2	...	3	2	1	confronti
n-1	n-2	...	3	2	1	scambi
1	*2*		*n-3*	*n-2*	*n-1*	*passo di iterazione*

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

Caso peggiore: esempio

20	8	5	0	-3

8	5	0	-3	20

5	0	-3	8	20

0	-3	5	8	20

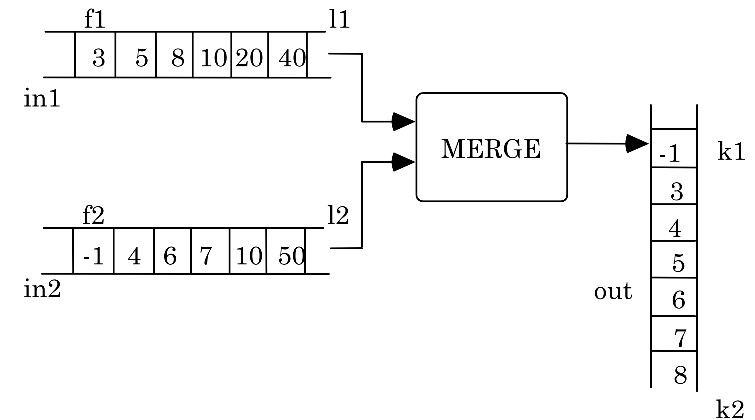
-3	0	5	8	20

Merge sort (ordinamento per fusione):

Utilizza, al proprio interno, l'algoritmo di ***fusione*** o ***merge***.

Merge:

Dati due vettori x, y ordinati in ordine crescente, con m componenti ciascuno, produrre un unico vettore z, di 2*m componenti, ordinato.



Algoritmo di merge:

Si scandiscono i due vettori di ingresso, confrontandone le componenti a coppie.

Merge dei vettori in1 (componenti da f1 ad l1) e in2 (componenti da f2 ad l2) nel vettore out (componenti da k1 a k2)

Indici i,j per scandire in1 e in2, indice k per scrivere su out.

Si confrontano in1[i] e in2[j]:

- *se in1[i] <= in2[j], scrive in1[i] nella componente k-esima di out (incrementa i, k);*
- *altrimenti, scrive in2[j] nella componente k-esima di out (incrementa j, k).*

Se la scansione di uno dei vettori è arrivata all'ultima componente, si copiano i rimanenti elementi dell'altro nel vettore out.

Complessità del merge:

Se in1, in2 hanno ciascuno n componenti:
    2*n confronti   (caso peggiore)
    2*n copie

2*n+2*n= 4*n = O(n)

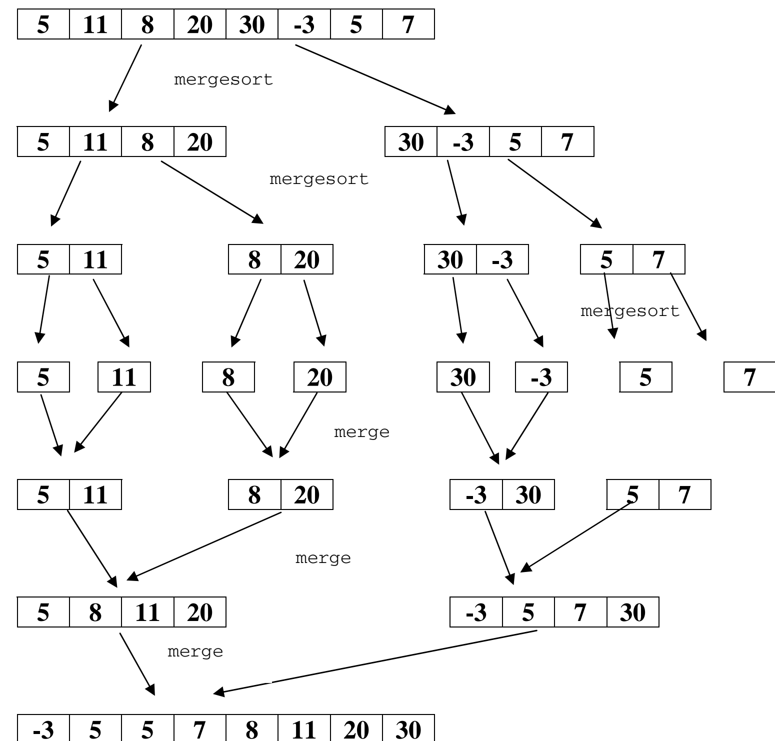
Ha complessità ***lineare***.

Merge sort (ordinamento per fusione):

È un algoritmo ***ricorsivo***.

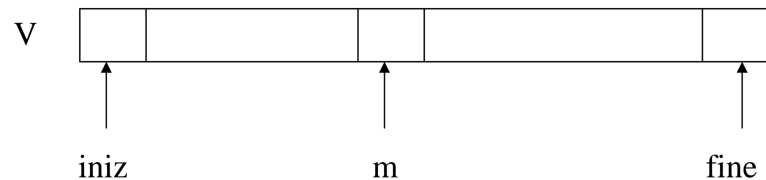
Il vettore di ingresso viene diviso in due sotto-vettori sui quali si richiama il merge sort.

Quando ciascun sotto-vettore è ordinato, i due vengono "fusi" attraverso la procedura di merge.



```
void mergesort (vettore v, int iniz, int
fine)
{   void merge(vettore v,int iniz1,int
iniz2,int fine);
    int m;
    if ( iniz < fine)
       {   m = (fine + iniz) / 2;
           mergesort(v, iniz , m);
           mergesort(v, m +1, fine);
           merge (v, iniz, m + 1,fine);
       }
}
```

Il parametro **v** rappresenta il vettore (o il sottovettore) da ordinare, **iniz** è l'indice del primo elemento e **fine** è l'indice dell'ultimo elemento.




```
#define N 8
typedef int vettore[N];

void merge (vettore v, int iniz1, int
iniz2,int fine)
{   vettore vout;
    int i, j, k;

i = iniz1; j = iniz2; k = iniz1;
/*confronto degli elementi correnti */
while (( i <= iniz2 -1) && ( j <= fine ))
    {   if (v [i] < v [j])
        {   vout [k] = v[i];
            i= i + 1; }
        else
        {   vout [k] = v[j];
            j= j + 1; }
        k = k + 1;
     }
/* copia degli elementi residui: */
while ( i <= iniz2 -1)
    {   vout [k] = v[i];
        i= i + 1;
        k = k + 1; }
while ( j <= fine )
    {   vout [k] = v[j];
        j= j + 1; k = k + 1; }

/* copia vout in v */
for (i = iniz1; i<= fine; i=i+1)
        v[i] = vout [i];
}
```

Chiamata:

```
merge(V,iniz, med+1,fine);
```

Complessità dell'algoritmo di merge-sort:

Il numero di attivazioni della procedura mergesort dipende dal numero di componenti del vettore da ordinare.

Per semplicità, caso di n potenza di due ($n=2^k$)

(0) 1 attivazione su un vettore di n ($=2^k$) componenti;
(1) 2 attivazioni su 2 vettori di n/2 ($=2^{k-1}$) componenti;
(2) 4 attivazioni su 4 vettori di n/4 ($=2^{k-2}$) componenti; ...
(i-1) 2^{i-1} attivazioni su 2^{i-1} vettori di $n/(2^{i-1})$ ($=2^{k-i+1}$) componenti;
...
(k) n attivazioni su n vettori di 1 componente.

$2^k=n \Rightarrow$
$k = \log_2 n$

Attivazioni di mergesort:

$1 + 2 + 4 + ... + 2^k = 2^0 + 2^1 + 2^2 + ... + 2^k =$

$= \Sigma_{(i=0..k)} 2^i = 2^{k+1} - 1 = O(n)$

Operazioni di confronto:

A ogni livello si fa il merge di M vettori di lunghezza L, dove:

	M	L
(0)	1	n
(1)	2	n/2
(2)	4	n/4
		...
(k+1)	n/2	2
(k)	n	1

poichè il numero di confronti nella procedura di *merge* è 2*L (caso peggiore), ogni livello compie al più:

$2*L*M/2 = L*M$ confronti per ogni livello

pertanto, il numero globale di confronti è:

$1*n + 2*n/2 + 4*n/4 + ... + n*1 = n*(k+1) = O(n*\log_2 n)$

La complessità asintotica dell'algoritmo di merge-sort è $O(n*\log_2 n)$:

$Costo_{merge_sort} = n*(k+1) = O(n*\log_2 n)$

In teoria è il "migliore" algoritmo di ordinamento (abbiamo assunto però nullo il costo di attivazione di una procedura).

Teorema:

Il problema dell'ordinamento di n elementi ha ***delimitazione inferiore*** $\Omega(n*\log_2 n)$.

Quick sort:

Come merge-sort, suddivide il vettore in due sotto-vettori, delimitati da un elemento “sentinella” (detto ***pivot***).

L’obiettivo è di avere nel primo sotto-vettore solo elementi minori o uguali al pivot e nel secondo sotto-vettore solo elementi maggiori.

```
void quicksort (vettore v, int iniz, int
fine)
{   int i, j, ipivot;
int  pivot, temp;
/* fase 1: */
<costruzione dei due sottovettori e
posizionamento del pivot v[ipivot]>

/* fase 2:*/
/* ricorsione sui due sottovettori */
if (<dim. del primo sottovettore non è 1>)
       quicksort (v, iniz, ipivot - 1);
if (<dim. del secondo sottovettore non è 1>)
       quicksort (v, ipivot + 1, fine);
}
```

Se nel primo sotto-vettore si incontra un elemento **`v[i]`** maggiore del **`pivot`** e nel secondo sotto-vettore un elemento **`v[j]`** minore del **`pivot`**, si scambiano **`v[i]`** e **`v[j]`**.

L’algoritmo è ***ricorsivo***: si richiama su ciascun sotto-vettore fino a quando non si ottengono sotto-vettori con un solo elemento.

A questo punto il vettore iniziale risulta ordinato.


```
void quicksort (vettore v, int iniz, int
fine)
{ int i, j, ipivot;
  int  pivot, temp;
  i=iniz; j=fine; ipivot=fine;
  pivot = v[ipivot];
/* individua la posizione del pivot e
  determina i due sottoinsiemi: */
do
{   while ((i < j)&&(v[i] <= pivot))
    i = i + 1;
    while ((j > i)&&(v[j] >= pivot))
        j = j - 1;
    if (i<j)
{   temp = v[i];
        v[i] = v[j];
        v[j] = temp;
}
} while (i < j);
/* posiziona il pivot: */
if ((i != ipivot) && (v[i] != v [ipivot]))
{   temp = v[i];
    v[i] = v[ipivot];
    v[ipivot] = temp;
    ipivot=i;
}
/* ricorsione sulle sottoparti*/
if (iniz < ipivot - 1)
    quicksort (v, iniz, ipivot - 1);
if (ipivot + 1< fine )
    quicksort (v, ipivot + 1, fine);
}
```

Esempio:

4	11	8	20	30	-3	5

`iniz=0, fine =6, ipivot=6, pivot=5.`

Prima iterazione del ciclo `do`: `i=0` e **`j=6`**.

Al termine del primo ciclo **`while`** si ottiene **`i=1`**.

Al termine del secondo ciclo **`while`** si ottiene **`j=5`**.

Vengono quindi scambiati tra loro **`v[1]`** e **`v[5]`**:

4	-3	8	20	30	11	5

Seconda iterazione del ciclo `do`: `i=1` e **`j=5`**.

Al termine del primo ciclo **`while`** si ottiene **`i=2`**.

Al termine del secondo ciclo **`while`** si ottiene **`j=2`**.

I valori di **`i`** e **`j`** coincidono: si esce dall'istruzione **`do`**.

Posizionamento del pivot: viene scambiato il pivot **`v[6]`** con l'elemento **`v[i]:`**

4	-3	5	20	30	11	8

Il pivot (l'elemento di valore 5) è quindi nella sua posizione definitiva.

Invocazione ricorsiva del **`quicksort`** sui due sottovettori contenenti rispettivamente gli elementi con indici appartenenti all'intervallo [0,1] e [3,6], con le chiamate **`quicksort(v,0,1)`** e **`quicksort(v,3,6)`**.

La chiamata **`quicksort(v,0,1)`** produce l'effetto:

-3	4	5	20	30	11	8

La chiamata **`quicksort(v,3,6)`** posiziona il nuovo pivot :

-3	4	5	8	30	11	20

e chiama ricorsivamente **`quicksort(v,4,6).`**

La chiamata **`quicksort(v,4,6)`** posiziona il nuovo pivot:

-3	4	5	8	11	20	30

Vengono individuati due sotto-vettori di dimensione unitaria e quindi già ordinati. A questo punto il vettore è completamente ordinato e l'esecuzione dell'algoritmo termina.

-3	4	5	8	11	20	30

Complessità dell'algoritmo di quicksort:

La procedura **`quicksort`** effettua un numero di confronti proporzionale alla dimensione L del vettore (L=fine-iniz+1).

Il numero di attivazioni di **`quicksort`** dipende dalla scelta del pivot.

Caso peggiore:
A ogni passo si sceglie un pivot tale che un sotto-vettore ha lunghezza 0:

attivazioni di **`quicksort`**: $k = n - 1$

numero di confronti:
al passo k si opera su un vettore di lunghezza L, ove:

k	L
1	n - 1
2	n - 2
...	
n-1	1

pertanto, il numero globale di confronti è:

$n-1 + n-2 + ... + 2 + 1 = n*(n-1)/2 = O(n^2)$

Anche se il vettore è già ordinato, il costo con questa scelta del pivot è $O(n^2)$.

Caso migliore:
A ogni passo si sceglie un pivot tale che il vettore si dimezza.

attivazioni di **`quicksort`**: $k = \log_2 n$

numero di confronti:
al passo k si opera su $M=2^k$ vettori di lunghezza L, ove:

k	M	L
0	1	n
1	2	n/2
2	4	n/4
	...	
$\log_2 n$	n	1

il numero di confronti ad ogni livello è L*M. Pertanto, il numero globale di confronti e':

$1*n + 2*n/2 + 4*n/4 +... + n*1 = n*(k+1) = O(n*\log_2 n)$

Caso medio:
La scelta casuale del pivot rende probabile la divisione in due parti aventi circa lo stesso numero di elementi.

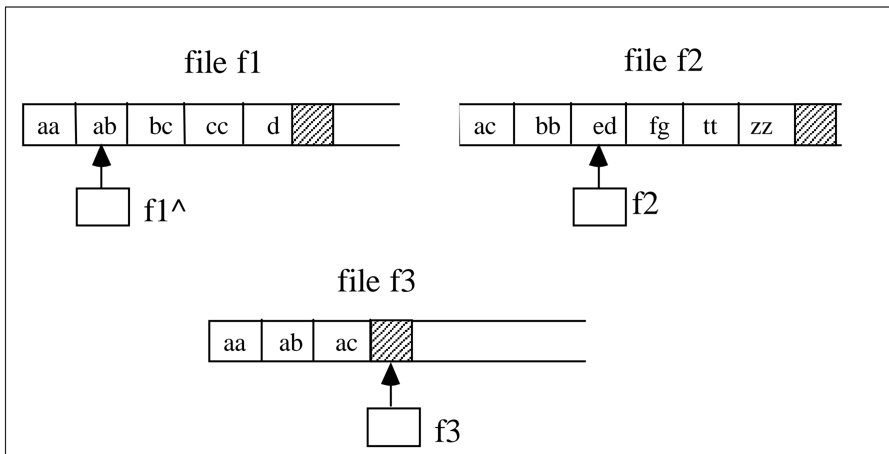
Si ha la stessa complessità del caso migliore:

$$O(n*\log_2 n)$$

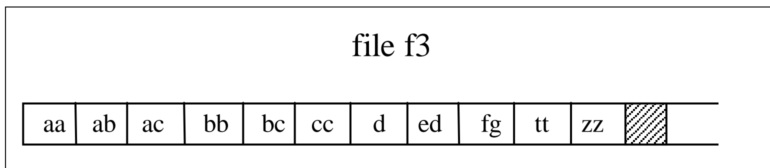
✐ Esercitazione 5.1:

Scrivere una procedura che esegue il merge di due file testo che contengono parole ordinate in senso lessicografico crescente in un terzo file, "merge.txt". Il contenuto del file merge.txt deve essere ordinato in senso lessicografico non decrescente (ci possono essere ripetizioni della stessa parola).

Merge di file ordinati:



Il risultato finale:



È un algoritmo importante perchè è utilizzato nell'ordinamento di file (sort-merge).


```
#include <stdio.h>
#include <string.h>
#define MAX 81
typedef char stringa[MAX];

main()
{
   FILE *f1, *f2, *f3;
   char nome1[10], nome2[10];
   stringa S1, S2;
   printf("Nome del primo file: ");
   scanf("%s", nome1);
   printf("Nome del secondo file: ");
   scanf("%s", nome2);
   f1=fopen(nome1, "r");
   f2=fopen(nome2, "r");
   f3=fopen("merge.txt", "w");
   fscanf(f1, "%s", S1);
   fscanf(f2, "%s", S2);

   while (!feof(f1) && !feof(f2))
   {   if (strcmp(S1,S2)<0)
       {   fprintf(f3, "%s\n", S1);
           fscanf(f1, "%s", S1);
       }
       else   if (strcmp(S1,S2)>0)
              {   fprintf(f3, "%s\n", S2);
                  fscanf(f2, "%s", S2);
              }
              else
              {   fprintf(f3, "%s\n", S1);
                  fscanf(f1, "%s", S1);
                  fprintf(f3, "%s\n", S2);
                  fscanf(f2, "%s", S2);
              }   }
```

```
   /* caso in cui termina f2 */
   while(!feof(f1))
   {  fprintf(f3, "%s\n", S1);
      fscanf(f1, "%s", S1);
   }

   /* caso in cui termina f1 */
   while(!feof(f2))
   {  fprintf(f3, "%s\n", S2);
      fscanf(f2, "%s", S2);
   }

   fclose(f1);
   fclose(f2);
   fclose(f3);
}
```


Funzione di ordinamento di libreria:

void qsort(void *base,size_t nelem,size_t width,
int (*fcmp)(void *elem1,void *elem2));

- **qsort** è una ***funzione generica di ordinamento*** (algoritmo quicksort) - libreria C - prototipo nell'header file stdlib.h
- **base** punta al primo elemento del vettore da ordinare
- **size_t** è un tipo definito in stdlib.h (in genere un unsigned long)
- **nelem** è il numero di elementi nel vettore
- **width** è la lunghezza di ogni elemento del vettore in byte
- **fcmp** è la funzione di confronto che deve essere definita dal programmatore e che deve essere passata alla qsort
  - i due **argomenti** sono i puntatori ai due elementi del vettore da confrontare
  - il **risultato** deve essere il seguente:

*elem1 < *elem2	un intero < 0
*elem1 == *elem2	0
*elem1 > *elem2	un intero > 0

Ordiniamo gli n elementi di un vettore di reali V

- base ⇒ V (che coincide con &V[0])
- nelem ⇒ n
- width ⇒ sizeof(V[0]) oppure sizeof(float)
- fcmp deve essere definita

```
#include <stdio.h>
#include <stdlib.h>
#define NMAX 50
int confronta(float *v1,float *v2);

main()
{ int n; float a[NMAX];
  n = leggi(a,NMAX);
  if(n == 0) exit(0);

  qsort(a,n,sizeof(float),confronta);

   scrivi(a,n);
}

int confronta (float *v1,float *v2)
{
  if(*v1 < *v2) return -1;
  else if(*v1 == *v2) return 0;
       else return 1;
}
```



```
int leggi(float v[],int DIM)
{
  int i,fine=0;
  clrscr();
  printf("\nInserire sequenza di reali:\n");
 /* terminata con l'inserimento di 0*/

  for (i=0;(i<DIM)&&(!fine);i++)
    {
      scanf("%f",&v[i]);
      if (v[i]==0)
       fine=1;
    }
  return i;
}


void scrivi(float v[],int num)
{
  int i;
  printf("\n Sequenza ordinata:");
  for (i=0;i<num;i++)
    printf("\n%f",v[i]);
}
```