



Ricerca ed ordinamento

Ing. Nadia Ranaldo
Dipartimento di Ingegneria
Università degli Studi del Sannio

Ricerca, ordinamento e fusione



- Ricerca (Searching)
 - Trovare un certo valore in un array
 - Due casi:
 - Array non ordinato
 - Array ordinato
- Ordinamento (Sorting)
 - Creare un array ordinato a partire da un array di elementi non ordinati
- Fusione (Merging)
 - Fondere due array ordinati in un unico array ordinato
- Questi concetti si possono applicare anche ad altre strutture dati utilizzate per contenere liste di elementi (come vedremo in seguito...)

2

Ricerca in un array non ordinato



- Algoritmo:
 - cerchiamo il valore richiesto nell'array a partire dal primo elemento e ci fermiamo:
 - quando troviamo l'elemento
 - o quando non abbiamo più elementi da analizzare
 - se il valore e' presente nell'array viene restituita la posizione del primo elemento in cui è stato trovato
 - altrimenti restituisce -1

Ricerca in un array non ordinato



```
int findInteger(int elem, int vet[], int n){  
    int i, trovato;  
    i=trovato=0;  
    while(!trovato && i<n)  
        if (elem == vet[i]) trovato=1;  
        else i++;  
    if (trovato)  
        return i;  
    else  
        return (-1);  
}
```

4

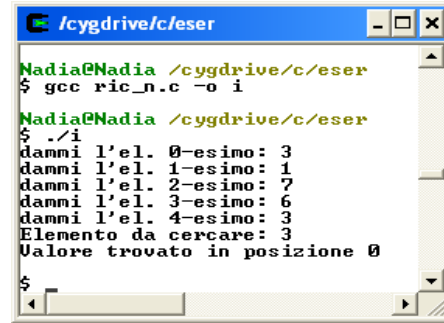
Ricerca in un array non ordinato



```

/* input riempimento ed array */
#include<stdio.h>
#define N 10
int findInteger(int elem, int vet[], int n);
main(){
  int elem, a[], pos;
  inputArray(a, N);
  printf("Elemento da cercare: ");
  scanf("%d", &elem);
  pos=findInteger(elem, a, N);
  if(pos==-1)
    printf("Valore non trovato");
  else
    printf("Valore trovato in posizione %d\n ",pos);
}

```



Ricerca in un array ordinato



- Ricerca lineare
- Ricerca binaria o logaritmica

Ricerca lineare



- Se l'array è ordinato in senso crescente, non è necessario arrivare alla fine dell'array per stabilire che l'elemento non è stato trovato ...
- Ci si può fermare appena si trova un elemento maggiore (o uguale) di quello dato ...
 - maggiore → non trovato !
 - uguale → trovato !
- Ovviamente, se l'elemento è maggiore di tutti quelli presenti nell'array, allora si visiterà l'intero array (caso peggiore) ...

Esempio



[4 6 7 10 12 15 18 20 24 30]

- La ricerca di 13 termina quando l'elemento corrente è 15 → fallimento
- La ricerca di 15 termina quando l'elemento corrente è 15 → successo
- La ricerca di 40 termina quando si sono visitati tutti gli elementi dell'array → fallimento

Programma ricerca lineare



```

...
i = 0;
trovato = 0;
while(i < n && !trovato)
    if (a[i] >= elem)
        trovato = 1;
    else i++;
if(trovato && (a[i] == elem))
    return i;
else return (-1);

```

trovato diventa 1 quando trovo un elemento >= elem

mi fa uscire dal ciclo

Short circuit

```

while(i < n && !trovato)
    if (a[i] >= elem) trovato = 1;
    else i++;
if(trovato && (a[i] == elem)) return i;
else return (-1);

```

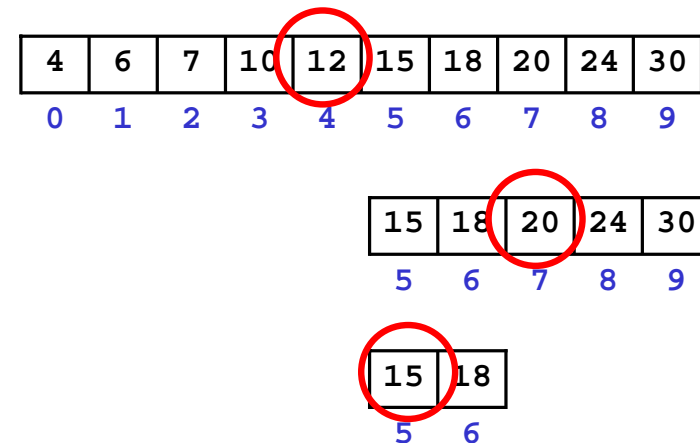
- trovato = 1 quando a[i] >= elem
- affinché l'elemento ci sia, deve essere vera la seguente espressione:
trovato && (a[i] == elem)
- nel valutare trovato && (a[i] == elem) potrebbe succedere (nel caso peggiore quando gli elementi dell'array sono tutti più piccoli di elem) che i == n. In questo caso trovato=0; poiché il C usa lo short circuit, non si valuta a[n]

Ricerca binaria (o logaritmica)



- Quando un array diventa molto grande la ricerca lineare è poco efficiente (pensando al caso peggiore)
- E' possibile seguire il ragionamento fatto dalle persone per cercare in un dizionario...
 - in modo da sfruttare maggiormente l'ordinamento
- Dividere l'array in due metà e confrontare l'elemento da cercare con l'elemento centrale dell'array
- uguali → trovato (... e ci si ferma)
- elemento dell'array maggiore → continuare la ricerca nella prima metà dell'array
- elemento dell'array minore → continuare la ricerca nella seconda metà dell'array
 - Dopo un confronto riesco ad eliminare metà degli elementi!

Esempio: cercare 15

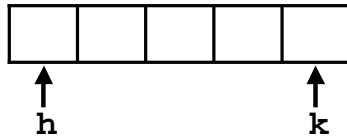


Algoritmo informale



elem : elemento da cercare

h e k: estremi dell'intervallo in cui cercare

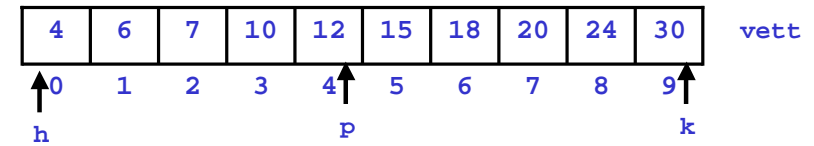


trovato: inizialmente vale 0. Il suo valore sarà 1 se trovo elem

finché non trovo elem (!trovato) e ci sono ancora elementi (h <= k):

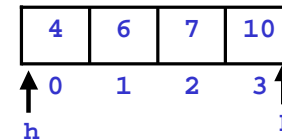
{

cerco l'elemento in posizione centrale $p = (h+k) / 2$

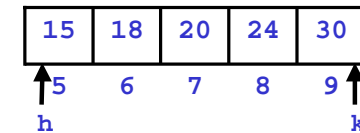


se sono uguali (vett[p]==elem) per es.: elem=12 → successo

se è maggiore (vett[p]>elem) per es.: elem=7 → k = p-1



se è minore (vett[p]<elem) per es.: elem=20 → h = p+1



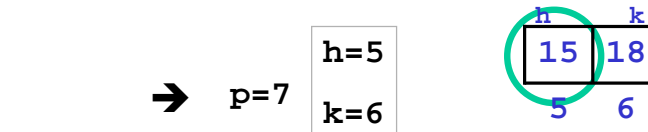
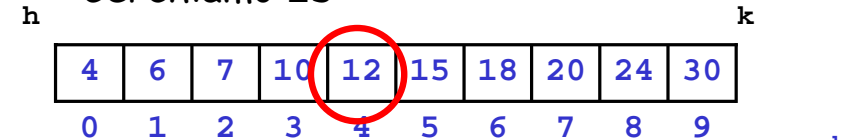
}

Programma



```
int p,h,k;
h = 0; k = n-1; /* estremi dell'intervallo in cui
ricercare*/
trovato = 0; /* inizialmente elem non è stato trovato*/
while (h <= k && !trovato) {
    p = (h + k) / 2; /* posizione centrale */
    if (vet[p] == elem)
        trovato = 1;
    else if(vet[p] > elem)
        k = p-1; /*la ricerca continua nella I metà*/
    else h = p+1; /*la ricerca continua nella II metà*/
}
if(trovato)
    return p;
else
    return (-1);
```

Cerchiamo 15



→ p=5 vett[p]=15 successo

Numero di confronti = 3

Ricerca lineare
Numero di confronti = 6

Confronto tra ricerca lineare e ricerca binaria



- Confronto dei due algoritmi di ricerca sulla base dell'efficienza
- Consideriamo il numero di confronti effettuati dai due algoritmi di ricerca nel caso peggiore
- La ricerca lineare richiede di eseguire nel caso peggiore N confronti (N è il numero degli elementi)
- Nella ricerca binaria invece il primo confronto fa diminuire il numero di elementi a N/2, il secondo a N/4, il terzo a N/8, fino a 1 elemento, caso in cui occorre fare solo un confronto
- Il numero di passi necessari per arrivare a questo punto è dato dal numero di volte che occorre dividere N per 2 al fine di ottenere 1:

$$(((N/2)/2)/2)/\dots/2=1 \text{ k volte} \Rightarrow N/2^k=1$$

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

17

Confronto tra ricerca lineare e ricerca binaria



- Quindi nel caso peggiore si visitano $\log_2 N$ elementi
- Esempio: cercare 80

5	7	8	10	17	20	25	30	33	36	38	50	60	70	80	90
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

33	36	38	50	60	70	80	90
8	9	10	11	12	13	14	15

4 ($\log_2 16$) confronti

60	70	80	90
12	13	14	15

Con la ricerca lineare:
15 confronti

80	90
14	15

18

Confronto tra N e $\log_2(N)$



N	$\log_2(N)$
8	3
16	4
64	6
256	8
1024 (1K)	10
16,384	14
131,072	17
262,144	18
524,288	19
1,048,576 (1Meg)	20
1,073,741,824 (1Gig)	30

19

Ordinamento



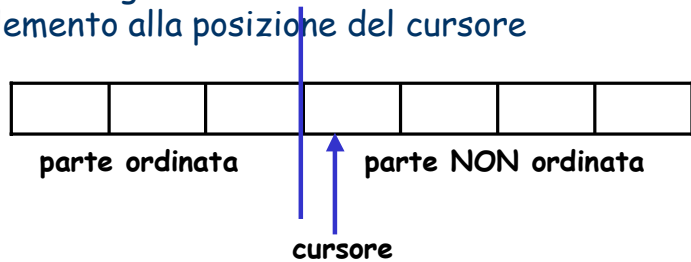
- Ordinare gli elementi di un vettore in ordine crescente o decrescente
 - Gli elementi possono essere interi, float, ma anche frasi (stringhe) da ordinare in ordine alfabetico
 - Il problema è lo stesso: data una lista di elementi ed un meccanismo per eseguire la comparazione tra due elementi, come riarrangiare gli elementi all'interno dell'array affinché tali elementi siano opportunamente ordinati?
- Esistono molti algoritmi
 - Selection sort (o ordinamento per minimi successivi)
 - Bubble sort (o ordinamento per scambi)
 - Insert sort
 - Etc.

20

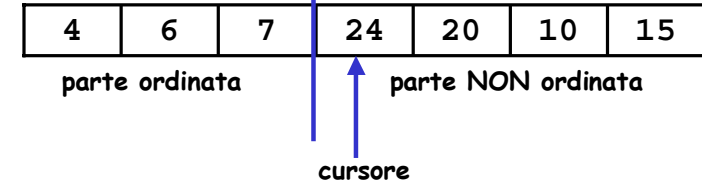
Selection Sort



- Si sistema al posto giusto un elemento alla volta
 - Si ripetono le operazioni per ogni elemento
 - Il vettore viene diviso in due parti da un indice (cursore) che scorre dalla prima alla penultima posizione. Ad ogni passo si cerca l'elemento minimo tra quelli della II parte e lo si mette nella posizione del cursore; quindi si fa avanzare il cursore di una posizione
 - Viene eseguito uno scambio tra il minimo trovato e l'elemento alla posizione del cursore

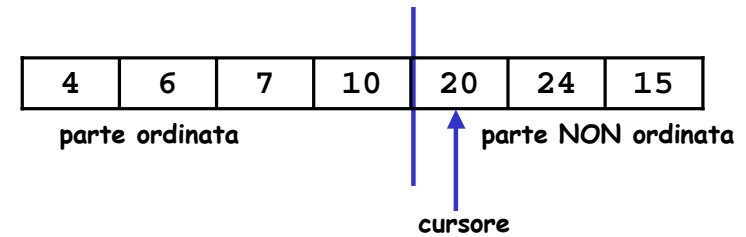


Idea scambio

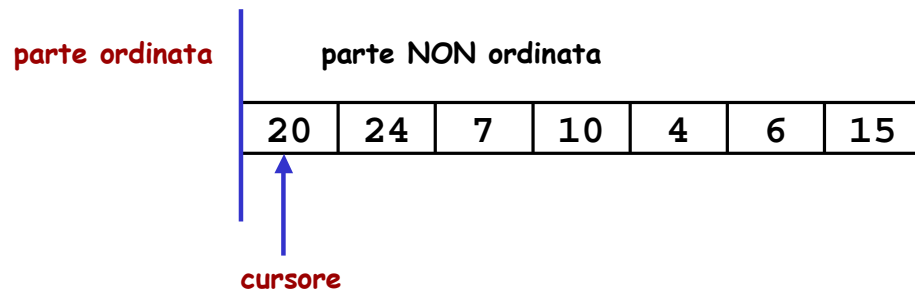


Trovo il minimo nella parte NON ordinata 10

Scambio



Situazione iniziale



Selection Sort



- Un esempio:

seq. originale:	3	9	6	1	2
il più piccolo è 1:	1	9	6	3	2
il più piccolo è 2:	1	2	6	3	9
il più piccolo è 3:	1	2	3	6	9
il più piccolo è 6:	1	2	3	6	9



```
void sortArray (int vett[], int n){
  /* input: array e riempimento */
  for(i = 0; i < n-1; i++)
  {
    Individua la posizione pmin dell'elemento
    minimo compreso tra le posizioni i e n-1 di
    vett

    Scambia gli elementi di posizione i e pmin
  }
```

cursore

25



```
/* individua la posizione pmin del minimo */
min = vett[i];
pmin = i;
for (j = i+1; j < n; j++)
  if (vett[j] < min) {
    min = vett[j];
    pmin = j;
  }

/* scambia gli elementi di posizione i e pmin */

temp = vett[i];
vett[i] = vett[pmin];
vett[pmin] = temp;
```

26



```
void sortArray(int vett[], int n){
  int i, j, min, pmin, temp;
  for(i = 0; i < n-1; i++) {
    min = vett[i];
    pmin = i;
    for (j = i+1; j < n; j++)
      if (vett[j] < min){
        min = vett[j];
        pmin = j;
      }
    temp = vett[i];
    vett[i] = vett[pmin];
    vett[pmin] = temp;
  }
```



```
#include<stdio.h>
#define N 7
void sortArray(int vett[], int n);
main() {
  int vett[N] = {1,110,23,4,55,1,7};
  int i;
  sortArray(vett,N);
  /* stampa il vettore ordinato */
  for (i=0; i<N; i++)
    printf("%d\n", vett[i]);
}
```

```
Nadia@Nadia /cygdrive
$ gcc select.c -o i
Nadia@Nadia /cygdrive
$ ./i
1
1
4
7
23
55
110
```

Efficienza dell' algoritmo di Selection Sort



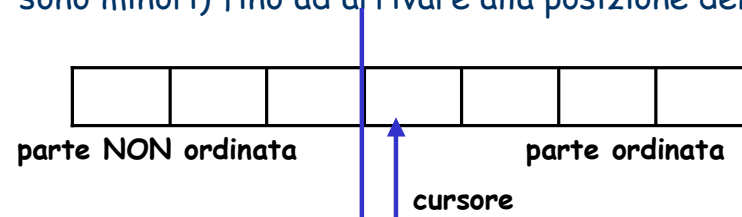
- Fissata la dimensione dei dati di input (N elementi nell'array), quanto è efficiente questo algoritmo di ordinamento?
- Consideriamo le operazioni di confronto che occorre effettuare:
 - Per ogni elemento dell'array (N), viene eseguito un ciclo per cercare l'elemento minimo, la prima volta tra N elementi, poi tra N-1, etc.
 - Il numero di operazioni in totale è proporzionale a
 - $N+(N-1)+(N-2)+\dots+1 = N(N+1)/2 = (N^2+N)/2$
 - Di conseguenza possiamo dire che l'algoritmo è **quadratico**, ovvero il numero di operazioni cresce in maniera proporzionale al quadrato di N (dimensione dei dati di input)

29

Bubble Sort

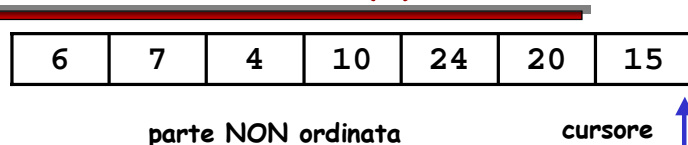


- Analogamente al Selection Sort, si sistema al posto giusto un elemento alla volta
 - Il vettore viene diviso in due parti da un indice (cursore) che scorre dall'ultima alla prima posizione. Ad ogni passo si cerca l'elemento massimo tra quelli della I parte (non ordinata) e lo si sposta nella posizione del cursore; quindi si fa decrescere il cursore di uno
 - L'elemento massimo viene spostato eseguendo degli scambi successivi tra elementi adiacenti (quando questi sono minori) fino ad arrivare alla posizione del cursore

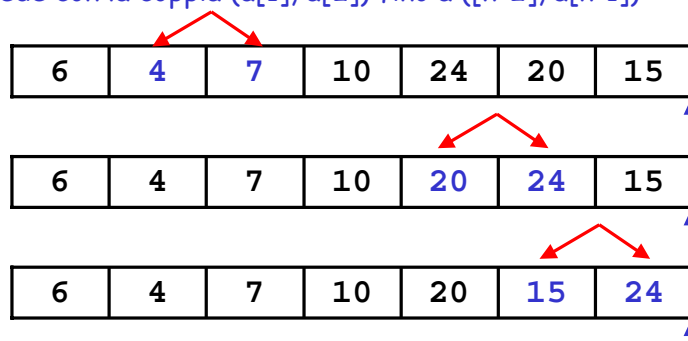


30

Idea (1)



- Inizialmente la parte non ordinata è tutto l'array quindi il cursore è la posizione n-1
- Si analizza la coppia (a[0], a[1]), se $a[0] > a[1]$ si esegue lo scambio (ovvero si scambia se la coppia non è ordinata)
- Si procede con la coppia (a[1], a[2]) fino a ([n-2], a[n-1])

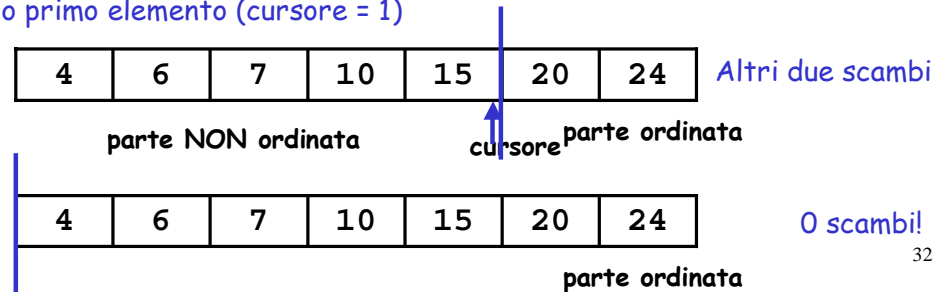


31

Idea (2)



- Al termine del ciclo l'elemento massimo occupa la posizione a[n-1], alla quale sarà pervenuto dopo un certo numero di scambi (da qui il paragone con la bolla che gorgoglia verso l'alto)
- Il procedimento viene ripetuto con riferimento alla nuova parte non ordinata e il cursore alla posizione n-2, e così via fino ad arrivare al solo primo elemento (cursore = 1)



32

0 scambi!

Algoritmo Bubble Sort: informale



```
void sortBubbleArray (int vett[], int n){
/* input: array e riempimento */

for(i = 0; i < n-1; i++)
{
    esegue lo scambio degli elementi adiacenti
    non ordinati tra le posizioni 0 e n-i
    (ad ogni passo n-i rappresenta il cursore

}
```

33

Algoritmo Bubble Sort (1)



```
/* esegue lo scambio degli elementi adiacenti
non ordinati tra le posizioni 0 e n-i */

for(j=0;j<n-i;j++) {
    if(vett[j]>vett[j+1]) {
        temp = vett[j];
        vett[j] = vett[j+1];
        vett[j+1] = temp;
    }
}
```

34

Algoritmo Bubble Sort (2)



```
void sortBubbleArray(int vett[], int n){
    int i, j, temp;
    for(i = 0; i < n-1; i++) {
        for(j=0;j<n-i;j++) {
            if(vett[j]>vett[j+1]) {
                temp = vett[j];
                vett[j] = vett[j+1];
                vett[j+1] = temp;
            }
        }
    }
}
```

35

Algoritmo Bubble Sort (3)



```
#include<stdio.h>
#define N 7
void sortBubbleArray(int vett[], int n);
main() {
    int vett[N] = {1,110,23,4,55,1,7};
    int i;
    sortBubbleArray(vett,N);
    /* stampa il vettore ordinato */
    for (i=0; i<N; i++)
        printf("%d\n", vett[i]);
}
```

```
~/cygdrive/c/eser
Nadia@Nadia /cygdrive/
$ gcc bubble.c -o b
Nadia@Nadia /cygdrive/
$ ./b
1
1
4
7
23
55
110
```

Bubble Sort: Considerazioni



- L'ultimo scambio effettuato in un ciclo determina la posizione a partire dalla quale la lista è ordinata:
 - Registrando la posizione u dove tale scambio è stato effettuato, è possibile limitare il successivo ciclo di scansione alla sottolista $a[1]...a[u]$
- Nel caso in cui in un ciclo non vengano effettuati scambi (come nell'esempio presentato), l'algoritmo può terminare precocemente

37

Algoritmo Bubble Sort (4)



```
void sortBubbleArray(int vett[], int n){
    int i, j, temp;
    int scambi = TRUE;
    i = 0;
    while(i < n && scambi) {
        scambi = FALSE;
        for(j=0; j<n-i; j++) {
            if(vett[j]>vett[j+1]) {
                temp = vett[j];
                vett[j] = vett[j+1];
                vett[j+1] = temp;
                scambi=TRUE;
            }
        }
        i++;
    }
}
```

38

Efficienza dell'algoritmo di Bubble Sort



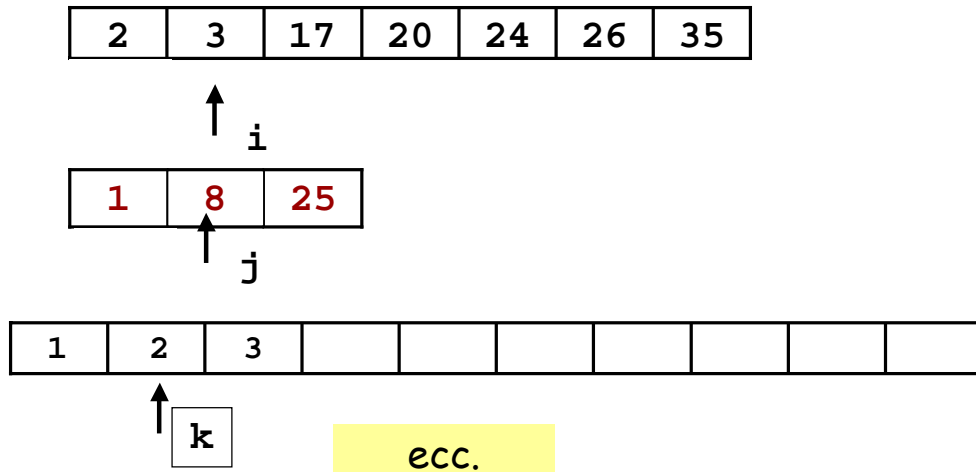
- Fissata la dimensione dei dati di input (N elementi nell'array), quanto è efficiente questo algoritmo di ordinamento?
- Consideriamo le operazioni di confronto e scambio che occorre effettuare:
 - N cicli in cui vengono analizzati un numero decrescente di elementi da N a 1
 - In ogni ciclo vengono effettuati confronti ed operazioni di scambio
 - Il numero di operazioni in totale è proporzionale a
 - $N+(N-1)+(N-2)+...+1=$
 $= N(N+1)/2 = (N^2+N)/2$
 - Di conseguenza possiamo dire che l'algoritmo è **quadratico**, analogamente all'algoritmo di Selection Sort
 - Ma in questo caso le operazioni ripetute non sono solo confronti ma anche scambi, quindi è più inefficiente rispetto al Selection Sort

Fusione



- A partire da due array ordinati si ricava un terzo array anch'esso ordinato
- La dimensione dei due array non necessariamente deve essere la stessa

40



```

...
int vet1[N], vet2[M], vet3[N+M];
int i=0, j=0, k=0;
do {
    if (vet1[i] <= vet2[j]){
        vet3[k] = vet1[i];
        i++;
        k++;
    } else {
        vet3[k] = vet2[j];
        j++;
        k++;
    }
} while(i < N && j < M);

```

```

if (i < N) for (; i < N; i++, k++) vet3[k] = vet1[i];
else for (; j < M; j++, k++) vet3[k] = vet2[j];

```