STRUTTURE DATI - ALBERI

Sommario

Introduzione
Strutture ricorsive
Alberi
Alberi binari di ricerca
Ricerca e ordinamento
Complessità

Introduzione

Strutture dati dinamiche – Il numero dei loro elementi crescono e decrescono durante l'esecuzione del programma

Liste concatenate – Si possono inserire ed eliminare elementi in qualunque posizione all'interno della lista

Pile – Inserimenti ed eliminazioni possono essere fatti solo sulla "testa" della pila

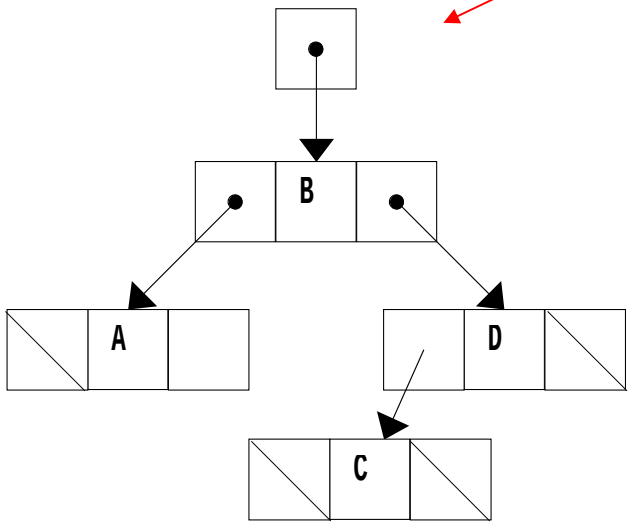
Code – Gli inserimenti possono essere fatti solo in "fondo" alla coda e le eliminazioni solo in "testa"

Alberi binari – Consentono la ricerca e l'ordinamento di dati in maniera veloce ed efficiente, come pure l'eliminazione e l'inserimento

Alberi

- Ogni nodo dell'albero può contenere due o più collegamenti
  - Le strutture viste finora hanno al più un collegamento
- Alberi binari
  - Ogni nodo ha 2 collegamenti
    - Tutti e due, uno o nessuno possono essere **`NULL`**
  - Il nodo radice è il primo nodo dell'albero.
  - Ogni collegamento fa riferimento ad un nodo *figlio* - *child*
  - Un nodo senza figli è detto nodo *foglia*

Puntatore alla radice



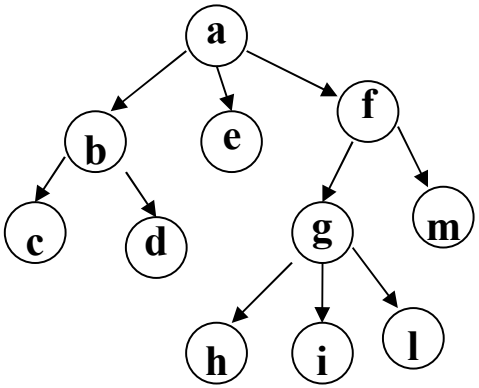
B è il nodo radice
D è il figlio ds. di B
A è figlio sin. di B (foglia)
C è figlio sin. di D (foglia)

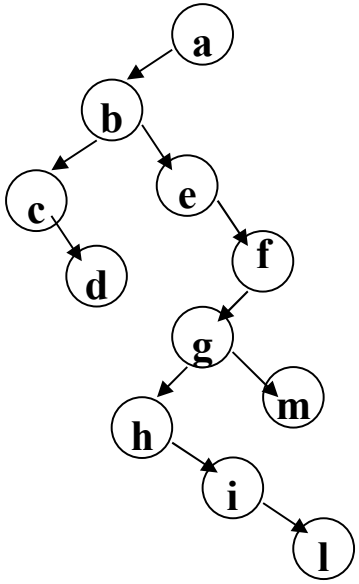
```
struct treeNode {
struct treeNode *leftPtr;
int data;
struct treeNode *rightPtr;
};
```

Trasformazione di alberi

Da un albero ordinato (da sinistra verso destra) A di n nodi è possibile ricavare un equivalente albero binario B di n nodi con la regola:

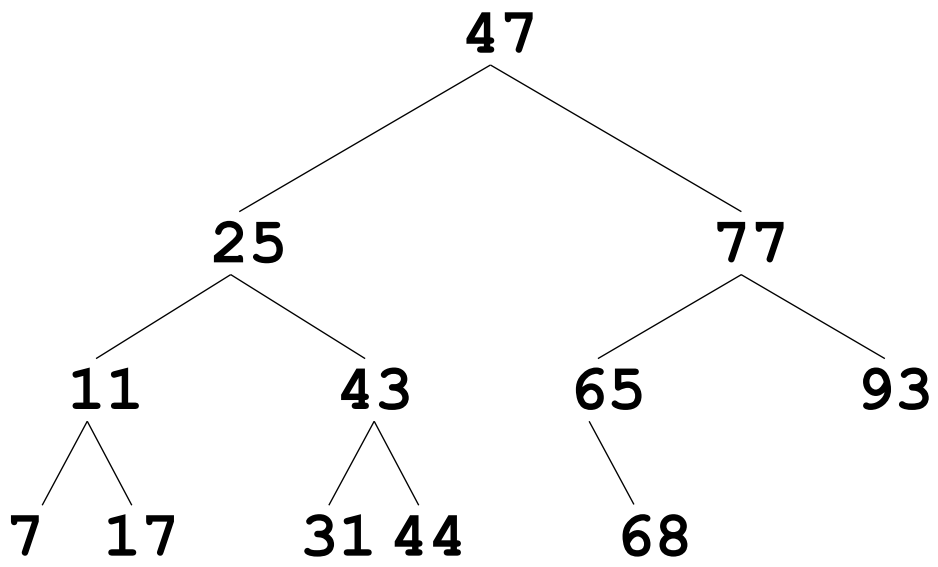
- La radice di A coincide con la radice di B;
- Ogni nodo b di B ha come radice del sottoalbero sinistro il primo figlio di b in A e come sottoalbero destro il fratello successivo di b in A.





Alberi binari di ricerca

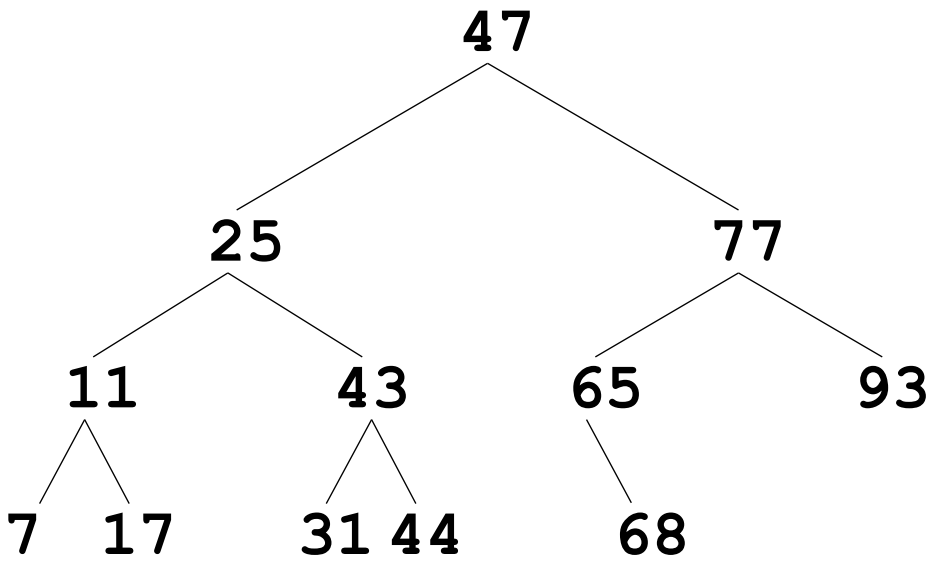
- Alberi binari di ricerca (ricorsivi)
  - I nodi di ogni sottoalbero di sinistra contengono valori < del nodo padre
  - I nodi di ogni sottoalbero di destra contengono valori > del nodo padre
  - Facilita l'eliminazione di doppioni
  - Ricerca veloce – per un albero "bilanciato" di n nodi, occorrono $\log_2 n$ confronti



Attraversamento degli alberi binari

- Sono funzioni ricorsive:
  - Attraversamento simmetrico – si ottengono i **nodi in ordine crescente**
    1. Si attraversa il sottoalbero sinistro in ordine simmetrico.
    2. Si elabora il valore del nodo (ad es., si stampa).
    3. Si attraversa il sottoalbero destro in ordine simmetrico.
  - Attraversamento anticipato:
    1. Si elabora il valore del nodo (ad es., si stampa).
    2. Si attraversa il sottoalbero sinistro in ordine anticipato.
    3. Si attraversa il sottoalbero destro in ordine anticipato.
  - Attraversamento differito:
    1. Si attraversa il sottoalbero sinistro in ordine differito.
    2. Si attraversa il sottoalbero destro in ordine differito.
    3. Si elabora il valore del nodo (ad es., si stampa).

Esempio

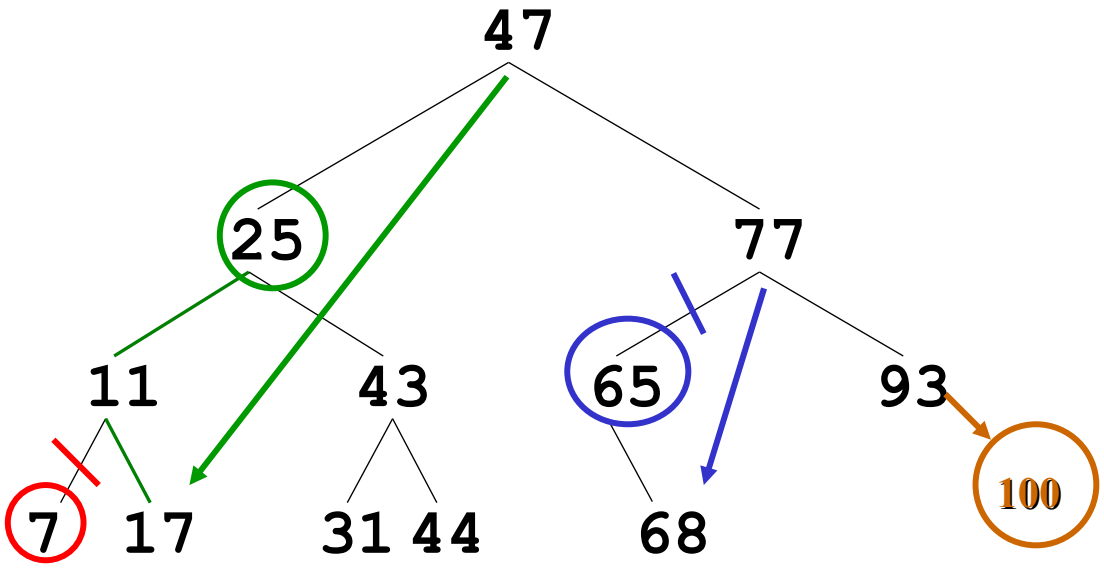


Simmetrico: 7 11 17 25 31 43 44 47 65 68 77 93

Anticipato: 47 25 11 7 17 43 31 44 77 65 68 93

Differito: 7 17 31 44 11 43 25 68 65 93 77 47

Inserimenti e cancellazioni



-1- Cancellazione foglia

-2- Cancellazione nodo intermedio senza figli a sin.

-3- Cancellazione nodo intermedio con figlio a sin.

-4- Inserimento nodo (foglia)

Alberi binari di ricerca: inserimenti e cancellazioni

Inserimento di un nuovo nodo:

Si aggiunge un nodo foglia nella posizione che gli compete: il puntatore (ds. o sin.) del nodo padre, punta al nuovo nodo.

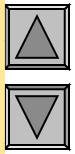
Cancellazione di un nodo:

Nodo foglia: Il puntatore del padre viene posto = NULL.

Nodo con 1 figlio: Il puntatore del padre viene posto = al puntatore al figlio.

Nodo con 2 figli: Si attraversa il sottoalbero sin. andando sempre a ds., fino ad un nodo con puntatore ds.=NULL. Questo è il nodo di sostituzione.

Il nodo padre di quello da eliminare punterà a ds. o a NULL (se il nodo da eliminare è foglia) o al suo (unico) figlio sin. Il puntatore ds. (sin.) del nodo di sostituzione punterà al sottoalbero ds. (sin) del nodo da eliminare.



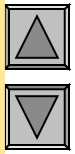
Outline

```
1  /* Fig. 12.19: fig12_19.c
2     Creazione di un albero binario ed attraversamento
3     in ordine simmetrico, anticipato, differito */
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <time.h>
7
8  struct treeNode {
9     struct treeNode *leftPtr;
10    int data;
11    struct treeNode *rightPtr;
12 };
13
14 typedef struct treeNode TreeNode;
15 typedef TreeNode *TreeNodePtr;
16
17 void insertNode( TreeNodePtr *, int );
18 void inOrder( TreeNodePtr );
19 void preOrder( TreeNodePtr );
20 void postOrder( TreeNodePtr );
21
22 int main()
23 {
24    int i, item;
25    TreeNodePtr rootPtr = NULL;
26
27    srand( time( NULL ) );
28
```

Definizione della struttura del nodo

Prototipi delle funzioni

```
29    /* insert random values between 1 and 15 in the tree */
30    printf( "The numbers being placed in the tree are:\n" );
31
32    for ( i = 1; i <= 10; i++ ) {
33       item = rand() % 15;
34       printf( "%3d", item );
35       insertNode( &rootPtr, item );
36    }
37
38    /* traverse the tree preOrder */
39    printf( "\n\nThe preOrder traversal is:\n" );
40    preOrder( rootPtr );
41
42    /* traverse the tree inOrder */
43    printf( "\n\nThe inOrder traversal is:\n" );
44    inOrder( rootPtr );
45
46    /* traverse the tree postOrder */
47    printf( "\n\nThe postOrder traversal is:\n" );
48    postOrder( rootPtr );
49
50    return 0;
51 }
52
53 void insertNode( TreeNodePtr *treePtr, int value )
54 {
55    if ( *treePtr == NULL ) {     /* *treePtr is NULL */
56       *treePtr = malloc( sizeof( TreeNode ) );
57
58       if ( *treePtr != NULL ) {
59          ( *treePtr )->data = value;
60          ( *treePtr )->leftPtr = NULL;
61          ( *treePtr )->rightPtr = NULL;
62       }
```



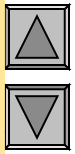
Outline

Inserisce numeri casuali compresi fra 1 e 15 nell'albero binario di ricerca

Chiamata delle funzioni di attraversamento dell'albero

Definizione della funzione di inserimento

```
63       else
64          printf( "%d not inserted. No memory available.\n",
65                  value );
66    }
67    else
68       if ( value < ( *treePtr )->data )
69          insertNode( &( ( *treePtr )->leftPtr ), value );
70       else if ( value > ( *treePtr )->data )
71          insertNode( &( ( *treePtr )->rightPtr ), value );
72       else
73          printf( "dup" );
74 }
75
76 void inOrder( TreeNodePtr treePtr )
77 {
78    if ( treePtr != NULL ) {
79       inOrder( treePtr->leftPtr );
80       printf( "%3d", treePtr->data );
81       inOrder( treePtr->rightPtr );
82    }
83 }
84
85 void preOrder( TreeNodePtr treePtr )
86 {
87    if ( treePtr != NULL ) {
88       printf( "%3d", treePtr->data );
89       preOrder( treePtr->leftPtr );
90       preOrder( treePtr->rightPtr );
91    }
92 }
```



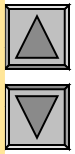
Outline

Funzione di inserimento (cont.)

Funzione attraversamento in ordine simmetrico

Funzione di attraversamento in ordine anticipato

```
93
94 void postOrder( TreeNodePtr treePtr )
95 {
96    if ( treePtr != NULL ) {
97       postOrder( treePtr->leftPtr );
98       postOrder( treePtr->rightPtr );
99       printf( "%3d", treePtr->data );
100   }
101}
```



Outline

Funzione di attraversamento in ordine differito

```
The numbers being placed in the tree are:
  7   8   0   6 14   1   0dup 13   0dup   7dup

The preOrder traversal is:
  7   0   6   1   8 14 13

The inOrder traversal is:
  0   1   6   7   8 13 14

The postOrder traversal is:
  1   6   0 13 14   8   7
```

Output

/* Creazione di un albero binario e visita in ordine anticipato. L'etichetta dei nodi è un valore intero, le occorrenze multiple dello stesso valore non vengono memorizzate */

```
#include <stdio.h>
#include <stdlib.h>
struct nodo {
  int inf;
  struct nodo *albSin;
  struct nodo *albDes;
};
struct nodo *albBin(void);
struct nodo *creaNodo(struct nodo *, int);
void anticipato(struct nodo *);

main()
{
struct nodo *radice;    /* puntatore alla radice
                           dell'albero  */

radice = albBin();      /*chiama la funzione per
                la creazione dell'albero binario */
printf("\nVISITA IN ORDINE
ANTICIPATO\n");
anticipato(radice);
}
```

```
/* Crea l'albero binario. Per ogni etichetta
      immessa dall'utente, invoca la funzione
      creaNodo. Ritorna al chiamante la radice
      dell'albero   */

struct nodo *albBin(void)
{
struct nodo *p = NULL;
struct nodo x;

do {
printf("\nInserisci una informazione (0 per
      finire): ");
  scanf("%d", &x.inf);

  if(x.inf!=0)
    p = creaNodo(p, x.inf);      /* chiama
                                      creaNodo() */
}
while (x.inf!=0);

return(p);                  /* resituisce la radice */
}
```


```
/* Visita ricorsivamente l'albero alla ricerca del
punto di  inserimento. Quando trova la posizione,
crea un nodo, vi  inserisce l'etichetta e ritorna il
puntatore a tale nodo.
Parametri in ingresso:
      p      e' il puntatore alla radice
      val    e' l'etichetta da inserire nel nodo   */

struct nodo *creaNodo(struct nodo *p, int val)
{
if(p==NULL) {       /* il punto di inserimento e'
                           stato reperito */
  /* Creazione del nodo */
  p = (struct nodo *) malloc(sizeof(struct nodo));
  p->inf = val;           /* inserimento di val in
elemento */
  p->albSin = NULL;       /*albero sinistro vuoto*/
  p->albDes = NULL;       /*albero destro vuoto */
}
else {           /* ricerca del punto di inserimento*/
if(val > p->inf)
    /* Visita il sottoalbero destro */
    p->albDes = creaNodo(p->albDes, val);
  else
    if(val < p->inf)
      /* Visita il sottoalbero sinistro */
      p->albSin = creaNodo(p->albSin, val);
}
return(p);                /* ritorna il puntatore
                                        alla radice
*/
}

/* Visita l'albero binario in ordine anticipato
*/

void anticipato(struct nodo *p)
{
if(p!=NULL) {
  printf("%d  ", p->inf);   /*visita la radice*/
  anticipato(p->albSin);    /*visita il

sottoalbero sinistro */
  anticipato(p->albDes);     /* visita il

sottoalbero destro*/
}
}
```


Modifica della funzione creaNodo, dove si calcola il numero di occorrenze di uno stesso valore, memorizzandole nel campo occorrenze del nodo stesso

```
Struct nodo {
int inf;
int occorrenze;
Struct nodo *albSin;
Struct nodo *albDes;
}
```

```
struct nodo *creaNodo2(struct nodo *p, int val)
{
if(p==NULL) {
  p = (struct nodo *) malloc(sizeof(struct nodo));
  p->inf = val;
  p->occorrenze = 1;
  p->albSin = NULL;
  p->albDes = NULL;
}
else {
  if(val > p->inf)
    p->albDes = creaNodo2(p->albDes, val);
  else
    if(val < p->inf)
      p->albSin = creaNodo2(p->albSin, val);
    else
      ++p->occorrenze;
}
return(p);
}
```


Esercizio

Dati due vettori di numeri reali, V1 e V2, di dimensione n=16, sia “Inf” un generico numero reale da allocare in un albero binario.

3. Scrivere una funzione che, visitando un albero binario, calcoli la somma dei valori già allocati.
4. Scrivere una funzione che, visitando un albero binario, calcoli il numero dei nodi già allocati.
5. Scrivere una funzione che costruisca un albero binario con il criterio:

- se “InfRadice”>= della media dei valori già allocati nell’albero, allocare “Inf” nel sottoalbero destro;
- se “InfRadice”< della media dei valori già allocati nell’albero, allocare “Inf” nel sottoalbero sinistro.

La media dovrà essere calcolata con le funzioni dei punti 1 e 2.

Scrivere un programma principale che preveda l’introduzione dei dati dei vettori V1 e V2 da tastiera ed utilizzi le funzioni 1-2 per costruire due alberi binari con il criterio della funzione al punto 3.

```
#include <stdio.h>
#include <stdlib.h>

#define MAXN 16
/* Creazione di un albero binario */
struct nodo {
   float inf;
   struct nodo *alb_sin;
   struct nodo *alb_des;
};
int Nnodi(struct nodo *p);
int Somma(struct nodo *p);
struct nodo *crea_nodo(struct nodo *p,float
      val,float M);

void main()
{
int v1[MAXN], v2[MAXN];
int j, n,Nn;
float M,S;
struct nodo *p1, *p2;
  p1=NULL;
  p2=NULL;
 for (j=0;j<MAXN;j++) scanf("%d",&v1[j]);
for (j=0;j<MAXN;j++) scanf("%d",&v2[j]);
```

```
/*L'esercizio prevede la introduzione dei dati da
tastiera.*/
for (j=0;j<MAXN;j++) {
                Nn=Nnodi(p1);
                S=Somma(p1);
                if(Nn>0)  M=S/Nn;
                  else M=0;
                p1=crea_nodo(p1,v1[j],M);
   }
   for (j=0;j<MAXN;j++) {
                Nn=Nnodi(p2);
                S=Somma(p2);
                if(Nn>0) M=S/Nn;
                  else M=0;
                p2=crea_nodo(p2,v2[j],M);
   }
}
```


```
/* funzione1*/
int Somma(struct nodo *p)
{
float v;
v=0;
if (p!=NULL){
      v=p->inf;
      v=Somma(p->alb_des)+v;
      v=Somma(p->alb_sin)+v;
 }
 return v;
}
/* funzione2*/
int Nnodi(struct nodo *p)
{
int v;
v=0;
if (p!=NULL){
      v=1;
      v=Nnodi(p->alb_des)+v;
      v=Nnodi(p->alb_sin)+v;
 }
 return v;
}
```

```
/* funzione 3 */

struct nodo *crea_nodo(struct nodo *p,float
val,float M)

{

if(p==NULL){

  p=(struct nodo*)malloc(sizeof(struct nodo));

  p->inf=val;

  p->alb_sin=NULL;

  p->alb_des=NULL;

}

else{

    if(p->inf >=M) p->alb_des=crea_nodo(p-
>alb_des,val,M);

    else p->alb_sin=crea_nodo(p-
>alb_sin,val,M);

}

return (p);

}
```


Algoritmi di ricerca e ordinamento

Vediamo degli algoritmi per:

- trovare un elemento in un array ordinato
- ordinare un array

Problema della ricerca: decidere se un intero si trova in un vettore.

Problema dell'ordinamento: Il vettore è ordinato se il primo elemento è minore del secondo, che è minore del terzo, ecc.

Es.:

Vettori ordinati:

-1 0 3 10 60 120 900

-100 4 20

0 1 2 3 4 50

Vettori non ordinati:

-1 0 3 10 9 120 900

-100 4 20 0

1 0 1 2 3 4 50

Ordinamento per selezione

E' un algoritmo semplice che si compone di n-1 iterazioni (n=dimensione del vettore A da ordinare):

0 - Si cerca la componente di valore più piccolo, A[i_min], in A[0…n-1], e si scambia con A[0];

1 - Si cerca la componente di valore più piccolo, A[i_min], in A[1…n-1], e si scambia con A[1];

…

…

i - All'iterazione i-ma, si cerca la componente di valore più piccolo, A[i_min], in A[i…n-1], e si scambia con A[i];

Si ripete fino all'iterazione (n-2)-ma. Il vettore è così ordinato.

```
Void SelectionSort(TipoVettore A, int n)
{
    int i,j,i_min;
    TipoElemVettore temp;
    for(i=0,i<n-1;i++) {
    /*ricerca del min in A[i…n-1]*/
        i_min=i;
        for(j=i+1;j<n;j++) {
            if(A(j)<A[i_min])
            i_min=j;
                if(i!=i_min) {
                temp=A[i_min];
                A[i_min]=A[i];
                A[i]=temp;
                        }
                    }
            }
}
```


Esempio

13	8	2	5	9	10
2	8	13	5	9	10
2	5	13	8	9	10
2	5	8	13	9	10
2	5	8	9	13	10
2	5	8	9	10	13

Ordinamento a bolle (bubblesort)

```
#include <stdio.h>
#define SIZE 10
main()
{
      int a[SIZE]={2,6,4,8,10,12,89,68,45,37};
      int i, pass, hold;
      printf("Data items in original order\n");
      for(i=0;i<=SIZE-1;i++)
          printf("%4d",a[i]);
      for(pass=1;pass <=SIZE-1;pass++)  /*passaggi*/
            for(i=0; <=SIZE-2;i++)
            if(a[i]>a[i+1]) {      /*scambio*/
                  hold = a[i];
            a[i] = a[i+1];
            a[i+1] = hold;
            }
      printf("Data items in ascending order\n");
      for(i=0;i<=SIZE-1;i++)
          printf("%4d",a[i]);
}
```

Trae vantaggio da un eventuale ordinamento parziale del vettore.

La tecnica prevede l'esecuzione di diversi passaggi, in ognuno dei quali viene confrontata una coppia di elementi adiacenti.

Tali valori vengono lasciati al loro posto se già in ordine crescente, altrimenti scambiati.

Esempio

1° passo	13	8	2	5	9	10
	8	13	2	5	9	10
	8	2	13	5	9	10
	8	2	5	13	9	10
	8	2	5	9	13	10
	8	2	5	9	10	13
2° passo	8	2	5	9	10	13
	2	8	5	9	10	13
	2	5	8	9	10	13
3° passo	2	5	8	9	10	13

Ordinamento a bolle – chiamata per indirizzo

```
#include <stdio.h>
#define SIZE 10
void bubblesort(int*, int);
main()
{
      int i,a[SIZE]={2,6,4,8,10,12,89,68,45,37};
      printf("Data items in original order\n");
      for(i=0;i<=SIZE-1;i++)
          printf("%4d",a[i]);
      bubblesort(a,SIZE);
      printf("Data items in ascending order\n");
      for(i=0;i<=SIZE-1;i++)
          printf("%4d",a[i]);
}
```

```
void bubblesort(int *array, int size)
{
int pass,j;
void swap(int *, int *);  /*funz.locale a
                            bubblesort*/
      for(pass=1;pass <=size-1;pass++)
      /*passaggi*/
            for(j=0; <=size-2;i++)
            if(array[j]>array[j+1]) {
            /*scambio*/
            swap(&array[j], &array[j+1]);
            }

            void swap(int *element1Ptr,
            int *element2Ptr)
            {
            int temp;
            temp=*element1Ptr;
            *element1Ptr=*element2Ptr;
            *element2ptr=temp;
            }
```


ordinamento a bolle ottimizzato

```
/* si introduce una variabile booleana per controllare se sono avvenuti scambi o meno*/

void BubbleSortOttimizzato(Tipovettore A, int n)
{
int i=0;
int j;
TipoElemVettore temp;
bool ordinato;
do {
        ordinato=TRUE;
        for(j=n-1;j>i;j--)
                if(A[j]<A[j-1]) {
                        temp=A[j];
                        A[j]=A[j-1];
                        A[j-1]=temp;
                        ordinato=FALSE;
                }
        i++;
    }while (!ordinato&&i<n-1);

}
```


ordinamento per inserimento

/*Ogni elemento del vettore viene posizionato nel punto che gli compete confrontandolo con i precedenti. Si scorre il vettore, quando un elemento è fuori posto, si sposta nella sua giusta posizione partendo dall'inizio del vettore*/

```
void InsertionSort(TipoVettore A, int n)
{
    int el1, el2;       /*el1=indice del prossimo elemento da sistemare*/
                        /*el2=indice dell'elemento da controllare*/
    TipoElemVettore val;        /*valore dell'elemento da sistemare*/
    for (el1=0;el1<n-1;el1++) {
        val=A[el1+1];           /*controlla e scala gli elementi partendo dall'ultimo sistemato*/
        el2=el1;
        while(el2>=0&&A[el2]>val){
            A[el2+1]=A[el2];
            el2--;
            }
        A[el2+1]=val;           /*sistema il valore da controllare nel posto rimasto libero*/
    }
}
```


ordinamento per fusione (merge sort)

Necessita di un vettore ausiliario.

Divide il vettore in due parti ed ordina ricorsivamente ciascuna parte.

Poi fonde i due sottovettori confrontando il primo elemento di entrambi e mettendo nel vettore ausiliario il più piccolo dei due.

L'indice di ciascun vettore è incrementato di uno tutte le volte che un elemento viene selezionato da quel vettore.

Si procede fino ad esaurimento di uno dei due vettori.

Gli elementi rimasti dell'altro vengono quindi copiati nel vettore ausiliario.

/* Fusione di due sequenze ordinate */

```
#include <stdio.h>
#define MAX_ELE 1000

main()
{
char vet1[MAX_ELE];      /* prima sequenza */
char vet2[MAX_ELE];      /* seconda sequenza */
char vet3[MAX_ELE*2];    /* merge */
int n;                   /* lunghezza prima sequenza */
int m;                   /* lunghezza seconda sequenza */

char aux;          /* variabile di appoggio per lo scambio */

int i, j, k, p, n1, m1;

do {
  printf("Lunghezza prima sequenza: ");
  scanf("%d", &n);
}
while(n<1 || n>MAX_ELE);

/* Caricamento prima sequenza */
for(i = 0;i <= n-1; i++) {
  printf("vet1 elemento n. %d: ",i+1);
  scanf("%1s", &vet1[i]);
}

do {
  printf("Lunghezza seconda sequenza: ");
  scanf("%d", &m);
}
while(m<1 || m>MAX_ELE);


/* Caricamento seconda sequenza */
for(i=0; i<=m-1; i++) {
  printf("vet2 elemento n. %d: ",i+1);
  scanf("%1s", &vet2[i]);
}

/* Ordinamento prima sequenza */
p = n; n1 = n;
do {
  k = 0;
  for(i = 0; i < n1-1; i++) {
    if(vet1[i]> vet1[i+1]) {
      aux = vet1[i]; vet1[i] = vet1[i+1]; vet1[i+1] = aux;
      k = 1; p = i+1;
    }
  }
n1 = p;
}
while(k==1);
```


```
/* Ordinamento seconda sequenza */
p = m; m1 = m;
do {
  k = 0;
  for(i=0; i<m1 - 1; i++) {
    if(vet2[i]>vet2[i+1]) {
      aux = vet2[i];
        vet2[i] = vet2[i+1];
        vet2[i+1] = aux;
      k = 1;
     p = i+1;
    }
  }
m1 = p;
}
while(k==1);
```

```
/* Fusione delle due sequenze
(merge) */
i = 0; j = 0; k = 0;
do {
  if(vet1[i]<=vet2[j])
    vet3[k++] = vet1[i++];
  else
    vet3[k++] = vet2[j++];
}
while(i<n && j<m);

if(i<n)
  for(; i<n; vet3[k++] = vet1[i++])
    ;
else
  for(; j<m; vet3[k++] = vet2[j++])
    ;
```


Ordinamento veloce (quicksort)

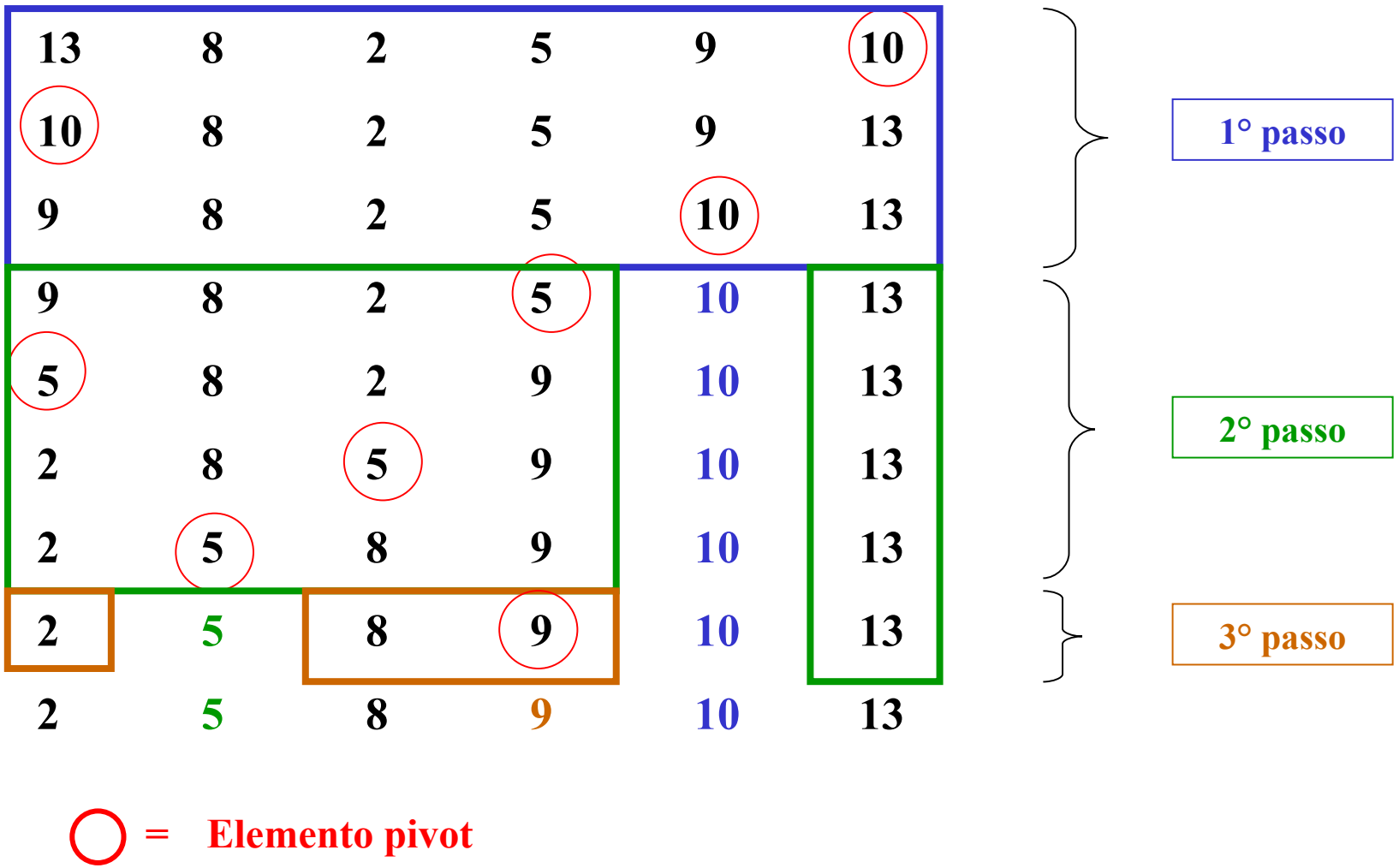
Si determina un elemento “perno” (pivot) e si pongono gli elementi minori del perno alla sua sinistra e quelli maggiori alla sua destra.

L’algoritmo si applica ricorsivamente ai due sottovettori, fino ad ottenere array con un solo elemento. A questo punto il vettore è ordinato. Su ogni sottovettore si procede come segue:

- Se x è l’elemento pivot, si esaminano le componenti dell’array in ordine di indice decrescente a partire dall’ultima, finché non si trova un elemento minore di x; quindi si esaminano le componenti dell’array a partire dalla prima, finché non si incontra un elemento maggiore di x, oppure tutto l’array è stato esaminato.

- Si scambiano i due elementi e si ripete il procedimento finché i due indici non si incontrano

Esempio



◯ = Elemento pivot

/* Ordinamento quicksort di un array di int */

```
#include <stdio.h>
#define N 10 /* numero elementi dell'array */
int v[N];    /* array contenente gli interi immessi */

void quick(int, int);
void scambia(int *, int *);

main()
{
int i;
for(i=0; i<N; i++) {  /*immissione dati*/
  printf("\nImmettere un intero n.%d: ",i);
  scanf("%d", &v[i]);
}

quick(0,N-1);     /* chiamata della procedura quick */

for(i=0; i<N; i++)       /* sequenza ordinata */
  printf("\n%d", v[i]);
putchar('\n');
}

/* Procedura ricorsiva "quick" */
void quick(int sin, int des)
{
  int i, j, media;
  media= (v[sin]+v[des]) / 2;            /*pivot*/
  i = sin;
  j = des;

do {
   while(v[i]<media) i = i+1;
   while(media<v[j]) j = j-1;
   if(i<=j) {
    scambia(&v[i], &v[j]);
    i = i+1;
    j = j-1;
   }
  }
  while (j>=i);

if(sin<j) quick(sin, j);  /* invocazione ricorsiva */
if(i<des) quick(i, des);  /* invocazione ricorsiva */
}

void scambia(int *a, int *b)
{
  int temp;

  temp = *a;
  *a = *b;
  *b = temp;
}
```


Ricerca di un elemento in un array

- Si effettua la ricerca su un *valore chiave*
- Ricerca lineare
  - E' la più semplice
  - Si confrontano tutti gli elementi con il valore chiave
  - Va bene per array di piccole dimensioni e non ordinati
  - **Parte inutile della ricerca**
    Quando ho trovato un elemento, il metodo continua ugualmente a verificare gli altri.
    È un problema se:
    1-ho vettori molto grandi
    2-devo fare spesso questa ricerca
    Soluzione alternativa: quando trovo l'elemento, mi fermo.

/* Ricerca sequenziale di un valore nel vettore */

```
#include <stdio.h>
#define MAX_ELE 1000   /* massimo
numero di elementi */

main()
{
char vet[MAX_ELE];
int i, n;
char c;

/* Immissione lunghezza della sequenza */
do {
  printf("\nNumero elementi: ");
  scanf("%d", &n);
}
while(n<1 || n>MAX_ELE);

/* Immissione elementi della sequenza */
for(i=0; i<n; i++) {
  printf("\nImmettere carattere n.%d: ",i);
  scanf("%1s", &vet[i]);
}
```

```
printf("Elemento da ricercare: ");
scanf("%1s", &c);

/* Ricerca sequenziale */
i = 0;
while(c!=vet[i] && i<n-1)  ++i;
if(c==vet[i])
  printf("\nElemento %c presente in
      posizione %d\n",c,i);
else
  printf("\nElemento non presente!\n");
}
```


Ricerca in vettore ordinato

Posso usare gli stessi metodi dei vettori non ordinati, oppure posso sfruttare l'ordinamento.

Vantaggio dell'ordinamento

Se cerco 4, e trovo un elemento maggiore, so che è inutile andare avanti:

[-2 -1 3 5 ...] Quando arrivo al 5, so che il 4 non lo trovo dopo, perché altrimenti il vettore non sarebbe ordinato.

Ricerca in un vettore ordinato: ricerca binaria

Dato un vettore v e un intero x:

- se x coincide con l'elemento medio di v, vmed, fine della ricerca
- se x è maggiore di vmed, prosegui da vmed in poi
- se è minore, prosegui prima di vmed

Alla fine, si arriva ad un vettore monodimensionale. Se non coincide con x, vuol dire che x non fa parte di v.

Esempio

Ricerca binaria dell'elemento 9 nel vettore:

2 5 8 9 10 13

9=8? No, 9>8. Proseguo nella seconda metà del vettore:

9 10 13

9=10? No, 9<10, proseguo nella prima metà del sottovettore:

9

Il sottovettore è uno scalare, ed è pari a 9.

9=9? Sì, elemento trovato

= elemento mediano del (sotto)vettore

```
/*ordinamento e ricerca binaria*/
#include <stdio.h>
main()
{
char vet[7];      /* array contenente i
caratteri immessi. Il programma          va
generalizzato per n generico */
int i,n,k,p;
char aux;        /* variabile di appoggio per
lo                    scambio */
char ele;             /* elemento da ricercare */
int basso, alto, pos;  /* var. usate per la
                      ricerca binaria */

/* Immissione caratteri */
n = 7;
for(i=0;i<=n-1; i++) {
  printf("vet %dº elemento: ", i+1);
  scanf("%1s", &vet[i]);
}
/* ordinamento ottimizzato */
p = n;
do {
  k = 0;
/*k vale 0 se non ci sono stati scambi,
altrimenti vale 1*/
  for(i=0; i<n-1; i++) {
    if(vet[i]>vet[i+1]) {
      aux = vet[i];
vet[i] = vet[i+1];
vet[i+1] = aux;
      k = 1; p = i+1;
    }
  }
/*il n. di confronti si interrompe dove
al passo precedente si è avuto l'ultimo
scambio*/
n = p;
}
while(k==1);
```


```
printf("\nElemento da ricercare: ");
scanf("%1s", &ele);

/* ricerca binaria */
n = 7;
alto = 0; basso = n-1; pos = -1;
do {
  i = (alto+basso)/2;
  if(vet[i]==ele)  pos = i;
  else
    if(vet[i]<ele)
      alto = i+1;
    else
      basso = i-1;
}
while(alto<=basso && pos==-1);

if(pos != -1)
  printf("\nElemento %c presente in posizione %d\n",ele,pos);
else
  printf("\nElemento non presente! %d\n", pos);
}
```


ricerca binaria ricorsiva in un vettore

```
/*ricerca elem nella parte di A compresa fra inf e sup*/
bool RicercaBinariaRic(int inf, int sup, TipoVettore A, TipoElemVettore elem, int*posiz)
{
      int med;
      bool trovato;
      if(inf>sup)
          trovato=FALSE;       /*la parte di vettore fra inf e sup è vuota*/
      else {
            med=(inf+sup)/2;
            if(elem==A[med]) {
                  *posiz=med;
                  trovato=TRUE;
            }
      else
          if(elem<A[med])
            trovato=RicercaBinariaRic(inf,med-1,A,elem,  posiz);       /*cerca nella parte
                                                                       inferiore*/
          else
            trovato=RicercaBinariaRic(med+1,sup,A,elem,  posiz);       /*cerca nella parte
                                                                       superiore*/
      }
      return trovato;
}
```


Ricerca in alberi binari

Un albero binario di ricerca è un albero binario in cui in ciascun nodo è memorizzato un elemento di un insieme in modo che:

-tutti gli elementi associati a nodi del sottoalbero sinistro di un qualunque nodo i sono più piccoli dell'elemento associato al nodo i,

- tutti gli elementi associati a nodi del sottoalbero destro di un qualunque nodo i sono più grandi dell'elemento associato al nodo i.

/* Ricerca ottimizzata - E' analoga alla ricerca binaria Si applica ad alberi binari di ricerca*/

```
void ricBin(struct nodo *p, int val, struct nodo **pEle)
{
if(p!=NULL)
  if(val == p->inf) {
    printf("  trovato ");
    *pEle = p;
  }
else
  if(val < p->inf) {
    printf("  sinistra");
    ricBin(p->albSin, val, pEle);
  }
  else {
    printf(" destra");
    ricBin(p->albDes, val, pEle);
  }
}
```


Efficienza degli algoritmi

Spazio di memoria/tempo che richiedono.

Devo tenere conto del fatto che:

- ho più algoritmi per lo stesso problema
- l'efficienza dipende dai dati

Per la ricerca in vettore ordinato ho la ricerca sequenziale e la ricerca binaria.

L'efficienza dipende dal vettore e dall'elemento da cercare.

Caso in cui la ricerca sequenziale è più veloce:

v[]={1, 2, 3, 4, 5, 6, 7, 8, 9};

x=1;

Caso in cui la ricerca binaria è più veloce:

v[]={0, 3, 4, 10, 12, 143, 159, 200};

x=10;

Valutazioni

Devo valutare quale algoritmo è migliore, ma questo dipende dai dati su cui gli algoritmi lavorano. Si danno valutazioni complessive:

caso migliore - caso peggiore - caso medio

In questo modo, posso dire quale algoritmo è il migliore complessivamente (senza specificare i dati di input).

- caso migliore
  - il minimo tempo che ci mette il programma (dati su cui ci mette di meno) a girare
- caso peggiore
  - il tempo che ci mette sui dati peggiori
- caso medio
  - devo specificare una distribuzione di probabilità sui dati

Dimensione dei dati

- Se il vettore ha pochi elementi, tutti gli algoritmi vanno bene.
- L'efficienza è importante quando ci sono grandi quantità di dati (vettori grandi).

Modello di costo

Suppongo che ogni istruzione richieda tempo=1.

Tempo di esecuzione = numero di istruzioni eseguite.

Valutazione in base al numero dei dati:

n = dimensione dei dati (grandezza del vettore)

T(n) = tempo impiegato dal metodo su un vettore di grandezza n.

Metodo dell'istruzione dominante: vado a vedere quante volte si esegue l'istruzione dentro i cicli maggiormente nidificati.

Ad es.:

Ricerca sequenziale fino alla fine: tempo T(n)=n in ogni caso.

Ricerca sequenziale in cui mi fermo quando trovo: T(n)=n nel caso peggiore,

T(n)=1 nel caso migliore.

Ricerca binaria: $T(n)=\log_2(n)$ nel caso peggiore, T(n)=1 nel caso migliore.

Costo della ricerca binaria

Il costo è $\log_2(n)$, se n è la dimensione del vettore.

Dimostrazione ``al contrario'': se ci vogliono x operazioni, quanto è grande il vettore?

una operazione: vettore grande 1

due operazioni: vettore grande 2

tre operazioni: vettore grande 4

La ricerca richiede una operazione in più se il vettore è grande il doppio.

Se la dimensione del vettore è esponenziale nel numero di operazioni, allora il numero di operazioni è logaritmico nella dimensione.

Esempio: se ho 8 elementi alla prima chiamata riduco a 4, poi a 2 poi a 1, quindi servono 3 chiamate. Se ho 16 elementi faccio 8, 4, 2, 1, ecc.

Notazione O

Ipotesi:

- ignoro le costanti moltiplicative, per cui $2n \sim n$
- ignoro i termini di ordine inferiore, per cui $n^2+3n+2 \sim n^2$

Uso la notazione O':

$n2+3n+2=O(n^2)$

Si dice che la complessità è $O(n^2)$

Complessità del SelectionSort

Quante operazioni vengono eseguite?

Considero un vettore di n elementi, senza specificare quali valori contiene.

Alla prima chiamata ricorsiva, faccio n iterazioni.

Alla seconda, faccio n-1 iterazioni, ecc.

Totale: $n+(n-1)+(n-2)+...+2+1 = n(n+1)/2$

Ignoro le costanti e i termini inferiori: ottengo n^2, quindi dico che la complessità è $O(n^2)$

Caso migliore o peggiore

Vengono eseguite $O(n^2)$ operazioni indipendentemente dai valori scritti nel vettore.

La complessità del caso migliore e peggiore coincidono: sono tutte e due $O(n^2)$

Complessità del BubbleSort

Vettore grande n, di cui non specifico i valori.

Il ciclo esterno ha n iterazioni.

Il ciclo interno ha n iterazioni la prima volta, poi n-1, ecc: di media, ho n/2 iterazioni.

Totale: $O(n^2)$

Questo vale sia nel caso migliore che nel caso peggiore.

Vantaggio del BubbleSort ottimizzato

Se faccio tutto il ciclo interno senza mai fare scambi, vuol dire che il vettore è ordinato.

Caso peggiore: devo fare tutto come prima, quindi ho $O(n^2)$

Caso migliore: il vettore è già ordinato.

In questo caso, faccio un'intera catena di confronti (eseguo una volta tutto il ciclo più interno), mi accorgo che trovato == true, e termino.

Costo di caso migliore: $O(n)$

Il metodo ottimizzato ha la stessa complessità nel caso peggiore, ma minore nel caso migliore.

Metodi di ordinamento - Complessità ottima

Nessun metodo di ordinamento può avere complessità minore di O(n).

Infatti, devo almeno verificare se il vettore è già ordinato.

Quindi, il BubbleSort ha complessità ottima nel caso migliore.

Esistono algoritmi di ordinamento che impiegano $O(n \log_2 n)$ nel caso peggiore.

Quindi: il BubbleSort è ottimo nel caso migliore, ma non nel caso medio.

Complessità del mergesort

Basato sul fatto che si possono fondere (merge) due array ordinati in un unico array ordinato in tempo lineare.

Ad ogni passo, l'elemento che viene messo nel vettore nuovo è il più piccolo fra quelli che mancano.

Costo di esecuzione della fusione: O(n) dove n è la dimensione complessiva dei due vettori.

Algoritmo complessivo

- se il vettore ha zero elementi, ritorna un vettore vuoto
- se il vettore ha un elemento, copialo in un nuovo vettore grande uno
- spezza il vettore in due parti
- ordina le due parti con due chiamate ricorsive
- restituisci il vettore ottenuto per fusione

A ogni passo, ho un costo lineare (escludendo il costo delle chiamate ricorsive).

In ogni chiamata ricorsiva, il vettore viene ancora spezzato e vengono fatte due chiamate.

Costo totale

Ogni sottovettore corrisponde a una chiamata ricorsiva.

Ogni chiamata ricorsiva ha costo pari alla dimensione del vettore passato.

Quindi, il totale di tutte le chiamate ricorsive che corrispondono a una certa riga ha costo n (dimensione del vettore originario).

Costo totale

Ogni riga ha costo n.

Ci sono $\log_2(n)$ righe.

Costo totale del mergesort: **$O(n \log_2(n))$**

Si può dimostrare che non esistono algoritmi più efficienti