[1 2 5 4 0]

Ricerca sequenziale

Se **disgiunti** restituisce la posizione o -1 se non c'è, se **non disgiunti** restituisce il numero di occorrenze

NON ordinati DISGIUNTI

```
int NonOrdDisg(int v[],int n, int x){
      int tro,k;

      tro = -1;
      k=0;
      while ((k < n) && (tro == -1)){
           if(x == v[k])
             tro = k;
           else
             k++;
      }
      return tro;
}
```

COMPLESSITÀ per tutti i tipi di ricerca
CONFRONTI
Peggiore n
Migliore 1
Medio n/2

```
/**versione con variabile booleana che restituisce solo la presenza o l'assenza
*  dell'elemento cercato
*/
bool NonOrdDisg (int v[], int n, int x) {
     int k=0;
     bool tro = false;

     while (k<n && !tro){      //tro == false
        if (x == v[k])
                tro = true;
        else
          k++;
     }
     return tro; //restituisce true se l'ha trovato, false altrimenti
}
```

[2 4 6 7]

Occorrenze

ORDINATI DISGIUNTI

```
int OrdDisg1(int v[],int n, int x) {
      //vettore ordinato a elementi disgiunti
      int tro,k;
      bool continua;

      tro = -1;
      k = 0  ;
      continua = true;

      while((k < n) && (tro == -1) && (continua == true)){
            if(x == v[ k ])
              tro = k;
            else
             if(v[ k ] > x )
                  continua = false ;      //forza l'uscita dal ciclo
             else
                  k++ ;
      }
      return tro ;
}
```

```
int OrdDisg2(int v[],int n, int x) {
      //vettore ordinato a elementi disgiunti
      int tro,k;

      tro = -1;
      k = 0  ;
      while((k < n) && (tro == -1)){
            if(x == v[ k ])
              tro = k;
            else
             if(v[ k ] > x )
                  k = n ;     //forza l'uscita dal ciclo
             else
                  k++ ;
      }
      return tro ;
}
```

CONFRONTI

Peggiore	n
Migliore	1

0 5 7 3 5

NON ordinati NON disgiunti

```
int NonOrdNonDisg(int v[],int n, int x) {
//vettore non ordinato a elementi non disgiunti restituisce le occorrenze
      int nx,k;

      nx=0;
      for (k=0; k<n; k++){
          if(x == v[k]){
            printf("\nL'elemento trovato in posizione %d", k);
            nx++;
          }
       }
      return nx; //numero di occorrenze
}
```

CONFRONTI n (c'è il for)

2 3 3 3 4 5 5

ORDINATI NON disgiunti

```
int OrdNonDisg1(int v[], int n, int x) {
//vettore ordinato a elementi non disgiunti
      int nx,k;
      bool continua;

      nx=0;
      k=0;
      continua = true;

      while(k<n && continua) {
           if(x == v[k]) {
            printf("\nL'elemento trovato in posizione %d", k);
            nx++;
            k++;
          }else
             if(v[k] > x)
                  continua = false; //fa uscire dal ciclo
             else
                  k++;
       }
      return nx; //numero di occorrenze
}


int OrdNonDisg2(int v[], int n, int x) {
```

```
//vettore ordinato a elementi non disgiunti
      int nx,k;

      nx=0;
      k=0;

      while(k<n) {
           if(x == v[k]) {
            printf("\nL'elemento trovato in posizione %d", k);
            nx++;
            k++;
          }else
             if(v[k] > x)
                  k = n; //fa uscire dal ciclo
             else
                  k++;
       }
      return nx; //numero di occorrenze
}
```

CONFRONTI

Peggiore	n
Migliore	1(non c'è e il primo è maggiore)

Ricerca binaria o dicotomica

La ricerca dicotomica realizza una ricerca su un albero binario ordinato e bilanciato. Alcune definizioni:
- **Livello** o profondità di un nodo = 1 + il livello del nodo padre, la radice ha livello 0;
- **Altezza** o profondità dell'albero = livello massimo fra i livelli di tutti i suoi nodi.
- Un albero è **bilanciato** se ha tutte le foglie al medesimo livello, ovvero se ogni foglia dell'albero ha la medesima distanza dalla radice.

L'altezza di un albero binario bilanciato è dato dal log2 N, con N il numero di nodi: se ho 7 elementi l'albero ha profondità 2, con 8 fino a 15 è 3, così via.

La ricerca binaria non usa mai più di log2 N +1 (logaritmo base 2 di N approssimato per difetto +1 ovvero profondità dell'albero + 1) confronti.

NON RICORSIVA

```
int ricercaBinariaNonRicorsiva(int v[], int n, int x) {
    int p,u,m,tro;     // indice del… p= primo, u=ultimo, m=medio
    p = 0;
    u = n-1;
    tro=-1;
    while((p <= u)&& tro==-1) {
        m = (p+u)/2;
        if(v[m]==x)
            tro=m;     // valore x trovato alla posizione m
        else if(v[m] < x)
               p = m+1;
            else       // x < v[m]
               u = m-1;
    }
    return tro;
}
```

Progressione aritmetica

Se il primo termine di una progressione aritmetica è *a* e la ragione è *d*, allora l'n-esimo termine della successione è dato da:

$$a_n = a + (n-1)d$$

Tale proprietà può essere estesa a un qualsiasi termine della progressione; si avrà quindi che:

$$a_r = a_s + (r-s)d$$

La somma dei numeri di una progressione aritmetica finita si chiama **serie aritmetica**. La somma *S* dei primi *n* valori di una progressione aritmetica è uguale a:

$$S_n = \frac{1}{2}n(a_1 + a_n)$$

dove a_1 è il primo termine e a_n l'n-esimo.

$$\sum_{k=1}^{n} k \qquad 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Per esempio per trovare la somma dei primi *n-1* interi positivi si calcola:

Se devo fare 1+2+…+n-1=1/2*(n-1)*(1+n-1)= n* (n-1) /2 = $(n^2 - n)/2$

http://www.ripmat.it/mate/q/qb/qbae.html (per la dimostrazione della formula)

ORDINAMENTO

RAFFINAMENTO Selection sort o per selezione

https://www.youtube.com/watch?v=tZgOfZ5sy64

```
void selectionSort(int v[],int n) {
/*Si cerca il minimo ad ogni ciclo e si fa al massimo un solo scambio a ciclo.
Quando sistemo il penultimo elemento automaticamente è sistemato l'ultimo*/
	int k,kmin,j;
	for(k = 0;k < n-1; k++) {
	  kmin = k;
	  for(j = k+1; j < n; j++){
		if(v[kmin] > v[j]) // confronti
			kmin = j;
	   }
	  if(kmin != k)
		scambio(&v[k], &v[kmin]); //scambi
	 }
	return;
}
```

COMPLESSITÀ
CONFRONTI
(n-1)+(n-2)+…+2+1=n(n-1)/2=$O(n^2)$
SCAMBI
Peggiore n-1=O(n)
Migliore 0

l'ordinamento per selezione ha lo svantaggio di non accorgersi se il vettore è ordinato, ma ha un'importante applicazione: poiché ciascun elemento viene spostato al più una volta, questo tipo di ordinamento è il metodo da preferire quando si devono ordinare file costituiti da record estremamente grandi e da chiavi molto piccole. Per queste applicazioni il costo dello spostamento dei dati è prevalente sul costo dei confronti e nessun algoritmo è in grado di ordinare un file con spostamenti di dati sostanzialmente inferiori a quelli dell'ordinamento per selezione.

Bubble sort o per scambio o per affioramento

https://www.youtube.com/watch?v=yIQuKSwPlro

```
void bubbleSort1(int vett[], int n){
  int k, sup;

  for (sup = n-1; sup > 0; sup--){
    for (k = 0; k < sup  ; k++){
      if (vett[k] > vett[k+1])
        scambio( &vett[k],&vett[k+1]);
    }
  }
}

in alternativa i cicli for possono essere scritti così
for (sup = n; sup > 1; sup--){
    for (k = 0; k < sup-1; k++){
```

COMPLESSITÀ

CONFRONTI

$n(n-1)/2=O(n^2)$

SCAMBI

Peggiore $n(n-1)/2=O(n^2)$

Migliore 0

```
void bubbleSort2(int vett[], int n) {
//se non si fanno scambi in un giro il vettore è ordinato
  int k,sup;
  bool sca;

  sup=n-1;

  sca=true;
  while ((sup>0) && sca==true) {
    sca=false;
    for (k = 0; k < sup; k++) {
      if (vett[k] > vett[k+1]){
        scambio( &vett[k],&vett[k+1]);
        sca=true;
      }
    }
    sup--;
  }
}

void bubbleSort3(int vett[], int n) {
  int k,sup,sca;

  sup= n-1 ;
  while ( sup>0 )  {
    sca=0 ;
    for (k = 0; k < sup ; k++) {
      if (vett[ k ] > vett[ k+1 ]){
        scambio(&vett[ k  ],&vett[ k+1 ]);
        sca=k ;
      }
    }
    sup=sca ;
  }
}
```

COMPLESSITÀ per tutti i tipi di bubble sort RAFFINATI

CONFRONTI

Peggiore $n(n-1)/2=O(n^2)$

Migliore $n-1=O(n)$

SCAMBI

Peggiore $n(n-1)/2=O(n^2)$

Migliore 0

Quicksort o ordinamento rapido

È l'algoritmo di ordinamento che ha, in generale, prestazioni migliori tra quelli basati su confronto. È stato ideato da Tony Hoare nel 1961.

Ad ogni stadio si effettua un ordinamento parziale di una sequenza di oggetti da ordinare. Assunto un elemento come perno (**pivot**) dello stadio, si confrontano con esso gli altri elementi e si posizionano alla sua sinistra i minori e a destra i maggiori, senza tener conto del loro ordine. Dopo questo stadio si ha che il perno è nella sua posizione definitiva. Successivamente si organizzano nuovi stadi simili nei quali si procede all'ordinamento parziale delle sottosequenze di elementi rimasti non ordinati, fino al loro esaurimento.

È basato sulla metodologia Divide et Impera:
Dividi: L'array A[p…u] viene "partizionato" (tramite spostamenti di elementi) in due sottoarray non vuoti A[p…q] e A[q+1…u] in cui: a ogni elemento di A[p…q] è minore o uguale ad ogni elemento di A[q+1…u]
Conquista: i due sottoarray A[p…q] e A[q+1…u] vengono ordinati ricorsivamente con QuickSort
Combina: i sottoarray vengono ordinati anche reciprocamente, quindi non è necessaria alcuna combinazione. A[p…u] è già ordinato.

COMPLESSITÀ

CONFRONTI

-

Peggiore	$n(n-1)/2=O(n^2)$
Migliore/Medio	$n \log n=O(n \log n)$

SCAMBI come confronti

Peggiore
Migliore

```
void QSort(int v[], int inf, int sup){
    int pivot = v[(inf + sup)/2];
    int i = inf;
    int j = sup;

    while (i <= j){
        while (v[i] < pivot) i++;  //si ferma sicuramente sul pivot
        while (v[j] > pivot) j--;  //si ferma sicuramente sul pivot
        // v[i] è >= pivot mentre v[j] <= pivot
        // se c'è un elemento a sx > pivot e uno a dx < si scambiano
        if (i < j)  //non sono nel posto giusto
            scambio (&v[i], &v[j]);
        if (i <= j){ // se < si prosegue, se == sono sul pivot e termina il ciclo
            i++;
            j--;
        }
            }
    if (inf < j) QSort (v, inf, j);
    if (i < sup) QSort (v, i, sup);
}
```