



Università degli Studi di Cagliari  
Corso di Laurea in Ingegneria Biomedica

# **ELEMENTI DI INFORMATICA**

[https://www.unica.it/unica/page/it/gianluca\\_marcialis](https://www.unica.it/unica/page/it/gianluca_marcialis)

A.A. 2021/2022

Docente: **Gian Luca Marcialis**

**LINGUAGGIO C**  
**Problemi notevoli**

# Sommario

- Introduzione
- La ricerca
  - Sequenziale
  - Binaria
- L'ordinamento
  - Selection Sort
- Misurare la «complessità»
- Appendice (non obbligatoria): Quick Sort

# Ricapitolando un po'...

- Fin qui abbiamo risolto alcuni problemi imparando delle tecniche base dell'algoritmica applicata alla programmazione
  - Assegnamento iterativo
    - Variabili cumulative
  - Estrazione di informazioni a partire da insiemi indicizzati di elementi (vettori)
- Questi problemi possono essere derivati da esse o concorrere ad esse
- Ciò porta alla indicazione di questi problemi come «idiomatici» nell'algoritmica

# Problemi visti

- Calcolo della media aritmetica
- Elevamento a potenza



**Tecniche algoritmiche iterative  
su variabili cumulatrici**

- Estrazione del massimo elemento
- Conteggio delle occorrenze



**Tecniche di RICERCA  
con cumulatori**

# La ricerca sequenziale

- Problema: cercare un elemento in un insieme
- Formulazione in C: scrivere una funzione che, ricevendo in ingresso un vettore di  $N$  interi ed un valore  $x$ , restituisca la posizione di  $x$  nel vettore (tra 0 e  $N-1$ ), se presente,  $N$  altrimenti
- Dal momento che alcuna informazione è data sul vettore, l'unico modo è fare una **ricerca sequenziale**: scandire il vettore dal primo all'ultimo elemento

# Soluzione

```
int ricerca_sequenziale(int *v, int x, int N)
{
    int i=0;

    while ( (i<N) && (v[i] !=x) ) i++;
    return i;
}
```

# La questione della «complessità»

- L'esecuzione di un'istruzione e l'impiego di certi tipi di dati, hanno un costo
  - Impiego del processore (tempo), memoria (occupazione di byte per i dati)
- Il costo dipende da quante iterazioni (for/while) e da quante biforcazioni (if) verranno fatte prima di trovare l'elemento x nel vettore, oppure nel non trovarlo
- «Complessità computazionale»
  - L'ammontare di massima nel caso medio o nel caso pessimo, in funzione della configurazione del vettore di ingresso, delle risorse impegnate dal sistema → dimensione del problema
  - Espresso come funzione dipendente da tale «dimensione»
    - Esempio: il numero degli elementi del vettore

# Complessità della ricerca sequenziale

- Caso medio: il valore si troverà in un punto qualunque del vettore – mediamente nella posizione  $N/2$
- Caso pessimo: il valore non si troverà nel vettore,  $N$  iterazioni
- In entrambi i casi, la complessità è legata al numero di confronti tra  $v[i]$  e  $x$  – questo numero è lineare con  $N$
- La complessità computazionale può dirsi «lineare» con  $N$ , dimensione del vettore:

$$\text{Complessità} = a * N + b$$

- Con  $a$  e  $b$  costanti di «slope» e di «bias» dipendenti dalla soluzione implementata.
- In generale, si indica questa complessità con  $O(N)$ , un infinito dipendente linearmente al crescere di  $N$



# La ricerca binaria

- Se il vettore è ordinato, ad esempio per valori crescenti
- Questa informazione ci permette di trovare una soluzione più efficiente alla semplice ricerca sequenziale, applicando la tecnica nota come “**divide et impera**”
- Il problema generale non lo si sa «gestire»
  - Allora lo si semplifica al punto in cui è possibile risolverlo
  - Si divide il problema generale in N problemi semplificati, per ognuno dei quali è nota la soluzione
  - La soluzione del problema generale si ottiene per «composizione» delle soluzioni di ciascuno degli N problemi semplificati risolti

# Algoritmo di ricerca binaria

- Si consideri il problema della ricerca di un elemento in un vettore ordinato come se si dovesse cercare il nome di un abbonato in un elenco telefonico
- Un algoritmo di ricerca sequenziale inizierebbe comunque dalla prima pagina
- Tuttavia, potremmo:
  1. Dividere l'elenco in due parti eguali
  2. Se il nome precede alfabeticamente quello della pagina centrale, ritorno a 1 considerando solo la prima metà; altrimenti
  3. Se il nome segue alfabeticamente quello della pagina centrale, ritorno a 1 considerando solo la seconda metà; altrimenti
  4. Se la pagina è l'unica dell'elenco, non è stato trovato il nome; altrimenti
  5. E' stato trovato il nome.

# Scrittura dell'algoritmo «divide et impera»

- So risolvere il problema: il nome che cerco corrisponde a quello che leggo?
- Allora mi metto nella condizione di poter essere sempre in questa condizione
- Se un primo tentativo fallisce, divido il problema in due sottoproblemi e li affronto alla stessa maniera
- Applico lo stesso principio a ciascuno dei sottoproblemi

# Se avessi un elenco telefonico

1. Sia  $t$  il numero di pagina iniziale dell'elenco,  $c$  quello finale
2. Se  $t - c$  è negativo
  - a) Restituisco «non trovato»
    - non ci sono più pagine nell'elenco, il nome non c'è
3. Altrimenti
  - a) Il numero di pagina centrale è dato da  $m = (t+c)/2$
  - b) Se il nome cercato è quello a pagina  $m$ 
    - i. Restituisco «trovato»
  - c) Se il nome da cercare precede quello a pagina  $m$ 
    - i. Torno a 2 considerando la pagina finale pari a  $m - 1$ , ovvero  $c = m - 1$
  - Altrimenti
    - i. Torno a 2 considerando la pagina iniziale pari a  $m + 1$ , ovvero  $t = m + 1$

# Dall'algoritmo al programma

- Problema semplificato di base: se il vettore è fatto di 1 elemento, mi basta controllare quello
- Se io indico con  $t$  l'indice di inizio del vettore o testa, e con  $c$  l'indice di fine del vettore o coda...
  - ...i valori iniziali degli indici di testa e di coda del vettore corrispondono, rispettivamente, a 0 e  $N-1$
  - ...l'indice dell'elemento centrale  $m$  è dato da  $(t+c)/2$
- Se l'elemento  $x$  è più piccolo dell'elemento centrale  $v[m]$ , cambia la coda perché devo cercare fra gli  $m$  elementi più piccoli con indici da 0 a  $m-1$
- Se l'elemento  $x$  è più grande, cambia la testa perché devo cercare fra gli  $N-m-1$  elementi più grandi con indici da  $m+1$  a  $N-1$
- La condizione di termine è che il valore relativo all'indice di testa diventi maggiore o uguale a quello di coda!

# Soluzione

```
int ricerca_binaria(int* v, int x)
{
    int K, m, t, c; /*K è il numero di iterazioni fatte prima di trovare
                    o non trovare l'elemento x in v*/

    t=0;
    c=N-1;
    K=0;
    while (t<=c)
    {
        m=(t+c)/2;
        if (v[m]==x)
            return K;
        if (v[m]<x)
            c=m-1;
        else
            t=m+1;
        K++;
    }

    return K;
}
```

# Complessità dell'algoritmo di ricerca binaria

- Come la ricerca sequenziale, la complessità dipende dal numero di confronti da fare.
- Vediamo il caso pessimo:
  - Supponiamo che il vettore sia fatto di  $N=2^K-1$  componenti ( $N$  dispari).
  - Ad ogni passo, questo insieme si divide in **due** insiemi di uguali dimensioni, pari a  $2^{K-1}-1$  componenti.
  - L'algoritmo termina quando arriviamo a un insieme di  $2^1-1=1$  componenti
  - Ciò implica che sono necessari **K** confronti per arrivare a trovare (o non trovare) il valore nel caso pessimo → la complessità dell'algoritmo va come  **$\log_2 N$**
- **Una complessità logaritmica è quella «ideale» per la risoluzione di qualsiasi problema, escludendo quella costante (indipendente da  $N$ )**

# Un esempio

- Sia  $v=[1 \ 5 \ 10 \ 25 \ 30 \ 31 \ 45]$ , e  $x=45$ .
- Lanciamo la funzione con:  $z=\text{ricerca\_binaria}(v, N, x)$  ;
- Si cerca 45 in  $v$  con  $N=7=2^3 - 1$  componenti ( $N=7 \rightarrow K=3$ )
- $m=(0+6)/2=3$
- $v[m]==x \rightarrow 25==45$  è falso
- Il vettore viene diviso in  $v1=[1 \ 5 \ 10]$  e  $v2=[30 \ 31 \ 45]$ .
- $v[m]<x \rightarrow 25<45$  è vero
- Si cerca 45 in  $v2$  con  $N=3=2^2 - 1$  componenti ( $N=3 \rightarrow K=2$ )
- $m=(4+6)/2=5$
- $v[m]==x \rightarrow 31==45$  è falso
- Il vettore viene spezzato in  $v21=[30]$  e  $v22=[45]$
- $v[m]<x \rightarrow 31<45$  è vero
- Si cerca 45 in  $v22$  con  $N=1=2^1 - 1$  componenti ( $N=1 \rightarrow K=1$ )
- $m=(6+6)/2=6$
- $v[m]==x \rightarrow 45==45$  è **VERO**
- Arriviamo a trovare 45 al terzo passaggio  $\rightarrow$  **K=3**



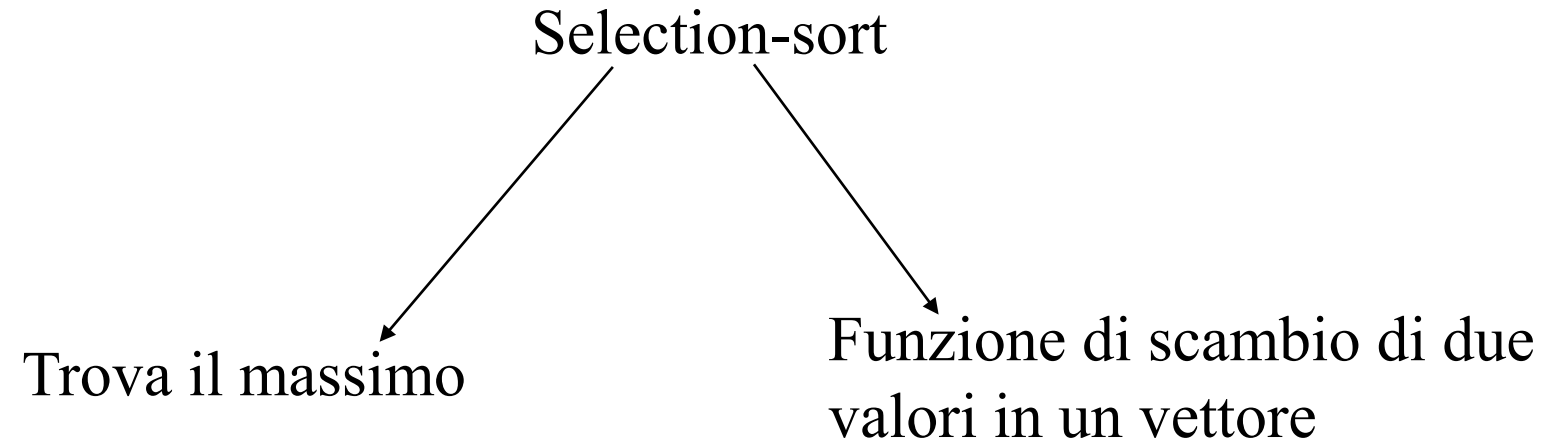
# Il problema dell'ordinamento ("sorting")

- Scrivere una funzione C che ricevendo in ingresso un vettore di N elementi interi, restituisca il vettore con gli elementi ordinati **in ordine decrescente**

# Una prima soluzione: l'ordinamento per selezione (“selection-sort”)

- L'ordinamento per selezione, o “selection-sort”, prevede semplicemente:
  1. Un confronto di tutti gli elementi del vettore tra loro per valutarne il massimo
  2. Il massimo viene spostato in cima al vettore
  3. Si ripetono le operazioni 1-2 sul vettore escludendo l'elemento in cima finché non si arriva ad un vettore di un solo elemento (quello in coda, il più piccolo)

# Vista top-down



# Selection-sort 1

- Procediamo modularmente (bottom-up), scrivendo una funzione che calcoli la posizione del massimo in un vettore di N interi tra le posizioni i e j

```
int massimo(int* v, int i, int j)
{
    int k, posmax;
    posmax=i;

    for(k=i+1; k<j; k++)
        if(v[k]>v[posmax]) posmax=k;
    return posmax;
}
```

# Selection-sort 2

- Ora possiamo passare al problema principale

```
void selection_sort(int* v, int N)
{
    int i, posmax;
    for(i=0; i<N-1; i++)
    {
        posmax=massimo(v,i,N);
        if(posmax!=i) //si può anche omettere
            scambia(v,i,posmax);
        /*ho usato la funzione implementata
           nel Cap. 2.4*/
    }
}
```

# Complessità di selection-sort

- Dipende:
  - dal numero di confronti operati durante la ricerca del massimo
  - dal numero di scambi effettuati ad ogni iterazione
- Esaminiamo il caso pessimo:
  - Alla i-esima iterazione:
    - La funzione massimo effettua  $N-1-i$  confronti
    - Viene eseguito 1 scambio
  - Quindi si ottiene:
    - $i = 0 \rightarrow$  numero confronti  $N-1$ , 1 scambio
    - $i = 1 \rightarrow$  numero confronti  $N-2$ , 1 scambio
    - ...
    - $i = N-2 \rightarrow$  numero confronti 1, 1 scambio
    - Totale  $\rightarrow N-1 + N-2 + \dots + 1$  confronti +  $N-1$  scambi
  - Associando costo unitario sia ai confronti che agli scambi (per semplicità), si ottiene:
    - Costo Totale  $\rightarrow N(N-1)/2 + N-1$
    - Per  $N$  molto grande si parla di complessità *asintotica* quadratica e si scrive:  $O(N^2)$

# Complessità di selection-sort

- Si noti che anche nel caso medio la complessità asintotica di selection-sort è quadratica
- Infatti, in media, gli scambi si ridurranno, mentre il numero di confronti sarà sempre quadratico
- Ma incidere sul numero degli scambi significa semplicemente incidere sulla componente lineare della complessità, quindi in generale la complessità di selection-sort è quadratica
- Esiste un algoritmo di ordinamento che, almeno nel caso medio, riduca la complessità → riduca il numero di confronti?

# Il selection-sort: implementazione «sintetica»

```
void selection-sort(int *v, int N)
{
    int i, j;

    for (i=0; i<N-1; i++)
        for (j=i+1; j<N; j++)
            if (v[j]>v[i])
                scambia(v, i, j);
}
```



# Struttura di un algoritmo “divide et impera”

```
tipo_soluzione divetimp (S)
{
    tipo_soluzione soluzione;
    if |S|<k /*se la dimensione del problema è accettabile*/
    {
        /*risolvi direttamente il problema*/
        return soluzione;
    }
    else
    {
        /*dividi S in sottoinsiemi S1, S2, ..., Sh*/
        soluzione=combina(divetimp(S1), ... , divetimp(Sh));
        return soluzione;
    }
}
```

# **Come misurare la complessità/l'efficienza di un programma?**

- Generalmente, l'efficienza un algoritmo dipende da:
  - Numero di iterazioni cicliche
  - Numero di condizioni if contenute nei blocchi di cicli for
- E' possibile dunque avere una stima dell'algoritmo contando il numero di confronti che viene svolto dallo stesso

# Selection-sort con il calcolo del numero approssimato di confronti

```
int selectionsort(int* v, int N)
{
    int i, j;
    int costo=0;

    for (i=0; i<N-1; i++)
        for (j=i+1; j<N; j++)
            if (v[i]>v[j])
            {
                costo++;
                scambia(v, i, j);
            }
    return costo;
}
```

# Modo alternativo: misurare il tempo

- Nella libreria `<time.h>` esistono alcuni tipi utili allo scopo
  - Il tipo `clock_t` permette di valutare il numero di *colpi di clock* emessi dal sistema ad un dato istante mediante la funzione `clock()`

- Esempio:

```
clock_t ct=clock();  
/* ct si stampa come intero */
```

- Quindi la differenza tra il numero di colpi di clock emessi prima e dopo l'esecuzione di una funzione ci da una misura *proporzionale* al tempo impiegato dalla funzione stessa

```
clock_t inizio, fine;  
inizio=clock();  
F(); /* eseguo una funzione generica */  
fine=clock();  
printf("%d -- %d", inizio, fine);
```

# Problema

- Considerare il seguente tipo di dato strutturato:

```
typedef struct  
{  
    char nome[20], cognome[20];  
    int eta, numero_telefonico;  
} Persona;
```

```
typedef struct  
{  
    Persona persone[50];  
    int num_persone;  
} VPersona;
```

# Esercizi

- Scrivere una funzione C che, ricevendo in ingresso una stringa `nome`, e una variabile di tipo `VPersone`, cerchi nella variabile se c'è un elemento il cui slot `nome` è uguale alla stringa in ingresso, e restituisca l'indice dell'elemento corrispondente.
  - Si implementi la funzione nel caso che l'archivio sia ordinato e non ordinato alfabeticamente per slot `nome`
- Scrivere una funzione C che, ricevendo in ingresso una variabile di tipo `VPersone`, scriva su un'altra variabile dello stesso tipo la stessa variabile d'ingresso ma ordinata per la componente `eta` di ciascun elemento dello slot `persone`.
  - Si adatti la funzione `select-sort` per variabili di tipo `Persona`

# Funzione di ricerca sequenziale

```
int cerca_persona(char nome[], VPersonne l)
{
    int i=0;

    while ( (i<l.num_personne) &&
           (strcmp(l.persone[i].nome,nome) !=0) ) i++;
    return i;
}
```

# Ora tocca a voi (homework)

- Adattate con criteri analoghi la funzione di ricerca binaria
- Adattate anche la funzione di selection-sort
- Misurate il tempo impiegato a cercare dei numeri a caso in un vettore ordinato (ricerca binaria) ed in un vettore non ordinato (ricerca sequenziale)
- Misurate il tempo impiegato  $T(N)$  ad ordinare un vettore al crescere della sua dimensione  $N$  e mettete in un grafico  $T(N)$



# Per saperne di più

- Ceri, Mandriola, Sbattella, *Informatica – arte e mestiere*, Capo. 7-8, McGraw-Hill
- Kernighan, Ritchie, *Il linguaggio C*, Cap. 4, Pearson-Prentice Hall

# Appendice: un algoritmo di ordinamento più efficiente

- Un'idea potrebbe essere: riuscire a partizionare il vettore in due parti, in modo tale che
  - tutti gli elementi di una parte (per esempio i primi  $N/2$  elementi) sono maggiori un valore di riferimento, detto “**perno**” o “**pivot**”
  - tutti gli elementi dell'altra (ad esempio i secondi  $N/2$  elementi) sono minori del perno
- A questo punto possiamo applicare ricorsivamente l'idea a ciascuna delle due partizioni, escludendo ovviamente il perno
  - Il perno può essere uno degli elementi del vettore
- Al termine di queste operazioni troviamo il vettore ordinato
- L'algoritmo che deriva da questa idea prende il nome di “quick sort”, o algoritmo di ordinamento veloce

# Implementazione in C

```
void quicksort(int *v, int p, int q) /*p=0, q=N-1 all'inizio*/
{
    int i, j, pivot;

    if(p<q) /* se il vettore non è di dimensione 1 o inferiore */
    {
        pivot=p;
        i=p;
        j=q;

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

        scambia(v, pivot, j);
        quicksort(v,p,j-1);
        quicksort(v,j+1,q);
    }
}
```

# Complessità di quicksort

- Si dimostra che, nel caso medio, la complessità di quicksort va come  **$N \log_2 N$**
- Caso medio: tutte le permutazioni dei valori del vettore sono equiprobabili
- Nel caso più favorevole, il perno avrà valore “centrale” rispetto agli altri  $N \rightarrow$  influisce sul numero di confronti effettuati da perno e sulla dimensione delle partizioni da esso determinate

# Il “quick sort”

- Supponiamo il vettore  $V$  di 10 interi
- Prendiamo ad esempio  $V[0]$  come perno
- Ora consideriamo il resto del vettore  $V - \{V[0]\}$  e, presi due indici  $p$  e  $q$ , l'uno in posizione  $V[1]$  e l'altro in  $V[9]$ , facciamoli scorrere l'uno incontro all'altro finché sono vere  $V[p] > V[0]$  e  $V[q] < V[0]$
- Quanto detto vale per l'ordine decrescente, per quello crescente basta invertire i versi di confronto di  $V[p]$  e  $V[q]$  con  $V[0]$

$V[0]$	$V[1]$	$V[2]$	$V[3]$	$V[4]$	$V[5]$	$V[6]$	$V[7]$	$V[8]$	$V[9]$
30	40	25	50	15	45	38	5	10	35

# Funzionamento del “quick sort”

- In altri termini,  $p$  si arresta non appena trova un elemento  $V[p]$  minore del perno,  $q$  non appena trova un elemento  $V[q]$  maggiore del perno
- A questo punto scambiamo  $V[p]$  con  $V[q]$  e ripetiamo il procedimento finché  $p \leq q$
- Infine scambiamo  $V[0]$  con l'elemento maggiore tra  $V[p]$  e  $V[q]$  in modo da ottenere che tutti gli elementi a sinistra del perno sono maggiori di esso, mentre tutti gli elementi a destra sono minori
- Si applicano i passaggi precedenti a ciascuna delle due parti di  $V$  fino al totale ordinamento del vettore

# Esempio di funzionamento: prima iterazione

0	1	2	3	4	5	6	7	8	9
<u>30</u>	40	25 p	50	15	45	38	5	10	35 q
<u>30</u>	40	35	50	15	45	38	5	10	25
<u>30</u>	40	35	50	38	45	15	5	10	25
45	40	35	50	38	<u>30</u>	15	5	10	25

# Esempio di funzionamento: seconda iterazione

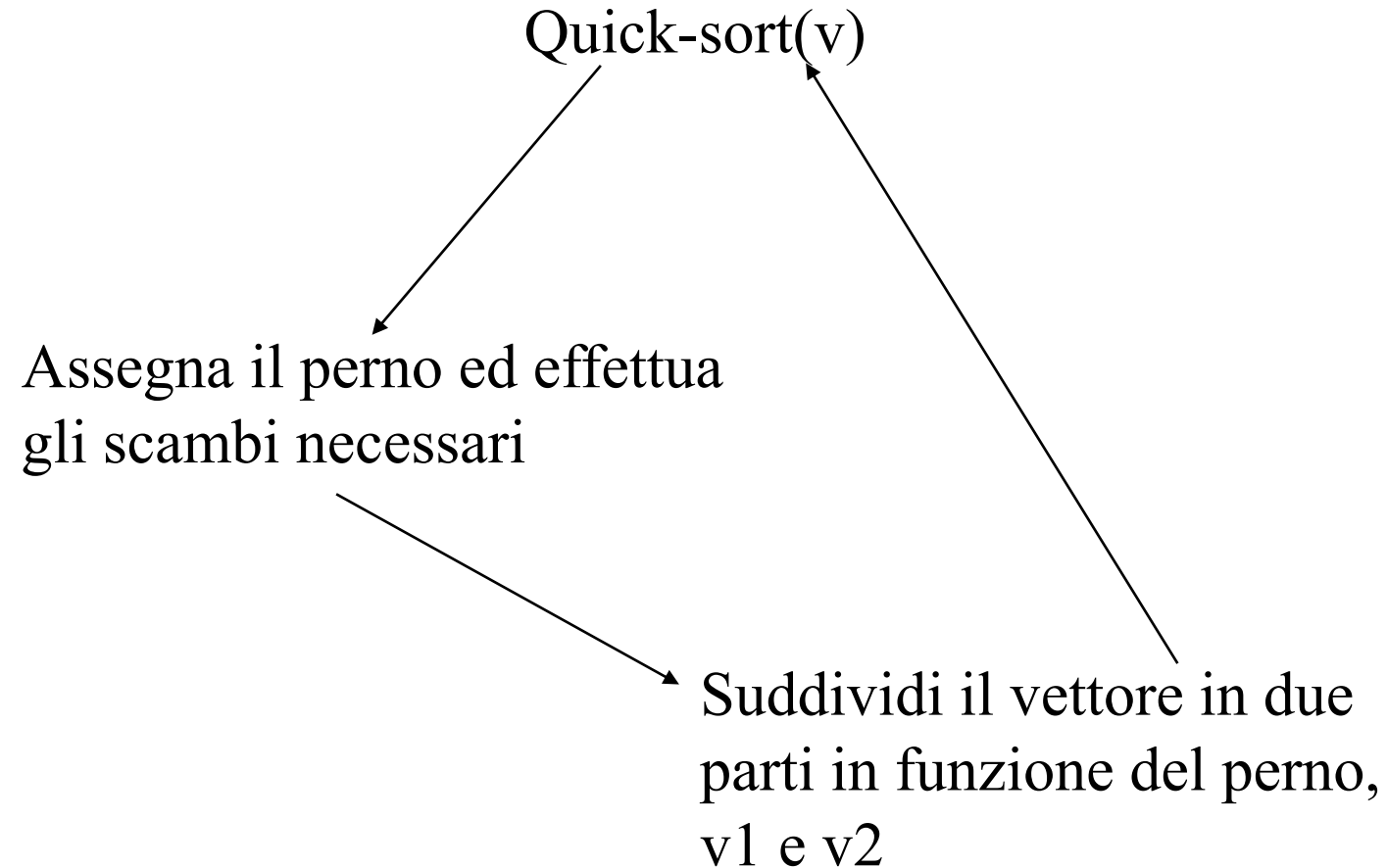
0	1	2	3	4	5	6	7	8	9
<u>45</u>	40	35	50	38	30	<u>15</u>	5	10	25
<u>45</u>	50	35	40	38	30	<u>15</u>	25	10	5
50	<u>45</u>	35	40	38	30	25	<u>15</u>	10	5



# Esempio di funzionamento: terza iterazione

0	1	2	3	4	5	6	7	8	9
50	45	<u>35</u>	40	38	30	25	15	<u>10</u>	5
50	45	38	40	<u>35</u>	30	25	15	10	5
50	45	40	<u>38</u>	35	30	25	15	10	5

# Procedimento ad alto livello



# Implementazione di Quick Sort in C

- Intestazione: QuickSort(V,N,p,q)
- Ingresso: V, N, p, q (inizializzati a 0 e N-1)
- Pseudo-codice:
- Se dimensione di V non è 1, o 0,
  - $p'=p+1$ ;  $q'=q$ ;
  - Ripeti
    - Ripeti
      - $p'=p'+1$ ;
    - Finché  $v[p'] \geq v[p]$
    - Ripeti
      - $q'=q'-1$ ;
    - Finché  $v[q'] \leq v[p]$
    - Scambia  $v[p']$  con  $v[q']$
    - $s=\text{argmax}(v[p'], v[q'])$
    - Scambia  $v[p]$  con  $v[s]$
  - Finché  $p' < q'$ ;
  - QuickSort(V,N,p,s-1);
  - QuickSort(V,N,s+1,q);

# Quicksort con il calcolo del numero dei confronti

```
int quicksort(int *v, int p, int q)
{
    int i, j, pivot, temp, costo=0;

    if(p<q)
    {
        pivot=p;
        i=p;
        j=q;

        while(i<j)
        {
            while((v[i]<=v[pivot])&&(i<q)) { i++; costo++;}
            while(v[j]>v[pivot]) { j--; costo++;}
            if(i<j) {scambia(v, i, j); costo++;}
            costo++;
        }

        scambia(v, pivot, j);
        costo=costo+quicksort(v,p,j-1);
        costo=costo+quicksort(v,j+1,q);
        costo++;
    }

    return costo;
}
```