Complessita degli algoritmi

Algoritmi di ricerca

e

Algoritmi di Ordinamento

Complessità

- Tra i problemi che ammettono soluzione esistono problemi "facili" e problemi "difficili"
  - complessità di un **problema**;
  - complessità di un **programma**;
  - **valutazione dell'efficienza** di un algoritmo.

- Efficienza = somma pesata fra **memoria occupata** e **tempo di esecuzione**

- Per valutare la **complessità dei programmi** è preferibile di norma privilegiare il tempo di esecuzione.

- Generalmente l'80% del tempo di esecuzione è dovuto a un 20% delle istruzioni e cioè a quelle nelle parti più interne dei cicli.

Esempio

- Moltiplicazione di due matrici quadrate di interi, A e B di dimezione NxN.

- Per calcolare la matrice risultante C, occorre:
  - ripetere N^2 volte il calcolo per determinare la cella C[i,j]
    - effettuare 2N letture,
    - N moltiplicazioni,
    - N-1 addizioni e
    - una scrittura per ciascuna cella C[i,j]

- In totale servono quindi:
  - $2N^3$ letture, N^3 moltiplicazioni, $N^2(N-1)$ addizioni, N^2 scritture
  - da cui il tempo richiesto dall'algoritmo (nell'ipotesi che tutte le operazioni richiedano un tempo paragonabile per essere svolte):

- **time Alg(C = A * B) (N)** = $2N^3 + N^3 + N^2(N-1) + N^2 = \mathbf{4N^3}$

Modello di costo

- ogni istruzione viene considerata di costo unitario
- il costo di una istruzione composta è pari alla somma dei costi delle istruzioni che la compongono
- il costo di una istruzione di ciclo è dato dalla somma del costo totale di esecuzione del test di fine ciclo e dal costo totale del corpo del ciclo

- Da cosa dipende la complessità di una algoritmo?

- Dall'algoritmo utilizzato
- Dalla dimensione dei dati di ingresso

La complessità dell'algoritmo viene dunque espressa in funzione della dimensione dei dati di ingresso

Esempio

N	NlogN	N^2	2^n
2	2	4	4
100	664	10000	$>10^{30}$
10000	13288	10^8	$>10^{3010}$

Se ogni operazione richiede un millisecondo si ha pertanto

$2 \equiv$ 2 millisec

$100 \equiv$ 0,1 sec

$664 \equiv$ 0,664 sec

$10000 \equiv$ 10000 millisec $\equiv$ 10 sec

$10^{30} \equiv 10^{30}$ millisec $\equiv 10^{27}$ sec $\approx 3 * 10^{19}$ anni

$10^{3010} \equiv 10^{3007}$ millisec $\equiv 1{,}6 * 10^{3005}$ sec $\approx 5 * 10^{2997}$ anni

Complessità

- Non è sempre possibile calcolare con esattezza la funzione f(n) che quantifica il tempo di esecuzione in funzione della dimensione dell'input. Si procede pertanto alla valutazione del comportamento asintotico di tale funzione:
  - caso peggiore
  - caso migliore

Complessità: O

- Limite superiore al comportamento asintotico
- Un programma ha complessità O(f(n)) se esistono opportune costanti a, b e n tali che il numero di istruzioni t(n) che vengono eseguite nel caso peggiore con input di dimensione n verifica

$$t(n) < a \cdot f(n) + b \qquad n > n'$$

- LA NOTAZIONE O FORNISCE UNA DELIMITAZIONE SUPERIORE AL COSTO DI ESECUZIONE DELL'ALGORITMO

Complessità: REGOLA 1

Il costo di un blocco di istruzioni è uguale al massimo fra i costi delle singole istruzioni del blocco

```
{   P       O(f(n))
    Q       O(g(n)
}
```

complessità del blocco

max(O(f(n)),O(g(n)))

Complessità:REGOLA 2

Un programma che richiede K volte l'esecuzione di una istruzione composta di complessità f(n) ha complessita

$$O(\Sigma \ f(n))$$

Complessità: Ω

- Limite inferiore al comportamento asintotico

- Un programma ha complessità Ω (g(n)) se esiste una costante c tale che il numero di istruzioni t(n) che vengono eseguite nel caso peggiore con input di dimensione n verifica

$$t(n) > c \cdot g(n)$$

per un numero finito di valori di n

- LA NOTAZIONE O FORNISCE UNA DELIMITAZIONE INFERIORE

La sequenza

- Insieme di elementi ordinati
- In genere tutti gli elementi di informazione della sequenza hanno la stessa struttura
- Le struttura dell'elemento è in genere complessa e viene vista come un insieme di campi di informazione
- Ad esempio la struttura di un elemento della sequenza di elementi di una rubrica telefonica è:
  - nome
  - cognome
  - numero

Campi chiave

- In una sequenza le operazioni di ricerca e ordinamento vengono effettuate scegliendo almeno un campo, definito chiave, che è quello utilizzato per la ricerca nella sequenza.
- In pratica ricerchiamo le informazioni di tutto l'elemento utilizzando una parte nota di esso, la chiave
- È molto comune la ricerca per più di un campo (**chiavi multiple**)
- Le chiavi vanno scelte in modo da ridurre le possibili omonimie
- Ad esempio nella rubrica cerchiamo principalmente per cognome e poi per nome

Esempio

1. Rossi Paolo 095456789
2. Rossi Carlo 095435612
3. Bianchi Agata 095353678

- La chiave costituita dal cognome + nome consente di selezionare un elemento
- Considerando lo scopo applicativo, la sequenza va ordinata per cognome e nome, in ordine non decrescente. Si ottiene così:

1. Bianchi Agata 095353678
2. Rossi Carlo 095435612
3. Rossi Paolo 095456789

RICERCA

Data una collezione di elementi trovare se esiste l'elemento di valore E

typedef ……. Elem
elem V[n]
int ricerca (elem *V, elem E, int NN);

V[] è la struttura che contiene la sequenza di elementi
E è la variabile che contiene il valore da ricercare nella sequenza
NN è il numero di elementi presenti nella sequenza

la funzione ritorna
l'indice dell'elemento E se è presente nella sequenza
-1 se l'elemento E non appartiene alla sequenza

RICERCA SEQUENZIALE

```
typedef  ……. Elem
elem V[n]

int ricerca_sequenziale (elem *V, elem E, int NN)
{     int t;
      for (t = 0; t<NN; t++)
            if (E == V[t]) return t
      return -1
}
```

caso peggiore O(NN)

caso medio N

RICERCA SEQUENZIALE con sentinella

Può essere utilizzata su una sequenza ordinata,

```
typedef  ……. Elem
elem V[n]

int ricerca_sentinella (elem *V, elem E, int NN)
{     int t;
      for (t = 0; t<NN && E <= V[t]; t++)
            if (E == V[t]) return t
      return -1
}
```

caso peggiore O(N)

caso medio N/2

RICERCA BINARIA

Può essere utilizzata su una sequenza ordinata,

```
typedef  ……. Elem
elem V[n]

int ricerca_binaria (elem *V, elem E, int NN)
{    int inf, sup, m;
     inf = 0;
     sup = NN-1;
     while (inf<=sup)
     {    m = (inf+sup)/2;
          if (E == V[m]) return m;
          if (E > V[m]) inf = m+1
          else sup = m-1;
     }
     return -1
}                              caso peggiore O(logN)
```

RICERCA BINARIA (impl ricorsiva)

```
typedef  ……. Elem
elem V[n]

int ric (elem *V, elem E, int inf, int sup)
{    int m;
     if (inf<=sup)
     { m = (inf+sup)/2;
       if (E == V[m]) return m;
       if (E > V[m]) return ric(V,E, m+1,sup)
          else return ric(V,E,inf, m-1);
     }
     return -1
}
```

RICERCA BINARIA (impl ricorsiva)

```
int ricerca_binaria (elem *V, elem E, int NN)
{     int inf, sup;
      inf = 0;
      sup = NN-1;
      return ric(V,E,inf,sup)
}
```

Ordinamenti interni ed esterni

- Gli ordinamenti interni sono fatti su sequenze in memoria centrale
- Gli ordinamenti esterni sono fatti su sequenze in memoria di massa

Operazioni elementari

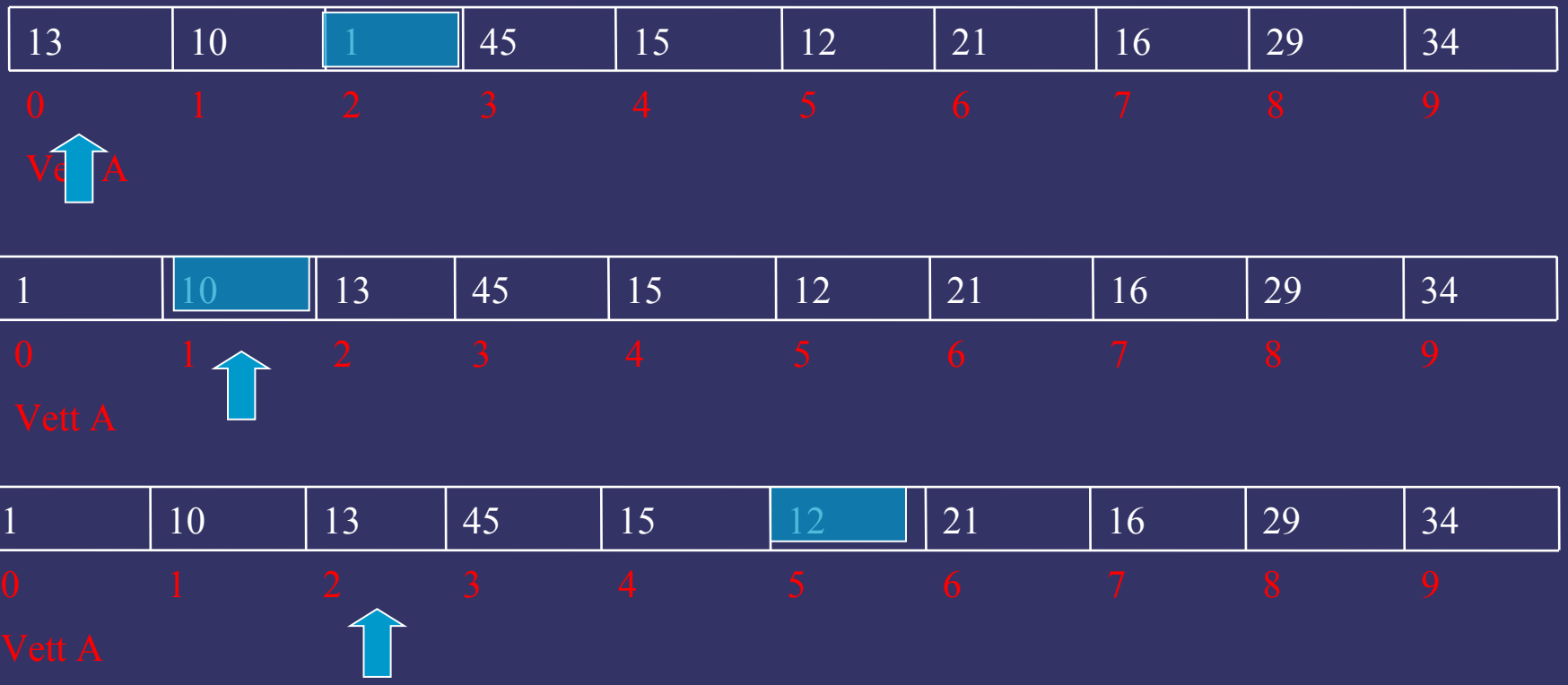
- Si nota come siano necessarie operazioni di
  - Confronto
  - Scambio

- Risulta inoltre intuitivo come possa essere necessario scandire più volte, magari su sottoinsiemi via via più piccoli la sequenza informativa

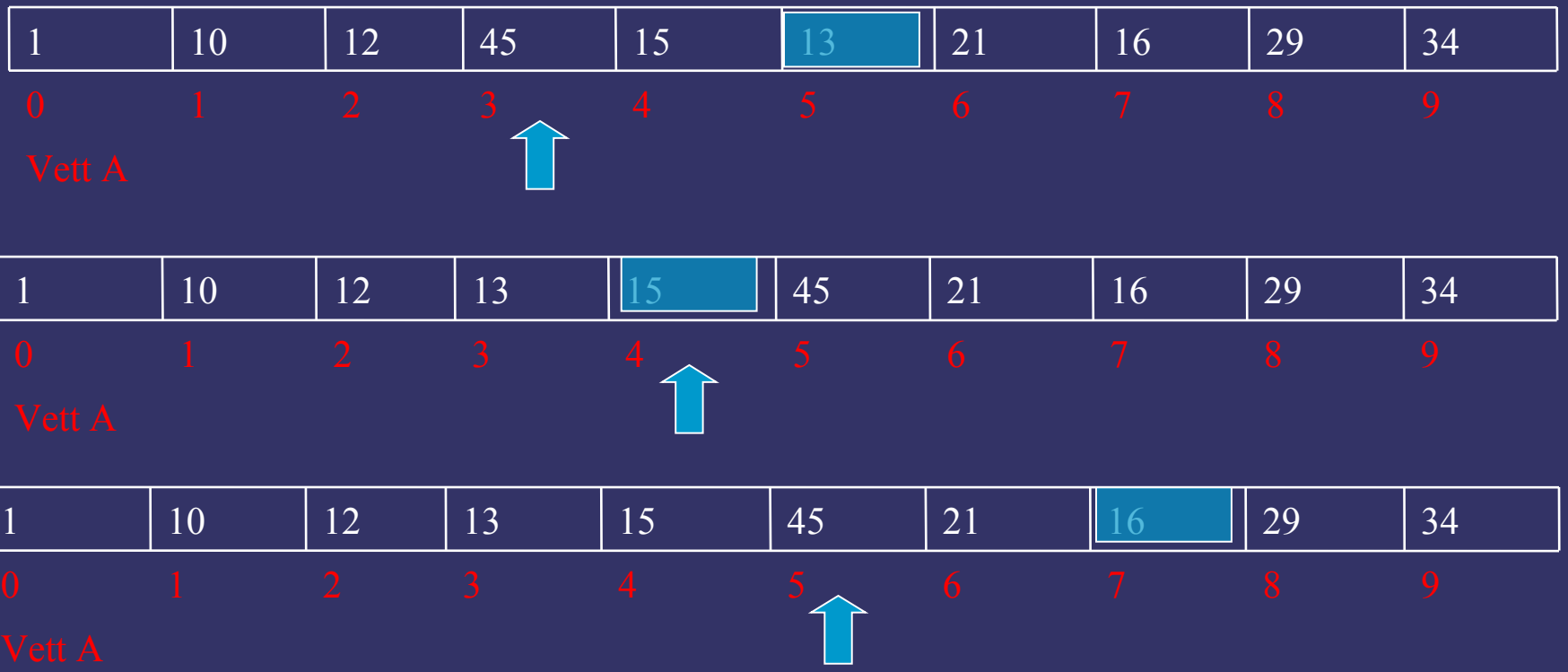
Algoritmi di ordinamento interno

- Selezione diretta
  - Selection sort
- Inserimento diretto
  - Insertion sort
- Scambio
  - Bubble sort
  - Quick sort

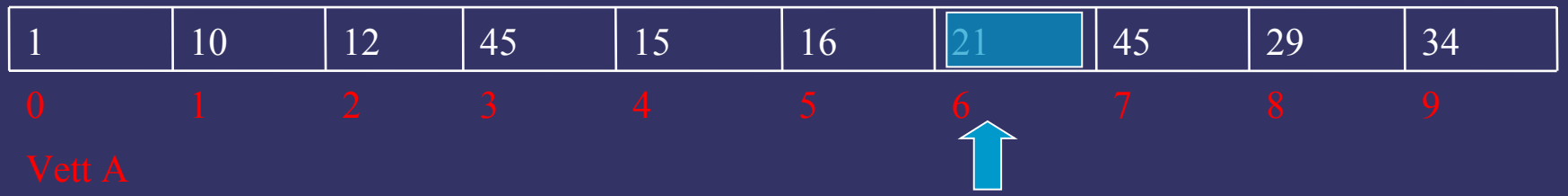
Selection sort

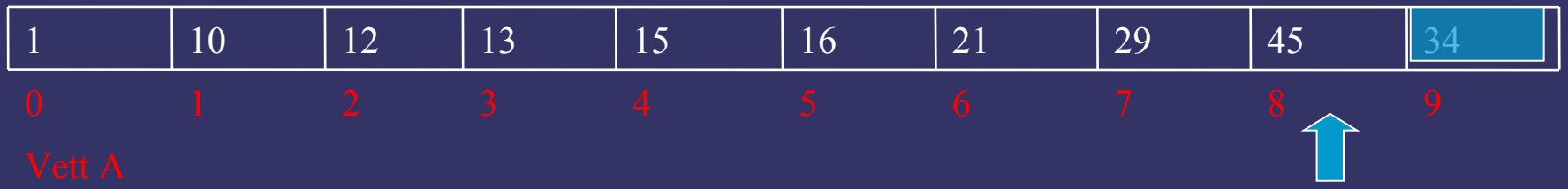


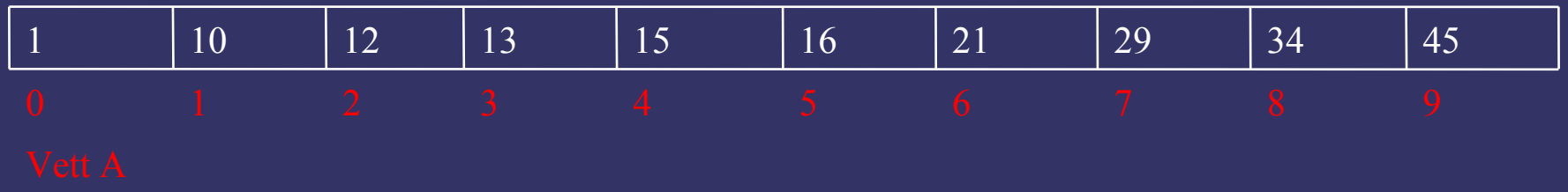
Selection sort



Selection sort





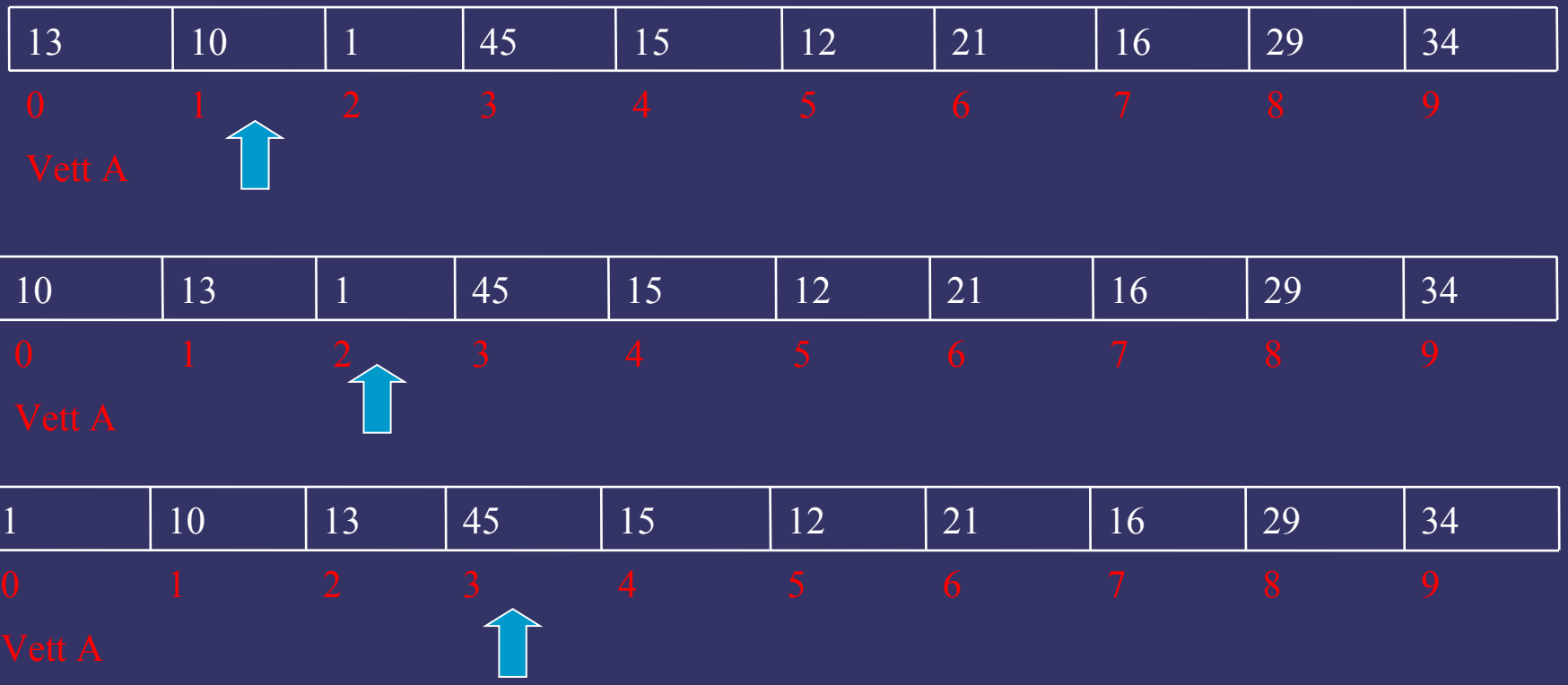


Ordinamento per selezione diretta

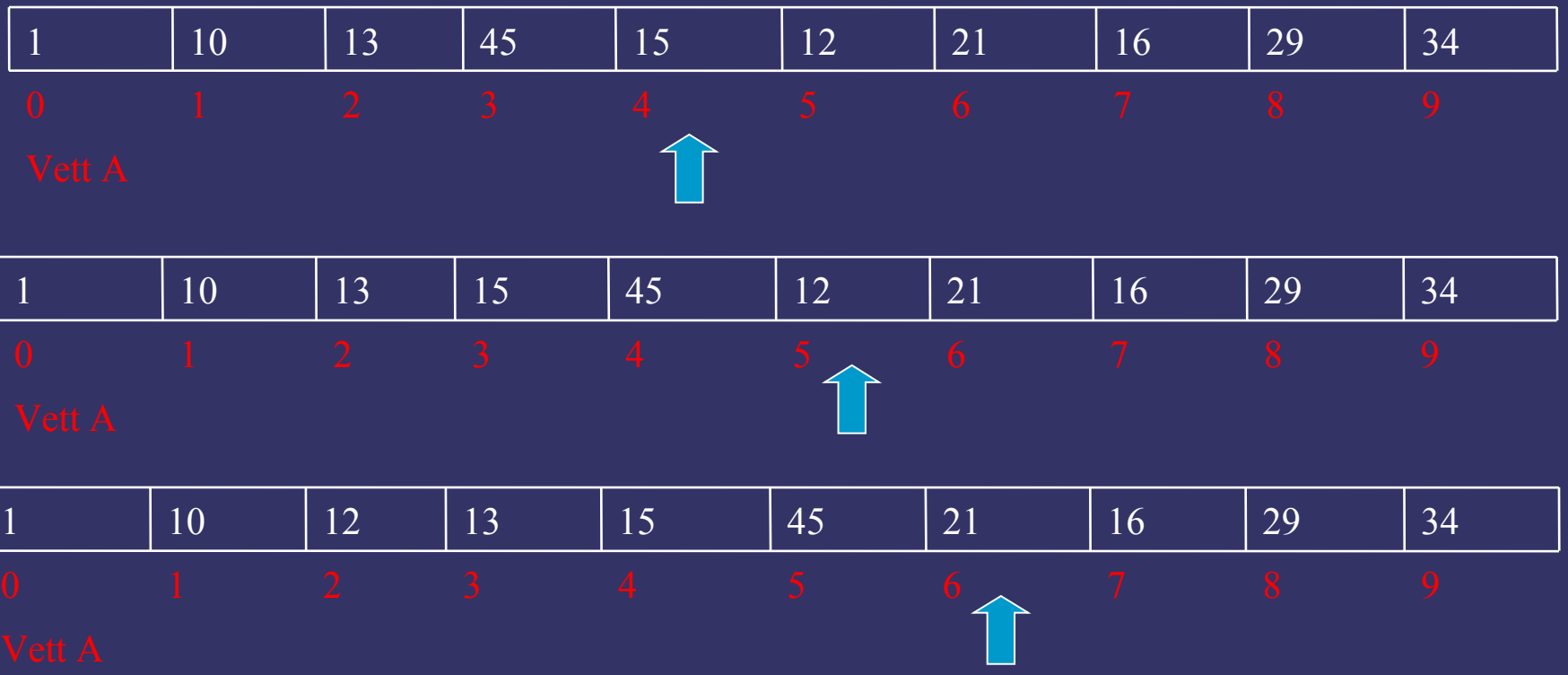
```
void selezione_diretta (elem *elemento, int NN)
{     int i, j, k, scambio;
      elem x;
      for (i = 0; i<NN-1; ++i)
      {     scambio = 0;
            k = i;
            x = elemento[i];
            for (j = i+1; j<NN; ++j)
            if (elemento[j]<x)  {
                  k = j;
                  x= elemento[j];
                  scambio=1;
             }
            if (scambio) {
                  elemento[k]=elemento[i];
                  elemento[i]=x;
            }
      }
}
```

Complessità: $O(n^2)$

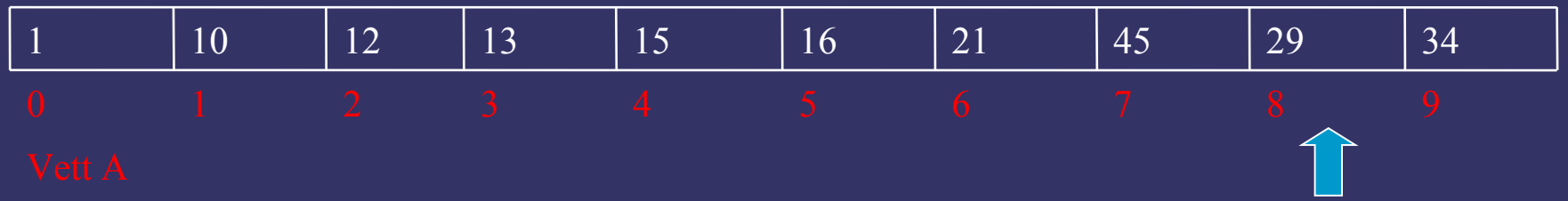
Insertion sort

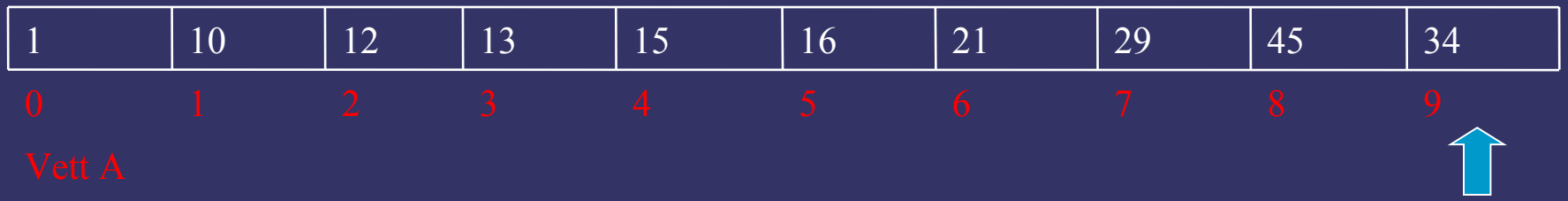


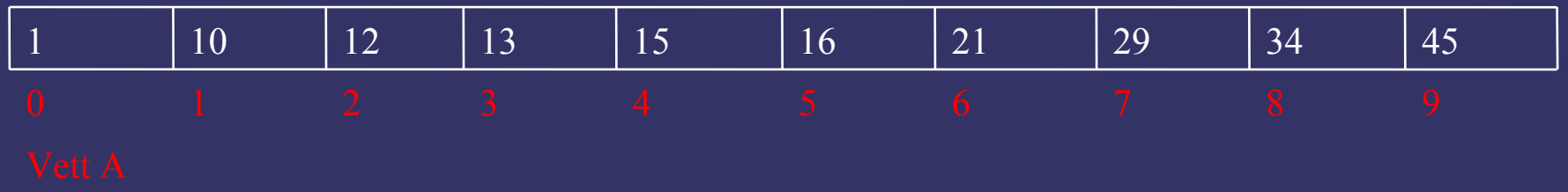
Insertion sort



Insertion sort







Ordinamento per inserimento diretto

```
void inserimento_diretto (elem *elemento, int NN)
{     int i, j, k;
      elem x;
      for (i = 1; i<NN; ++i)
      {     x = elemento[i];
            for (j = i-1; j>=0 && x<elemento[j]; j--)
                  elemento[j+1]= elemento[j];
            elemento[j+1]=x;
      };
}
```

Complessità

- Caso peggiore: $O(n^2)$
- Caso migliore: $O(n)$
- Caso medio: $O(n^2)$

Bubble Sort (cont.)

```
void BubbleSort(elem *v, int NN) {

    int scambio,i,j;
    scambio=0;
    for (j=0;j<NN-1 && !scambio; j++)   {
     scambio=1;
     for (i=NN-1;i>j;i--){
            if (v[i]<v[i-1]) {
                scambia(v,i,i-1);
                scambio=0;
            }
     }
    }
}
```

Bubble Sort

```
void scambia(elem *v, int i, int j)
{     int tmp;
      tmp = v[i];
       v[i] = v[j];
       v[j] = tmp;
}
```

Complessità del BubbleSort

- L'algoritmo BubbleSort esegue alla prima iterazione al più n-1 confronti/scambi.
- Alla seconda iterazione n-2 confronti/scambi
- Le iterazioni sono in totale al più n-1
- Il numero totale al più di confronti/scambi è:

  (N-1)+(n-2)+…1 ovvero possiamo dire che la complessità e dell'ordine:

  $O(n^2)$

QuickSort

- Si tratta di un algoritmo evoluto che ha complessità computazionale O(n) pari a n*log(n) ed inoltre usa la ricorsione.
- Ricordiamo che la ricorsione consiste nell'esprimere F(n) in funzione di F(n-1) ed inoltre conoscere il valore iniziale F(0)
- Gli algoritmi ricorsivi sono più semplici ed eleganti, ma la loro esecuzione comporta, a causa della annidarsi della funzione che si richiama da se un uso a volte esagerato dello stack

Approccio informale

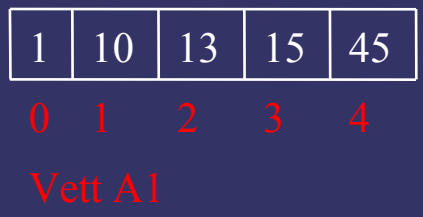
Il ragionamento descritto prima non funziona sempre.

Supponiamo di divedere il vettore A a metà e di ordinare i due pezzi, separatamente ottenendo i due vettori A1 e A2. Il vettore ricomposto non è però ordinato.

Il vincolo da porre è chiaro:

possiamo applicare questo metodo solo se il massimo elemento in A1 è inferiore o uguale al minimo elemento di A2.

L'operazione che crea due parti con la suddetta caratteristica si dice partizionamento del vettore.



Ovviamente potrebbe crearsi una zona intermedia con valori uguali che separa i due sottovettori; in questo caso il tempo di calcolo si riduce ulteriormente

Step del QuickSort

- il vettore da ordinare viene delimitato dall'indice del primo elemento (inf) e dall'indice dell'ultimo elemento (sup).
- viene scelto un elemento del vettore di indice compreso tra inf e sup, chiamato pivot, scelto del tutto casualmente. Potrebbe essere l'elemento di indice (inf+sup)/2.
- vengono utilizzati due indici i,j, che vengono inizializzati a i=inf e j=sup.
- l'indice i viene incrementato di 1 fino a quando l'elemento di indice i non è maggiore o uguale al pivot.
- l'indice j viene decrementato di 1 fino a quando l'elemento di indice j non è minore o uguale al pivot.
- nel caso in cui i<j, gli elementi di indici i e j vengono scambiati, e poi i viene incrementato e j decrementato (in modo unitario)
- nel caso in cui i=j, non si effettua lo scambio, ma i viene incrementato di 1 e j viene decrementato di 1

Step del QuickSort (cont.)

1. se i>j allora la fase di partizionamento termina. In questo modo risulta che:
   - tutti gli elementi di indice appartenente a inf,..,j sono minori o uguali del pivot
   - tutti gli elementi di indice appartenente a i,..,sup sono maggiori o uguali del pivot
   - tutti gli elementi di indice appartenente a j+1,..,i-1 sono uguali del pivot
2. l'algortimo QuickSort viene applicato ricorsivamente al sottovettore individuato dagli indici (inf,j), se tale sottovettore contiene almeno un elemento, ossia se inf<j
3. l'algortimo QuickSort viene applicato ricorsivamente al sottovettore individuato dagli indici (i,sup), se tale sottovettore contiene almeno un elemento, ossia se i<sup

Esempi del QuickSort

- Primo esempio (**caso migliore**): verrà considerato il caso in cui il vettore originario è decomposto nella prima passata in due sottovettori di dimensione uguale. Il pivot viene scelto pari al mediano degli elementi contenuti nel vettore

- Secondo esempio (**caso peggiore**): il vettore originario è decomposto in due sottovettori, di cui il primo ha dimensione uguale alla dimensione originaria meno 1, e l'altro ha una dimensione unitaria. Il pivot coincide con l'elemento massimo del vettore.

- Il terzo esempio (**caso generico**) si riferisce ad un comportamento compreso tra il peggiore e il migliore.

QuickSort: caso migliore

Se scegliamo come pivot l'elemento di indice m = (inf+sup)/2, ovvero pvett[m]=15, casualmente questo coincide con l'elemento mediano m del vettore.

L'elemento mediano è quello tale che il numero di elementi del vettore più grandi di lui è circa uguale al numero di elementi del vettore che più piccoli di lui.

Si consideri il seguente vettore di n=10 elementi:

13	10	1	45	15	12	21	16	29	34
0	1	2	3	4	5	6	7	8	9
i									j

Il numero di elementi più piccoli di 15 è 4, mentre il numero di elementi più grandi di 15 è 4.

Codifica C del QuickSort

```
void QSort (elem v[], int inf, int sup){
    elem pivot;
    int i,j;
    pivot=v[(inf+sup)/2];
    i=inf; j=sup;
    do {
     while (v[i]<pivot) i++;
     while (v[j]>pivot) j--;
     if (i<j) scambia(v,i,j);
     if (i<=j) {i++;j--;}
    } while (i<=j);
    if (inf<j) QSort(v,inf,j);
    if (i<sup) QSort(v,i,sup);
}
```

```
void scambia (elem v[], int i, int j){
    elem tmp=v[i];
    v[i]=v[j];
    v[j]=tmp;
}
```