COMPLESSITÀ DEGLI ALGORITMI

- **Tra i problemi che ammettono soluzione ne esistono di più *"facili"* e di più *"difficili"***
- ***Teoria della complessità (anni '70):***
  - **complessità di un *problema***
  - **complessità di un *programma***
  - **valutazione dell'efficienza di un *algoritmo***
- Un programma richiede ***spazio di memoria*** e ***tempo di calcolo***
- Per valutare la ***complessità dei programmi***, ci concentreremo sul secondo aspetto

COMPLESSITÀ DI UN ALGORITMO

- **Come valutare la complessità *di uno specifico algoritmo*?**
  - ***Contando il numero di operazioni* aritmetiche, logiche, di accesso ai file, etc.**
- *Ipotesi semplificative:*
  - ogni operazione abbia costo unitario
  - il tempo globalmente impiegato sia proporzionale al numero di operazioni eseguite.
- *Non ci si riferisce ad una specifica macchina*

ESEMPIO

- Per moltiplicare due matrici quadrate NxN di interi (C = A x B), occorre:
  - ripetere N^2 volte il calcolo del valore C[i,j]
  - per calcolare C[i,j], effettuare 2N letture, N moltiplicazioni, N-1 addizioni e 1 scrittura
- Totale: $2N^3$ letture, N^3 moltiplicazioni, $N^2*(N-1)$ addizioni, N^2 scritture
- Tempo richiesto:
  (ipotesi: stesso tempo per tutte le operazioni):

$$\textbf{time}_{\textbf{Alg(C = AxB)}}\textbf{(N)} = 2N^3+N^3+N^2(N-1)+N^2 = \mathbf{4N^3}$$

MOTIVAZIONI

- **Perché valutare la complessità di un algoritmo?**
  - per scegliere l'algoritmo *più efficiente*
- ***Da cosa dipende* la complessità di un algoritmo?**
  - dall'algoritmo stesso (ovvio...)
  - *dalla **"dimensione" dei dati** a cui l'algoritmo si applica*

La ***complessità*** dell'algoritmo viene dunque espressa in funzione della ***dimensione dei dati*** **(in spazio)**

MOTIVAZIONI

- Si consideri un algoritmo che risolve il generico problema P
- Avere

$$\mathbf{time_{Alg(P)}(N) = 2^N}$$

è <u>molto diverso</u> da avere

$$\mathbf{time_{Alg(P)}(N) = 4*N^3}$$

perché *cambia l'ordine di grandezza* del problema

ORDINI DI GRANDEZZA

- Tanto per quantificare:

N	$N*\log_2 N$	N^2	N^3	2^N
2	2	4	8	4
10	33	100	10^3	$> 10^3$
100	664	10.000	10^6	$>> 10^{25}$
1000	9.966	1.000.000	10^9	$>> 10^{250}$
10000	13.288	100.000.000	10^{12}	$>> 10^{2500}$

- Se un elaboratore esegue 1000 operazioni/sec, un algoritmo il cui tempo sia dell'ordine di 2^N richiede:

N	*tempo*
10	1 sec
20	1000 sec (17 min)
30	10^6 sec (>10giorni)
40	(>>10 anni)

COMPORTAMENTO ASINTOTICO

<u>Problema</u>:

- **individuare con esattezza $time_A(N)$ è spesso *molto difficile***
- ***interessa capire con grandi moli di dati***
  - con N piccolo, in pratica, qualunque algoritmo è OK
  - ***è con N grande che la situazione può diventare critica* (in particolare: per $N \to \infty$)**
- **Per questo ci interessa il *comportamento approssimmato e asintotico* della funzione $time_A(N)$**

COMPORTAMENTO ASINTOTICO

- **Anche individuare il comportamento asintotico di $time_A(N)$ non è sempre semplice**
- **D'altronde, *interessa* non tanto l'espressione esatta, quanto *l'ordine di grandezza***
  - costante al variare di N
  - lineare, quadratico... (polinomiale) al variare di N
  - logaritmico al variare di N
  - esponenziale al variare di N
- **Si usano notazioni che "danno un'idea" del *comportamento asintotico* della funzione**

NOTAZIONI ASINTOTICHE

- **<u>Limite superiore</u> al comportamento asintotico di una funzione (notazione *O*)**
  - **quando esistono tre costanti a, b, N' tali che**

    $time(N) < a\, g(N) + b \quad \forall N > N'$

    e si scrive $time(N) = O(g(N))$
- **<u>Limite inferiore</u> al comportamento asintotico di una funzione (notazione Ω)**
  - **quando esistono due costanti c, N' tali che**

    $time(N) > c\, f(N) \quad \forall N > N'$

    e si scrive $time(N) = \Omega(f(N))$

NOTAZIONI ASINTOTICHE

- **<u>Limite superiore</u> al comportamento asintotico di una funzione (notazione *O*)**
  - **quando esistono tre costanti a, b, N' tali che**

    $time(N) < a\, g(N) + b \quad \forall N > N'$

    e si scrive $time(N) = O(g(N))$

La funzione g(N) costituisce un *limite <u>superiore</u>* al costo dell'algoritmo (time(N))

- **<u>Limite inferiore</u> al comportamento asintotico di una funzione (notazione Ω)**
  - **quando esistono due costanti c, N' tali che**

    $time(N) > c\, f(N) \quad \forall N > N'$

    e si scrive $time(N) = \Omega(f(N))$

La funzione f(N) costituisce un *limite <u>inferiore</u>* al costo dell'algoritmo (time(N))

COMPORTAMENTO ASINTOTICO

<u>Caso particolare:</u>

- **se esiste una funzione f(N) tale che**

  $time_A(N) = O(f(N)) = \Omega(f(N))$
- **allora f(N) costituisce una *valutazione esatta* del costo dell'algoritmo.**

In questo caso, infatti, *le due delimitazioni inferiore e superiore coincidono*, e dunque caratterizzano compiutamente time(N)

ESEMPIO

- **Si supponga che per un certo algoritmo sia**

  $time_A(N) = 3*N^2 + 4*N + 3$
- **Poiché $3*N^2 + 4*N + 3 \leq 4*N^2 \quad \forall N>3$, si può dire che $time_A(N) = O(N^2)$**
- **D'altronde, $3*N^2 + 4*N + 3 > 3*N^2 \quad \forall N>1$, e quindi $time_A(N) = \Omega(N^2)$**

La funzione $f(N)= N^2$ è perciò una valutazione esatta del costo di questo algoritmo

CLASSI DI COMPLESSITÀ

- **Le notazioni O e Ω consentono di *dividere gli algoritmi in classi,* in base all'ordine di grandezza della loro complessità.**
  - costante 1, ... k, ...
  - sotto-lineare log N oppure N^k con k<1
  - lineare N
  - sovra-lineare N*log N, e N^k con k>1
  - esponenziale c^N oppure N^N
- **Obiettivo: dati due algoritmi, *capire se sono della stessa complessità o se uno è "migliore" (più efficiente, meno complesso) dell'altro***

ALGORITMO MIGLIORE

- **Dati due algoritmi A1 e A2 che risolvono lo stesso problema P, *A1 è migliore di A2* nel risolvere il problema P se:**
  - **$time_{A1}(N)$ è $O(time_{A2}(N))$**
  - **$time_{A2}(N)$ *non è* $O(time_{A1}(N))$**
- Ad esempio, se per due algoritmi A e B risulta:
  - $time_A(N) = 3 N^2 + N$
  - $time_B(N) = N \log N$

  l'algoritmo B è migliore di A

COMPLESSITÀ DI UN PROBLEMA

- **Finora ci siamo interessati della complessità del *singolo algoritmo* che risolve un certo problema**
- **Ora interessa capire se *il problema in quanto tale* abbia una sua complessità, cioè se sia "intrinsecamente facile" o "intrinsecamente difficile" *indipendentemente dall'algoritmo* che possiamo inventare per risolverlo**

COMPLESSITÀ DI UN PROBLEMA

Diremo allora che un problema ha:

- ***delimitazione superiore* O(g(N)) alla sua complessità se *esiste ALMENO UN algoritmo* che lo risolve con complessità O(g(N))**
- ***delimitazione inferiore* Ω(f(N)) alla sua complessità se *OGNI algoritmo* che lo risolve è di complessità *ALMENO* Ω(f(N))**

COMPLESSITÀ DI UN PROBLEMA

D

•

Il problema *non può essere più complesso* di O(g(N)), perché almeno un algoritmo che lo risolve con tale complessità esiste
Però potrebbe essere più semplice (possiamo essere noi a non aver trovato l'algoritmo migliore)

Per dire che il problema **è *sicuramente* di complessità Ω(f(N)) bisogna dimostrare che *non può esistere* *un algoritmo migliore,*** ossia che qualunque altro algoritmo che possiamo inventare avrà comunque almeno quella complessità

CLASSI DI PROBLEMI

Diremo che un problema ha complessità:

- ***lineare,* se ogni algoritmo che lo risolve ha delimitazioni di complessità O(N) e Ω(N)**
- ***polinomiale,* se ogni algoritmo risolvente ha delimitazioni di complessità O(N^k) e Ω(N^k)**
- **Problema intrattabile:** un problema per cui non esistono algoritmi risolventi di complessità polinomiale (esempio: commesso viaggiatore)

ALGORITMI OTTIMALI

Diremo che un algoritmo è *ottimale* se

- ***l'algoritmo stesso ha complessità* O(f(N))**
- ***la delimitazione inferiore* alla complessità del problema è Ω(f(N))**

È piuttosto ovvio: se il problema in quanto tale ha complessità Ω(f(N)), e l'algoritmo in questione ha appunto complessità O(f(N)), di meglio non potremo mai trovare

VALUTAZIONI DI COMPLESSITÀ

- **Come valutare la complessità *in pratica* ?**

Concetto di *ISTRUZIONE DOMINANTE*

- **Dato un algoritmo A il cui costo è t(N), una sua istruzione viene detta *dominante* se esistono opportune costanti a, b, N' tali che**

$$t(N) < a\, d(N) + b \qquad \forall N > N'$$

dove d(N) indica *quante volte* l'istruzione dominante viene eseguita

VALUTAZIONI DI COMPLESSITÀ

- **Com**

Conce

- **Dato** **sua istruzione** **detta *dominante* se esistono opportune costanti a, b, N' tali che**

L'idea di fondo è che l'istruzione dominante venga eseguita un numero di volte *proporzionale alla complessità dell'algoritmo*, che perciò risulta essere O(d(N))

Negli algoritmi di
- ordinamento di un array di elementi
- ricerca di un elemento in un array di elementi

l'istruzione dominante è *il confronto* fra elementi

ESEMPIO

Ricerca esaustiva di un elemento in un array

```
boolean ricerca (int v[], int el){
  int i=0;
  boolean trovato=false;
  while (i<N) {
    if (el == v[i])
      trovato = true;
    i++;
  }
  return trovato;
}
```

istruzioni dominanti

N+1 confronti nel `while`
N confronti nell' `if`
→ *costo lineare* **O(N)**

DIPENDENZA DAI DATI DI INGRESSO

- **Spesso accade che il costo di un algoritmo dipenda non solo dalla *dimensione* dei dati di ingresso, ma anche dai *particolari valori dei dati di ingresso***
  - ad esempio, un algoritmo che ordina un array può avere un costo diverso secondo se l'array è "molto disordinato" o invece "quasi del tutto ordinato"
  - analogamente, un algoritmo che ricerca un elemento in un array può costare poco, se l'elemento viene trovato subito, o molto di più, se l'elemento si trova "in fondo" o è magari del tutto assente

DIPENDENZA DAI DATI DI INGRESSO

- **In queste situazioni occorre distinguere diversi casi:**
  - **caso *migliore***
  - **caso *peggiore***
  - **caso *medio***
- **Solitamente la complessità si valuta sul *caso peggiore***
- **Tuttavia, poiché esso è di norma assai raro, spesso si considera anche il *caso medio***
  - Caso medio: ogni elemento è *equiprobabile*

ESEMPIO

Per la ***ricerca sequenziale*** in un array, il costo dipende dalla posizione dell'elemento cercato.

- ***Caso migliore***: l'elemento è il primo dell'array $\rightarrow$ un solo confronto
- ***Caso peggiore***: l'elemento è l'ultimo o non è presente $\rightarrow$ N confronti, costo *lineare* ***O*(N)**
- ***Caso medio***: l'elemento può con egual probabilità essere il primo (1 confronto), il secondo (2 confronti), ... o l'ultimo (N confronti)

$$\sum \text{Prob(el(i))} * i = \sum (1/N) * i = (N+1)/2 = \boldsymbol{O(N/2)}$$

ALGORITMI DI ORDINAMENTO

- **Scopo: *ordinare una sequenza di elementi* in base a una certa *relazione d'ordine***
  - **lo scopo finale è ben definito** $\rightarrow$ ***algoritmi equivalenti***
  - **diversi algoritmi possono avere *efficienza assai diversa***
- **Ipotesi:**
  gli elementi siano memorizzati in un array

ALGORITMI DI ORDINAMENTO

Principali algoritmi di ordinamento:

- ***naïve sort*** (semplice, intuitivo, poco efficiente)
- ***bubble sort*** (semplice, un po' più efficiente)
- ***shell sort*** (generalizza bubble, efficienza media)
- ***insert sort*** (intuitivo, abbastanza efficiente)
- ***quick sort*** (non intuitivo, alquanto efficiente)
- ***merge sort*** (non intuitivo, molto efficiente)

Per "misurare le prestazioni" di un algoritmo, conteremo quante volte viene svolto il ***confronto fra elementi dell'array***

NAÏVE SORT

- **Molto intuitivo e semplice, è il primo che viene in mente**

  Specifica (sia **n** la dimensione dell'array **v**)

```
while (<array non vuoto>) {
  <trova la posizione p del massimo>
  if (p<n-1) <scambia v[n-1] e v[p] >
  /* invariante: v[n-1] contiene il massimo */
  <restringi l'attenzione alle prime n-1 caselle
   dell' array, ponendo n' = n-1 >
}
```

NAÏVE SORT

Codifica

```
public static void naiveSort(int v[])
{int len = v.length;
 int p;
 while (len>1) {
   p = trovaPosMax (v,len);
   if (p < len-1)
          scambia(v,p,len-1);
   len--;
 }
}
```


NAÏVE SORT

Codifica

```
public static void naiveSort(int v[])
{int len = v.length;
 int p;
 while (len>1) {
   p = trovaPosMax (v,len);
   if (p < len-1)
          scambia(v,p,len-1);
   len--;
 }
}
```

La dimensione dell'array cala di 1 a ogni giro

NAÏVE SORT

Codifica

```
static public int trovaPosMax
      (int v[], int n)
{ posMax=0;
  for (int i=1; i<n; i++)
     if (v[posMax] < v[i]) posMax = i;
  return posMax;
}
```

All'inizio si assume v[0] come max di tentativo

Si scandisce l'array e, se si trova un elemento maggiore del max attuale, lo si assume come nuovo max, memorizzandone la posizione

NAÏVE SORT

Valutazione di complessità

- Il numero di *confronti* necessari vale *sempre*:

$$(N-1) + (N-2) + (N-3) + \ldots + 2 + 1 = \\ = N^*(N-1)/2 \quad = \quad O(N^2/2)$$

- *Nel caso peggiore*, questo è anche il numero di scambi necessari (in generale saranno meno)
- ***Importante*: la complessità non dipende dai particolari dati di ingresso**
  - **l'algoritmo fa gli stessi confronti sia per un array disordinato, sia per un array *già ordinato!!***

BUBBLE SORT

- **Corregge il difetto principale del naïve sort: quello di *non accorgersi se l'array, a un certo punto, è già ordinato***
- **Opera per *"passate successive"* sull'array:**
  - **a ogni "passata", considera una ad una *tutte le possibili coppie di elementi adiacenti*, scambiandoli se risultano nell'ordine errato**
  - **così, dopo ogni passata, l'elemento massimo è in fondo alla parte di array considerata**
- **Quando non si verificano scambi, l'array è ordinato, e l'algoritmo termina**

BUBBLE SORT

- **Corregge il difetto principale del naïve sort: quello di *non accorgersi se l'array, a un certo punto, è***

Può accadere anche alla prima "passata", se l'array è già ordinato

- **Opera per *"p***
  - **a ogni "passata", considera una a**

Accorgendosi di array già ordinati, l'algoritmo evita lavoro inutile

- **Quando non si verificano scambi, l'array è ordinato, e l'algoritmo termina**

BUBBLE SORT

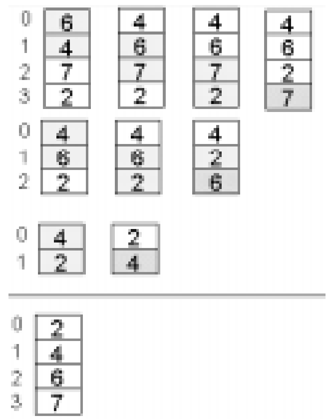
```
static   public   void   bubbleSort
  (int v[], int n){
 boolean ordinato = false;
 while (n>1 && !ordinato){
   ordinato = true;
   for (int i=0; i<n-1; i++)
     if (v[i] > v[i+1]) {
      scambia(v,i,i+1);
      ordinato = false; }
   n--;
 }
}
```

Continua solo se l'array non è ancora ordinato

A ogni iterazione ipotizza che l'array sia ordinato, poi verifica: se deve fare anche solo uno scambio, non era vero

BUBBLE SORT

Esempio



Iª passata (dim. = 4) al termine, 7 è a posto

IIª passata (dim. = 3) al termine, 6 è a posto

IIIª passata (dim. = 2) al termine, 4 è a posto

array ordinato

BUBBLE SORT

<u>*Valutazione di complessità*</u>

- Caso peggiore: numero di *confronti* identico al precedente → **O(N^2/2)**
- ***Nel caso migliore, però, basta una sola passata***, con N-1 confronti → **O(N)**
- *Nel caso medio,* i confronti saranno compresi fra N-1 e N^2/2, a seconda dei dati di ingresso

SHELL SORT

- **È una variante del bubble sort**
- **Opera anch'esso per passate successive, considerando coppie di elementi...**
- **...ma *non sono più coppie adiacenti*, bensì coppie di elementi *a distanza "gap"***
- **Questa distanza gap:**
  - **all'inizio è la metà della dimensione dell'array**
  - **poi, viene ridotta per dimezzamenti successivi**

SHELL SORT

- **Ad esempio, su un vettore di 8 elementi:**
  - **il gap iniziale è 4 → si considerano le 4 coppie di elementi di posizione (0,4), (1,5), (2,6), (3,7)**
  - **poi, il gap si riduce a 2, e si considerano le 6 coppie di elementi di posizione (0,2), (1,3),(2,4), ...**
  - **infine, il gap si riduce a 1, e si considerano le coppie di elementi adiacenti, come nel bubble**
- **In caso di scambio, *i confronti vengono retro-propagati* fino all'inizio del vettore, potendo quindi generare nuovi scambi**

SHELL SORT

Esempio: v = [20, 4, 12, 14, 10, 16, 2]

- **Inizialmente, dim=7, gap = 3**
  - **v = [20, 4, 12, 14, 10, 16, 2]**
    20>14 → scambio (nessuna retropropagazione)
    Risultato: **v = [14, 4, 12, 20, 10, 16, 2]**
  - **v = [14, 4, 12, 20, 10, 16, 2]**
    20>2 → scambio
    Risultato: **v = [14, 4, 12, 2, 10, 16, 20]**
    Retropropagazione: **v = [14, 4, 12, 2, 10, 16, 20]**
    14>2 → scambio
    Risultato: **v = [2, 4, 12, 14, 10, 16, 20]**

...

SHELL SORT

Situazione: v = [2, 4, 12, 14, 10, 16, 20]

- **Ora gap = 3/2 = 1** ***(coppie adiacenti)***
  - **v = [2, 4, 12, 14, 10, 16, 20]**
    nessuno scambio né retropropagazione
    ...
  - **v = [2, 4, 12, 14, 10, 16, 20]**
    14>10 → scambio
    Risultato: **v = [2, 4, 12, 10, 14, 16, 20]**
    Retropropagazione: **v = [2, 4, 12, 10, 14, 16, 20]**
    12>10 → scambio
    Risultato: **v = [2, 4, 10, 12, 14, 16, 20]**

...

SHELL SORT

Situazione: v = [2, 4, 10, 12, 14, 16, 20]

- **gap vale sempre 1** ***(coppie adiacenti)***
  - **v = [2, 4, 10, 12, 14, 16, 20]**
    nessuno scambio né retropropagazione
  - **v = [2, 4, 10, 12, 14, 16, 20]**
    nessuno scambio né retropropagazione
- **fine algoritmo**

Valutazione di complessità

- difficile da calcolare
- è stato stimato un caso medio pari a $\mathbf{O(N^{1.7})}$

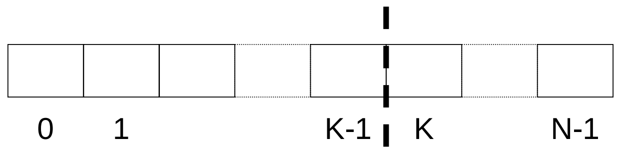
INSERT SORT

- **Approccio originale: per ottenere un array ordinato basta** ***costruirlo ordinato, inserendo gli elementi al posto giusto fin dall'inizio***
- **Idealmente, il metodo costruisce un nuovo array, contenente gli stessi elementi del primo, ma ordinato**
- **In pratica,** ***non è necessario costruire davvero un secondo array*****, in quanto le stesse operazioni possono essere svolte direttamente sull'array originale: così, alla fine esso risulterà ordinato**

INSERT SORT

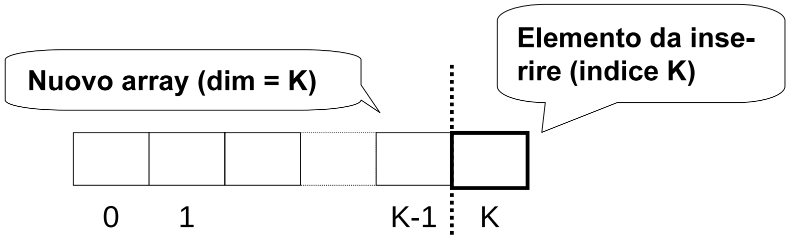
Scelta di progetto

- ***"vecchio" e "nuovo" array condividono lo stesso array fisico di N celle (da 0 a N-1)***
- ***in ogni istante, le prime K celle (numerate da 0 a K-1) costituiscono il nuovo array***
- ***le successive N-K celle costituiscono la parte residua dell'array originale***



INSERT SORT

- Come conseguenza della scelta di progetto fatta, **in ogni istante *il nuovo elemento da inserire si trova nella cella successiva alla fine del nuovo array, cioè* la (K+1)-esima** (il cui indice è K)



INSERT SORT

Specifica

```
for (int k=1; k<n; k++)
```

*<inserisci alla posizione k-esima del nuovo array **l'elemento minore fra quelli rimasti nell'array originale**>*

All'inizio (k=1) il nuovo array è la sola prima cella

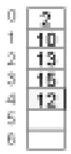
Codifica

```
static void insertSort(int v[], int n){
  int i;
  for (k=1; k<n; i++)
    insMinore(v,k);
}
```

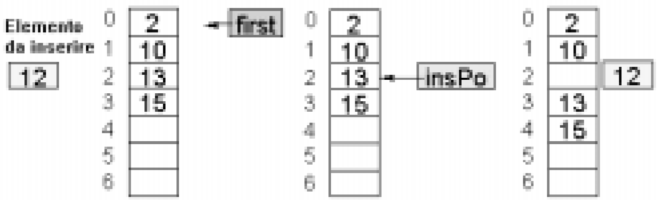
Al passo *k*, la demarcazione fra i due array è alla posizione k

INSERT SORT

Esempio



Scelta di progetto: se il nuovo array è lungo K=4 (numerate da 0 a 3) l'elemento da inserire si trova nella cella successiva (di indice K=4).



INSERT SORT

Specifica di insMinore()

```
static void insMinore(int v[], int pos)
```

{ *<determina la posizione in cui va inserito il nuovo elemento>*

<crea lo spazio spostando gli altri elementi in avanti di una posizione>

<inserisci il nuovo elemento alla posizione prevista>

}

INSERT SORT

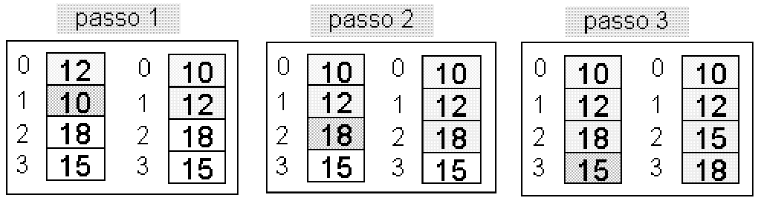
Codifica di insMinore()

```
static void insMinore(int v[], int pos){
 int i = pos-1, x = v[pos];
  while (i >= 0 && x < v[i])
  {
   v[i+1]= v[i]; /* crea lo spazio */
   i--;
  }
  v[i+1]=x;    /* inserisce l'elemento */
}
```

Determina la posizione a cui inserire x

INSERT SORT

Esempio



INSERT SORT

Valutazione di complessità

- *Nel caso peggiore* (array ordinato al contrario), richiede 1+2+3+…+(N-1) confronti e spostamenti $\rightarrow$ **O(N²/2)**
- *Nel caso migliore* (array già ordinato), bastano solo N-1 confronti (senza spostamenti)
- ***Nel caso medio*,** a ogni ciclo il nuovo elemento viene inserito nella posizione centrale dell'array $\rightarrow$ 1/2+2/2+…+(N-1)/2 confronti e spostamenti
  Morale: O(N²/4)

QUICK SORT

- **Idea base: *ordinare un array corto è molto meno costoso che ordinarne uno lungo*.**
- **Conseguenza: può essere utile *partizionare l'array in due parti e ordinarle separatamente*.**
- **<u>In pratica</u>:**
  - **si suddivide il vettore in due "sub-array", delimitati da un elemento "sentinella" (*pivot*)**
  - **il primo array deve contenere solo elementi *minori o uguali* al pivot, il secondo solo elementi *maggiori* del pivot.**

QUICK SORT

Algoritmo ricorsivo:

- **i due sub-array ripropongono un problema di ordinamento *in un caso più semplice* (array più corti)**
- **a forza di scomporre un array in sub-array, si giunge a un array di un solo elemento, che è già ordinato (*caso banale*)**

QUICK SORT

Struttura dell'algoritmo

- **scegliere un elemento come pivot**
- **partizionare l'array nei due sub-array**
- **ordinarli separatamente (*ricorsione*)**

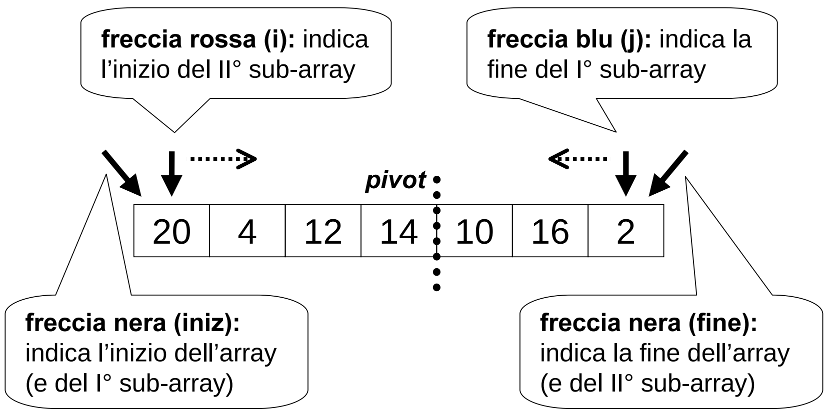
L'operazione-base è il *partizionamento dell'array nei due sub-array*. Per farla:

- se il primo sub-array ha un elemento > pivot, e il secondo array un elemento < pivot, questi due elementi vengono *scambiati*

Poi si riapplica quicksort ai due sub-array

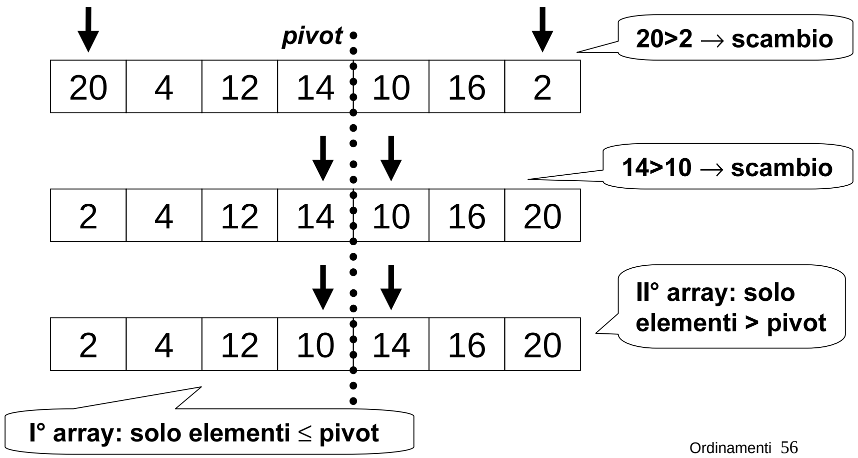
QUICK SORT

Esempio: *legenda*



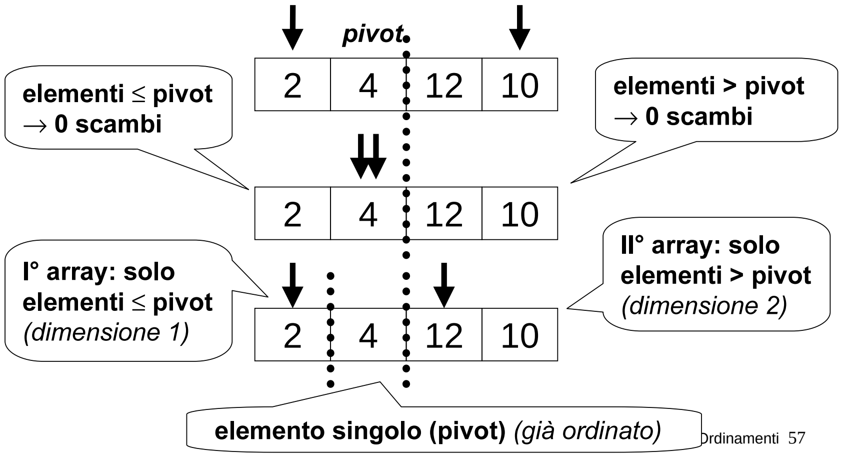
QUICK SORT

Esempio (ipotesi: si sceglie 14 come pivot)



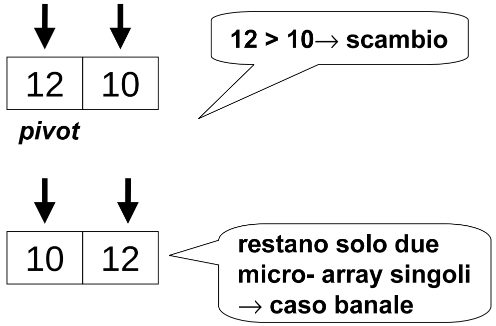
QUICK SORT

Esempio (passo 2: ricorsione sul I° sub-array)



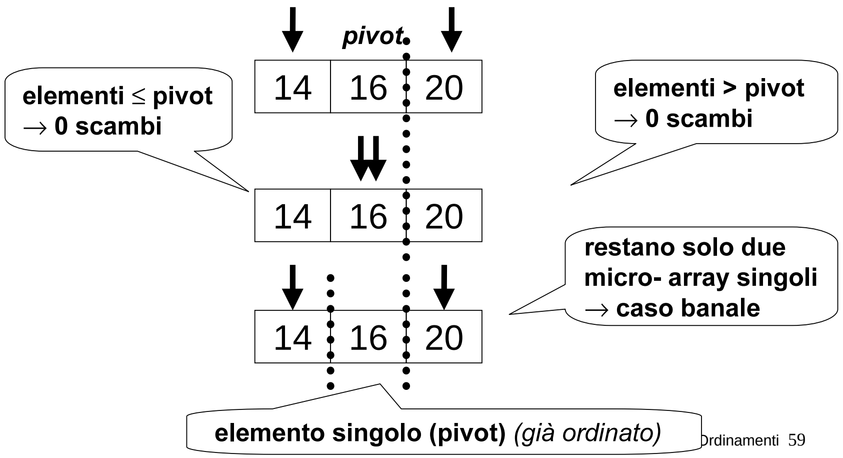
QUICK SORT

Esempio (passo 3: ricors. sul II° sub-sub-array)



QUICK SORT

Esempio (passo 4: ricorsione sul II° sub-array)



QUICK SORT

Specifica

```
static public void quickSort
  (int v[],int iniz,int fine){
 if (<vettore non vuoto> )
   <scegli come pivot l'elemento mediano>
   <isola nella prima metà array gli elementi minori o
     uguali al pivot e nella seconda metà quelli maggiori >
   <richiama quicksort ricorsivamente sui due sub-array,
     se non sono vuoti >
}
```

QUICK SORT

Codifica

```
static public void quickSort
  (int v[],int iniz,int fine){
 int i, j, pivot;
 if (iniz<fine) {
   i = iniz,  j = fine;
   pivot = v[(iniz + fine)/2];
```

<isola nella prima metà array gli elementi minori o uguali al pivot e nella seconda metà quelli maggiori >

<richiama quicksort ricorsivamente sui due sub-array, se non sono vuoti >

```
}
```


QUICK SORT

Codifica

```
… quickSort(int v[],int iniz,int fine){
 int i, j, pivot;
 if (iniz<fine) {
   i = iniz,  j = fine;
   pivot = v[(iniz + fine)/2];
```

<isola nella prima metà array gli elementi minori o uguali al pivot e nella seconda metà quelli maggiori >

```
   if (iniz < j) quickSort(v, iniz, j);
   if (i < fine) quickSort(v, i, fine);
}
```


QUICK SORT

Codifica

<isola nella prima metà array gli elementi minori o uguali al pivot e nella seconda metà quelli maggiori >

```
 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);
```

<invariante: qui j<i, quindi i due sub-array su cui applicare la ricorsione sono (iniz,j) e (i,fine) >

QUICK SORT

La complessità dipende dalla scelta del pivot:

- se il pivot è scelto male (uno dei due sub-array ha lunghezza zero), i confronti sono **$O(N^2)$**
- **se però il pivot è scelto bene (in modo da avere due sub-array di egual dimensione):**
  - **si hanno $\log_2 N$ attivazioni di quicksort**
  - **al passo k si opera su 2^k array, ciascuno di lunghezza $L = N/2^k$**
  - **il numero di confronti ad ogni livello è sempre N (L confronti per ciascuno dei 2^k array)**
- **Numero globale di confronti: $O(N \log_2 N)$**

QUICK SORT

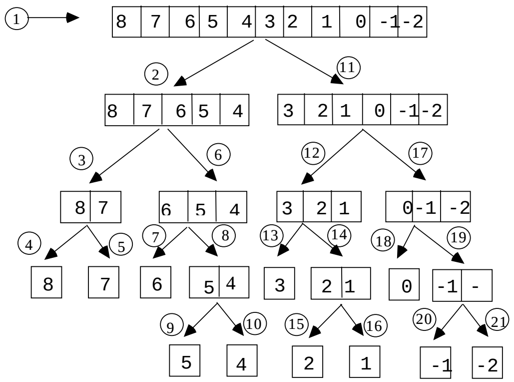
- **Si può dimostrare che O(N $\log_2$ N) è un limite inferiore alla complessità del *problema dell'ordinamento di un array***
- **Dunque, *nessun algoritmo, presente o futuro, potrà far meglio di* O(N $\log_2$ N)**
- **Però, il quicksort raggiunge questo risultato *solo se il pivot è scelto bene***
  - **per fortuna, la suddivisione in sub-array uguali è la cosa più probabile nel caso medio**
  - **l'ideale sarebbe però che tale risultato fosse raggiunto <u>*sempre*</u>*:* a ciò provvede il *Merge Sort***

MERGE SORT

- **È una variante del quick sort che produce *sempre* due sub-array di egual ampiezza**
  - **così, ottiene sempre il caso ottimo O(N*$\log_2$ N)**
- ***In pratica:***
  - **si spezza l'array in due parti *di ugual dimensione***
  - **si ordinano separatamente queste due parti *(chiamata ricorsiva)***
  - **si fondono i due sub-array ordinati così ottenuti in modo da ottenere un unico array ordinato.**
- **Il punto cruciale è l'algoritmo di fusione (*merge*) dei due array**

MERGE SORT

<u>Esempio</u>



MERGE SORT

<u>Specifica</u>

```
static public void mergeSort(int v[],
     int first, int last, int vout[])

{if (<array non vuoto>){
  <partiziona l'array in due metà>
  <richiama mergeSort ricorsivamente sui due sub-array,
   se non sono vuoti>
  <fondi in vout i due sub-array ordinati>
  }
}
```

MERGE SORT

Codifica

```
static public void mergeSort(int v[],
            int first, int last, int vout[])
{int mid;
  if ( first < last ) {
    int mid = (last + first) / 2;
    mergeSort(v, first, mid,  vout);
    mergeSort(v, mid+1, last, vout);
    merge(v, first, mid+1, last, vout);
  }
}
```

mergeSort() si limita a suddividere l’array: è merge() che svolge il lavoro

MERGE SORT

Codifica di merge()

```
static private void merge(int v[], int i1,
  int i2, int fine, int vout[]){
  int i=i1, j=i2, k=i1;
  while ( i <= i2-1 &&  j <= fine ) {
    if (v[i] < v[j]) vout[k] = v[i++];
    else vout[k] = v[j++];
    k++;
  }
  while (i<=i2-1) { vout[k] = v[i++]; k++; }
  while (j<=fine) { vout[k] = v[j++]; k++; }
  for (i=i1; i<=fine; i++) v[i] = vout[i];
}
```


ESPERIMENTI

- **Verificare le valutazioni di complessità che abbiamo dato non è difficile**
  - **basta predisporre un programma che “conti” le istruzioni di confronto, incrementando ogni volta un’apposita variabile intera ...**
  - **... e farlo funzionare con diverse quantità di dati di ingresso**
- **Farlo può essere molto significativo**

ESPERIMENTI

- **Risultati:**

N	$N^2/2$	$N^2/4$	$N \log_2 N$	naive sort	bubble sort	insert sort	quick sort	merge sort
15	112	56	59	119	14	31	57	39
45	1012	506	247	1034	900	444	234	191
90	4050	2025	584	4094	2294	1876	555	471
135	9112	4556	955	9179	3689	4296	822	793

- per problemi semplici, anche gli algoritmi “poco sofisticati” funzionano abbastanza bene, *a volte meglio degli altri*
- quando invece il problema si fa complesso, la differenza diventa ben evidente.