Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Teoria della complessità
Fondamenti e casi concreti

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
IL PRESENTE MATERIALE È RISERVATO AL PERSONALE DELL'UNIVERSITÀ DI BOLOGNA E NON PUÒ ESSERE UTILIZZATO AI TERMINI DI LEGGE DA ALTRE PERSONE O PER FINI NON ISTITUZIONALI

COMPLESSITÀ DEGLI ALGORITMI

- Tra i problemi che ammettono soluzione ne esistono di più *"facili"* e di più *"difficili"*.
- *Teoria della complessità (anni '70):*
  - valutazione della complessità di un *algoritmo*
  - valutazione della complessità di un *problema* (indipendentemente dallo specifico algoritmo)
  - valutazione dell'efficienza di un *algoritmo*
- Tali valutazioni si fanno con riferimento a due parametri: *spazio di memoria occupato* e *tempo di calcolo*.

COMPLESSITÀ DI UN ALGORITMO

- La complessità di uno *specifico algoritmo* si valuta formulando un *modello di costo*
  - non ci si riferisce ad una specifica macchina
  - tipicamente ci si basa sul *numero di operazioni*
- Ipotesi semplificative:
  - ogni operazione elementare abbia *costo unitario*
  - ogni invocazione di funzione, metodo o costruttore sia valutata in base al corpo del metodo, più il costo dei metodi o funzioni da essa chiamati
  - il tempo globale impiegato sia *proporzionale al numero di operazioni considerate*.

ESEMPIO

- Per moltiplicare due matrici quadrate N×N di interi (C = A × 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):

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

MOTIVAZIONI

- Valutare la complessità di un algoritmo serve per *scegliere l'algoritmo più efficiente*
- Da cosa dipende la complessità di un algoritmo?
  - dall'algoritmo stesso (ovvio...)
  - *dalla "dimensione" dei dati* trattati.

In effetti, se un algoritmo A risolve un generico problema P con un tempo dell'ordine di:

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

la sua efficienza è ben diversa rispetto a un algoritmo B in cui:

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

perché l'andamento di queste funzioni è totalmente diverso!

MIPS e PROCESSORI: anni 2000

Per valutare praticamente l'impatto di queste funzioni, facciamoci un'idea di *quante operazioni al secondo* possa gestire un processore:

AMD Athlon	3,561 MIPS at 1.2 GHz	2.967 MIPS/MHz	2000
AMD Athlon XP 2400+	5,935 MIPS at 2.0 GHz	2.967 MIPS/MHz	2002
Pentium 4 Extreme Edition	9,726 MIPS at 3.2 GHz	3.039 MIPS/MHz	2003
ARM Cortex A8	2,000 MIPS at 1.0 GHz	2.0 MIPS/MHz	2005
AMD Athlon FX-57	12,000 MIPS at 2.8 GHz	4.285 MIPS/MHz	2005
AMD Athlon 64 3800+ X2 (Dual Core)	14,564 MIPS at 2.0 GHz	7.282 MIPS/MHz	2005
Xbox360 IBM "Xenon" Triple Core	19,200 MIPS at 3.2 GHz	2.0 MIPS/MHz	2005
PS3 Cell BE (PPE only)	10,240 MIPS at 3.2 GHz	3.2 MIPS/MHz	2006
AMD Athlon FX-60 (Dual Core)	18,938 MIPS at 2.6 GHz	7.283 MIPS/MHz	2006
Intel Core 2 Extreme X6800	27,079 MIPS at 2.93 GHz	9.242 MIPS/MHz	2006
Intel Core 2 Extreme QX6700	49,161 MIPS at 2.66 GHz	18.481 MIPS/MHz	2006
P.A. Semi PA6T-1682M	8,800 MIPS at 2.0 GHz	4.4 MIPS/MHz	2007
Intel Core 2 Extreme QX9770	59,455 MIPS at 3.2 GHz	18.580 MIPS/MHz	2008
Intel Core i7 Extreme 965EE	76,383 MIPS at 3.2 GHz	23.860 MIPS/MHz	2008
AMD Phenom II X4 940 Black Edition	42,820 MIPS at 3.0 GHz	14.273 MIPS/MHz	2009

MIPS (**M**illion **I**nstruc-tions **P**er **S**econd)

Nel 2006, era ragionevole prendere come riferimento prudenziale **20.000 MIPS**

20.000 MIPS = ***20 miliardi*** di operazioni al secondo

MIPS e PROCESSORI: anni 2020

Per valutare praticamente l'impatto di queste funzioni, facciamoci un'idea di *quante operazioni al secondo* possa gestire un processore:

Intel Core i7 5960X	298,190 MIPS at 3.5 GHz	85.2	10.65	
Raspberry Pi 2	4,744 MIPS at 1.0 GHz	4.744	1.186	
Intel Core i7 6950X	320,440 MIPS at 3.5 GHz	91.55	9.16	2016
ARM Cortex A73 (4-core)	71,120 MIPS at 2.8 GHz	25.4	6.35	2016
ARM Cortex A75	?	?	8.2-9.5	2017
ARM Cortex A76	?	?		
ARM Cortex A77	?	?		
ARM Cortex A78	?	?		
AMD Ryzen 7 1800X	304,510 MIPS at 3.7 GHz	82.3	10.29	2017
Intel Core i7-8086K	221,720 MIPS at 5.0 GHz	44.34	7.39	
Intel Core i9-9900K	412,090 MIPS at 4.7 GHz	87.68	10.96	2018
AMD Ryzen 9 3950X	749,070 MIPS at 4.6 GHz	162.84	10.18	2019
AMD Ryzen Threadripper 3990X	2,356,230 MIPS at 4.35 GHz	541.66	8.46	2020

10 anni dopo, c'è chi arriva a 16x volte

Se vogliamo stare nel mid-range, possiamo assumere come riferimento 300.000 MIPS

15 anni dopo, beh..

MIPS/MHz

MIPS/MHz/core

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}	$>> 10^{2500}$

- 2006: eseguendo 20 miliardi di operazioni/sec, un algoritmo il cui tempo sia dell'ordine di $\mathbf{2^N}$ richiede:
  - N=20 $t = 2^{20} / 20*10^9 = 52\ \mu s$
  - N=30 $t = 2^{30} / 20*10^9 = 53\ ms$
  - N=50 $t = 2^{50} / 20*10^9 = 5.6*10^4\ s = 15$ ore
  - N=60 $t = 2^{60} / 20*10^9 = 5.7*10^7\ s = 667$ giorni
  - N=70 $t = 2^{70} / 20*10^9 = 5.9*10^{10}\ s = 1871$ anni

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}	$>> 10^{2500}$

- 2020: eseguendo 300 miliardi di operazioni/sec, un algoritmo il cui tempo sia dell'ordine di 2^N richiede:
  - N=20 t= 2^{20} / 300*10^9= 3.5 µs
  - N=30 t= 2^{30} / 300*10^9= 3.5 ms
  - N=50 t= 2^{50} / 300*10^9= 3733 s = 1 ora
  - N=60 t= 2^{60} / 300*10^9= $3.8*10^6$ s = 44 giorni
  - N=70 t= 2^{70} / 300*10^9= $3.9*10^9$ s = 124 anni

In 15 anni, la «infattibilità» si è spostata solo da N=60 a N=70

COMPORTAMENTO ASINTOTICO

PROBLEMA:

- individuare con esattezza l'espressione di $time_A(N)$ è spesso *molto difficile*
- D'altronde, interessa capire cosa succede *quando i dati sono di grandi dimensioni*
  - con N piccolo, qualunque algoritmo alla fine va bene
  - *è con N grande che la situazione può diventare critica* (in particolare: per $N \to \infty$)
- Per questo interessa il *comportamento asintotico* della funzione $time_A(N)$.

COMPORTAMENTO ASINTOTICO: ANDAMENTO

- Anche individuare il comportamento asintotico di $time_A(N)$, però, *spesso non è semplice*
- D'altronde, non interessa l'espressione esatta, ma una *indicazione del suo andamento*
  - 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

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

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

    e si scrive $\text{time}(N) = O(g(N))$

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

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

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

NOTAZIONI ASINTOTICHE

- Limite superiore
  una funzione (n
  - quando esiston

    time (N) < a g(N) + b ∀N > N'

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

La funzione g(N) costituisce un *limite superiore* al costo dell'algoritmo (time(N)).

- Limite inferiore a
  una funzione (n
  - quando esistono

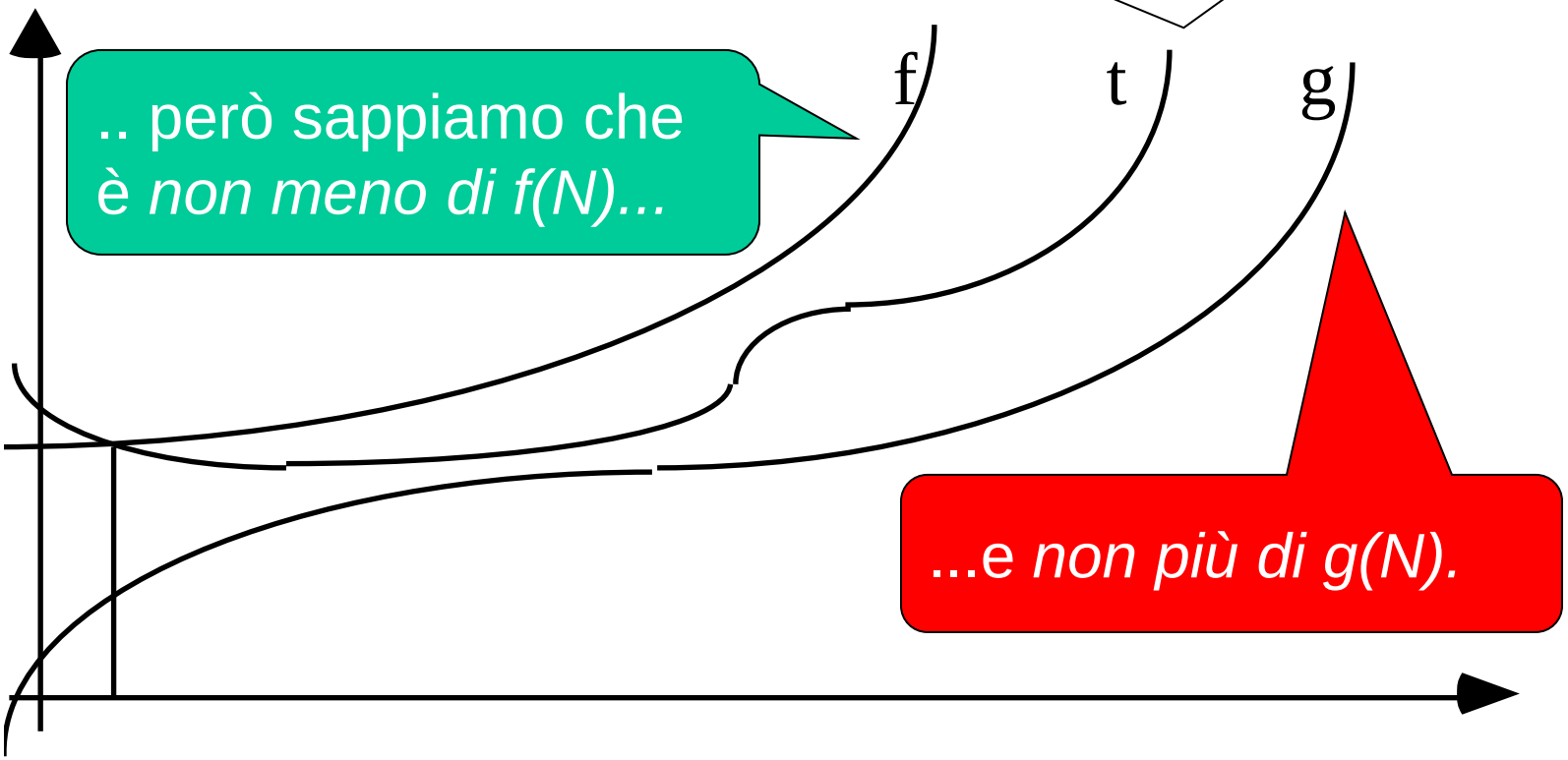
    time (N) > c f(N) ∀N > N'

  e si scrive time(N) = Ω (f(N))

La funzione f(N) costituisce un *limite inferiore* al costo dell'algoritmo (time(N)).

INTERPRETAZIONE GRAFICA

Magari non sapremo esattamente come è fatta **time(N)** per $N \to \infty$ ...



UN CASO PARTICOLARE

Un caso particolare, ma di grande interesse, è quando le due funzioni g(N) e f(N) coincidono:

- 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 time(N)
  - da notare che *non interessano i fattori di scala*, ma solo l'andamento.

ESEMPIO

- Si supponga che per un certo algoritmo sia

$$time_A(N) = 3*N^2 + 4*N + 3$$

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

  La funzione **f(N)= N^2** costituisce perciò una *valutazione esatta* di questo algoritmo.
  - non interessa che il fattore di scala sia 3, 4, o altro

CLASSI DI COMPLESSITÀ

- Le notazioni O e Ω consentono di *definire diverse classi di complessità:*
  - costante — *qualsiasi costante (esempio: 1)*
  - sotto-lineare — $\log N$ oppure N^k con $k<1$
  - lineare — N
  - sovra-lineare — $N*\log N$
  - polinomiale — N^k con $k>1$
  - esponenziale — c^N oppure N^N
- In questo modo, diventa più immediato *confrontare diversi algoritmi* per capire se sono *"della stessa complessità"* o se uno sia “migliore” 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.

Dalla complessità di un algoritmo alla complessità di un problema

COMPLESSITÀ DI UN PROBLEMA

- Finora ci siamo interessati alla complessità di un *singolo algoritmo* per un dato problema.
- Ora interessa capire *se il problema in quanto tale* abbia una sua complessità, cioè se sia *intrinsecamente facile* o *intrinsecamente difficile* *indipendentemente dallo specifico algoritmo* usato
  - tale algoritmo potrebbe essere già noto..
  - …ma anche *ancora da inventare!*

DEFINIZIONE

Diremo allora che un problema ha:

- *delimitazione superiore* $O(g(N))$ alla sua complessità se *esiste* *almeno un algoritmo* che lo risolve ha complessità $O(g(N))$.
- *delimitazione inferiore* $\Omega(f(N))$ alla sua complessità se *ogni algoritmo* che lo risolve è di complessità *almeno* $\Omega(f(N))$.

DEFINIZIONE

Questo equivale a dire che *il problema non può essere più complesso* di O(g(N)), dato che esiste un algoritmo che lo risolve con tale complessità.

Però potrebbe essere più semplice, in quanto potremmo non aver (ancora) trovato l'algoritmo migliore.

Al contrario, per dire che *il problema in quanto tale è di complessità Ω(f(N))* bisogna dimostrare che *non può esistere un algoritmo migliore:*

ovvero, che qualunque algoritmo che possiamo inventare avrà di certo *almeno* quella complessità.

CLASSI DI PROBLEMI

Diremo che un problema è *trattabile* se la sua complessità è:

- *lineare,* se ogni algoritmo che lo risolve ha delimitazioni di complessità $O(N)$ e $\Omega(N)$
- *polinomiale,* se ogni algoritmo risolvente ha delimitazioni di complessità $O(N^k)$ e $\Omega(N^k)$

Diremo invece che è *intrattabile* se la sua complessità è *esponenziale,* ossia se non esistono algoritmi di complessità polinomiale che lo risolvono (es. problema del commesso viaggiatore).

ALGORITMI OTTIMALI PER UN DATO PROBLEMA

Il concetto di *complessità di un problema* permette di definire la nozione di *algoritmo ottimale per un problema* nei seguenti termini:

- l'algoritmo stesso ha complessità O(f(N))
- la delimitazione inferiore alla complessità del problema è proprio $\Omega(f(N))$.

  Infatti, se il problema in quanto tale ha complessità $\Omega(f(N))$, nessun algoritmo potrà fare di meglio, anche in futuro; ergo, se l'algoritmo considerato ha appunto tale complessità, fa già quanto di meglio si possa fare e dunque è ottimo.

Valutazione di complessità
Istruzioni dominanti

VALUTAZIONI DI COMPLESSITÀ

- Poiché valutare con precisione il costo delle singole istruzioni è impossibile, si introduce il 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

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

  - dove d(N) indica *quante volte* viene eseguita *l'istruzione dominante*.
- L'idea è che *l'istruzione dominante caratterizzi l'algoritmo* e sia eseguita un numero di volte *proporzionale* alla sua complessità: *t(N) = O(d(N))*

ISTRUZIONI DOMINANTI

- Ma come si riconoscono – o come si scelgono – le istruzioni dominanti?
  - l'istruzione dominante è per definizione tale se *caratterizza l'essenza stessa* dell'algoritmo
  - occorre dunque *valutare caso per caso* quale sia il tipo di istruzione che condiziona il conseguimento dell'obiettivo, che "guida" l'algoritmo.
- Ad esempio, negli algoritmi di ordinamento e di ricerca di elementi in strutture dati, l'istruzione dominante è di solito 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<v.length){
     if (el == v[i])
       trovato = true;
     i++;
   }
   return trovato;
}
```

istruzioni dominanti

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

Valutazione di complessità
Dipendenza dai dati d'ingresso

DIPENDENZA DAI DATI DI INGRESSO

- Spesso il costo di un algoritmo non dipende solo dalla *dimensione* dei dati di ingresso, ma anche dai *particolari valori* di quei dati
  - 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.
- Occorre perciò distinguere fra diversi casi.

CASO PEGGIORE vs. CASO MEDIO

- In base al valore dei dati, si distingue fra:
  - caso *migliore*
  - caso *peggiore*
  - caso *medio*
- Solitamente si considera il *caso peggiore*
- Tuttavia, poiché esso – per fortuna – è spesso anche raro, si considera anche il *caso medio*
- Il caso medio si valuta di solito considerando tutte le situazioni come *equiprobabili.*

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 → un solo confronto
- *Caso peggiore*: l'elemento è l'ultimo o non è presente → 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)

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

Valutazione di complessità in algoritmi di ordinamento

ALGORITMI DI ORDINAMENTO

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

PRINCIPALI ALGORITMI

- Considereremo gli algoritmi di ordinamento:
  - naïve sort (semplice, intuitivo, poco efficiente)
  - bubble sort (semplice, un po' più efficiente)
  - insert sort (intuitivo, abbastanza efficiente)
  - shell sort (non intuitivo, abbastanza efficiente)
  - quick sort (non intuitivo, alquanto efficiente)
  - merge sort (non intuitivo, molto efficiente)
- Per valutarne la complessità, assumeremo come istruzione dominante il *confronto fra elementi* dell'array.

NAÏVE SORT (1/3)

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

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

```
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 (2/3)

Codifica

```
void naiveSort(int[] v){
 int n = v.length;
 while (n>1) {
   int p = trovaPosMax(v,n);
   if (p<n-1) { // scambio
     int t=v[p]; v[p]=v[n-1]; v[n-1]=t;
   }
   n--;
 }
}
```

funzione ausiliaria

La dimensione logica dell'array cala di uno a ogni ciclo, poiché dopo ogni iterazione l'ultima cella è certamente ordinata.

NAÏVE SORT (3/3)

Codifica

```
int trovaPosMax(int[] v){
   int posMax=0;
   for (int i=1; i<v.length; 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

Valutazione di complessità

- Il numero di *confronti* è fisso (non dipende dai dati d'ingresso) e vale:

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

- Dunque, l'algoritmo *fa gli stessi confronti in tutti i casi*, sia che l'array sia totalmente disordinato, sia che sia perfino *già ordinato!!*
- Nel caso peggiore, questo è anche il numero di scambi necessari (in generale saranno meno)

BUBBLE SORT (1/3)

- 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 (2/3)

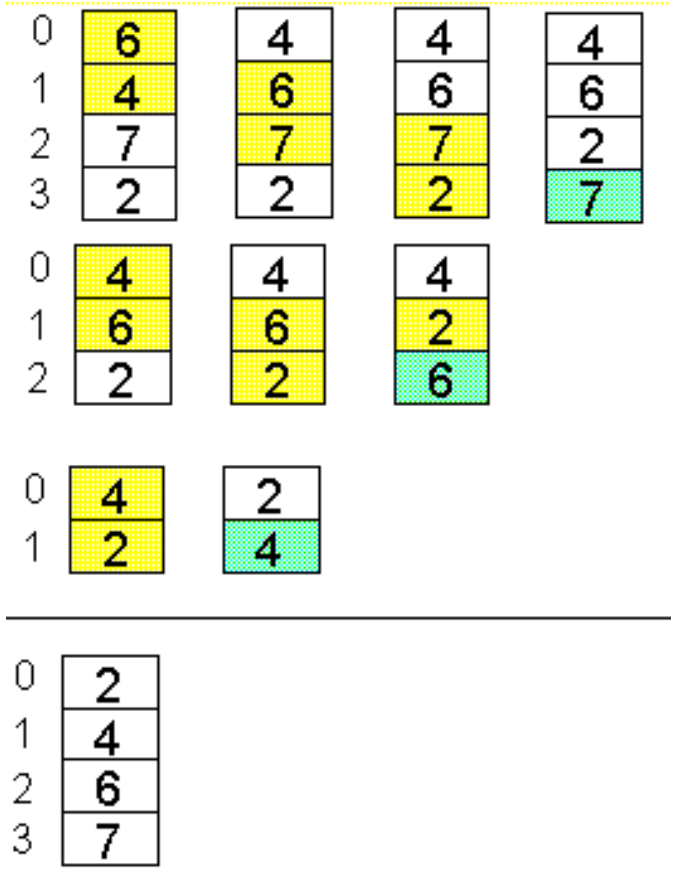
Codifica

```
void bubbleSort(int[] v){
  boolean ordinato = false;
  int n = v.length;
  while (n>1 && !ordinato){
    ordinato = true;
    for (int i=0; i<n-1; i++)
      if (v[i]>v[i+1]) {
       {int t=v[p]; v[p]=v[n-1]; v[n-1]=t;}
       ordinato = false; }
    n--;
 }
}
```

Continua solo se l'array non è ancora ordinato.

BUBBLE SORT (3/3)

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: VALUTAZIONE

Valutazione di complessità

- Caso peggiore: numero di confronti identico al-l'algoritmo 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 in numero compreso fra N-1 e $N^2/2$, a seconda dei dati di ingresso.

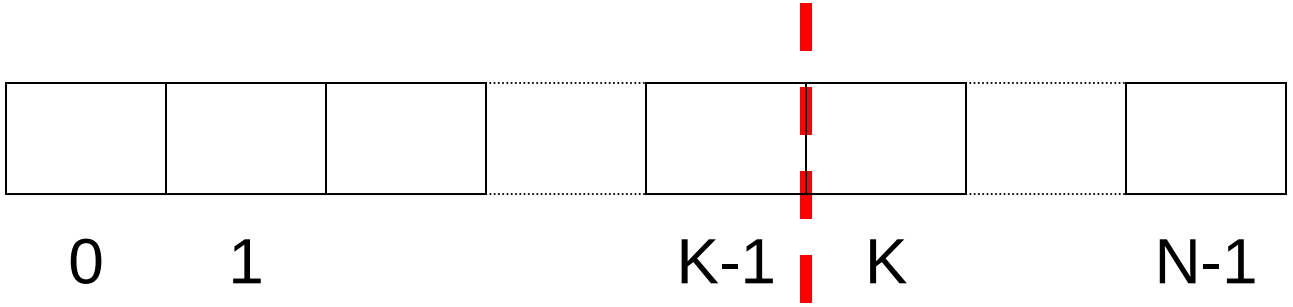
INSERT SORT (1/8)

- 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*, perché le stesse operazioni possono essere svolte direttamente sull'array originale, che così alla fine risulterà ordinato.

INSERT SORT (2/8)

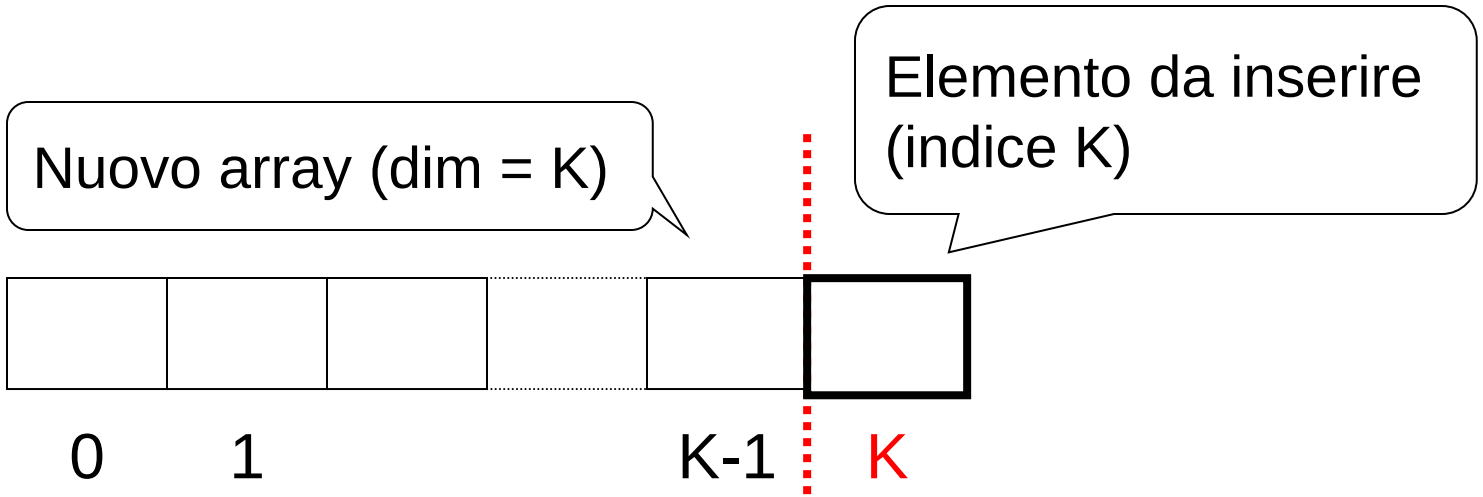
Scelta di progetto

- “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 (3/8)

- Per costruzione, *in ogni istante* l'*elemento da inserire è il primo del vecchio array, che si trova nella cella (K+1)-esima* (di indice K)



INSERT SORT (4/8)

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

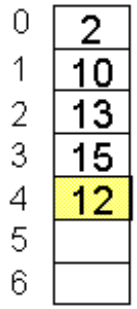
```
void insertSort(int[] v, int n){
  for (int k=1; k<n; k++)
     insMinore(v,k);
}
```

funzione ausiliaria

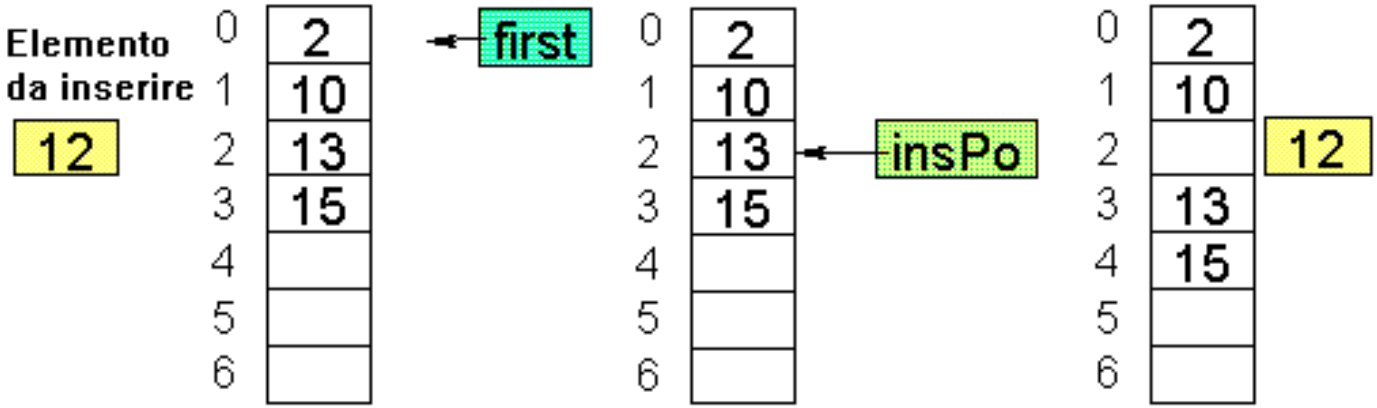
Al passo *k*, la demarcazione fra i due array (nuovo e vecchio) è alla posizione k.

INSERT SORT (5/8)

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 (6/8)

Specifica di insMinore()

```
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 (7/8)

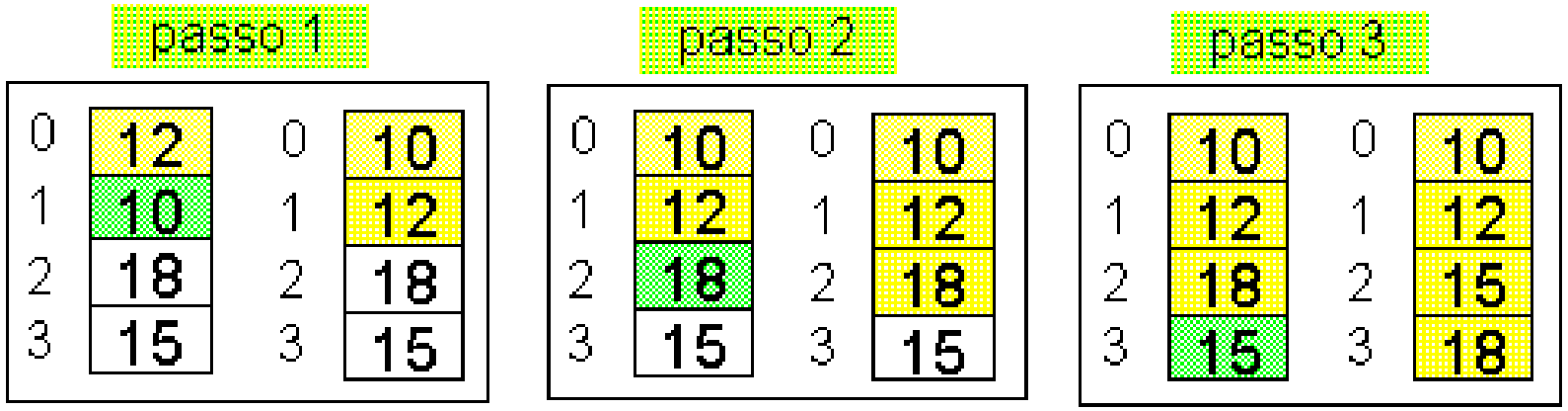
Codifica di insMinore()

```
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 (8/8)

Esempio



INSERT SORT: VALUTAZIONE

Valutazione di complessità

- Nel caso peggiore (array ordinato a rovescio), richiede 1+2+…+(N-1) *confronti e spostamenti*
  → $O(N^2/2)$
- 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*
  → 1/2+2/2+…+(N-1)/2 *confronti e spostamenti*
  → $O(N^2/4)$

INSERT SORT vs BUBBLE SORT

- Nel caso medio ($O(N^2/4)$), *l'insert sort è circa il doppio più efficiente del bubble sort* ($O(N^2/2)$), pur rimanendo un algoritmo *quadratico.*
  - l'esperienza indica che l'insert sort va bene per ordinare insiemi di al più qualche migliaio di elementi
  - non va usato se gli elementi da ordinare sono più di qualche migliaio, o se si devono ordinare ripetutamente insiemi di centinaia di elementi

SHELL SORT (1/5)

- Lo Shell sort (Donald Shell, 1959) è una variante del bubble sort , che però opera con *coppie di elementi non adiacenti,* ma *a distanza "gap"*
  - all'inizio, *gap* è metà della dimensione dell'array
  - poi viene ridotta per dimezzamenti successivi.
  - in caso di scambio, *i confronti vengono retro-propagati* e possono generare nuovi scambi.
- Deve la sua fama al fatto di essere stato *il primo a scendere sotto il limite di complessità quadratica*
  - migliora l'insert sort, ma è molto più complicato
  - poco usato dopo l'invenzione di quicksort

SHELL SORT (2/5)

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 (3/5)

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 (4/5)

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

SHELL SORT: VALUTAZIONE

Valutazione di complessità

- Difficile da calcolare in dettaglio: è stato stimato un *caso medio pari a* $O(N^{1.7})$
- intuitivamente, migliora l'insert sort perché riesce a *portare avanti più velocemente* gli elementi verso la destinazione

SHELL SORT vs INSERT SORT & BUBBLE SORT

- Lo shell short è il migliore della famiglia degli algoritmi "quadratici"
- Nel caso medio, si è rivelato *circa il doppio più efficiente dell'insert sort* (il suo concorrente diretto) e *circa 5 volte più veloce del bubble sort*
  - l'esperienza dice che dà il meglio di sé su insiemi fino ad alcune migliaia di elementi (es. 5000), o per ordinamenti ripetuti di insiemi di minori dimensioni
  - non adatto a insiemi (molto) grandi.

QUICK SORT (1/13)

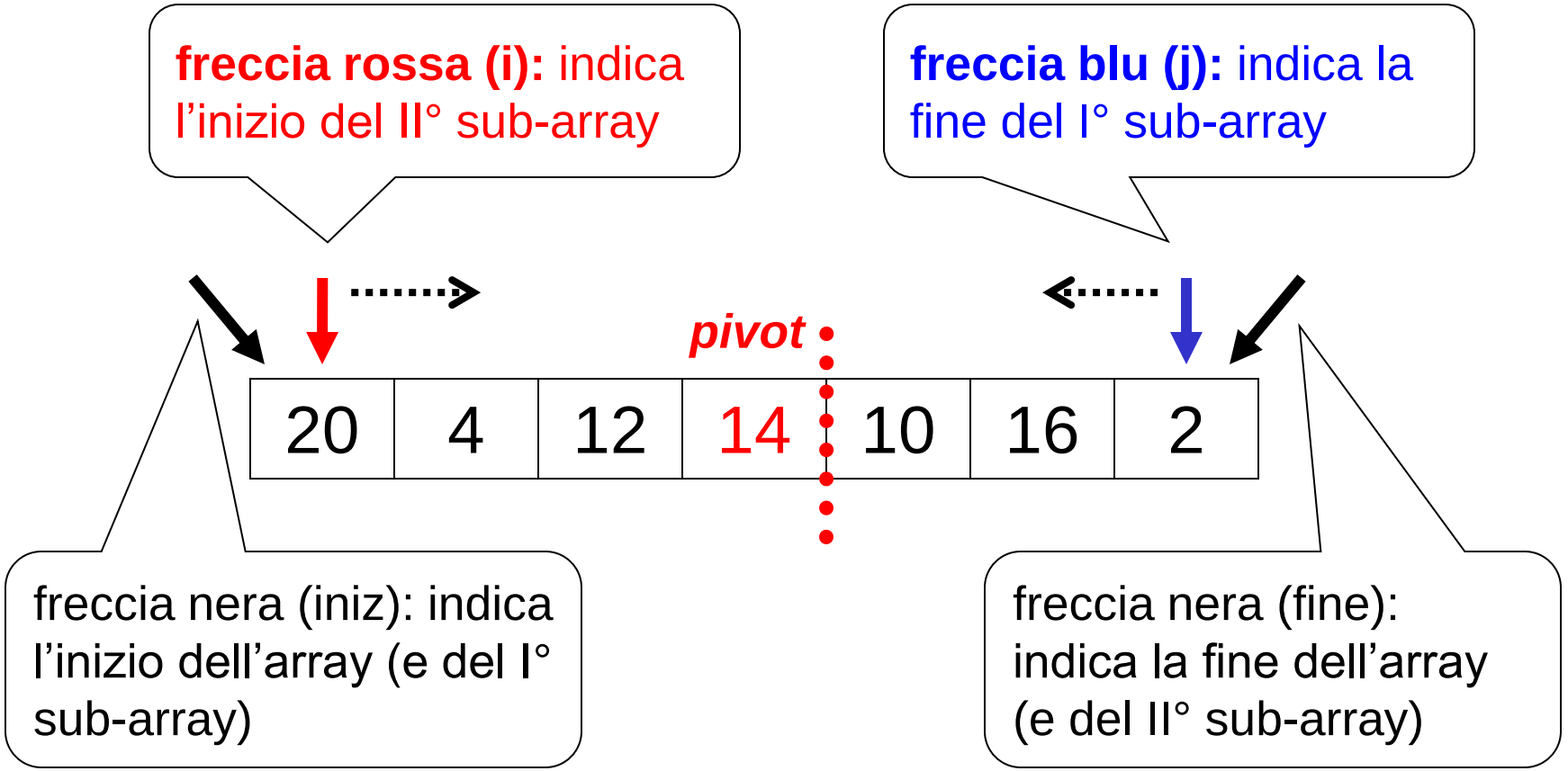
- Idea base: *ordinare un array corto è molto meno costoso che ordinarne uno lungo*.
- Conseguenza: può essere utile *partizionare l'array in due parti, ordinarle separatamente* (ricorsione!) e infine *fonderle insieme.*
- In pratica:
  - si suddivide l'array in due "sub-array", delimi-tati da un elemento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

QUICK SORT (2/13)

- L'operazione base è il *partizionamento dell'array nei due sub-array*
  - per farlo, si cerca nel primo sub-array un elemento *maggiore* del pivot, e nel secondo array un elemento minore: indi, questi due elementi vengono *scambiati*

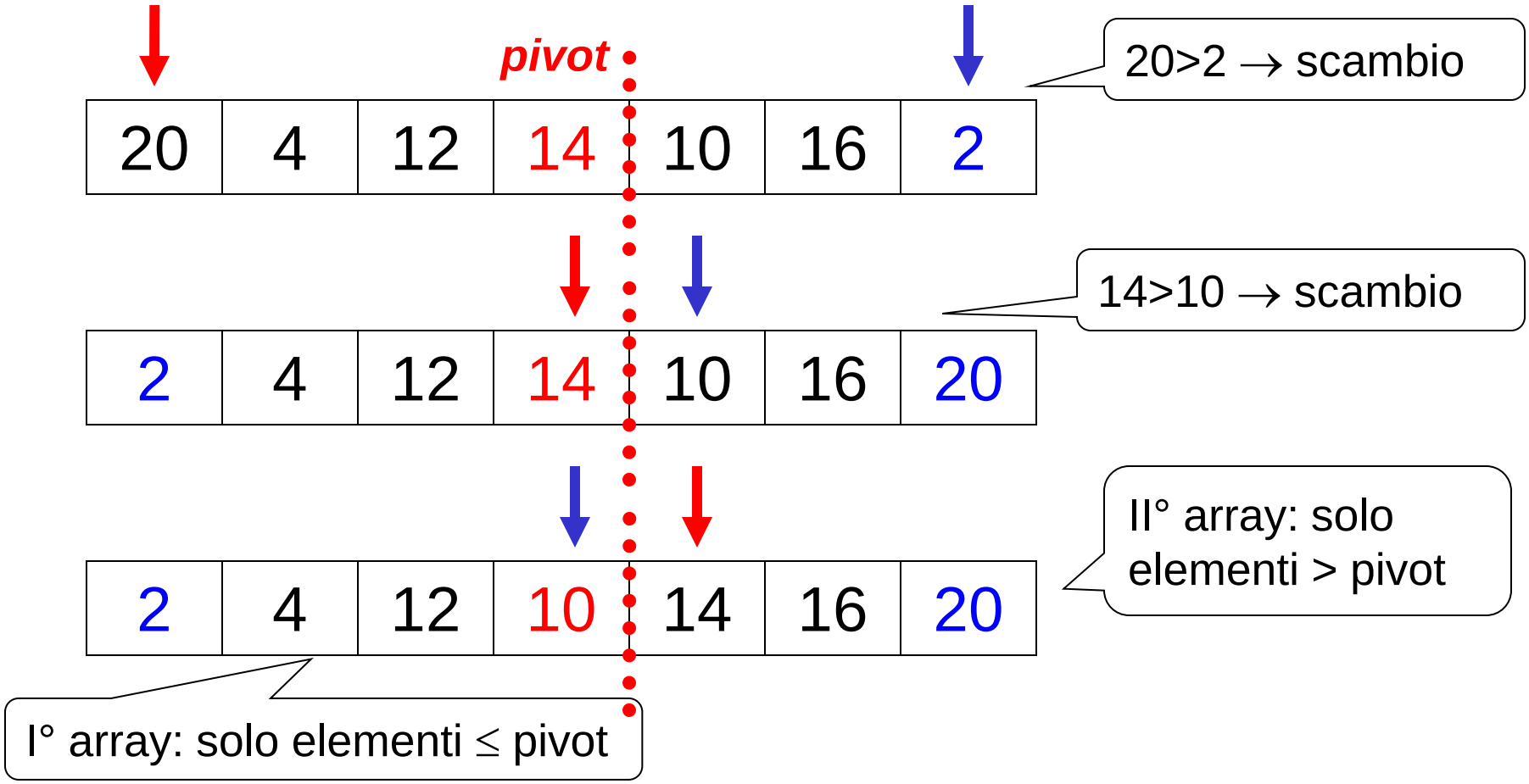
QUICK SORT (3/13)

Esempio: *legenda*



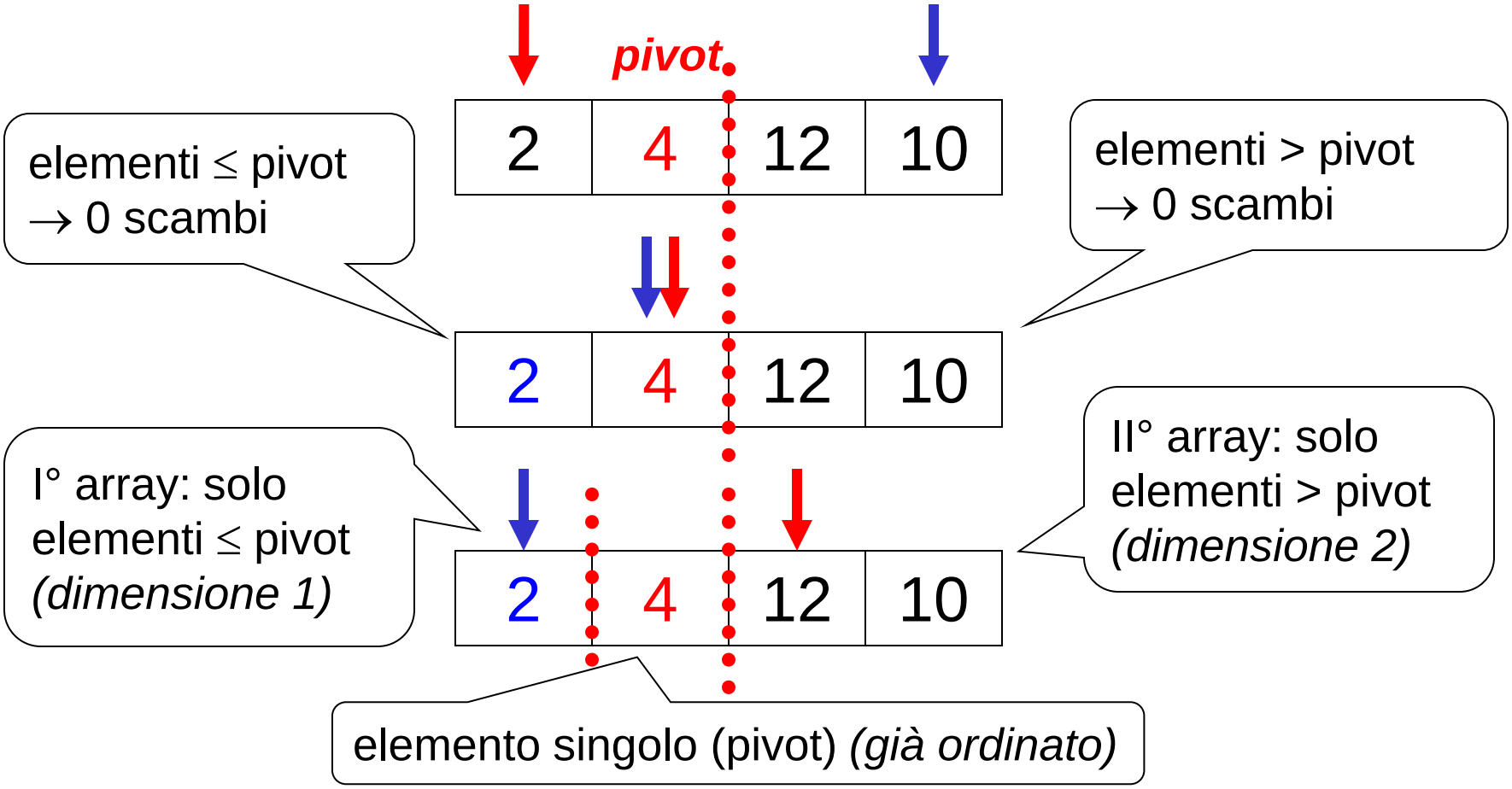
QUICK SORT (4/13)

Esempio (ipotesi: si sceglie 14 come pivot)



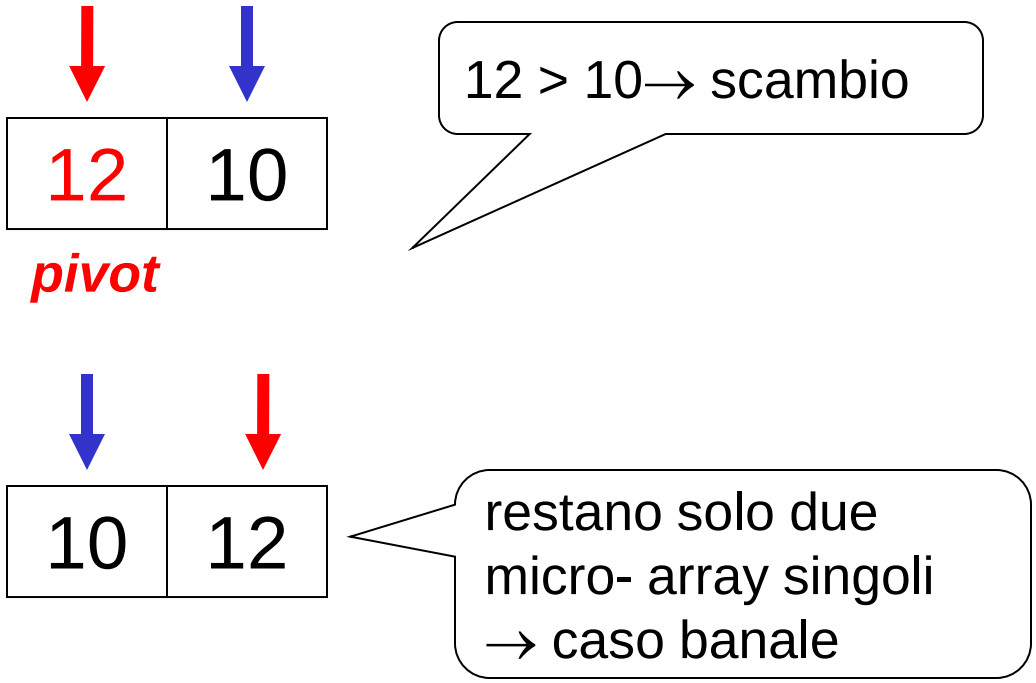
QUICK SORT (5/13)

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



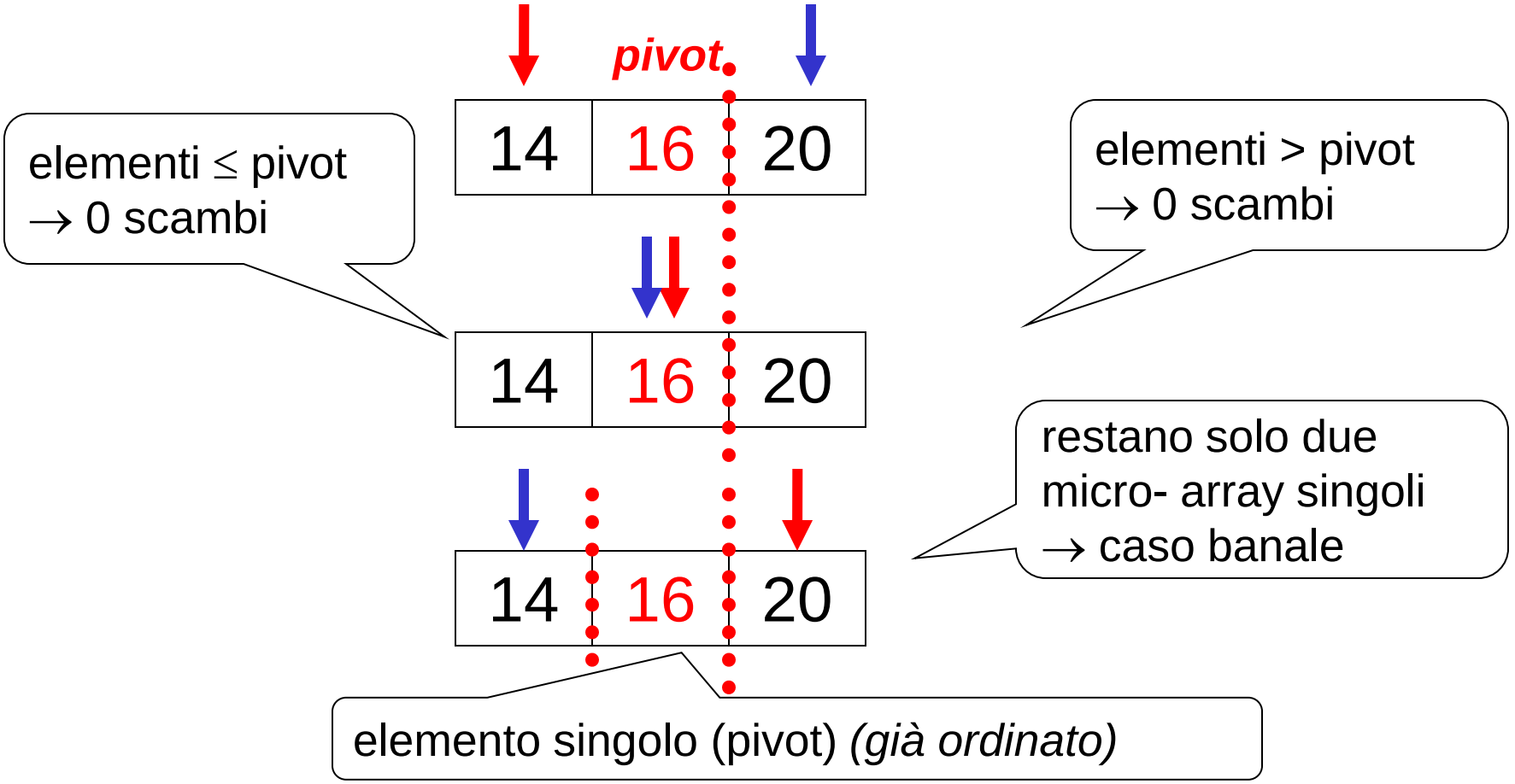
QUICK SORT (6/13)

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



QUICK SORT (7/13)

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



QUICK SORT (8/13)

Specifica

```
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 (9/13)

Codifica

```
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 (10/13)

Codifica

```
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 >

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

QUICK SORT (11/13)

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], &v[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 (12/13)

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 $L = N/2^k$
  - perciò 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 (13/13)

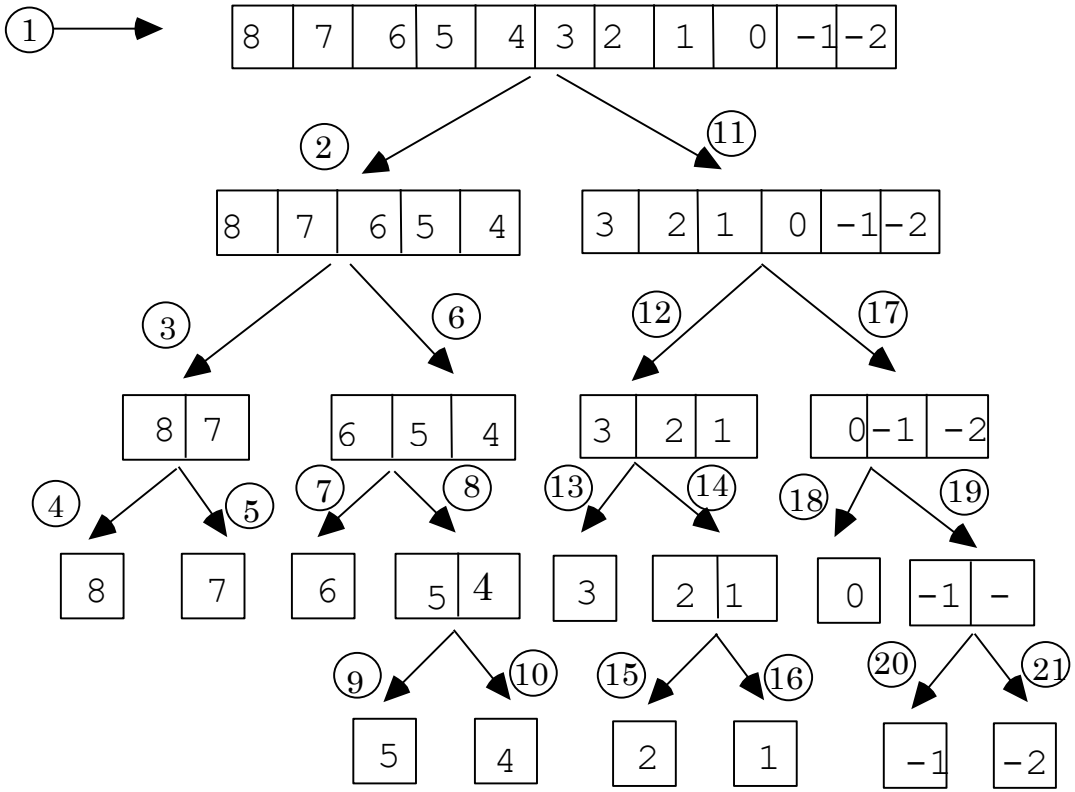
- Si dimostra che $O(N \log_2 N)$ è il *limite inferiore* di complessità del *problema* di ordinamento (sequenziale) di un array
  - dunque, *nessun algoritmo sequenziale, presente o futuro, potrà mai far meglio* di $O(N \log_2 N)$
  - per andare oltre servono *algoritmi paralleli* operanti su *più processori*
- Ma se il pivot non è scelto bene?
  - quicksort garantisce O(NlogN) *solo se i due sub-array hanno egual dimensione*, che però per fortuna è anche *la situazione più probabile* nel caso medio
  - per raggiungere sempre tale risultato → *Merge Sort*

MERGE SORT (1/5)

- 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 (2/5)

Esempio



MERGE SORT (3/5)

Specifica

```
void mergeSort(int v[], int iniz, int fine,
               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>
  }
}
```

mergeSort() si limita a suddividere l'array: è qui che si svolge davvero il lavoro.

MERGE SORT (4/5)

Codifica

```
void mergeSort(int v[], int iniz, int fine,
                        int vout[]) {
  int mid;
  if ( first < last ) {
     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 (5/5)

Codifica di merge()

```
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

Caso medio:

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

$N^2/2$

$N \div N^2/2$

$N^2/4$

$\geq N \log N$

$N \log N$

- Per problemi semplici, gli algoritmi “poco sofisticati” funzionano abbastanza bene, *a volte meglio degli altri*
- Quando però il problema si fa complesso, *la differenza diventa evidente.*

ALGORITMI PARALLELI

- Finora abbiamo considerato sempre *algoritmi sequenziali,* operanti su un unico processore
- Se però si dispone di *più processori* operanti *in parallelo,* si aprono nuove possibilità
- La complessità si valuta con due *nuovi concetti*:
  - Speed-up: misura quanto un algoritmo parallelo è più veloce del *miglior algoritmo sequenziale noto* per lo stesso problema.
  - Efficienza (Eff): dà una *misura percentuale* della bontà dell'algoritmo parallelo rispetto al *miglior algoritmo sequenziale noto* per lo stesso problema.

SPEED UP

- Lo Speed-up (Su) è definito come

$$Su = Ts / Tp \quad \leq \mathbf{P}$$

dove:

  - Ts = tempo del miglior algoritmo sequenziale noto nel caso peggiore
  - Tp = tempo dell'algoritmo parallelo con **P processori**

- Nel caso ideale, Su = P, poiché P processori possono migliorare la velocità al più di P volte (Tp =Ts/P)
- In generale, l'algoritmo parallelo non sarà P volte più veloce, ma un po' meno → Tp > Ts/P → Su < P
- Il caso Su>P è di scarso interesse, poiché indica che l'algoritmo seq.le considerato è inefficiente e se ne può trovare uno migliore semplicemente svolgendo in serie i passi dell'algoritmo parallelo.

SPEED UP (2)

Lo Speed-up si dice:

- ideale (o lineare) quando *Su = P.*
- assoluto quando Ts fa riferimento a un *algoritmo sequenziale ottimo.*
- relativo quando Ts è il tempo di esecuzione dell'*algoritmo parallelo su un solo processore.*

EFFICIENZA

- L'Efficienza (Eff) è definita come

$$Eff = Su / P \quad \leq 1$$

dove Su è lo Speed-up definito prima.

Essa misura quanto tempo-processore è speso per risolvere il problema rispetto a quello speso per gestire la comunicazione e la sincronizzazione.

- Eff=1 per algoritmi con speedup lineare (o che girano su un singolo processore)
- Eff<<1 per algoritmi difficili da parallelizzare (spesso Eff=1/log P, dunque Eff $\rightarrow$0 quando P aumenta…)

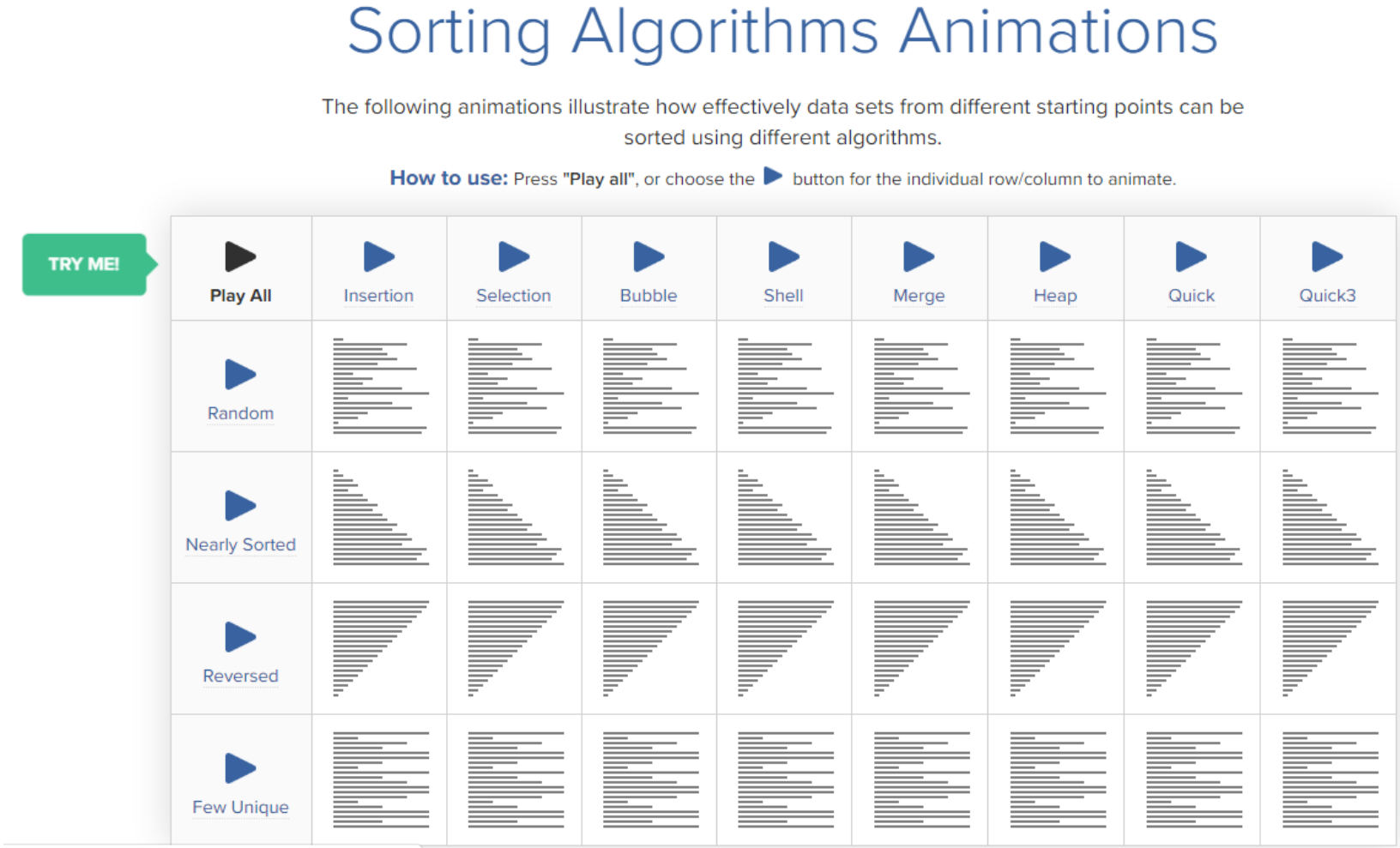
ALCUNI ALGORITMI PARALLELI

Problemi di sort → Ts = N log N

- *Odd-Even Transposition Sort*
  - complessità nel caso peggiore = O(N)
  - Speed up = (N log N) / N = O(log N)
  - Efficienza = O((log N)/N)
- *Shear Sort*
  - complessità nel caso peggiore = $O(N^{1/2} \log N)$
  - Speed up = $(N \log N)/(N^{1/2} \log N) = O(N^{1/2})$
  - Efficienza = $O(N^{-1/2}) = O(1/\sqrt{N})$

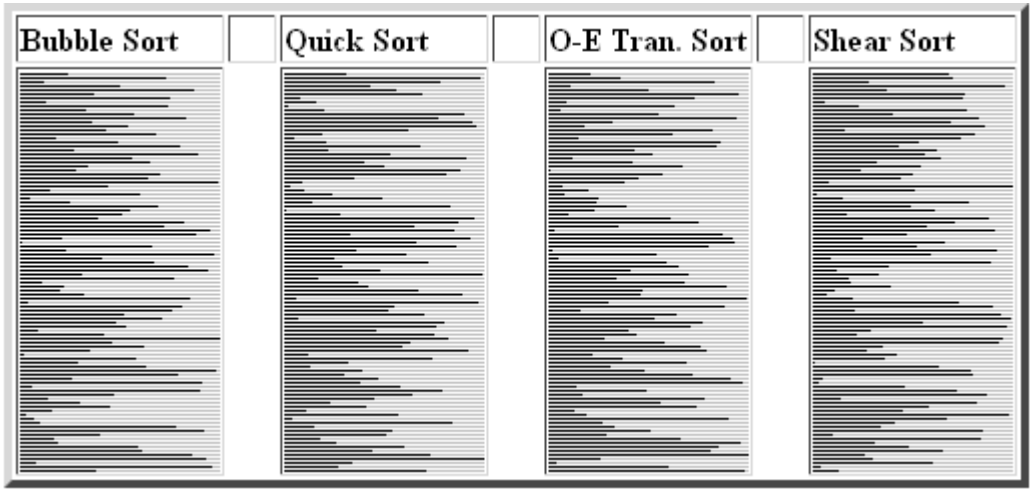
ESPERIMENTO ANIMATO

www.sorting-algorithms.com

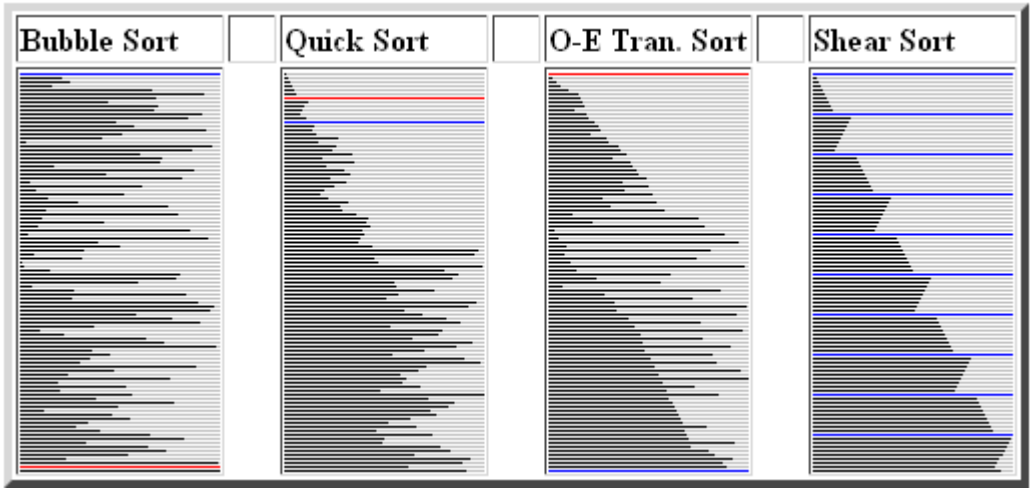


ESPERIMENTO: INIZIO

www.sorting-algorithms.com



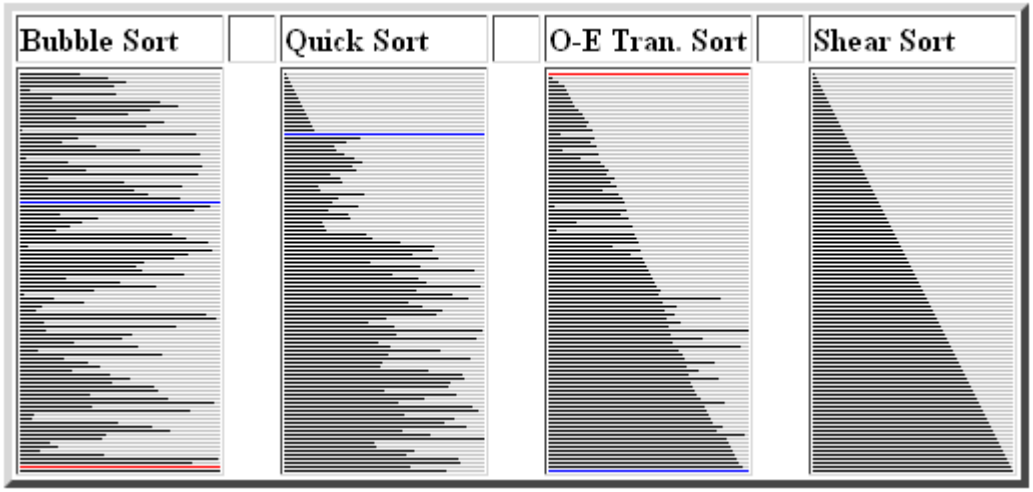
Inizio



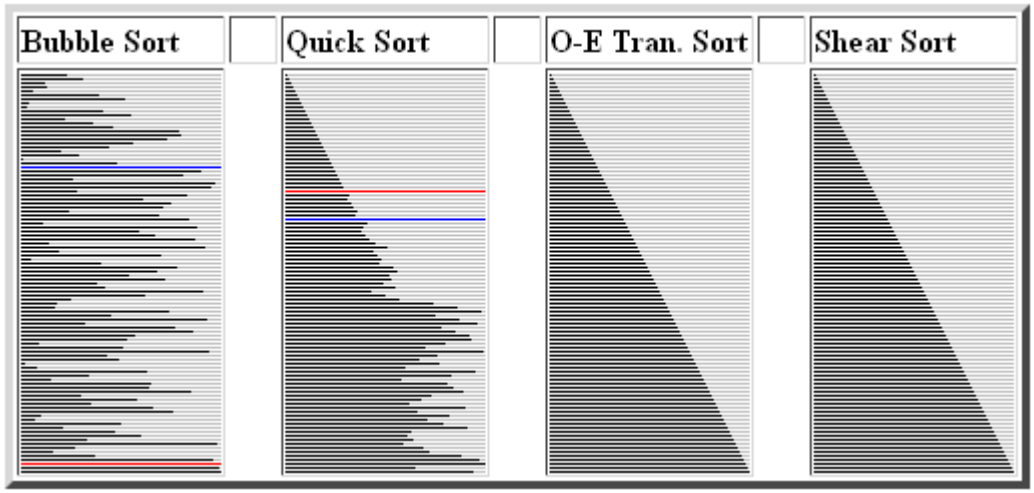
Dopo 4 s

ESPERIMENTO: DOPO UN PO'...

www.sorting-algorithms.com



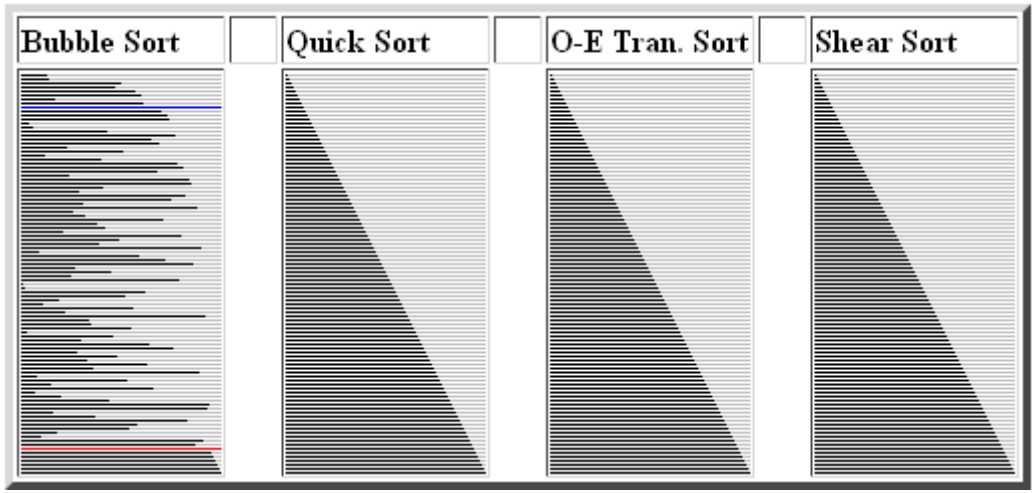
Dopo 6 s



Dopo 8 s

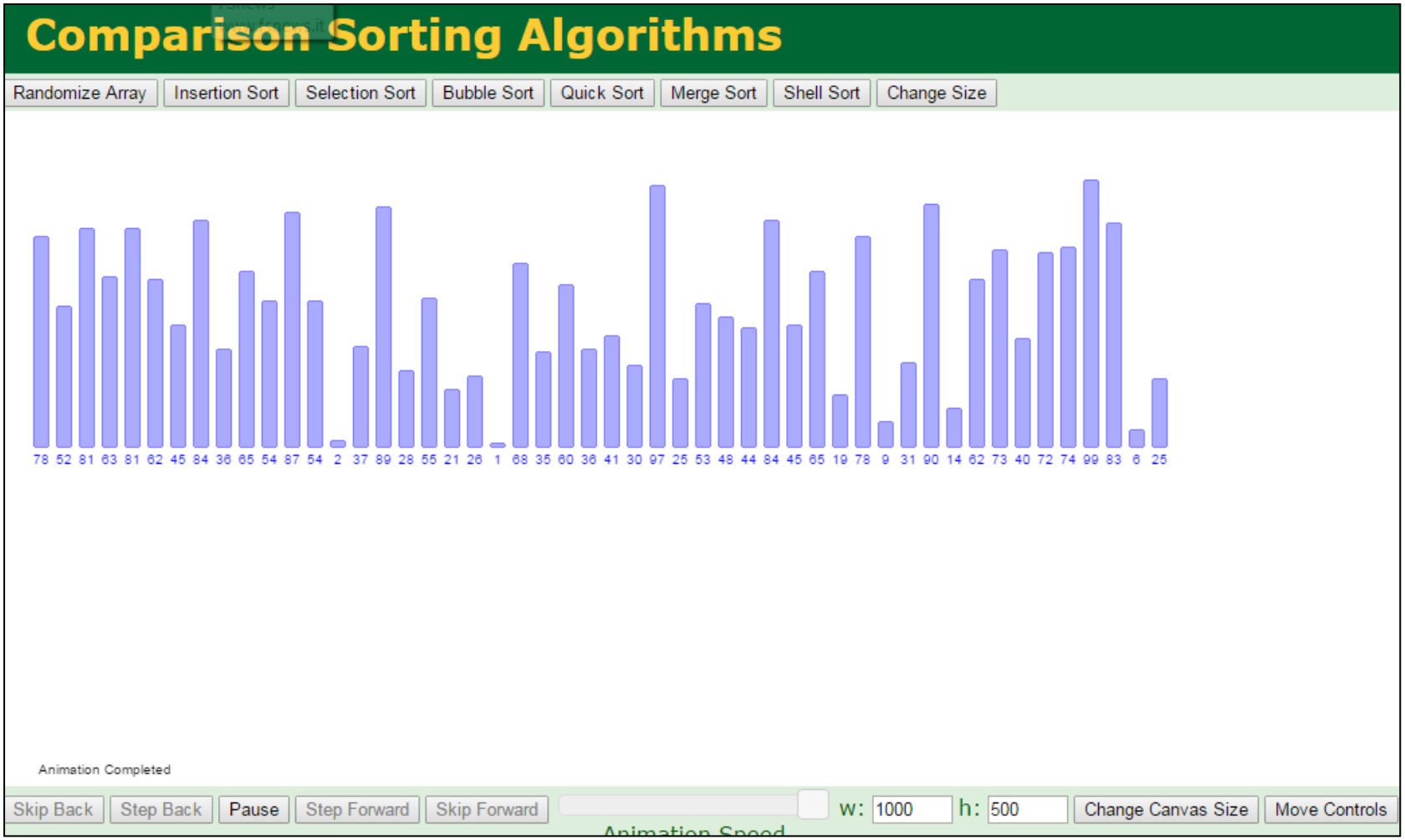
ESPERIMENTO: DOPO 25 sec

www.sorting-algorithms.com



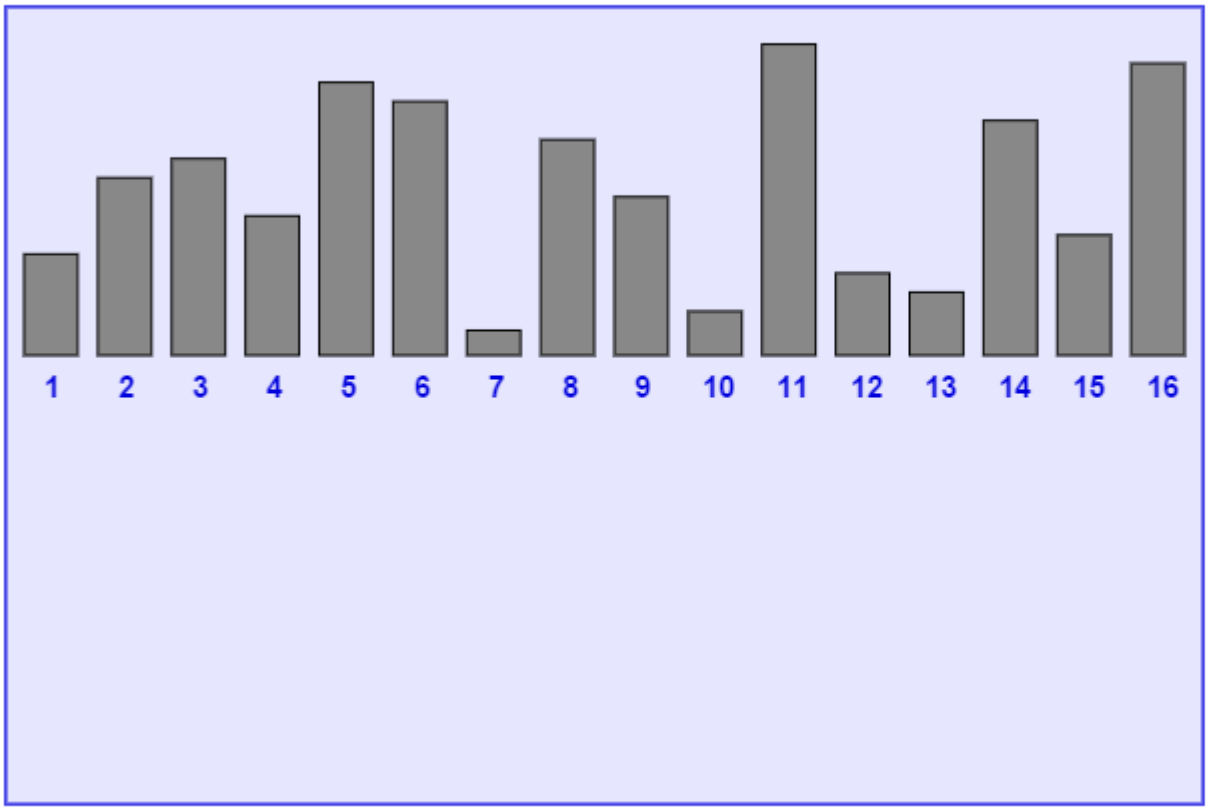
Quanto al bubble.. aspettiamo.. aspettiamo.. qualche minuto ☺

https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html



http://math.hws.edu/eck/jsdemo/sortlab.html

HTML5 Canvas Demo: Sorting Algorithms



Selection Sort ▾

New Sort

Fast

Run

Pause

Step

Comparisons: 0

Copies: 0

Click "Run" or "Step" to begin sorting.

Valutazione di complessità in algoritmi di ricerca

ALGORITMI DI RICERCA

Per cercare un elemento in un array:

- se non vi sono ipotesi sull'esistenza di una qualche forma di ordinamento, occorre *controllare tutti gli elementi uno ad uno*
  - cioè fino a che non si trova l'elemento, o l'array è finito
- *se però l'array è ordinato* la ricerca può essere fatta in modo *molto più efficiente, sfruttando l'ordinamento per "andare a colpo sicuro"* a reperire l'elemento richiesto
  - o appurare la sua assenza

RICERCA BINARIA (1)

- L'algoritmo emula ciò che si fa quando si cerca, a mano, una parola in un dizionario:

  si apre il dizionario "circa alla posizione"
  in cui dovrebbe trovarsi la parola da cercare

- Si confronta l'elemento da cercare con *quello di posizione mediana* nell'array
  - così, o l'elemento viene trovato subito..
  - ..oppure si sa *dove continuare la ricerca*
    - a sinistra (fra gli elementi minori), se l'elemento mediano è maggiore di quello richiesto
    - a destra (fra gli elementi maggiori) in caso contrario.

RICERCA BINARIA (2)

Dunque:

1) si confronta l'elemento da cercare con *quello di posizione mediana*

2) se è l'elemento cercato, la ricerca si conclude con successo
   - altrimenti, la ricerca prosegue
     - *nella metà di sinistra* dell'array se l'elemento mediano è maggiore di quello richiesto
     - *nella metà di destra* dell'array se l'elemento mediano è minore di quello richiesto.

ANALISI DI COMPLESSITÀ

Valutazione di complessità

- A ogni iterazione si dimezza lo spazio di ricerca
- *Caso migliore*: l'elemento viene trovato al primo colpo
  $\rightarrow$ 1 confronto
- *Caso peggiore*: l'elemento non è presente
  $\rightarrow$ occorre controllare tutto l'array
  - a ogni passo si effettua *un confronto*
  - ad ogni passo la dimensione dell'array si dimezza
    $\rightarrow$ al passo K, l'array è lungo $N / 2^{K-1}$
  - ci si ferma quando l'array è lungo 1, ossia quando $2^{K-1} \geq N$, ossia quando $K = 1 + \log_2 N$
- Numero di confronti: $O(\log_2 N)$

PARALLELIZZARE SEMPRE..?

- La ricerca binaria è un caso *tristemente noto* perché *più si parallelizza, meno ci si guadagna*
  - nel caso sequenziale
    - a ogni passo, con 1 confronto si divide in 2 lo spazio di ricerca
    - ci si ferma dopo k passi, quando $2^k = N \rightarrow k = \log_2 N$
  - nel caso parallelo
    - a ogni passo, avendo P processori si fanno P confronti che dividono lo spazio di ricerca in P+1 parti
    - ci si ferma dopo k passi, quando $(P+1)^k = N \rightarrow k = \log_{P+1} N$
- La base del logaritmo *aumenta linearmente con P*

PARALLELIZZARE SEMPRE..?

- Cosa significa?
  - aumentando i processori, naturalmente, ci si guadagna in senso assoluto…
  - .. ma *il guadagno aumenta solo logaritmicamente rispetto al numero di processori aggiunti*, ossia lo Speed-up è pari a

$$Su = Ts/Tp = \log_2 N / \log_{P+1} N = \log_2(P+1)$$

- Ne segue che al crescere di P *l'efficienza peggiora,* perché (log P) / P → 0:

$$Eff = (\log_2(P+1)) / P$$

PARALLELIZZARE SEMPRE..?

Graficando:

# processori (P)	$\log_{P+1} N$ (N=1000)	$\frac{\log_2 N}{\log_{P+1} N}$	$\log_2 (P+1)$	Efficienza
1	9,97	1,00	1,00	100%
2	6,29	1,59	1,59	79,5%
3	4,98	2,00	2,00	66,7%
…	…	…	…	…
9	3,00	3,32	3,32	36,9%
…	…	…	…	…

Come si vede, *al crescere di P l'efficienza peggiora* rispetto al top 100% del caso sequenziale

PARALLELIZZARE SEMPRE..?

- Questo è l'opposto di ciò che si desidererebbe da un approccio parallelo
  - sarebbe naturale aspettarsi che aumentare i processori fosse «in assoluto» una buona scelta..
  - .. invece, non sempre è così: la ricerca *non ammette un'efficiente soluzione parallela*
  - per approfondire:
    `www.cs.umd.edu/~gasarch/TOPICS/ramsey/parasearch.pdf`