Department of Information Engineering

Ordinamento e ricerca

Rilevazione delle prestazioni

When you learnt quick sort, merge sort, bubble sort, insertion sort but n < 1000

tmtz

Ordinamento e ricerca
[e analisi delle prestazioni]
(capitolo 13)

Department of Information Engineering

Motivazioni

- Problema **molto** frequente nella elaborazione dei dati
  - **Ordinamento** dei dati stessi, secondo un criterio prestabilito
  - Per esempio: nomi in ordine alfabetico, numeri in ordine crescente…
- Studieremo diversi algoritmi per l'ordinamento, introducendo anche uno **strumento analitico** per la **valutazione delle prestazioni** di tali algoritmi
- Inoltre, su dati ordinati è possibile **effettuare ricerche** in modo **efficiente**, e studieremo algoritmi adeguati
- Studieremo degli algoritmi **ricorsivi** ed **efficienti**, ma **non tutti gli algoritmi ricorsivi sono efficienti**...

Department of Information Engineering

Ordinamento per selezione

Ordinamento per selezione

- Per semplicità, analizzeremo prima gli algoritmi per ***ordinare un insieme di numeri*** (interi) memorizzato in un array
  - vedremo poi come si estendono semplicemente al caso in cui si debbano elaborare oggetti anziché numeri
- Prendiamo in esame un array **a** da ordinare in senso crescente

11	9	17	5	12
a[0]	a[1]	a[2]	a[3]	a[4]

→

5	9	11	12	17
a[0]	a[1]	a[2]	a[3]	a[4]

Department of Information Engineering

Ordinamento per selezione

11	9	17	5	12
a[0]	a[1]	a[2]	a[3]	a[4]

5	9	17	11	12

- Per prima cosa, bisogna trovare ***l'elemento dell'array contenente il valore minimo***, come sappiamo già fare
  - in questo caso è il numero **5** in posizione **a[3]**
- Essendo l'elemento minore, la sua posizione corretta nell'array ordinato è **a[0]**
  - in **a[0]** è memorizzato il numero **11**, da spostare
  - non sappiamo quale sia la posizione corretta di **11**
    - lo spostiamo temporaneamente in **a[3]**
  - quindi, ***scambiamo*** **a[3]** con **a[0]**

Department of Information Engineering

Ordinamento per selezione

a[0]	a[1]	a[2]	a[3]	a[4]
5	9	17	11	12

la parte colorata dell'array è già ordinata

- La **parte sinistra** dell'array è **già ordinata** e non sarà più considerata
  - Dobbiamo ancora ordinare la **parte destra**
- Ordiniamo la parte destra **con lo stesso algoritmo**
  - cerchiamo l'elemento contenente il valore minimo, che è il numero **9** in posizione **a[1]**
  - dato che **è già** nella prima posizione della parte da ordinare (la posizione **a[1]**), **non c'è bisogno** di fare scambi

a[0]	a[1]	a[2]	a[3]	a[4]
5	9	17	11	12

Ordinamento per selezione

5	9	17	11	12
a[0]	a[1]	a[2]	a[3]	a[4]

- Proseguiamo per ordinare la parte di array che contiene gli elementi **a[2]**, **a[3]** e **a[4]**
  - il valore minimo è il numero **11**, contenuto nell'elemento **a[3]**
  - scambiamo **a[3]** con **a[2]**

5	9	11	17	12

5	9	11	17	12
a[0]	a[1]	a[2]	a[3]	a[4]

Ordinamento per selezione

5	9	11	17	12
a[0]	a[1]	a[2]	a[3]	a[4]

- Ora l'array da ordinare contiene **a[3]** e **a[4]**
  - Il valore minimo è il numero **12**, contenuto nell'elemento **a[4]**
  - scambiamo **a[4]** con **a[3]**

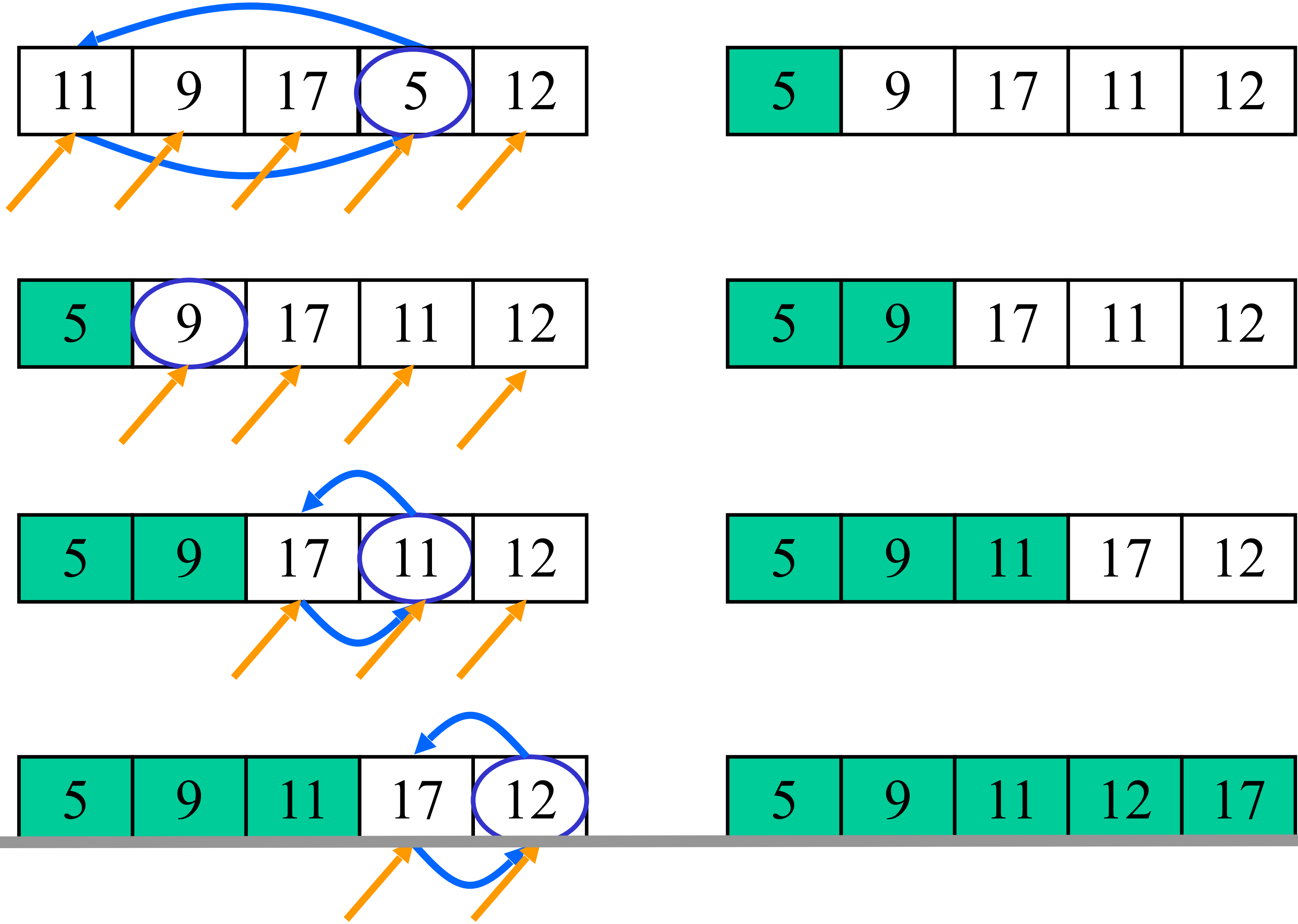
5	9	11	12	17

5	9	11	12	17
a[0]	a[1]	a[2]	a[3]	a[4]

- A questo punto la parte da ordinare contiene **un solo elemento**, quindi è **ovviamente ordinata**

Ordinamento per selezione

= accesso in lettura = accesso in scrittura



La classe ArrayAlgs

- Riprendiamo la nostra classe **ArrayAlgs**
  - L'abbiamo introdotta quando abbiamo studiato semplici algoritmi su array
  - Contiene una collezione di **metodi statici** per l'elaborazione di array
    - Per ora trattiamo array di numeri **interi**
    - Più avanti tratteremo generici array di **Object**
- In particolare **ArrayAlgs** conterrà le realizzazioni dei vari algoritmi di **ordinamento** e **ricerca** da noi studiati
  - È una "**classe di utilità**", come la classe **Math**

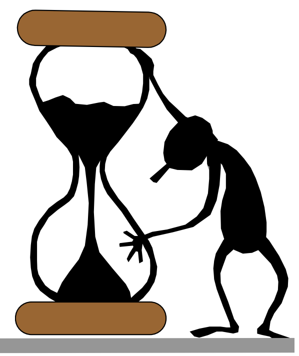
Department of Information Engineering

Il metodo selectionSort

```
public class ArrayAlgs{...
    public static void selectionSort(int[] v, int vSize)
    {
        for (int i = 0; i < vSize - 1; i++)
        {   int minPos = findMinPos(v, i, vSize-1);
            if (minPos != i) swap(v, minPos, i);
        }
    } //abbiamo usato due metodi ausiliari, swap e findMinPos
    private static void swap(int[] v, int i, int j)
    {   int temp = v[i];
        v[i] = v[j];
        v[j] = temp;
    }
    private static int findMinPos(int[] v, int from, int to)
    {   int pos = from;
        int min = v[from];
        for (int i = from + 1; i <= to; i++)
            if (v[i] < min)
            {   pos = i;
                min = v[i];   }
        return pos;
    }
...
}
```

Ordinamento per selezione

- L'algoritmo di ordinamento per selezione è corretto ed è in grado di ordinare qualsiasi array
  - Allora perché studieremo altri algoritmi di ordinamento?
  - A cosa serve avere **diversi algoritmi** per risolvere lo **stesso problema**?
- Vedremo che esistono algoritmi di ordinamento che, a parità di dimensioni del vettore da ordinare, **vengono eseguiti più velocemente**



Department of Information Engineering

È tutto chiaro? …

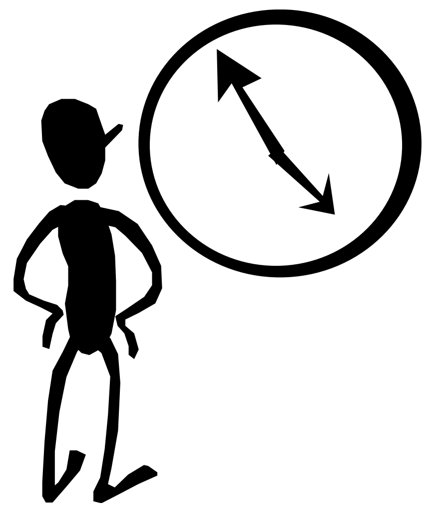
1. Perché nel metodo swap serve la variabile temp? Cosa succede se si assegna semplicemente a[j] ad a[i] e a[i] ad a[j]?
2. Quali sono i passi compiuti da selectionSort per ordinare la sequenza 654321?

Rilevazione delle prestazioni

Department of Information Engineering

Rilevazione delle prestazioni

- Le **prestazioni** di un algoritmo vengono valutate in funzione della **dimensione** dei dati trattati
  - Per valutare **l'efficienza** di un algoritmo si misura il **tempo** in cui viene eseguito su insiemi di dati di dimensioni via via maggiori
- Il tempo **non va misurato con un cronometro**, perché parte del tempo reale di esecuzione non dipende dall'algoritmo
  - caricamento della JVM
  - caricamento delle classi del programma
  - lettura dei dati dallo standard input
  - visualizzazione dei risultati



Rilevazione delle prestazioni

- Il tempo di esecuzione di un algoritmo va misurato all'interno del programma
- Si usa il metodo statico

  ```
  System.currentTimeMillis()
  ```

  che, ad ogni invocazione, restituisce un numero di tipo **long** che rappresenta
  - il numero di millisecondi trascorsi da un **evento di riferimento** (la **mezzanotte del 1 gennaio 1970**)
- Ciò che interessa è la **differenza** tra due valori
  - si invoca **System.currentTimeMillis( )** immediatamente **prima e dopo** l'esecuzione dell'algoritmo (escludendo le operazioni di input/output dei dati)

La classe SelSortTester

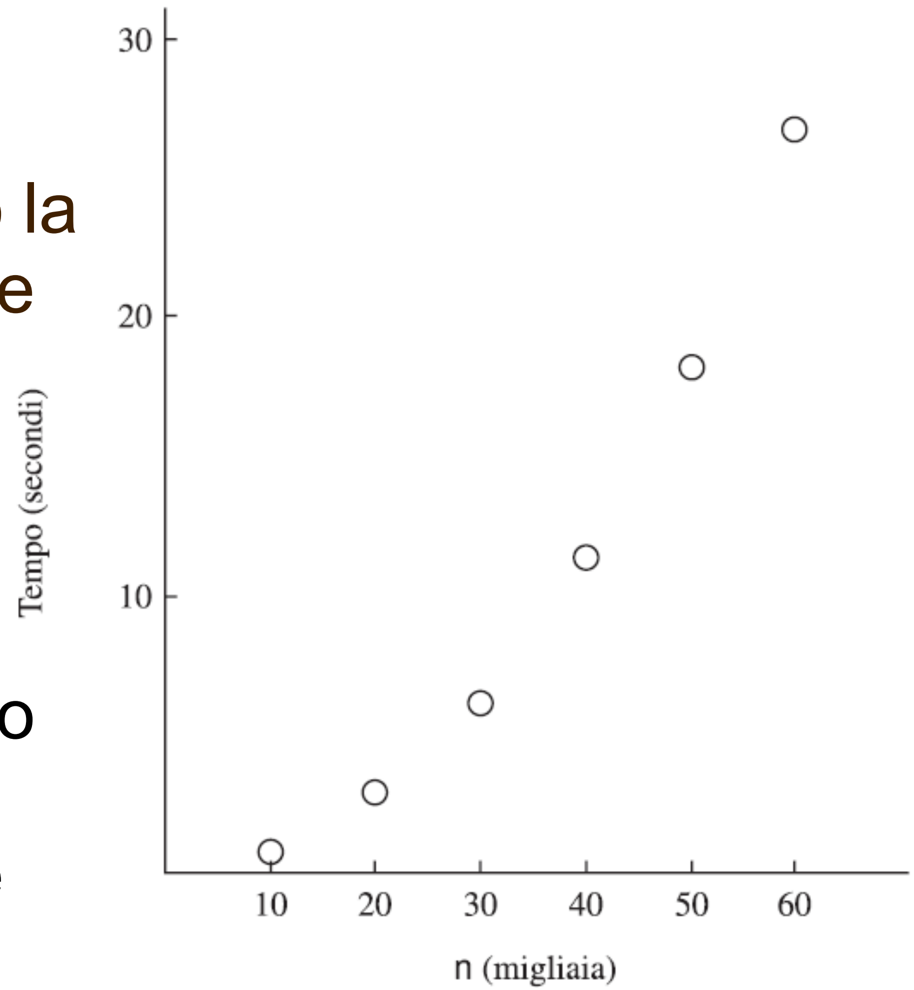
```
import java.util.Scanner;
public class SelSortTester
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Dimensione dell'array? ");
        int n = in.nextInt();
        // costruisce un array casuale
        int[] a = ArrayAlgs.randomIntArray(n, 100);
        int aSize = a.length;
        // usa System.currentTimeMillis() per misurare il tempo
        long time = System.currentTimeMillis();
        ArrayAlgs.selectionSort(a, aSize);
        time = System.currentTimeMillis() -time;

        System.out.println("Tempo trascorso: " + time + " ms");
    }
}
```

Depa

Rilevazione delle prestazioni

- Eseguiamo l'ordinamento di array con diverse dimensioni (**n**), contenenti numeri interi casuali
  - Per ogni valore di **n** ripetiamo la misura molte volte, per trovare un valore medio di **T(n)**
- Si nota che l'andamento del tempo di esecuzione **non è lineare**
  - se **n** diventa il doppio, il tempo diventa circa il **quadruplo**
  - Quindi il tempo di esecuzione ha **andamento quadratico** (parabola tipo $\mathbf{T(n) = a\, n^2}$)



Department of Information Engineering

Analisi teorica delle prestazioni

Department of Information Engineering

Analisi teorica delle prestazioni

- **Modello di costo**: il tempo d'esecuzione di un algoritmo dipende in generale dai seguenti fattori:
  1. Il numero di **operazioni primitive** (o **passi base**) eseguite dall'algoritmo (ad es., istruzioni macchina del processore)
  2. Che a sua volta dipende dalla **dimensione dei dati** da elaborare (per esempio, la lunghezza dell'array da ordinare)
  3. Il **valore dei dati** da elaborare (ad esempio, un array già ordinato, ordinato al contrario, con valori casuali, ...)

- Dato un algoritmo, vogliamo stimare una funzione ***T(n)*** che ne descrive il tempo di esecuzione ***T*** **unicamente** in funzione della dimensione ***n*** dei suoi dati
  - Tramite **analisi teorica**, senza esperimenti numerici
  - Anzi, **senza realizzare e compilare** un algoritmo!

Department of Information Engineering

Analisi teorica delle prestazioni

- Cosa si intende per **operazione primitiva** (passo base)?
- È una operazione che ha **tempo di esecuzione (circa) costante**, indipendente da valori e tipi dei dati. Per es.:
  - Una assegnazione di valore a una variabile
  - Una operazione aritmetica o logica tra variabili e/o costanti numeriche e booleane
  - Un accesso in lettura/scrittura a un elemento di un array
  - Invece un enunciato contenente una invocazione di un metodo **non** è un'operazione primitiva
- Nel caso di **selectionSort**, il tempo di esecuzione si stima contando il numero di **accessi in lettura/scrittura** a un elemento dell'array, in funzione della lunghezza **n** dell'array (dimensione dei dati del problema)

Department of Information Engineering

Analisi teorica delle prestazioni

- Cosa si intende per **dimensione dei dati** di un algoritmo?
- A seconda del problema, la dimensione dell'input assume significati diversi
  - La **grandezza di un numero** (per esempio in problemi di calcolo)
  - Il **numero di elementi** su cui lavorare (ad esempio in **problemi di ordinamento**)
  - Il **numero di bit** che compongono un numero
  - ...
- Indipendentemente dal tipo di dati, indichiamo sempre con ***n*** la dimensione dell'input.

Analisi teorica delle prestazioni

- Come si tiene conto del **valore dei dati**?
  - Se l'algoritmo contiene cicli e decisioni, il numero di operazioni primitive dipende anche dal valore dei dati
  - Ma noi vogliamo **T(n)** come funzione solo di **n**.
- Di solito si stima **per eccesso** il tempo di esecuzione ***T(n)***, ovvero di ottenere una **stima** **“di caso peggiore”**
  - A seconda del problema, la definizione di “caso peggiore” assume significati diversi
  - Per es., per un algoritmo di ordinamento il caso peggiore è quello in cui l’array in input è **ordinato alla rovescia.**
- Si possono anche fare stime di
  - **caso migliore** (per es. array in ingresso già ordinato)
  - **caso medio** (con ipotesi statistiche, ad es. array in input contenente numeri casuali)

Department of Information Engineering

Prestazioni di selectionSort

- Esaminiamo il codice Java del metodo **selectionSort**
  - Potremmo anche esaminare lo pseudo-codice
- Conteggio degli accessi all'array nella **prima** iterazione del ciclo esterno (ovvero per **i=0**)
  - Per trovare l'elemento minore si fanno **n** accessi
  - Per scambiare due elementi si fanno **quattro** accessi
    - **Caso peggiore**: ipotizziamo che serva sempre lo scambio
  - In totale si fanno quindi **(n+4)** accessi
- Ora l'algoritmo deve ordinare la parte rimanente, cioè un array di **(n-1)** elementi
  - serviranno quindi **((n-1) + 4)** accessi
- E così fino al passo con **(n-(n-2))=2** elementi, incluso

Prestazioni di selectionSort

- Il conteggio totale degli accessi in lettura/scrittura è quindi

```
T(n) = (n+4)+((n-1)+4)+…+(3+4)+(2+4)
     = n+(n-1)+…+3+2 + (n-1)*4
     = n*(n+1)/2 - 1 + (n-1)*4
     = 1/2*n² + 9/2*n - 5
```

- Si ottiene quindi una **equazione di secondo grado** in **n**, che giustifica l'andamento **parabolico** dei tempi rilevati sperimentalmente

$$\boxed{\sum_{i=1}^{n} i = \frac{n(n+1)}{2}}$$

Department of Information Engineering

Prestazioni di selectionSort

- Sommatoria notevole: $$\boxed{\sum_{i=1}^{n} i = \frac{n(n+1)}{2}}$$
  - La si ritrova molto spesso nella stima di tempi di esecuzione di cicli annidati.
- Dimostrazione di questa uguaglianza
  - Basta disporre gli addendi in questo ordine

```
 1  +    2    +    3    + ... +   n   +
 n  + (n-1)  + (n-2)  + ... +   1   =
____________________________________
n+1 +   n+1   +   n+1   + ... +   n+1
```

n volte

Department of Information Engineering

Andamento asintotico delle prestazioni

$$T(n) = 1/2*n^2 + 9/2*n - 5$$

- Facciamo un'ulteriore semplificazione, tenendo presente che ci interessano le prestazioni per **valori elevati** di **n** (**andamento asintotico**)
- Per esempio, se **n** vale 1000
  - $1/2*n^2$ vale 500000
  - $9/2*n-5$ vale 4495, circa **1%** del totale
- quindi diciamo che l'**andamento asintotico** dell'algoritmo in funzione di **n** è

$$T(n) \sim 1/2*n^2$$

Department of Information Engineering

Andamento asintotico delle prestazioni

$$\boxed{T(n) \sim 1/2*n^2}$$

- Facciamo un'ulteriore **semplificazione**
  - ci interessa soltanto valutare cosa succede al tempo d'esecuzione **T(n)** se **n**, ad esempio, **raddoppia**
  - Nel caso in esame il tempo di esecuzione **quadruplica**

    $T(n) = 1/2*n^2$

    $T(2n) = 1/2*(2n)^2 = 1/2*4*n^2$

    $= 4*T(n)$
- Si osserva che $T(2n) = 4*T(n)$
  - Questo è vero in generale nel caso in cui $T(n) = cn^2$
  - **Indipendentemente** dal fatto che sia presente un fattore moltiplicativo `1/2`, o qualsiasi altro fattore moltiplicativo **c**.

Department of Information Engineering

È tutto chiaro? …

1. Se si aumenta di 10 volte la dimensione dei dati, come aumenta il tempo richiesto per ordinare i dati usando **selectionSort**?

Department of Information Engineering

Department of Information Engineering

Notazione "O-grande"

- Si dice quindi che
  - per ordinare un array con l'algoritmo di selezione si effettua un numero di accessi che è ***dell'ordine* di** n^2
- Per esprimere sinteticamente questo concetto si usa la notazione **O-grande** e si dice che il numero degli accessi è $O(n^2)$
- Dopo aver ottenuto una formula che esprime l'andamento temporale dell'algoritmo, si ottiene la notazione "O-grande" considerando soltanto il **termine che si incrementa più rapidamente** all'aumentare di n, **ignorando** coefficienti costanti

Notazione "O-grande", "Ω", "Θ"

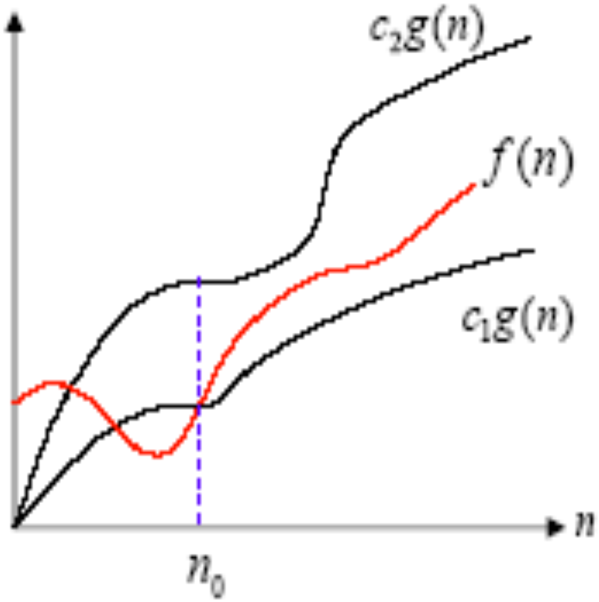
- La notazione $f(n)=O(g(n))$ significa: f **non cresce più velocemente** di g
  - Ma è possibile che f cresca molto più lentamente
  - Esempio: $f(n) = n^2 + 5n - 3$ è $O(n^3)$, ma è anche $O(n^{10})$

- Esistono ulteriori notazioni per descrivere il modo in cui una funzione cresce
  - $f(n)=\Omega(g(n))$ significa: f **non cresce più lentamente** di g
    - Ovvero $g(n) = O(f(n))$
  - $f(n)=\Theta(g(n))$ significa: f **cresce con la stessa velocità** di g
    - Ovvero $f(n) = O(g(n))$ **e** $f(n) = \Omega(g(n))$

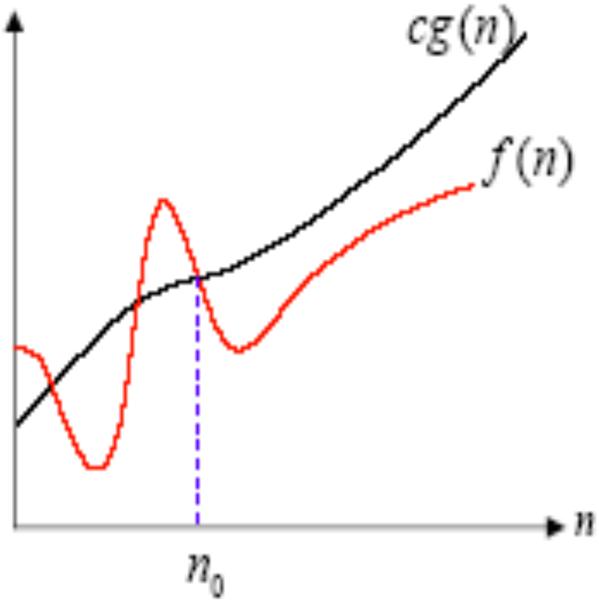
Notazione "O-grande", "Ω", "Θ"

- ***f*(*n*) = *O*(*g*(*n*))** esiste una costante ***C>0*** tale che

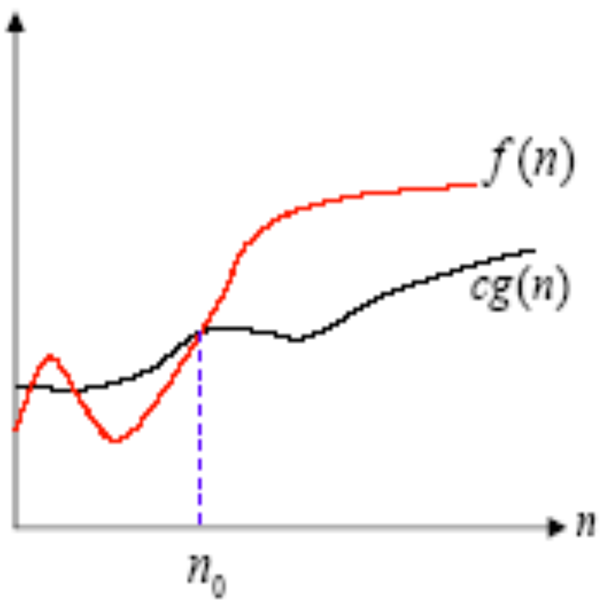
  f(n)/g(n)* < *C *definitivamente*
- ***f*(*n*) = *Ω*(*g*(*n*))** … ***f(n)/g(n)* > *C*** *definitivamente*
- ***f*(*n*) = *Θ*(*g*(*n*))** … $C_1 < f(n)/g(n) < C_2$ *definitivamente*



$f(n) = \Theta(g(n))$



$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$

Department of Information Engineering

Ordini di complessità

- Un **algoritmo** può essere classificato in funzione delle proprie **prestazioni**
  - Un algoritmo è considerato **efficiente** se il suo tempo di esecuzione (caso peggiore) è **al più polinomiale**
  - Un algoritmo è considerato **inefficiente** se il suo tempo di esecuzione (caso peggiore) è **almeno esponenziale**
- Un **problema algoritmico** può essere classificato in funzione della propria **complessità** (ovvero le prestazioni del più veloce algoritmo che lo risolve)
  - Un problema è considerato **trattabile** se la sua complessità è **al più polinomiale**
  - Un problema è considerato **non trattabile** se la sua complessità è **almeno esponenziale**

Department of Information Engineering

Ordini di complessità

- Ipotesi: una istruzione di Java viene eseguita in *10^{-7} s*

(1 anno ≈ 3E7 secondi)

(convenzione anglosassone: 1 “bilione” = 10^9; 1 “trilione” = 10^{12})

	f(n)	n=10	n=20	n=50	n=100	n=300
Problemi trattabili (polinomiali)	n^2	1/100000 secondi	1/25000 secondi	1/4000 secondi	1/1000 secondi	9/1000 secondi
	n^5	1/100 secondi	0.32 secondi	31.2 secondi	16.7 minuti	2.8 giorni
Problemi non trattabili	2^n	1/10000 secondi	0.1 secondi	3.57 anni	40 trilioni di secoli	Un numero a 74 cifre di secoli
	n^n	16.7 minuti	332 bilioni (miliardi) di anni	Un numero a 69 cifre di secoli	Un numero a 184 cifre di secoli	Un numero a 727 cifre di secoli

Cicli annidati: analisi delle prestazioni

- Riesaminiamo la struttura di **SelectionSort**
  - È realizzato tramite **due cicli annidati** di questo tipo

```
for (int i = 0; i < n; i++)
    //... operazioni primitive
    for (int j = i; j < n; j++)
        //... operazioni primitive
```

- Per stimarne il tempo di esecuzione dobbiamo stimare il numero di operazioni primitive eseguite **nel ciclo interno**.
  - Per **i = 0** vengono eseguite **n** volte. Per **i = 1** vengono eseguite **n-1** volte. ...
  - Il numero totale è quindi $$\boxed{\sum_{i=1}^{n} i = \frac{n(n+1)}{2}} \longleftarrow \boxed{= O(n^2)}$$

- Due cicli annidati del tipo appena esaminato **hanno sempre prestazioni $O(n^2)$**

Ordinamento per fusione (*MergeSort*)

Department of Information Engineering

MergeSort

- Presentiamo ora un algoritmo di ordinamento “**per fusione**” (**MergeSort**) che vedremo avere **prestazioni migliori** di quelle dell’ordinamento per selezione
- Dovendo ordinare il vettore seguente

5	7	9	1	8

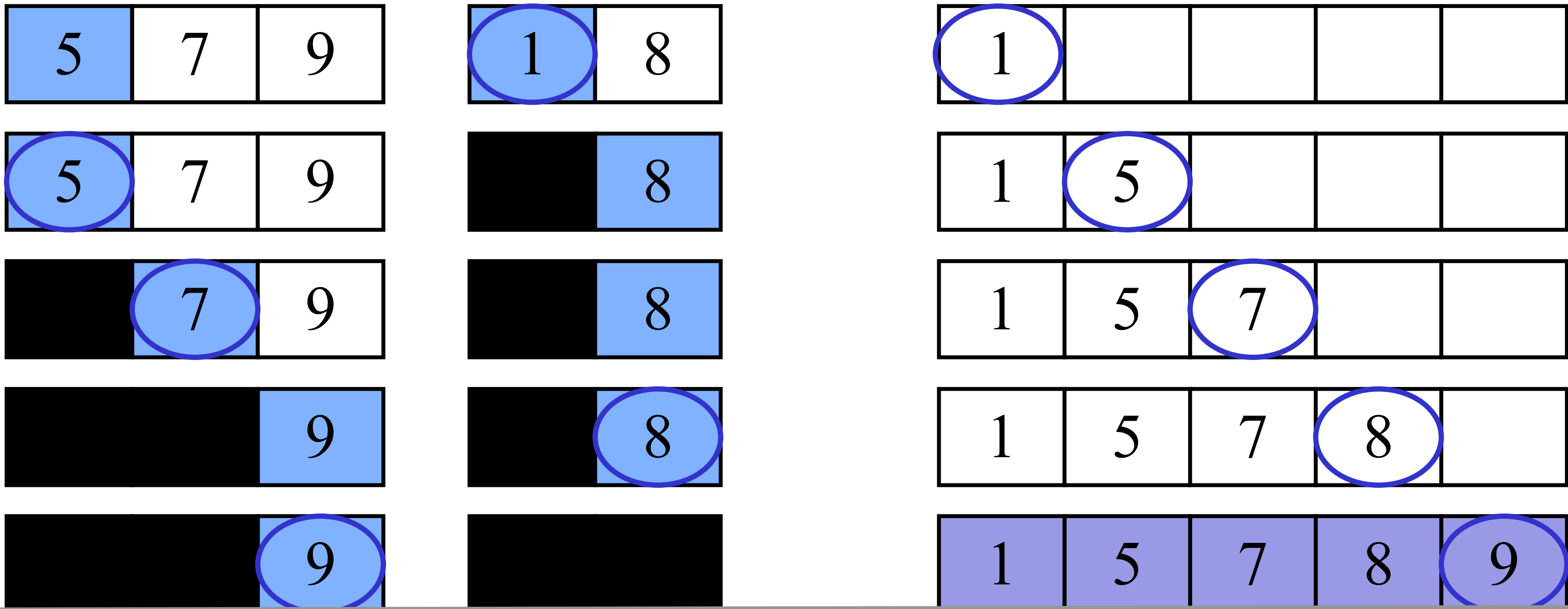
- Lo dividiamo in due parti (circa) uguali

5	7	9

1	8

MergeSort

- Supponiamo che le due parti siano **già ordinate**
  - Allora è facile costruire il vettore ordinato, prendendo sempre il **primo elemento** da uno dei due vettori, scegliendo **il più piccolo**



MergeSort

- Ovviamente, nel caso generale le due parti del vettore non saranno ordinate
- Possiamo però **ordinare ciascuna parte** ripetendo il processo
  - **dividiamo** il vettore in due parti (circa) uguali
  - **ordiniamo** ciascuna delle due parti, separatamente
  - **uniamo le due parti** ora ordinate, nel modo visto
    - questa ultima fase si chiama **fusione** (**merge**)
- C'è una situazione in cui le due parti **sono sicuramente ordinate**
  - quando contengono **un solo elemento**

Department of Information Engineering

MergeSort

- Si delinea quindi un **algoritmo ricorsivo** per ordinare un array, chiamato **MergeSort**
- **Caso base**: se l'array contiene meno di due elementi, è già ordinato
- **Altrimenti**
  - si divide l'array in due parti (circa) uguali
  - si ordina la prima parte usando **MergeSort**
  - si ordina la seconda parte usando **MergeSort**
  - si fondono le due parti ordinate usando l'algoritmo di **fusione** (**merge**)

Department of Information Engineering

Realizzazione di mergeSort

```
public class ArrayAlgs
{   ...
    public static void mergeSort(int[] v, int vSize)
    {
        if (vSize < 2) return; // caso base
        int mid = vSize / 2; //dividiamo circa a meta'
        int[] left = new int[mid];
        int[] right = new int[vSize - mid];
        System.arraycopy(v, 0, left, 0, mid);
        System.arraycopy(v, mid, right, 0, vSize-mid);
        // passi ricorsivi: ricorsione multipla (doppia)
        mergeSort(left, mid);
        mergeSort(right, vSize-mid);
        // fusione (metodo ausiliario)
        merge(v, left, right);
    }
                                              // continua
```

Realizzazione di mergeSort

```
                                          // continua
  private static void merge(int[] v, int[] v1, int[] v2)
  {
      int i = 0, i1 = 0, i2 = 0;
      while (i1 < v1.length && i2 < v2.length)
          if (v1[i1] < v2[i2])
              // prima si usa i, poi lo si incrementa...
              v[i++] = v1[i1++];
          else
              v[i++] = v2[i2++];
      while (i1 < v1.length)
          v[i++] = v1[i1++];
      while (i2 < v2.length)
          v[i++] = v2[i2++];
  }
  ...
}
```

È tutto chiaro? …

1. Eseguire manualmente l'algoritmo **mergeSort** sull'array 87654321

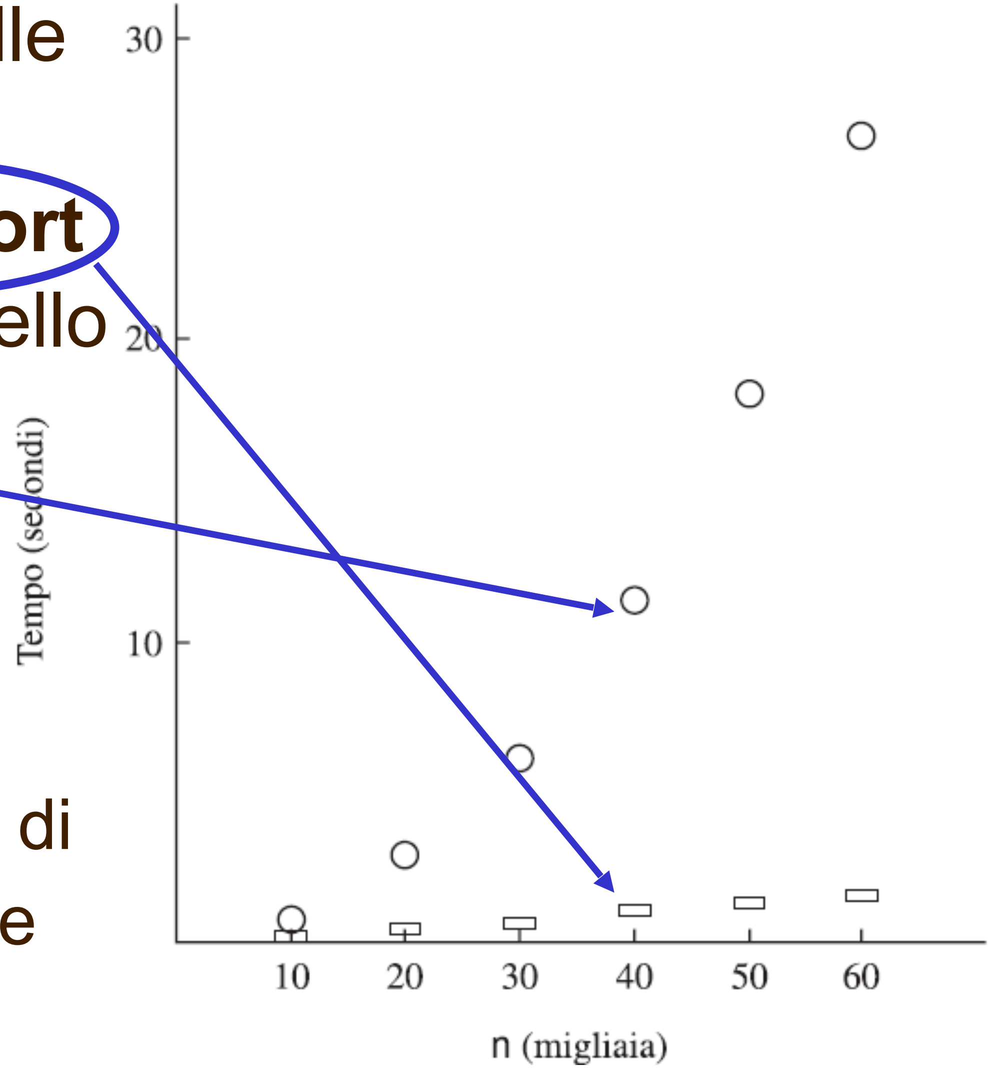
Department of Information Engineering

Prestazioni di MergeSort

Department of Information Engineering

Prestazioni di MergeSort

- Rilevazioni sperimentali delle prestazioni mostrano che l'ordinamento con **MergeSort** è **molto** più efficiente di quello con **selectionSort**
- Cerchiamo di fare una valutazione teorica delle prestazioni
  - Chiamiamo `T(n)` il numero di accessi richiesti per ordinare un array di `n` elementi



Department of Information Engineering

Prestazioni di MergeSort

- Analizziamo i tempi di esecuzioni delle tre fasi
  - Creazione di due sottoarray
  - Ordinamento dei due sottoarray (invocazioni ricorsive)
  - Fusione dei due sottoarray
- La **creazione** dei due sottoarray richiede `2n` accessi
  - Perchè tutti gli `n` elementi devono essere letti e scritti
- Le **invocazioni ricorsive richiedono `T(n/2)` ciascuna**
  - **Per definizione**: **T(n)** è il tempo di esecuzione di mergeSort su un array di dimensione **n**
- La fusione richiede
  - **2n** accessi ai sottoarray ordinati (ogni elemento da scrivere nell'array finale richiede la lettura di due elementi, uno da ciascuno dei due array da fondere)
  - Più **n** accessi in scrittura nell'array finale

Department of Information Engineering

Prestazioni di MergeSort

- Sommando tutti i contributi: $\boxed{T(n) = 2T(n/2) + 5n}$
- Questa è (definizione) una **equazione di ricorrenza**
- Soluzione: per trovare un'espressione esplicita di ***T(n)*** in funzione di **n** si procede per **sostituzioni successive**, fino ad arrivare al caso base
  - $T(n/2)=2T(n/4)+5(n/2)$, $T(n/4)=2T(n/8)+5(n/4)$, ecc.
  - Inoltre $T(1)=1$ (è il **caso base**, ordinamento con **n=1**)
- Quindi:

$$T(n) = 2(\ 2T(n/4) + 5(n/2)\ ) + 5n = 4T(n/4) + 2*5n =$$
$$= \ldots\text{(dopo k sostituzioni)} \ldots = 2^k T(n/2^k) + k*5n$$

- Dal termine $T(n/2^k)$ si vede che il caso base è raggiunto dopo $k = \log_2 n$, sostituzioni, ovvero quando $n/2^k = 1$

Department of Information Engineering

Prestazioni di MergeSort

$$T(n) = 2^k T(n/2^k) + k*5n = nT(1) + 5n*log_2 n$$

per $k=log_2 n$

ma $T(1)=1$

- Per trovare la notazione "O-grande" osserviamo che
  - il termine $5n*log_2 n$ cresce più rapidamente di n
  - il fattore **5** è ininfluente, come ogni costante moltiplicativa
  - Nelle notazioni "O-grande" non si indica la base dei logaritmi, perché log_a si può trasformare in log_c con un fattore moltiplicativo, che va ignorato $\log_a b = \frac{\log_c b}{\log_c a}$
- In conclusione, l'algoritmo **MergeSort** ha tempi di esecuzione

$$\boxed{T(n) = O(n \log n)}$$

- e ha quindi prestazioni migliori di $O(n^2)$

Department of Information Engineering

Ricorsione e stima delle prestazioni

Department of Information Engineering

- Quando un algoritmo contiene almeno una invocazione ricorsiva, il suo tempo di esecuzione **T(n)** può sempre essere espresso da una **equazione di ricorrenza**
  - **Ovvero**: una equazione che descrive una funzione in termini del valore che essa assume su valori di input più piccoli

$$\boxed{T(n) = aT(n/b) + f(n)}$$

- Uno dei metodi (non l'unico) per risolvere queste equazioni è il **metodo per sostituzione**
  - Di cui abbiamo visto un esempio analizzando **mergeSort**
- Il **Master Theorem** fornisce una stima dell'ordine di **T(n)** in funzione di **a**, **b**, **f(n)**

Department of Information Engineering

Ordinamento per inserimento (cfr. argomenti avanzati 13.2)

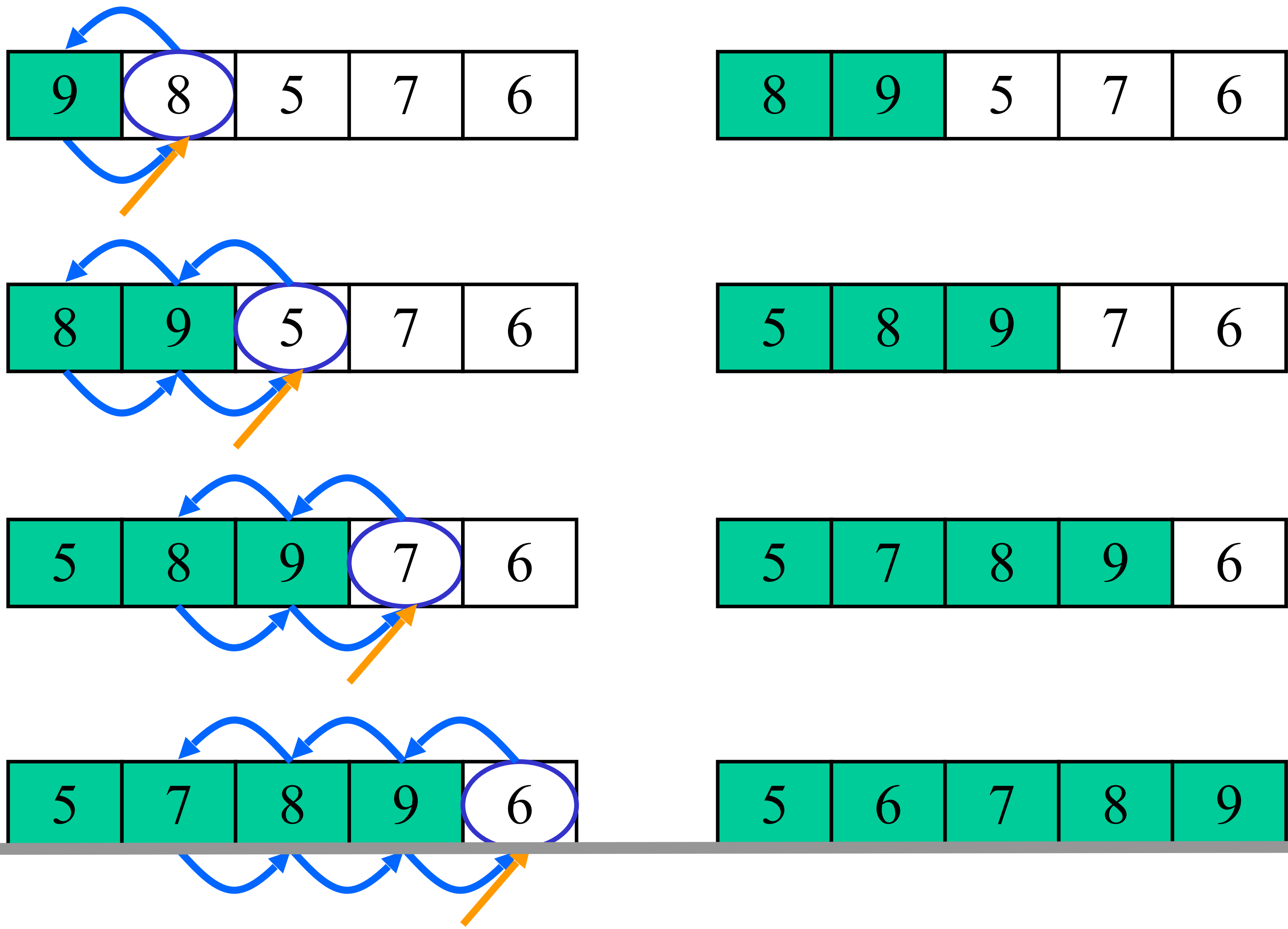
Ordinamento per inserimento

- L’algoritmo di ordinamento per inserimento
  - inizia osservando che il **sottoarray di lunghezza unitaria** costituito dalla prima cella dell’array è ordinato (essendo di lunghezza unitaria)
  - estende verso destra la parte ordinata, **inserendo** nel sottoarray ordinato il primo elemento alla sua destra
    - per farlo, il nuovo elemento viene spostato verso sinistra finché non si trova nella sua posizione corretta, spostando verso destra gli elementi intermedi

9	8	5	7	6
a[0]	a[1]	a[2]	a[3]	a[4]

Ordinamento per inserimento

= accesso in lettura = accesso in scrittura



Realizzazione di insertionSort

```
public class ArrayAlgs
{   ...
    public static void insertionSort(int[] v,int vSize)
    {   // il ciclo inizia da 1 perché il primo
        // elemento non richiede attenzione
        for (int i = 1; i < vSize; i++)
        {
            int temp = v[i]; //nuovo el. da inserire
            // j va definita fuori dal ciclo perche` il
            // suo valore finale viene usato in seguito
            int j;
            //sposta di uno verso destra tutti gli el. a
           // sin. di temp e > di temp partendo da destra
            for (j = i; j > 0 && temp < v[j-1]; j--)
                v[j] = v[j-1];
            v[j] = temp; // inserisci temp
        }
    }
  ...
}
```

Prestazioni di insertionSort

- Ordiniamo con inserimento un array di `n` elementi
- Il ciclo esterno esegue `n-1` iterazioni
  - A ogni iterazione vengono eseguiti
    - **2 accessi** (uno prima del ciclo interno e uno dopo)
    - **il ciclo interno**
- Il ciclo interno esegue **3 accessi** per ogni sua iterazione
  - ma quante iterazioni esegue?
    - **dipende** da come sono ordinati i dati!

Department of Information Engineering

Prestazioni di insertionSort

- **Caso peggiore**: i dati sono ordinati a rovescio
- Ciascun nuovo elemento inserito richiede lo spostamento di tutti gli elementi alla sua sinistra, perché **deve essere inserito in posizione 0**

`T(n) = (2 + 3*1) + (2 + 3*2) +`

- `(2 + 3*3) + ... + (2 + 3*(n-1))`
- `= 2(n-1) + 3[1+2+...+(n-1)]`
- `= 2(n-1) + 3n(n-1)/2`
- $= O(n^2)$

- **Osservazione:** la struttura di **insertionSort** è quella dei **due cicli annidati** esaminati in precedenza (analoga a quella di **selectionSort**)

Department of Information Engineering

Prestazioni di insertionSort

- **Caso migliore**: i dati sono già ordinati
- Il ciclo più interno non esegue mai iterazioni
  - richiede un solo accesso per la prima verifica
- Il ciclo esterno esegue `n-1` iterazioni
  - A ogni iterazione vengono eseguiti
    - 2 accessi (uno prima del ciclo interno ed uno dopo)
    - 1 accesso (per verificare la condizione di terminazione del ciclo interno)
- $T(n) = 3*(n-1) = O(n)$

Department of Information Engineering

Prestazioni di insertionSort

- **Caso medio**: i dati sono in ordine casuale
- Ciascun nuovo elemento inserito richiede **in media** lo spostamento di metà degli elementi alla sua sinistra
- $T(n) = (2 + 3*1/2) + (2 + 3*2/2) +$
- $(2 + 3*3/2) + ... +$
- $(2 + 3*(n-1)/2)$
- $= 2(n-1) + 3[1+2+...+(n-1)]/2$
- $= 2(n-1) + 3[n(n-1)/2]/2$
- $= O(n^2)$

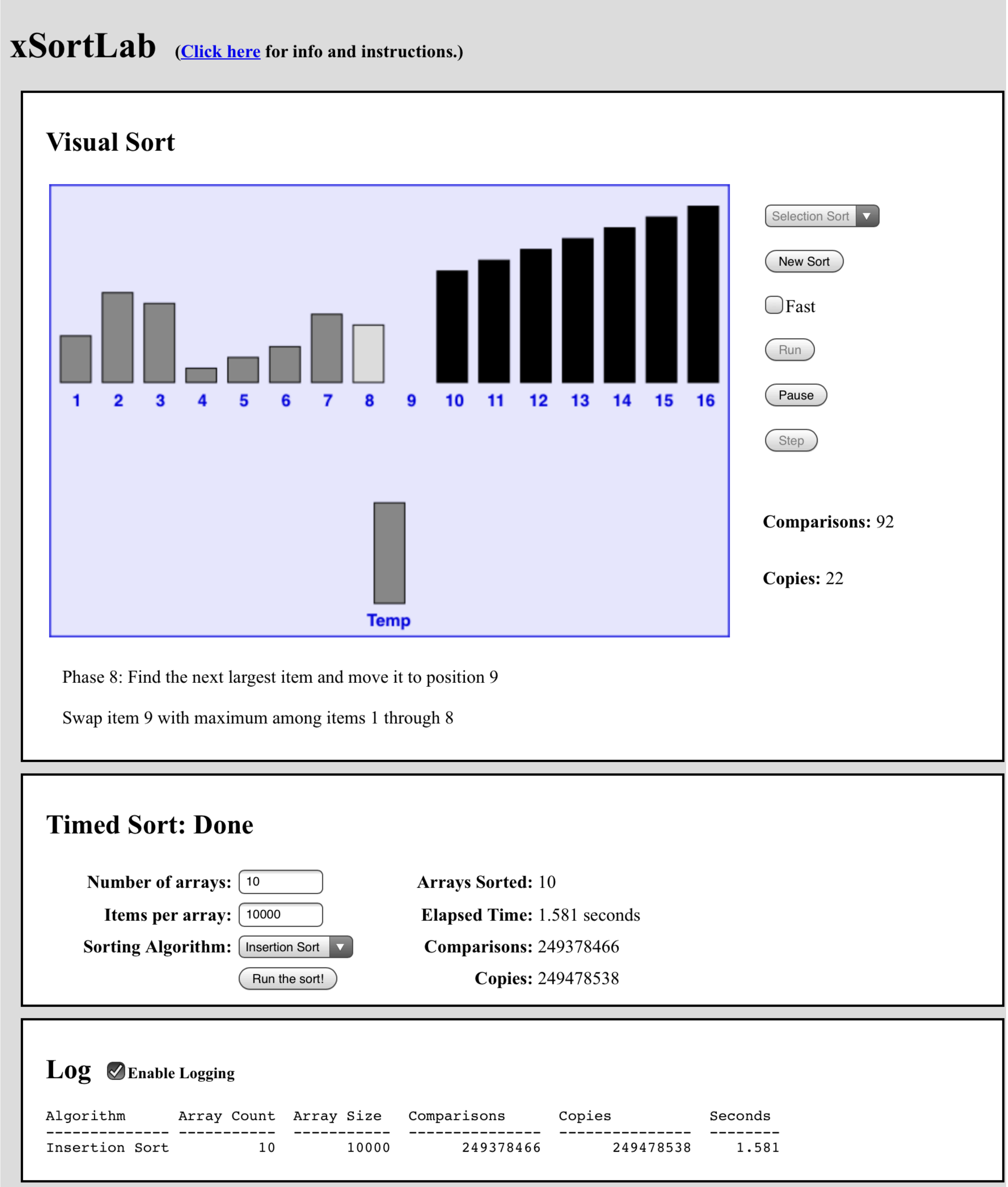
Department of Information Engineering

Confronto tra ordinamenti

	caso migliore	caso medio	caso peggiore
merge sort	n lg n	n lg n	n lg n
selection sort	n^2	n^2	n^2
insertion sort	n	n^2	n^2

- Se si sa che l'array è "quasi" ordinato, è meglio usare l'ordinamento per inserimento
- **Esempio notevole**: un array che **viene mantenuto ordinato** per effettuare ricerche, inserendo ogni tanto un nuovo elemento e poi riordinando
- http://math.hws.edu/eck/js/sorting/xSortLab.html

Department of Information Engineering



- http://math.hws.edu/eck/js/sorting/xSortLab.html

È tutto chiaro? …

1. Eseguire manualmente l’algoritmo **insertionSort** sull’array 87654321 e sull’array 23456781

Department of Information Engineering