Un esempio di calcolo di complessità: insertion sort

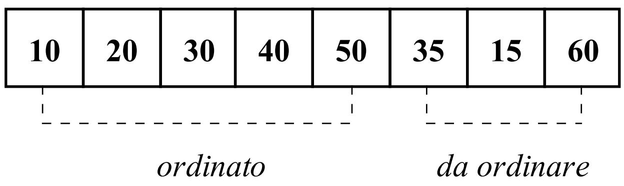
Vediamo su un esempio come si può calcolare la complessità di un algoritmo....

L'esempio è un metodo semplice per ordinare arrays: insertion sort, o inserimento diretto (bisogna però dire che tra i metodi semplici il migliore, dal punto di vista dell'efficienza, è quello che abbiamo visto per primo: selection sort).

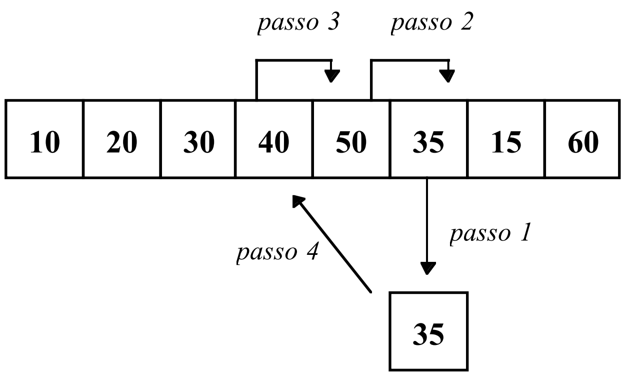
L'idea di base, per l'inserimento diretto, è illustrata in Figura 1.

Figura 1.

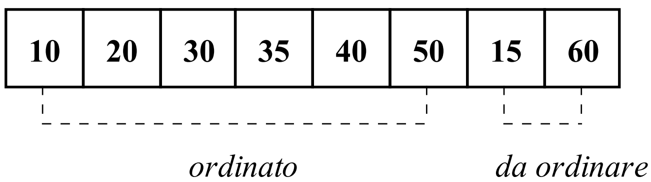
Supponiamo di aver già ordinato una parte dell'array:



Ora sistemiamo il primo degli elementi "fuori posto", cioè 35
(la casella in basso rappresenta una variabile ausiliaria):



Quindi, l'array diventa:



All'inizio, la parte ordinata è costituita da un solo elemento: il primo.

Insertion sort sotto forma di “pezzo di programma”.

Supponiamo di avere un programma in cui ci siamo definiti ed abbiamo “riempito” un array:

a : array [1 ... n] di interi.   Ora vogliamo ordinarlo; per farlo usiamo:

- j   per indicare l’elemento da “sistemare”
- i   per scorrere l’array da j-1 a 1
- temp   per salvare il valore dell’elemento da “sistemare”

Lo pseudo-codice per il pezzo di programma che ordina l’array a è:

```
per j = 2, 3, ..., n :   {   temp ← a[j]
                             /* ora inseriamo a[j] al suo posto */
                             i ← j - 1
                             while i > 0 e a[i] > temp  do {  a[i+1] ← a[i]
                                                              i ← i-1
                                                           }
                             a[i+1] ← temp
                         }
```

Nota: nella condizione del while, e è un and stile C : se (i > 0) è falso, non si passa a valutare la seconda parte della condizione (infatti a[i] non sarebbe definito).

Costo dell’algoritmo

Per valutare il costo dell’algoritmo, traduciamolo in un “diagramma di flusso”, traducendo il “per” ed il while; vedere Fig. 2.

Ci interessa valutare il costo di tutto l’algoritmo in funzione di n (cioè il numero di elementi nell’array), che è un buon paramentro per caratterizzare la dimensione dell’input dell’algoritmo di ordinamento (che è appunto l’array).

Il costo di eseguire ciascuna delle istruzioni contenute nelle “scatole” di Fig. 2, come pure il costo dei test contenuti negli “ovali” di Fig. 2 è costante: non dipende da n (inoltre è “piccolo”, in quanto ogni istruzione/test corrisponde a poche istruzioni macchina).

Questo è visualizzato in Fig 3, dove, al posto delle istruzioni ed dei test abbiamo indicato, per ciascuna: il costo (usando delle costanti C1, C2,...) ed il numero di volte che viene eseguita nel caso peggiore.

Qual’è il caso peggiore ? È quello in cui gli elementi sono in ordine rovesciato (ad esempio, con n=5: 50, 40, 30, 20, 10); ad ogni passo, l’elemento j-mo va confrontato con tutti i precedenti e questi vengono tutti traslati di un posto.

Figura 2 - Diagramma dell'algoritmo

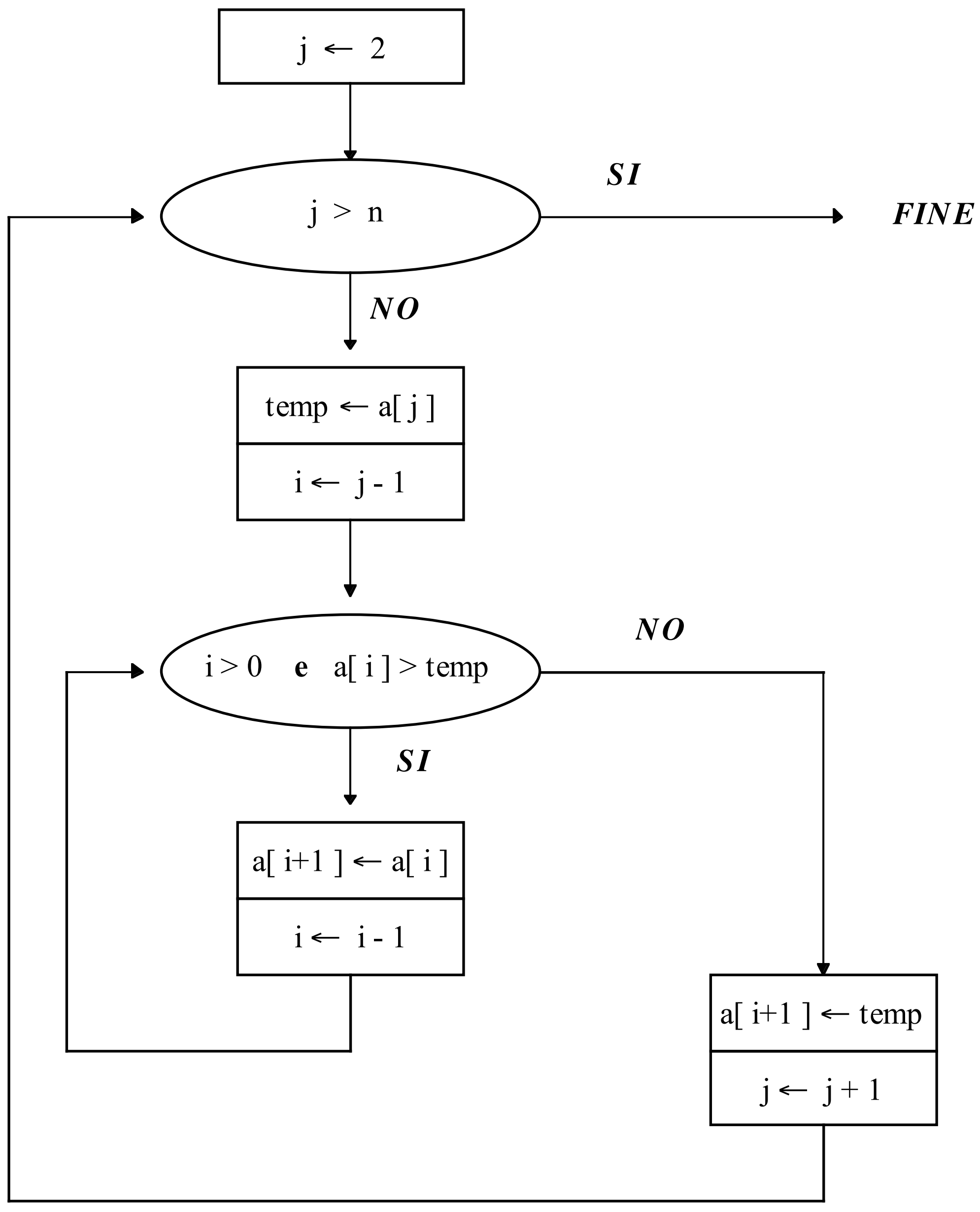
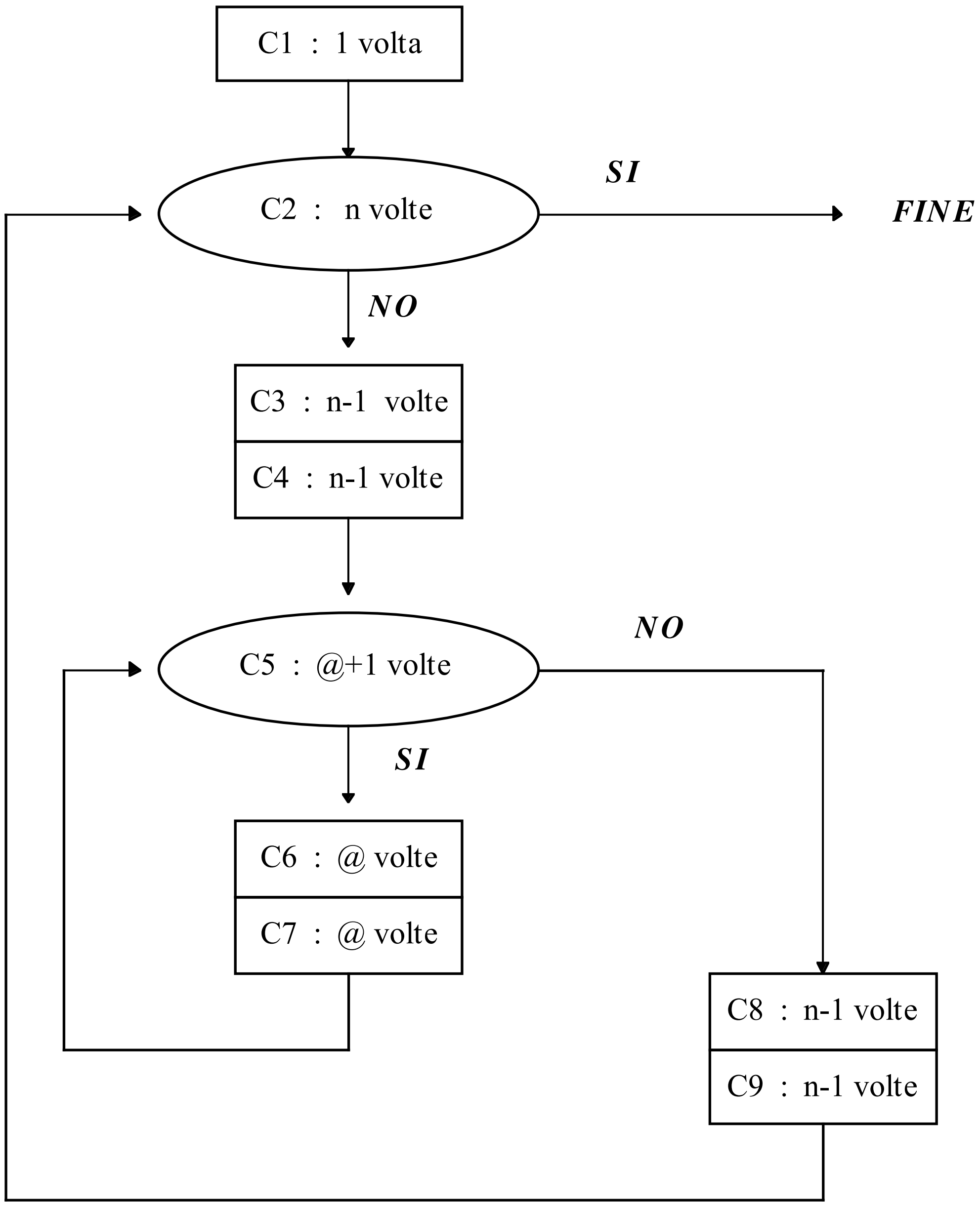


Figura 3 - Costi



In **Fig. 3** dovrebbe essere tutto chiaro, a parte il valore di **@**.

Quello che abbiamo detto a proposito del caso peggiore, però, fa capire che

al passo j-mo, il corpo del while (cioè le due istruzioni $a[i+1] \leftarrow a[i]$; $i \leftarrow i-1$)

si esegue j-1 volte, perchè bisogna traslare tutti gli elementi da 1 a j-1.

Allora $@ = \sum_{(j=2\ldots n)} (j-1) = \sum_{(k=1\ldots n-1)} k$

Poichè questo genere di sommatorie si incontra spesso, vediamo come si ottiene il risultato.

$@ = \sum_{(k=1...n-1)} k$ Allora:

$$\begin{array}{lllllllll} @ & = & 1 & + & 2 & + & ..... & + & n-1 \\ @ & = & n-1 & + & n-2 & + & .... & + & 1 \\ \end{array}$$

==================================

$$\begin{array}{lllllllll} 2\,@ & = & n & + & n & + & ..... & + & n \end{array} \quad \text{(n-1 addendi)}$$

Quindi $@ = n\,(n-1)/2$.

Adesso possiamo sommare tutti i costi; se indichiamo con T_{IS} la funzione complessità tempo dell'algoritmo, abbiamo:

$T_{IS}\ (n) = C1 + n\ C2 + ... + (n(n-1)/2 + 1)\ C5 + ...\ = a\,n^2 + b\,n + c$

con a, b, c costanti opportune; a > 0.

Isando una notazione che vedremo poi, si scrive:

$$T_{IS}\ (n) \in \Theta(n^2)$$

Come alternativa: prima di calcolare il valore di @ facciamo un po' di somme e semplifichiamo.

$T_{IS}\ (n) = C1 + n\ C2 + ... + (@ + 1)\ C5 + ... =$

$a\,n + b@ + c$ con a, b, c opportune costanti intere e positive

Poichè $@ = 1 + 2 + ...+ (n-1)$ si ha subito: $p^2 < @ < n^2$ dove $p = (n-1)$ div2

Quindi, sempre usando la notazione che vedremo:

@ è in $\Theta(n^2)$ e domina sul resto; dunque: $T_{IS}\ (n) \in \Theta(n^2)$

Se gli elementi dell'array non sono interi ....

Nei conti che abbiamo fatto, abbiamo valutato costante il costo di confrontare 2 elementi dell'array (istruzione: a[i] > temp) e quello di copiarli (istruzione: a[i+1] ← a[i]). Questo è corretto se gli elementi sono interi, reali, caratteri, record di piccola dimensione,....

Se invece abbiamo un array di stringhe di lunghezza "arbitraria" (quindi un array di arrays o di liste), allora dobbiamo calcolare anche il costo dei confronti e degli spostamenti.

Per la precisione, T_{IS} deve avere 2 parametri: n (numero di elementi) e ls (che caratterizza la lunghezza delle stringhe nell'array). Il problema è come scegliere ls, visto che le stringhe hanno lunghezza arbitraria. Nell'ottica del caso peggiore, in genere si sceglie ls = lunghezza massima delle stringhe nell'array. A questo punto, il caso peggiore è quello in cui le stringhe hanno (quasi) tutte lunghezza (vicina alla) massima.

In questo caso, il costo dei confronti è lineare in ls (cioè, della forma: a ls + b, con a >0).

Per il costo di copiatura, le cose sono diverse: un array di stringhe di lunghezza variabile è un array di puntatori a liste o array dinamici che contengono le stringhe; allora, l'istruzione a[i+1] $\leftarrow$ a[i], copia semplicemente un puntatore e quindi è a costo costante.

In conclusione abbiamo: $\boldsymbol{T_{IS}}$ (n, ls) $\in \Theta(ls\ n^2)$.

Si possono fare i conti direttamente sulla Fig. 1

Per calcolare la complessità di insertion sort non è necessario scrivere il codice; è sufficiente ragionare sulla Fig. 1. Poniamo:

Cf_j = numero di confronti tra elementi dell'array al passo j-mo

M_j = numero di "spostamenti" (copiature, assegnazioni) di elementi al passo j-mo.

Si vede che, per ogni j : $M_j = Cf_j + 1$ sempre

$Cf_j = 1$ nel caso migliore

$Cf_j = j-1$ nel caso peggiore

Poichè j varia da 2 ad n, si ottiene (calcolando una sommatoria analoga a quella precedente) che, per tutto l'algoritmo:

- nel caso migliore, il numero di confronti è **an+b** (a, b costanti, a>0; quindi si fanno $\Theta(n)$ confronti);
- nel caso peggiore, il numero di confronti è $\mathbf{cn^2 + dn + p}$ (c, d, p costanti, c>0; quindi si fanno $\Theta(n^2)$ confronti).

Si capisce, inoltre, che un programma (una procedura) non scema per insertion sort avrà un costo che dipende solo dal numero di confronti (ed il loro costo) e dal numero di spostamenti (ed il loro costo). Noti il numero ed il costo di confronti e spostamenti, si ha subito la complessità dell'algoritmo. Si dice che confronti e spostamenti sono *operazioni dominanti* nell'algoritmo (vedere Operazioni dominanti).

==============================

Insertion sort come procedura (questo va visto dopo la parte " Regole empiriche ....")

Vedere l'algoritmo come "pezzo di programma" non è molto naturale; la cosa piú ovvia è scriverlo come procedura:

procedura insert_sort (aa array [1 .. n] of integer) con aa parametro IN-OUT, quindi per riferimento
  dichiarazioni di j, i, temp
  pseudo-codice come sopra, cambiando a con aa.

I conti di complessità, allora si riferiscono ad una generica chiamata

insert_sort(a) con a array [1 .. n] of integer

I conti non cambiano, salvo che per una cosa: bisogna valutare il costo del "passaggio dei parametri". In questo caso, tale costo è costante, perchè il passaggio per riferimento equivale a "passare alla procedura" un puntatore (in C poi, per i parametri array si passa sempre e comunque solo il puntatore al 1° elemento).