Appendice B

Algoritmi e Complessità

1. Introduzione

Un *algoritmo A* è una procedura passo-passo per risolvere un *problema P*. Un problema *P* è caratterizzato dall'insieme I delle sue *istanze*. L'algoritmo *A* risolve *P* se ne determina la soluzione per ogni sua istanza *I*.

Esempio di algoritmo che calcola il massimo tra *n* interi:

```
procedure MAX_INT(a_1, . . . , a_n: integers)
      max := a_1;
      for i := 2 to n do
            if max < a_i then max := a_i ;
      end_for /* max contiene il massimo */
end_procedure
```

Per poter fornire una soluzione significativa, gli algoritmi devono rispettare alcune proprietà:

- *Input*: L'algoritmo deve avere un input *I* contenuto in un insieme definito I;
- *Output*: Da ogni insieme di valori in input, l'algoritmo produce un insieme di valori in uscita che comprende la soluzione;
- *Determinatezza*: I passi dell'algoritmo devono essere definiti precisamente;
- *Finitezza*: Un algoritmo deve produrre la soluzione in un numero di passi finito (eventualmente molto grande) eseguibili in un tempo finito per ogni possibile input *I* definito su I;
- *Generalità*: L'algoritmo deve essere valido per ogni insieme di dati *I* contenuto in I e non solo per alcuni;
- *Efficienza*: L'algoritmo deve limitare il più possibile l'utilizzo delle risorse di calcolo, in termini di tempo di calcolo e occupazione di memoria.

Problematiche di questo tipo sono trattate dall'analisi della complessità computazionale degli algoritmi:

- Complessità temporale;
- Complessità spaziale.

2. Efficienza

Siamo particolarmente interessati a valutare l'efficienza di un algoritmo, dal punto di vista della sua complessità computazionale dal punto di vista temporale. Un algoritmo potrebbe impiegare poco tempo per risolvere alcune "buone" istanze del problema e invece impiegare molto tempo per risolvere le sue "cattive" istanze.

Come possiamo allora dire se un algoritmo è efficiente o meno? E tra diversi algoritmi efficienti per uno stesso problema, qual è l'algoritmo più efficiente?

Ci sono tre principali approcci per valutare le prestazioni di un algoritmo:

- *Analisi sperimentale*: testare le prestazioni su una classe di istanze;
- *Analisi del caso medio*: si sceglie una distribuzione di probabilità delle istanze del problema e tramite analisi statistica si deriva il tempo di calcolo atteso dell'algoritmo;
- *Analisi del caso peggiore*: fornisce un limite superiore sul numero di passi (operazioni fondamentali) che l'algoritmo può eseguire su qualsiasi istanza del problema.

Le *prime due tecniche* hanno diverse *controindicazioni*:

- dipendenza dal compilatore e dal calcolatore impiegato; l'analisi può richiedere diverso tempo;
- l'analisi dipende dalla distribuzione statistica; è difficile scegliere quella più opportuna; sono richiesti calcoli matematici piuttosto complicati.

L'*analisi del caso peggiore* non ha questi difetti. E' il *metodo più popolare* per valutare l'efficienza di un algoritmo. Il suo svantaggio è che tale valutazione può essere influenzata dalla presenza di istanze cosiddette "patologiche".

- *Analisi del caso peggiore*

L'obiettivo è definire un limite superiore sul tempo necessario per risolvere una qualsiasi istanza del problema. Siccome il tempo è strettamente legato alla performance del computer impiegato, in questa analisi il tempo di calcolo è valutato come numero di operazioni elementari (somme, prodotti, confronti, ecc.) attraverso una funzione di tempo di calcolo che dipende dalla dimensione dell'istanza I del problema P. Tale funzione rappresenta la complessità (nel caso peggiore) dell'algoritmo.

- *Dimensione dell'istanza di un problema*

Il tempo di calcolo di un algoritmo è quindi funzione della *complessità dell'istanza I* del problema P che si vuole risolvere, cioè della sua *dimensione*. La *dimensione* di I è tipicamente valutata come il *numero* di *bit* necessari a *codificare I*, cioè la sua *lunghezza*, *length*(I). Un intero x può essere codificato con $\log x$ bit.

L'analisi del caso peggiore esprime il *tempo* di *calcolo* di un algoritmo con una *funzione* della *lunghezza* dell'*input* (istanza) I.

3. Notazione Big-O

Per valutare la complessità di un algoritmo con l'analisi del caso peggiore si impiega la *notazione Big-O*. Questa permette di rappresentare asintoticamente la crescita di una funzione.

Siano f e g due funzioni tali che $f, g : \Re \rightarrow \Re$. Diremo che $f(n) = O(g(n))$ se esistono due costanti positive c e n_0, tali che $\forall\ n \geq n_0$ si ha che $f(n) \leq c\ |g(n)|$.

Ciò vuol dire che in senso asintotico $f(n)$ cresce *non più velocemente* di $g(n)$.

Esempio:

- $f(n) = n^2 + 2n + 1 = O(n^2)$.
  Infatti $f(n) = n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2 = 4n^2$, quindi considerando $g(n) = n^2$, $c = 4$ e $n_0 = 1$, si

ha $f(n) \leq c\,|g(n)|$, per $n \geq n_0$. Si noti che per questo esempio vale anche il contrario cioè $g(n) \leq c'\,|f(n)|$, $n \geq n'_0$. In tal caso $f(n)$ e $g(n)$ si dicono dello *stesso ordine*.

Altri esempi di Big-O:

- $1 + 2 + \ldots + n = O(n^2)$, infatti: $1 + 2 + \ldots + n \leq n + n + \ldots + n = n^2$, allora $1 + 2 + \ldots + n \leq c\,n^2$, per $n \geq n_0$ con $c = 1$ e $n_0 = 1$;
- $\log(n!) = O(n \log n)$, infatti: $\log(n!) = \log(1 \cdot 2 \cdot 3 \cdot \ldots \cdot n) \leq \log(n \cdot n \cdot n \cdot \ldots \cdot n) = \log n^n = n \log n$ allora $\log(n!) \leq c\,(n \log n)$, con per $n \geq n_0$ $c = 1$ e $n_0 = 1$.

La notazione Big-O indica il *termine più dominante* nella funzione di tempo di calcolo.

Se $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$ allora $f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$.

Ad esempio:

- $n \log n + n^2 = O(n^2)$;
- $n + n = O(n)$.

Se $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$ allora $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.

Altro esempio di Big-O:

- $3n \log(n!) + (n^2 + 3) \log n = O(n^2 \log n)$.

Se $f(n) = a_q n^q + a_{q-1} n^{q-1} + \ldots + a_0$, con $a_q, a_{q-1}, \ldots, a_0$ numeri reali, allora $f(n) = O(n^q)$.

4. Notazione Big-Θ e Big-Ω

Si noti che se $f(n) = O(g(n))$, non è detto che valga il contrario. Ad esempio:

- $10\,n^2 = O(n^q)$, per ogni $q \geq 2$, ma $n^3 \neq O(n^2)$.

Ovviamente siamo interessati alla valutazione (bound) più stretta.

Se $f(n) = O(g(n))$ e $g(n) = O(f(n))$, allora la valutazione asintotica della crescita di $f(n)$ attraverso $g(n)$ è stretta e si scrive $f(n) = \Theta(g(n))$ e $g(n) = \Theta(f(n))$.

Ad esempio:

- $10\,n^2 = \Theta(n_2)$;
- un polinomio $P^q(n)$, di grado q, è $\Theta(n^q)$.

Dire che $f(n) = \Theta(g(n))$ significa che $f(n)$ cresce *asintoticamente* con la *stessa velocità* di $g(n)$.

Esempi di Big-Θ:

- $1 + 2 + \ldots + n = O(n^2)$, ma è anche $\Theta(n^2)$, perché $1 + 2 + \ldots + n = \sum_{i=1}^{n} i = n(n + 1)/2$ e $n^2 = O(n(n + 1)/2)$;
- $2n \log(n!) + 10n^4 + 10n^2 = O(n^4)$, ma più precisamente $2n \log(n!) + 10n^4 + 10n^2 = \Theta(n^4)$;
- $2n^2 + 5n(\log n^3 + 1) = O(n^2)$, ma più precisamente $2n^2 + 5n(\log n^3 + 1) = \Theta(n^2)$;
- $a_q n^q + a_{q-1} n^{q-1} + \ldots + a_0 = O(n^q)$, ma più precisamente $a_q n^q + a_{q-1} n^{q-1} + \ldots + a_0 = \Theta(n^q)$.

Con Big-Ω si specifica invece un limitazione inferiore sulla velocità di crescita asintotica di una funzione.

Siano f e g due funzioni tali che $f, g : \Re \rightarrow \Re$. Diremo che $f(n) = \Omega(g(n))$ se esistono due costanti positive c e n_0, tali che $\forall\ n \geq n_0$ si ha che $c\ |g(n)| \leq f(n)$.

Ciò vuol dire che in senso asintotico $f(n)$ cresce *almeno* tanto *velocemente* quanto $g(n)$.

Ad esempio:

- $10\ n^2 = \Omega(n^q)$, per ogni $q \leq 2$, ma $n \neq \Omega(n^2)$.

Le definizioni di Big-O e Big-Ω permettono di dare una definizione alternativa a Big-Θ.

Siano f e g due funzioni tali che $f, g : \Re \rightarrow \Re$. Diremo che $f(n) = \Theta(g(n))$, se $f(n) = \Omega(g(n))$ e $f(n) = O(g(n))$.

5. Complessità di un Algoritmo

Un *algoritmo* è detto *operare* in *tempo* $O(f(n))$ se esistono due costanti positive c e n_0, tali che $\forall\ n \geq n_0$ il tempo di calcolo richiesto dall'algoritmo è *al più* pari a $c\ |f(n)|$.

In sostanza, un algoritmo è detto operare in tempo $O(f(n))$ se il suo tempo di calcolo cresce in senso asintotico *non più velocemente* di $f(n)$.

Si noti che:

- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^a) < O(b^n) < O(n!)$, con $1 < a, b < n$.

La seguente figura riporta stime di tempi di calcolo ipotizzando che il calcolatore esegua un operazione base su un bit in 10^{-9} sec.

Dimensione del problema	Numero di operazioni su bit eseguite					
n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	$3 \cdot 10^{-9}$ sec	10^{-8} sec	$3 \cdot 10^{-8}$ sec	10^{-7} sec	10^{-6} sec	$3 \cdot 10^{-3}$sec
10^2	$7 \cdot 10^{-9}$ sec	10^{-7} sec	$7 \cdot 10^{-7}$ sec	10^{-5} sec	$4 \cdot 10^{13}$ anni	$*$
10^3	$1 \cdot 10^{-8}$ sec	10^{-6} sec	$1 \cdot 10^{-5}$ sec	10^{-3} sec	$*$	$*$
10^4	$1.3 \cdot 10^{-8}$ sec	10^{-5} sec	$1 \cdot 10^{-4}$ sec	10^{-1} sec	$*$	$*$
10^5	$1.7 \cdot 10^{-8}$ sec	10^{-4} sec	$2 \cdot 10^{-2}$ sec	10 sec	$*$	$*$
10^6	$7 \cdot 10^{-8}$ sec	10^{-3} sec	$3 \cdot 10^{-2}$ sec	17 min	$*$	$*$

Stime di tempi di calcolo supponendo che il calcolatore esegua un'operazione base su un bit in 10^{-9} sec

La stima Big-O non permette di avere una valutazione effettiva del tempo di calcolo impiegato da un algoritmo: questo dipende dalla velocità v del calcolatore che usiamo. Ci permette però di valutare come le prestazioni del calcolatore impattano su tali tempi.

Supponiamo di confrontare i tempi di calcolo impiegati da un algoritmo su due calcolatori CPU1 e CPU2, il secondo 100 volte più veloce ($v_2 = 100\ v_1$).

Assumiamo che in certo tempo (es. 1 sec) CPU1 risolva un'istanza di dimensione n_1 e vogliamo stimare quale possa essere la dimensione n_2 dell'istanza che CPU2 è in grado di risolvere nello stesso tempo.

Se l'algoritmo opera in tempo $O(n^2)$, allora possiamo stimare che vale la seguente proporzione $(n_2)^2/(n_1)^2 = v_2/v_1 = 100$. Quindi, nello stesso tempo, CPU2 dovrebbe risolvere un'istanza di dimensione $n_2 = 10\ n_1$.

Se l'algoritmo opera in tempo $O(2^n)$, allora possiamo stimare che vale la seguente proporzione $2^{n_2}/2^{n_1} = v_2/v_1 = 100$. Quindi, nello stesso tempo, CPU2 dovrebbe risolvere un'istanza di dimensione $n_2 = n_1 + \log 100 \cong n_1 + 7$.

La notazione Big-O specifica un limite superiore sulla complessità di un algoritmo. La notazione Big-Ω specifica un limite inferiore sulla complessità di un algoritmo (sempre in riferimento al caso peggiore).

Un *algoritmo* è detto *operare* in *tempo* $\Omega(f(n))$ se esistono due costanti positive c' e n'_0, tali che $\forall\ n \geq n'_0$ il tempo di calcolo richiesto dall'algoritmo è *almeno* pari a $c'\,|f(n)|$

In sostanza, un algoritmo è detto operare in tempo $\Omega(f(n))$ se il suo tempo di calcolo cresce in senso asintotico *non meno velocemente* di $f(n)$.

La notazione Big-Θ fornisce invece sia un limite superiore che inferiore sulla complessità dell'algoritmo.

Un *algoritmo* è detto *operare* in *tempo* $\Theta(f(n))$ se opera sia in tempo $O(f(n))$ che in tempo $\Omega(f(n))$.

In sostanza, un algoritmo è detto operare in tempo $\Theta(f(n))$ se il suo tempo di calcolo, nel caso peggiore, cresce in senso asintotico *velocemente tanto quanto* $f(n)$.

La seguente tabella riporta degli esempi di classi di complessità di un algoritmo:

Complessità	Terminologia
$O(1)$	Complessità costante
$O(\log n)$	Complessità logaritmica
$O(n)$	Complessità lineare
$O(n \log n)$	Complessità $n \log n$
$O(n^a)$	Complessità polinomiale
$O(a^n)$, con $a > 1$	Complessità esponenziale
$O(n!)$	Complessità fattoriale

Un algoritmo con complessità non superiore a $O(n^a)$, dove n è la dimensione dell'input e $a > 1$, è detto *algoritmo (tempo-)polinomiale*, gli altri sono detti *algoritmi (tempo-)esponenziale*.

Un algoritmo polinomiale viene spesso denominato come *algoritmo efficiente*.

6. Algoritmi di Ricerca

Gli *algoritmi di ricerca* hanno una importanza di particolare interesse. Di seguito è riportato l'algoritmo di *ricerca lineare* (o *sequenziale*) in una lista (di interi).

```
procedure LINEAR_SEARCH(x: integer; a_1, ..., a_n: integers)
      i := 1;
      while (i ≤ n) and (x ≠ a_i) do
            i := i + 1;
      end_while
      if i ≤ n then posizione := i;
      else posizione := 0;
      /* posizione contiene la posizione di x */
end_procedure
```

La complessità dell'algoritmo **LINEAR_SEARCH** è $O(n)$:

- Prima del ciclo `while` c'è un assegnamento ad una variabile. All'interno del ciclo `while` vengono effettuati due confronti: uno per verificare se si è arrivati alla fine della lista e l'altro per confrontare x con un termine della lista; infine all'interno del `while` viene fatta una somma. Successivamente viene eseguito un confronto fuori dal ciclo. Considerando il caso peggiore, ovvero quello in cui l'elemento non è contenuto nella lista, sono eseguite n iterazioni nel ciclo `while` più una nella quale si verifica la fine della lista, quindi $2n + 1$ confronti che sommati al confronto fuori dal ciclo comportano un totale di $2n + 2$ confronti. A questi vanno sommati le n operazioni di somma all'interno del `while` + una prima e dopo del `while`. In totale $2n + 4$ operazioni fondamentali. Quindi la ricerca lineare richiede $O(n)$ operazioni elementari. In particolare è facile mostrare che queste sono $\Theta(n)$.

Si noti invece che nel caso *migliore* l'algoritmo esegue $O(1)$ operazioni.

Di seguito è riportato l'algoritmo di *ricerca binaria* in una lista *ordinata* (di interi).

```
procedure BINARY_SEARCH(x: integer; a_1, ..., a_n: non-
decreasing integers)
      i := 1; j := n ;
      while i < j do
            m := ⌊(i + j)/2⌋ ;
            if x > a_m then i := m + 1;
            else j := m;
      end_while
      if x = a_i then posizione := i;
      else posizione := 0 ;
      /* posizione contiene la posizione di x */
end_procedure
```

La complessità dell'algoritmo **BINARY_SEARCH** è $O(\log n)$:

- Contiamo per semplicità solo i confronti. Assumiamo che la lista contenga n interi con $2k - 1 < n \leq 2k$ (quindi, $k = \lceil \log n \rceil$). Ad ogni passo (iterazione) dell'algoritmo, le variabili i e j sono confrontate per vedere se la lista ristretta ha più di un termine e, se $i < j$, viene eseguito un confronto per determinare se x è maggiore del termine mediano della lista in considerazione. Al primo passo, la ricerca è limitata ad al più $2k$ termini e vengono effettuati due confronti (uno per la condizione del ciclo `while` e l'altro all'interno del ciclo); ad ogni passo

successivo vengono eseguiti due confronti su di una lista che è grande la metà di quella del passo precedente. Quindi il numero delle iterazioni all'interno del ciclo `while` è al più pari a k. Per l'uscita dal ciclo while viene effettuato un ulteriore confronto. Infine usciti dal ciclo `while` viene eseguito un ulteriore confronto e quindi complessivamente saranno stati eseguiti al più $2k + 2$ confronti, ovvero $2\lceil \log n \rceil + 2$ confronti. Quindi, l'algoritmo di ricerca binaria richiede $O(\log n)$ confronti.

Di seguito è riportato l'algoritmo di *ricerca lineare* (o *sequenziale*) in una matrice $m \times n$ (di interi).

```
procedure MATRIX_SEARCH(x: integer; a11, …, a1n, …, am1, …,
amn: integers)
      i := 1; j := 1;
      while (i ≤ n) and (x ≠ aij) do
            if j < n then j := j + 1;
            else
                  i := i + 1;
                  j := 1;
            end_if
      end_while
      if i ≤ n then posizione := (i, j);
      else posizione := (0, 0);
      /* posizione contiene la posizione di x */
end_procedure
```

La complessità dell'algoritmo **`MATRIX_SEARCH`** è $O(mn)$.

- Esaminiamo per semplicità solo il numero dei confronti. All'interno del ciclo `while` vengono effettuati tre confronti: uno per verificare se si è terminata la scansione della matrice, il secondo per confrontare x con un termine della matrice e il terzo per verificare se si è terminata la scansione della corrente riga. Per l'uscita dal ciclo viene effettuato un ulteriore confronto. Infine viene eseguito un confronto fuori dal ciclo. Considerando il caso peggiore, ovvero quello in cui l'elemento non è contenuto nella matrice, sono eseguite $m \cdot n$ iterazioni nel ciclo `while`, quindi $3mn$ confronti che sommati al confronto per uscire dal ciclo e a quello fuori dal ciclo comportano un totale di $3mn + 2$ confronti. Quindi la ricerca lineare all'interno della matrice richiede $O(mn)$ confronti.

Di seguito è riportato l'algoritmo di *ricerca binaria* in una matrice $m \times n$ in cui ogni riga contiene una lista *ordinata* (di interi).

```
procedure ORDERED_MATRIX_SEARCH(x: integer; (a11, …, a1n),
…, (am1, …, amn): non-decr. lists of integers)
      i := 0; row := 0;  column := 0;
      repeat
            i := i + 1;  h := 1;  j := n;
            while (h < j) do
                  m := ⌊(h + j)/2⌋;
                  if x > aim then h := m + 1;
                  else j := m;
            end_while
            if x = aih then row := i;  column := h;  i := n;
      until (i ≥ m)
      /* row e column definiscono posizione di x */
end_procedure
```

La complessità dell'algoritmo **ORDERED_MATRIX_SEARCH** è $O(m \log n)$.

- Esaminiamo per semplicità solo il numero dei confronti. L'algoritmo è costituito da un ciclo `while` annidato all'interno di un ciclo `repeat-until`. Ad ogni iterazione del ciclo `repeat-until` esterno viene eseguito un ciclo `while` interno nel quale si effettua una ricerca binaria (analoga al **BINARY_SEARCH**) all'interno di una certa riga della matrice costituita da una lista di n interi non decrescenti, che pertanto richiede $O(\log n)$ confronti; a queste va aggiunto un confronto in cui si determina se sono state scandite tutte le righe della matrice. Siccome il ciclo esterno viene ripetuto al più m volte allora il numero complessivo dei confronti è $m(O(\log n) + O(1))$, cioè $O(m \log n)$.

6. Algoritmi di Ordinamento

Gli *algoritmi di ordinamento* consentono di ordinare in senso non-decrescente (oppure non-crescente) una lista di n elementi (ad es. interi).

Ci sono diversi algoritmi di ordinamento. Ad esempio quelli basati su confronti-scambi sono:

- **SELECTION_SORT**;
- **BUBBLE_SORT**;
- **INSERTION_SORT**;

} $O(n^2)$

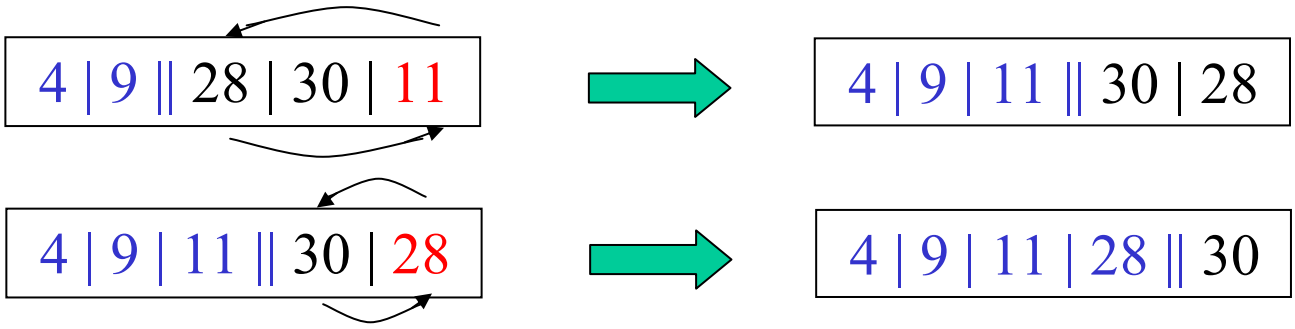
- **QUICK_SORT**;
- **MERGE_SORT**;
- **HEAP_SORT**;

} $O(n \log n)$

- …

Impiegano metodi differenti gli uni dagli altri e alcuni sono più efficienti degli altri.

L'algoritmo **SELECTION_SORT** opera su una sequenza non ordinata come se fosse composta da due sotto-sequenze: la prima *ordinata* e costituita da elementi non più grandi dell'altra che *non è ordinata*. Nella sequenza non ordinata, ricerca l'elemento *minimo* e lo scambia con l'elemento in prima posizione. Questo diventa l'ultimo elemento della sotto-sequenza ordinata (vedi figura).



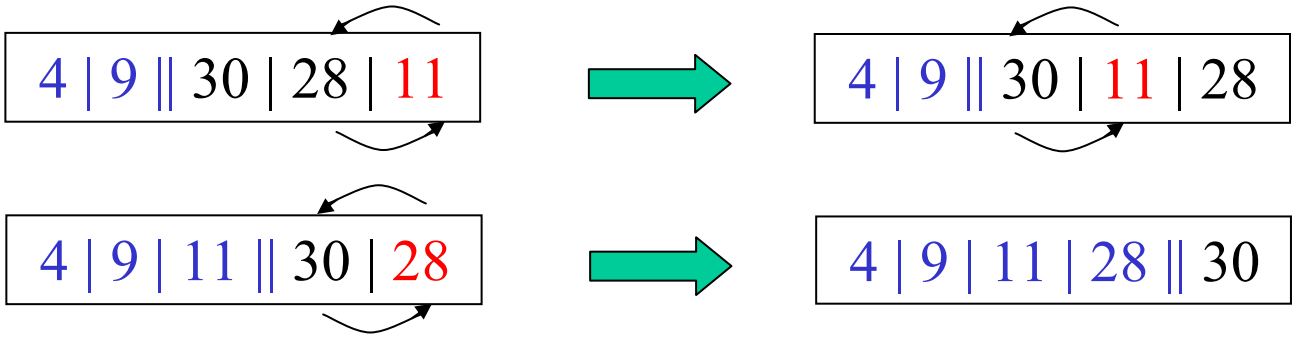
Quando la sotto-sequenza non ordinata contiene un solo elemento l'ordinamento è completato.

Di seguito è riportato l'algoritmo di ordinamento **SELECTION_SORT**.

```
procedure SELECTION_SORT(a_1, ..., a_n: integers)
     for j := 1 to n – 1 do
          h := j;
          for i := j + 1 to n do
               if a_h > a_i then h := i;
          end_for // h è la posizione del min
          if h ≠ j then // scambia di posto a_h e a_j
               x := a_j;
               a_j := a_h;
               a_h := x;
          end_if
     end_for
// gli n interi sono ordinati in modo non-decr.
end_procedure
```

L'algoritmo **SELECTION_SORT** opera in tempo $O(n^2)$ effettuando $O(n)$ scambi.

Anche l'algoritmo **BUBBLE_SORT** opera su una sequenza non ordinata come se fosse composta da due sotto-sequenze: la prima *ordinata* e costituita da elementi non più grandi dell'altra che *non è ordinata*. Sulla sequenza non-ordinata, partendo dal fondo, fa risalire il minimo confrontando ed eventualmente scambiando coppie di elementi adiacenti (vedi figura).



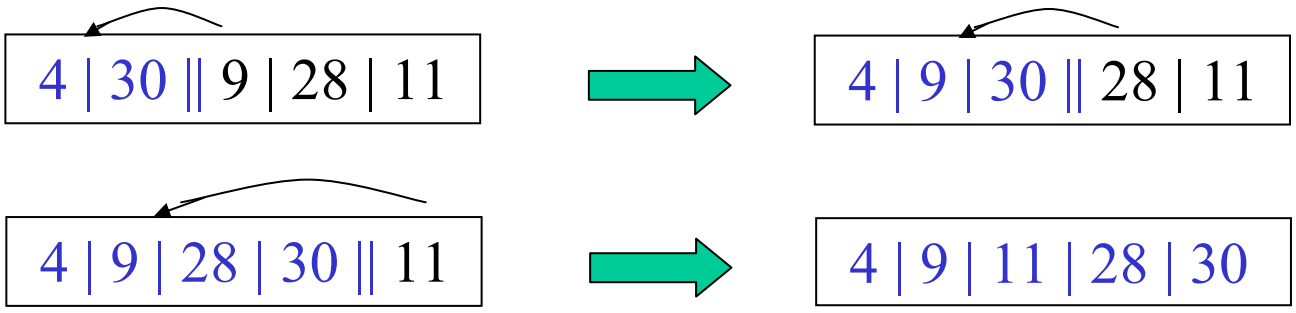
Quando nella sotto-sequenza non ordinata non si eseguono scambi (è ordinata) l'algoritmo termina.

Di seguito è riportato l'algoritmo di ordinamento **BUBBLE_SORT**.

```
procedure BUBBLE_SORT(a_1, ..., a_n: integers)
     repeat
          scambio := false; j := 2;
          for i := n downto j do
               if a_{i-1} > a_i then //scambia a_{i-1} e a_i
                    x := a_{i-1}; a_{i-1} := a_i; a_i := x;
                    scambio := true;
               end_if
          end_for // il minimo è in pos. j – 1
          j := j + 1;
     until (scambio = false)
// gli n interi sono ordinati in modo non-decr.
end_procedure
```

L'algoritmo **BUBBLE_SORT** opera in tempo $O(n^2)$, ma è mediamente più efficiente di **SELECTION_SORT**, anche se gli scambi sono $O(n^2)$.

Anche l'algoritmo **INSERTION_SORT** opera su una sequenza non ordinata come se fosse composta da due sotto-sequenze: la prima *ordinata* e l'altra *no*. Dalla sequenza non-ordinata viene prelevato un elemento (ad esempio il primo) e viene *inserito* nella posizione corretta nella sotto-lista ordinata (vedi figura).



Quando la sotto-sequenza non ordinata è vuota l'algoritmo termina.

Di seguito è riportato l'algoritmo di ordinamento **INSERTION_SORT**.

```
procedure INSERTION_SORT(a_1, ..., a_n: integers)
     for j := 2 to n do
          x := a_j;
          i := j – 1;
          while (i > 0) and (a_i > x) do
               //shift adiacenti
               a_{i+1} := a_i;
               i := i – 1;
          end_while
          a_{i+1} := x;
     end_for
// gli n interi sono ordinati in modo non-decr.
end_procedure
```

L'algoritmo **INSERTION_SORT** opera in tempo $O(n^2)$, effettuando $O(n^2)$ scambi. Sperimentalmente è fra i più veloci su liste di pochi elementi. Potremmo quindi pensare che un'analisi più raffinata della complessità possa svelare un valore migliore di $O(n^2)$ per questo algoritmo. Tuttavia non è così. Infatti, contando il numero di scambi che l'algoritmo effettua se la lista di elementi è ordinata in modo contrario (strettamente decrescente), si ha che il numero degli scambi è $\sum_{j=2}^{n} (j-1) = \sum_{j=1}^{n-1} j = n(n-1)/2 = \Theta(n^2)$ scambi.

Si dimostra che un qualsiasi algoritmo per confronti-scambi effettua $\Omega(n \log n)$ confronti. Quindi dal punto di vista asintotico i migliori algoritmi di ordinamento di tale categoria sono quelli che operano in tempo $O(n \log n)$. La seguente tabella riassume l'analisi per vari algoritmi di ordinamento.

algoritmo	**migliore**	**medio**	**peggiore**	**scambi**	**mem. add.**	**in place**
SELECTION_SORT	$O(n2)$	$O(n2)$	$\Theta(n2)$	$O(n)$	$O(1)$	Sì
BUBBLE_SORT	$O(n)$	$O(n2)$	$\Theta(n2)$	$O(n2)$	$O(1)$	Sì
INSERTION_SORT	$O(n)$	$O(n2)$	$\Theta(n2)$	$O(n2)$	$O(1)$	Sì
QUICK_SORT	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$*	$O(n \log n)$*	$O(\log n)$	Sì
MERGE_SORT	$O(n \log n)$	$O(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$	No
HEAP_SORT	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	Sì