

Introduzione alla correttezza degli algoritmi

Massimo Benerecetti

Università di Napoli, “Federico II”, Napoli

Questo documento contiene una breve introduzione alle dimostrazioni di correttezza degli algoritmi. Nella prima sezione del documento verrà fornito un richiamo alla tecnica di dimostrazione per induzione. La seconda sezione illustra, tramite semplici esempi, le problematiche connesse alle dimostrazioni di correttezza degli algoritmi. Le sezioni 3 e 4 descrivono le tecniche di dimostrazione per algoritmi ricorsivi e iterativi, rispettivamente. In sezione 5 verrà trattato il problema della dimostrazione di terminazione per algoritmi iterativi e ricorsivi. Infine, la sezione 6 presenta un’applicazione degli argomenti trattati in questa dispensa alla dimostrazione completa della correttezza dell’algoritmo di ordinamento InsertionSort.

1 Principio di Induzione

Il *Principio di Induzione Matematica* è una importante tecnica di dimostrazione, particolarmente adatta a dimostrare proprietà universali che in cui intervengano numeri interi.

Principio di Induzione Semplice. Sia n_0 un intero e sia $P(n)$ un enunciato (una proprietà) che ha senso per ogni $n \geq n_0$. Se:

1. $P(n_0)$ è vero
2. per ogni $n > n_0$, $P(n)$ vero implica $P(n + 1)$ vero

allora l’enunciato $P(n)$ è vero per tutti i numeri naturali $n \geq n_0$.

Esempio 1. Mostriamo che che la somma dei primi n numeri interi naturali è $\frac{n(n+1)}{2}$, cioè mostriamo che vale l’uguaglianza:

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

per ogni intero $n \geq 1$.

In questo caso $P(n)$ è $\sum_{i=1}^n i = \frac{n(n+1)}{2}$, mentre $P(n+1)$ è $\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$. Vediamo che:

- $P(1)$ è vera: infatti il primo membro vale 1 e il secondo membro $\frac{1(1+1)}{2} = 1$;
- supponendo vero $P(n)$, si ottiene che $P(n+1)$ è vero. Scomponendo la sommatoria opportunamente e utilizzando la verità dell'ipotesi induttiva $P(n)$ secondo cui è $\sum_{i=1}^n i = \frac{n(n+1)}{2}$, otteniamo:

$$\sum_{i=1}^{n+1} i = \left[\sum_{i=1}^n i \right] + (n+1) = \left[\frac{n(n+1)}{2} \right] + (n+1) = \frac{[n(n+1) + 2(n+1)]}{2} = \frac{(n+1)(n+2)}{2}$$

che è la tesi.

Pertanto la proprietà è vera per tutti gli $n \geq 1$.

Esempio 2. Dimostrare che la somma dei primi n numeri naturali dispari (non nulli) è n^2 , cioè che vale l'uguaglianza:

$$\sum_{i=1}^n (2i-1) = n^2$$

per ogni intero $n \geq 1$.

- $P(1)$ è vero: infatti il primo membro è $2 \cdot 1 - 1 = 1$, mentre il secondo membro è $1^2 = 1$.
- Supposto $P(n)$ vero, cioè supposto che valga l'uguaglianza $\sum_{i=1}^n (2i-1) = n^2$, proviamo

$$P(n+1), \text{ cioè l'uguaglianza } \sum_{i=1}^{n+1} (2i-1) = (n+1)^2$$

Scomponendo opportunamente la sommatoria e utilizzando l'ipotesi induttiva, otteniamo:

$$\sum_{i=1}^{n+1} (2n-1) = \left[\sum_{i=1}^n (2n-1) \right] + 2 \cdot (n+1) - 1 = n^2 + 2 \cdot (n+1) - 1 = n^2 + 2n + 1 = (n+1)^2$$

che è la tesi.

Pertanto la proprietà $P(n)$ è vera per tutti gli $n \geq 1$.

Principio di Induzione Forte. Sia n_0 un intero e sia $P(n)$ un enunciato (una proprietà) che ha senso per ogni $n \geq n_0$. Se:

1. $P(n_0)$ è vero
2. per ogni $n > n_0$, $P(k)$ vero per ogni $n_0 \leq k < n$ implica $P(n)$ vero

allora l'enunciato $P(n)$ è vero per tutti i numeri naturali $n \geq n_0$.

Esempio 3. Vogliamo provare che ogni numero naturale $n \geq 2$ è prodotto di numeri primi.

- $P(2)$: l'affermazione è vera per $n = 2$, poiché esso stesso è un numero primo.
- Consideriamo un generico $n > 2$ e supponiamo che tutti i numeri da 2 fino ad $n-1$ siano prodotti di numeri primi. Cerchiamo di provare che, allora, anche n lo deve essere.

Infatti, se n è esso stesso un numero primo, allora l'affermazione è banalmente vera. Se, invece, n non è primo, allora, per definizione di numero primo, si può esprimere come prodotto di due numeri interi più piccoli di n , cioè $n = k_1 \cdot k_2$, con $k_1 < n$ e $k_2 < n$.

Essendo sia k_1 che k_2 interi minori di n , per ipotesi induttiva sono entrambi esprimibili come prodotto di numeri primi. Cioè, $k_1 = p_1 \cdot \dots \cdot p_r$, con p_i primo per ogni $1 \leq i \leq r$, e $k_2 = q_1 \cdot \dots \cdot q_s$, con q_j primo per ogni $1 \leq j \leq s$.

Quindi, sostituendo, avremo che:

$$n = p_1 \cdot \dots \cdot p_r \cdot q_1 \cdot \dots \cdot q_s$$

che dimostra che anche n è prodotto di numeri primi.

Concludiamo, allora, che l'affermazione è vera per ogni intero $n \geq 2$.

2 Correttezza degli algoritmi

In questo documento diremo che un algoritmo è *corretto* se il suo valore di output soddisfa una determinata condizione (detta *postcondizione*) per ogni istanza di input che, a sua volta, soddisfa una determinata condizione (detta *precondizione*). Intuitivamente, la postcondizione è una condizione che esprime formalmente la nozione informale di “risultato corretto” dell’algoritmo, mentre la precondizione è la condizione che identifica le istanze di input “ragionevoli”, cioè quelle per le quali ci si aspetta che l’algoritmo fornisca risultati significativi. Sia la precondizione che la postcondizione sono fortemente correlate col problema specifico che l’algoritmo è inteso risolvere.

Ad esempio, il seguente semplice algoritmo è pensato per calcolare la somma tra due numero reali.

Algorithm ALG-1(X,Y)

```
1 S = X;  
2 S = S + Y;  
3 return S
```

In tal caso la correttezza può essere espressa dalle seguenti condizioni:

Correttezza $\left\{ \begin{array}{l} \textit{Precondizione: } X \text{ e } Y \text{ numeri reali.} \\ \textit{Postcondizione: } \text{Se } z = \text{ALG-1}(X,Y), \text{ allora } z = X+Y. \end{array} \right.$

La formulazione completa della correttezza di un algoritmo, detta *correttezza totale* viene formulata nel nel seguente modo.

Correttezza totale: L’algoritmo termina per tutte le istanze di input che soddisfano la precondizione e restituisce il risultato corretto.

La dimostrazione di correttezza di un algoritmo viene, tipicamente, suddivisa nella dimostrazione delle due seguenti proprietà:

- *Correttezza parziale:* Se l’algoritmo termina, esso restituirà il risultato corretto (che, cioè, soddisfa la postcondizione);
- *Terminazione:* L’algoritmo termina per tutte le istanze di input che soddisfano la precondizione.

È evidente che la dimostrazione delle due proprietà sopra riportate fornisce la dimostrazione della correttezza totale.

Una dimostrazione di correttezza consiste essenzialmente in una dimostrazione matematica.

Il primo passo è, dunque, quello di tradurre le operazioni eseguite dall'algoritmo in opportune relazioni matematiche tra le variabili coinvolte, che ne descrivano adeguatamente l'effetto.

Consideriamo l'esempio dell'algoritmo ALG-1(X,Y) riportato sopra. Le uniche operazioni eseguite dall'algoritmo sono assegnamenti del valore di un'espressione matematica ad una variabile. Appare, quindi, naturale rappresentare ogni variabile dell'algoritmo con una variabile matematica. Ad esempio, possiamo rappresentare le variabili X, Y e S con le variabili matematiche x, y e s (intese come variabili su un dominio reale), rispettivamente. Sebbene possa apparire naturale tradurre l'assegnamento $S = X$ con la relazione di uguaglianza $s = x$, non così è per il secondo assegnamento. Se volessimo, infatti, rappresentare l'assegnamento $S = S + Y$ con la relazione $s = s + y$, otterremmo una descrizione matematica errata dell'operazione di assegnamento (non esiste, infatti, alcuna coppia di valori reali per s e y , eccettuato il caso banale $y = 0$, tale che la loro somma sia uguale ad uno dei due numeri). In generale, mentre l'uguaglianza è una relazione che può sussistere tra i valori di variabili o espressioni (e che, quindi, non ha alcun effetto sul loro valore), l'assegnamento è un'operazione che modifica il valore di una variabile. È chiaro, infatti, che le occorrenze della variabile S a destra e a sinistra dell'assegnamento svolgono ruoli diversi e assumono valori differenti durante l'esecuzione dell'algoritmo. In particolare, l'occorrenza della variabile S a destra dell'assegnamento si riferisce al valore che la variabile S assume prima dell'assegnamento, mentre quella a sinistra si riferisce al valore che la variabile S assume dopo dell'assegnamento. Al fine di poter rappresentare i differenti valori che la stessa variabile dell'algoritmo può assumere durante l'esecuzione è, quindi, necessario rappresentare la stessa variabile con più variabili matematiche. Le variabili matematiche associate alla stessa variabile dell'algoritmo intendono rappresentare i possibili stati (valori) che essa può assumere durante l'esecuzione. Tornando all'esempio dell'algoritmo ALG-1(X,Y), assoceremo alla variabile S tre variabili matematiche, che chiameremo s_0, s_1 e s_2 , che rappresentano il valore che S assume inizialmente, dopo il primo assegnamento e dopo il secondo assegnamento, rispettivamente. Poiché né X né Y compaiono a sinistra di un assegnamento, è sufficiente rappresentarle ciascuna di esse con una singola variabile matematica (x e y). Possiamo ora tradurre l'algoritmo ALG-1(X,Y) con le seguenti relazioni matematiche:

$$s_1 = x$$

$$s_2 = s_1 + y$$

Poiché il valore di ritorno dell'algoritmo è proprio il valore di S dopo il secondo assegnamento, il valore finale dell'algoritmo è quindi rappresentato dalla variabile matematica s_2 . Sfruttando le proprietà della relazione di uguaglianza (in particolare, il fatto che se sussistono le relazioni $a = b$ e $c = a + d$ allora sussiste anche la relazione $c = b + d$), otteniamo che, a partire dalle relazioni che descrivono l'algoritmo segue anche la relazione $s_2 = x + y$. In altre parole, il valore ritornato dall'algoritmo è proprio la somma dei valori forniti in input. Ciò dimostra la verità della proprietà di correttezza sopra individuata.

Consideriamo ora il seguente algoritmo.

Algorithm ALG-2(X)

```

1  if ( $X < 0$ ) then
2      $S = -X$ 
3  else
4      $S = X$ 
5  return  $S$ 

```

Correttezza $\left\{ \begin{array}{l} \textit{Precondizione: } X \text{ numero reale.} \\ \textit{Postcondizione: } \text{Se } z = \text{ALG-2}(X), \text{ allora } z = |X|. \end{array} \right.$

Seguendo le considerazioni sopra riportate, rappresenteremo con la variabile x la variabile X e con le variabili s_0 e s_1 la variabile S . In base alla semantica del costrutto **if ... then ... else ...**, l'assegnamento $S = -X$ viene eseguito quando il valore di X è negativo, mentre l'assegnamento $S = X$ viene eseguito quando il valore di X è positivo o nullo. Possiamo, quindi, tradurre l'algoritmo con le seguenti relazioni (dove \rightarrow è il simbolo che denota l'implicazione logica):

$$\begin{aligned} (x < 0) &\rightarrow (s_1 = -x) \\ (x \geq 0) &\rightarrow (s_1 = x) \end{aligned}$$

Dalle relazioni sopra riportate segue immediatamente la seguente:

$$s_1 = \begin{cases} -x & \text{se } x < 0 \\ x & \text{se } x \geq 0 \end{cases}$$

Per la definizione di valore assoluto, questo implica $s_1 = |x|$. Essendo s_1 la rappresentazione del valore di S al termine dell'algoritmo, segue che il valore ritornato dall'algoritmo è proprio il valore assoluto del dato X in input.

3 Correttezza di algoritmi ricorsivi

Una delle tecniche più utilizzate per dimostrare la correttezza di un algoritmo ricorsivo è quella di sviluppare, sfruttando la struttura induttiva dell'algoritmo, una dimostrazione per induzione. Tipicamente, la dimostrazione procede per induzione sulla dimensione dell'istanza di input ricevuta dall'algoritmo. In altre parole, si vuole dimostrare, per ogni valore $i \geq 0$ della dimensione dell'input, che per un'istanza di input arbitraria di dimensione i che soddisfi la precondizione, l'algoritmo, se termina (correttezza parziale), restituisce il risultato desiderato (postcondizione). Per completare la dimostrazione di correttezza (correttezza totale) è poi sufficiente dimostrare che l'algoritmo termina per ogni istanza in input che soddisfa la precondizione.

Consideriamo, a titolo di esempio, il seguente algoritmo:

Algorithm BINSEARCH(A[],K,P,R)

```

1  if P < R then
2    Q = ⌊ (P+R)/2 ⌋
3    if K ≤ A[Q] then
4      return BinSearch(A[],K,P,Q)
5    else
6      return BinSearch(A[],K,Q+1,R)
7  else
8    if P = R and A[P] = K then
9      return P
10  else
11    return -1

```

Correttezza $\left\{ \begin{array}{l} \textit{Precondizione:} \text{ L'array } A \text{ è ordinato. Quindi, posto } n = \textit{length}(A), \\ \text{ l'assunzione è esprimibile come segue: } \forall 1 \leq x < n. A[x] \leq A[x+1]. \\ \textit{Postcondizione:} \text{ Se } k \text{ è un elemento presente nell'array } A \text{ (} k \in A \text{) e} \\ \text{ } z = \text{BINSEARCH}(A,K,1,N) \text{ , allora } k = A[z]. \end{array} \right.$

Iniziamo con l'individuare le variabili matematiche che utilizzeremo per rappresentare le variabili dell'algoritmo. Poiché ogni variabile dell'algoritmo viene assegnata al massimo una volta durante l'esecuzione della stessa chiamata ricorsiva, sarà sufficiente utilizzare una singola variabile matematica per ciascuna variabile dell'algoritmo. La variabile a (che, con un abuso di notazione, nel seguito verrà intesa per semplicità a volte come insieme a volte come sequenza) sarà utilizzata per rappresentare l'array A . Per convenzione, indicheremo con $a[p..r]$ l'insieme di elementi della sequenza a in essa contenuti tra la posizione p e la posizione r . Le variabili p, r, k e q verranno, invece, utilizzate di seguito per rappresentare le variabili P, R, K e Q , rispettivamente.

Al fine di dimostrare la proprietà di correttezza sopra riportata, dimostreremo la seguente proprietà più generale:

Correttezza $\left\{ \begin{array}{l} \textit{Precondizione:} \text{ } P \text{ e } R \text{ sono interi tali che } 0 \leq P, R \leq \textit{length}(A) \text{ e l'array} \\ \text{ } A \text{ è ordinato. L'assunzione è esprimibile come segue: } \forall P \leq x < \\ \text{ } R. A[x] \leq A[x+1]. \\ \textit{Postcondizione:} \text{ Se } k \text{ è un elemento presente in } A[P..R] \text{ (} k \in A[P..R] \text{) e} \\ \text{ } z = \text{BINSEARCH}(A,K,P,R) \text{ , allora } k = A[z]. \end{array} \right.$

Nel caso specifico, dimostreremo che, posto $i = r - p + 1$ (cioè la dimensione della porzione dell'array compresa tra p e r):

$$\forall i \geq 0, \text{ se } k \in a[p..r] \text{ e } z = \text{BINSEARCH}(a, k, p, r), \text{ allora } k = a[z].$$

La dimostrazione procede come segue:

- *Caso base 1* per $i = 0$. In questo caso $p < r$. La condizione dell'**if...then...else** è quindi falsa e l'algoritmo procede ad eseguire il ramo **else** (riga 7). Poiché $p \neq r$, la condizione a riga 8 è falsa e viene, quindi, eseguita la riga 11. Il valore ritornato è quindi -1 . Poiché, però, se $i = 0$ la partizione è vuota, la proprietà di correttezza è banalmente verificata in quanto $k \notin a$.
- *Caso base 2* per $i = 1$. In questo caso $p = r$. La condizione dell'**if...then...else** è, ancora una volta, falsa e l'algoritmo procede ad eseguire il ramo **else** (riga 7). Questa volta, però, $p = r$ e abbiamo due casi possibili.
 - Se $k \in a$ allora deve essere vera la condizione $a[p] = k$. In tal caso l'algoritmo restituisce proprio il valore p .
 - Se $k \notin a$ allora deve essere vera la condizione $a[p] \neq k$ e il valore restituito è -1 .

In entrambi i casi, la proprietà di correttezza vale.

- *Caso induttivo* per $i > 1$. Assumiamo, per *ipotesi induttiva*, che una qualsiasi chiamata all'algoritmo $\text{BINSEARCH}(a, k, p, r)$, con $r - p + 1 < i$, soddisfi la postcondizione, cioè che, se $k \in a[p..r]$ e z è il valore restituito, allora $a[z] = k$.

Consideriamo ora una qualsiasi chiamata all'algoritmo con parametri p e r tali che $r - p + 1 = i$. Poiché $i > 1$, $p < r$ e la condizione a riga 1 è sicuramente vera. L'algoritmo procede, quindi, ad eseguire la riga 2. L'assegnamento a riga 2 viene tradotto nella relazione $q = \lfloor \frac{p+r}{2} \rfloor$. È immediato verificare che $p \leq q < r$. Infatti:

- poiché $p < r$, allora, sommando p ad entrambi i membri, otteniamo che $p+p < p+r$. Da quest'ultima e dividendo entrambi i membri per 2, otteniamo che $\frac{p+p}{2} < \frac{p+r}{2}$, che è equivalente a $p < \frac{p+r}{2}$. Poiché p ed r sono due valori interi, è immediato concludere che $p \leq \lfloor \frac{p+r}{2} \rfloor = q$;
- analogamente, poiché $p < r$, allora, sommando r ad entrambi i membri, otteniamo che $p+r < r+r$. Da quest'ultima e dividendo entrambi i membri per 2, otteniamo che $\frac{p+r}{2} < \frac{r+r}{2}$, che è equivalente a $\frac{p+r}{2} < r$. Poiché $\lfloor \frac{p+r}{2} \rfloor \leq \frac{p+r}{2}$, per transitività otteniamo che $q = \lfloor \frac{p+r}{2} \rfloor < r$.

A questo punto, abbiamo due casi, in relazione alla verità o meno della condizione a riga 3.

- Se la condizione è vera, allora $k \leq a[q]$. In tal caso, se fosse vero che $k \in a[p..r]$, allora necessariamente sarebbe vero che $k \in a[p..q]$. L'algoritmo procede chiamando ricorsivamente se stesso sulla sottosequenza da p a q (cioè eseguirà, alla riga 4, la chiamata $\text{BINSEARCH}(A, K, P, Q)$). La dimensione della sottosequenza passata a questa chiamata è strettamente minore di i . Infatti, essendo $p \leq q < r$, è immediato verificare che $0 \leq q - p + 1 < r - p + 1$. Per ipotesi induttiva, sappiamo che se $k \in a[p..q]$, allora la chiamata $\text{BINSEARCH}(A, K, P, Q)$ restituirà un valore z tale

che $a[z] = k$. Poiché la chiamata originaria $\text{BINSEARCH}(A, K, P, R)$ restituisce lo stesso valore risultante dalla chiamata $\text{BINSEARCH}(A, K, P, Q)$, avremo quindi che il valore z restituito sarà tale che $a[z] = k$, come richiesto.

- Se la condizione è falsa, allora $k > a[q]$. In tal caso, se fosse vero che $k \in a[p..r]$, allora necessariamente sarebbe vero che $k \in a[q + 1..r]$. L'algoritmo procede chiamando ricorsivamente se stesso sulla sottosequenza da $q + 1$ a r (cioè eseguirà la chiamata $\text{BINSEARCH}(A, K, Q+1, R)$ a riga 6). Anche in questo caso, la dimensione della sottosequenza passata a questa chiamata è strettamente minore di i . Infatti, essendo $p \leq q < r$, è immediato verificare che $0 \leq r - (q + 1) + 1 < r - p + 1$. Per ipotesi induttiva, sappiamo che se $k \in a[q + 1..r]$, allora la chiamata $\text{BINSEARCH}(A, K, Q+1, R)$ restituirà un valore z tale che $a[z] = k$. Poiché, anche in questo caso, la chiamata originaria $\text{BINSEARCH}(A, K, P, R)$ restituisce lo stesso valore risultante dalla chiamata $\text{BINSEARCH}(A, K, Q+1, R)$, avremo quindi che il valore z restituito sarà tale che $a[z] = k$, come richiesto.

In conclusione, ogni volta che l'algoritmo viene chiamato su una partizione di dimensione $i \geq 0$ che contiene al suo interno il valore k , esso restituirà un valore z tale che $a[z] = k$. Questo completa la dimostrazione della proprietà di *correttezza parziale*.

Esercizi

[3.1] Si consideri il seguente algoritmo ricorsivo:

Algorithm ARB-SEARCH(T, K)

```

1  if (T ≠ Nil) then
2      if (T→Key < K) then
3          R = ARB-SEARCH(T→SX, K)
4      else if (T→Key > K) then
5          R = ARB-SEARCH(T→DX, K)
6      else
7          R = T
8      return R
    else
9      R = Nil
10 return R

```

Si formalizzi la seguente proprietà di correttezza: supposto T un ABR, se K è una chiave presente in T, l'algoritmo restituisce il puntatore ad un nodo di T che ha come chiave K. Si dimostri, successivamente, la correttezza dell'algoritmo per induzione sull'altezza di T.

[3.2] Si consideri il seguente algoritmo ricorsivo:

Algorithm EXP-RIC(A, M)

```

1  if (M > 0) then
2      if Pari(M) then
3          B = A * A
4          R = EXP-RIC(B,M/2)
5      else
6          R = A * EXP-RIC(A,M-1)
7      else
8          R = 1
9      return R

```

Si dimostri, per induzione sulla dimensione di M, che, se M è un intero non negativo (e A è in valore reale qualsiasi), l'algoritmo restituisce A elevato ad esponente M.

[3.3] Si consideri il seguente algoritmo ricorsivo:

Algorithm COMB(N,K)

```

1  if (K = 1) then
2      return N
3  else if (N = K) then
4      return 1
5  else
6      return COMB(N-1,K-1) + COMB(N-1,K)

```

Assumendo N e K interi non negativi con $K \leq N$, si dimostri, per induzione sulla dimensione di N, che l'algoritmo restituisce il coefficiente binomiale $\binom{N}{K}$. (Si ricorda che $\binom{N}{K} = \frac{N!}{K!(N-K)!}$).

[3.4] Si consideri il seguente algoritmo ricorsivo:

Algorithm POLY(A[],D,X)

```

1  return POLY-RIC(A[],D,X,0)

```

Algorithm POLY-RIC(A[],D,X,R)

```

1  if D ≥ 0 then
2      return POLY-RIC(A[],D-1,X,A[D] + X * R)
3  else
4      return R

```

Assumendo che D sia un valore intero non negativo e che l'array A[] contenga i valori a_0, a_1, \dots, a_D nelle posizioni A[0], A[1], \dots , A[D], si dimostri che POLY(A[],D,X) calcola e restituisce il valore $\sum_{i=0}^D a_i \cdot x^i$. A tal fine, si suggerisce di formulare un'opportuna proprietà di correttezza per l'algoritmo POLY-RIC(A[],D,X,R), da dimostrare per induzione su D, e di utilizzare la correttezza di quest'ultimo per stabilire la correttezza dell'algoritmo principale.

4 Correttezza di algoritmi iterativi e invarianti di ciclo

Nei paragrafi precedenti abbiamo visto come sia possibile effettuare la dimostrazione di correttezza di semplici algoritmi, ricorsivi e non, tramite una traduzione delle operazioni e dei costrutti di controllo (sequenze, condizionali e ricorsione) in relazioni matematiche. Vediamo ora come procedere nel caso in cui l'algoritmo contenga costrutti iterativi. Come visto sopra, la traduzione in relazioni matematiche di operazioni (e dei costrutti di controllo) consiste nell'esprimere l'effetto delle operazioni tramite opportune relazioni tra le variabili dell'algoritmo prima e dopo l'operazione. Nel caso della presenza di costrutti iterativi il problema si complica, poiché le operazioni presenti nel corpo del costrutto vengono eseguite un numero *variabile* di volte (e che spesso dipende dall'input). Si noti che l'esecuzione di un costrutto iterativo è equivalente alla ripetizione in sequenza delle operazioni presenti nel suo corpo. Possiamo, quindi, tradurre ciascuna variabile assegnata all'interno del corpo in un insieme di variabili matematiche, ciascuna associata ad una particolare esecuzione dell'assegnamento della variabile. Ciò è dovuto al fatto che un assegnamento all'interno di un costrutto iterativo è eseguito ad ogni iterazione ed è quindi necessario associare alla variabile dell'algoritmo in esso assegnata una diversa variabile matematica per ogni sua esecuzione. Chiaramente, il numero di variabili matematiche necessarie per ogni variabile è potenzialmente infinito (il costrutto iterativo potrebbe, infatti, non terminare). Consideriamo, ad esempio, il seguente algoritmo iterativo:

Algorithm ARRAYSUM(A[],N)

```

1  S = 0
2  I = 1
3  while (I ≤ N) do
4      S = S + A[I]
5      I = I + 1
6  return S

```

È facile osservare che, se il valore del parametro N è non superiore alla lunghezza dell'array A[], l'algoritmo calcola la somma di tutti i valori contenuti tra la posizione 1 e la posizione N all'interno dell'array A[] fornito in ingresso, restituendo, alla terminazione dell'algoritmo, il valore di tale somma nella variabile S. Tale proprietà di correttezza può essere espressa come segue:

Correttezza $\left\{ \begin{array}{l} \textit{Precondizione: } N \text{ numero intero con } 1 \leq N \leq \textit{length}(A). \\ \textit{Postcondizione: } \textit{Se } z = \textit{ARRAYSUM}(A[],N), \textit{ allora } z = \sum_{j=1}^N A[j]. \end{array} \right.$

Seguendo le considerazioni fatte all'inizio di questa sezione, assoceremo alla variabile S e alla variabile I le variabili matematiche s_0, s_1, s_2, \dots e le variabili i_0, i_1, i_2, \dots , rispettivamente. Utilizzeremo una variabile matematica A per rappresentare l'array (è sufficiente una singola

variabile poiché l'array non viene mai modificato). Come per gli esempi precedenti, la variabile indicizzata con 0 (ad esempio s_0) corrisponde al valore della variabile dell'algoritmo prima di ogni assegnamento e, in generale, la variabile indicizzata con k (ad esempio s_k) corrisponde al valore della variabile dell'algoritmo immediatamente dopo il k -esimo assegnamento. In particolare, s_1 (i_1 , rispettivamente) corrisponde al valore della variabile S (I, rispettivamente) immediatamente prima dell'inizio della prima esecuzione del ciclo **while**. Più in generale, per $k \geq 1$, s_k (i_k , rispettivamente) corrisponde al valore della variabile S (I, rispettivamente) immediatamente prima dell'inizio della k -esima esecuzione del ciclo **while**. Si noti, inoltre, che il numero di variabili matematiche associate a ciascuna variabile dell'algoritmo è potenzialmente infinito.

Come già osservato, sebbene l'esecuzione di un costrutto iterativo sia equivalente alla ripetizione in sequenza delle operazioni presenti nel suo corpo, non è in genere possibile (né ragionevole) eliminare il costrutto iterativo, traducendolo in una sequenza di istruzioni e procedere alla dimostrazione come nel caso di assenza di costrutti iterativi. La lunghezza della sequenza corrispondente non è, infatti, predicibile in generale e potrebbe addirittura essere infinita. Al fine di dimostrare la proprietà di correttezza sopra riportata, procederemo riducendo la proprietà in proprietà più semplici, la cui dimostrazione è più agevole. Tali proprietà più semplici sono riconducibili a proprietà del ciclo contenuto nell'algoritmo. L'idea di base è quella di individuare una o più proprietà che vengono preservate ad ogni iterazione del ciclo iterativo. Ad esempio, il ciclo **while** dell'algoritmo ARRAYSUM ad ogni nuova iterazione aggiunge al valore corrente della variabile S il valore contenuto in una specifica locazione dell'array A[]. Analogamente, ad ogni iterazione, la variabile I, viene incrementata di una unità. In altre parole, I conta le iterazioni eseguite del ciclo, indicando all'inizio di ogni iterazione il numero dell'iterazione successiva da eseguire. La variabile S, invece, contiene la somma di tutti i valori contenuti nelle locazioni dell'array comprese tra la locazione 1 e la locazione corrispondente al valore di $k - 1$. Possiamo, quindi, esprimere questa proprietà nel modo seguente: all'inizio dell'iterazione k -esima del ciclo:

$$s_k = \sum_{j=1}^{k-1} A[j] \quad \text{e} \quad i_k = k \quad (1)$$

In generale, una proprietà che viene preservata ad ogni iterazione di un ciclo prende il nome *invariante di ciclo*. Come vedremo, gli invarianti di ciclo, se correttamente individuati, sono di grande utilità al fine di dimostrare la proprietà di correttezza di un algoritmo iterativo. Supponiamo, ad esempio, che la proprietà (1) sia effettivamente un invariante del ciclo **while** dell'algoritmo ARRAYSUM(A[],N). Supponiamo, inoltre, che il ciclo termini immediatamente prima della k_{fin} -esima iterazione (k_{fin} indica, cioè, il numero dell'iterazione del ciclo prima della quale la condizione del **while** è falsa). Sarebbe di conseguenza verificato che:

$$s_{k_{fin}} = \sum_{j=1}^{k_{fin}-1} A[j] \text{ e } i_{k_{fin}} = k_{fin}$$

Poiché i valori delle variabili i_k (per ogni $k \geq 1$) sono valori interi non negativi, affinché k_{fin} sia il valore d'indice per cui il ciclo termina, è necessario che all'inizio dell'iterazione di indice k_{fin} la condizione del **while** sia falsa. In altre parole, vale $i_{k_{fin}} > n$. Per l'invariante, avremmo che $i_{k_{fin}} = k_{fin}$ e, quindi, $k_{fin} > n$ sarebbe vero. Ora, essendo stato eseguito con successo il ciclo di indice $k_{fin} - 1$, la stessa condizione doveva essere falsa all'inizio di quella iterazione, cioè $k_{fin} - 1 \leq n$. Poiché però k_{fin} è un valore intero, le relazioni $k_{fin} > n$ e $k_{fin} - 1 \leq n$ implicano necessariamente che valgono $k_{fin} = n + 1$ e $k_{fin} - 1 = n$. Sotto

queste condizioni, la prima parte dell'invariante ($s_{k_{fin}} = \sum_{j=1}^{k_{fin}-1} A[j]$) diventa equivalente a

$s_{k_{fin}} = \sum_{j=1}^n A[j]$. Poiché $s_{k_{fin}}$ è proprio il valore ritornato dall'algoritmo, la proprietà di

correttezza è, dunque, verificata. Ciò mostra che, se riusciamo a dimostrare che la proprietà (1) è effettivamente un invariante del ciclo, allora (se l'algoritmo termina) il valore restituito è proprio quello corretto.

Resta ora, quindi, solo da dimostrare la verità dell'invariante ipotizzato. Di seguito è riportato l'algoritmo dell'esempio, in cui è stato inserito l'invariante (1) nel punto in cui si suppone sia sempre verificato (k indica l'iterazione corrente del ciclo).

Algorithm ARRAYSUM(A[],N)

```

1  S = 0
2  I = 1
3  while (I ≤ N) do  INV: { $s_k = \sum_{j=1}^{k-1} A[j]$  e  $i_k = k$ }
4      S = S + A[I]
5      I = I + 1
6  return S
```

Dimostriamo ora che la proprietà (1) è effettivamente un invariante del ciclo, che, cioè, viene preservato ad ogni iterazione. Poiché il numero di cicli eseguiti è un numero intero non negativo, è possibile dimostrare per induzione sul numero di cicli eseguiti che l'invariante è verificato. In altre parole, dimostrare che è un invariante di ciclo significa dimostrare che la seguente proprietà è verificata:

$$\forall k \geq 1, s_k = \sum_{j=1}^{k-1} A[j] \text{ e } i_k = k.$$

La dimostrazione è per induzione su k (con $k \geq 1$).

- *Caso base* Il caso base individuato è per $k = 1$. Questo caso corrisponde al caso in cui la prima esecuzione del ciclo deve ancora avvenire. Dall'algoritmo risulta evidente che $s_1 = 0$ e $i_1 = 1$. La proprietà (1) per $k = 1$, che corrisponde a $s_1 = \sum_{j=1}^0 A[j] = 0$ e $i_1 = 1$, è quindi verificata.
- *Caso induttivo* Assumiamo, per ipotesi induttiva, che la proprietà (1) valga prima dell'inizio dell'esecuzione del ciclo k -esimo. Dunque, che valga $s_k = \sum_{j=1}^{k-1} A[j]$ e $i_k = k$. L'esecuzione della k -esima iterazione comporta l'esecuzione, nell'ordine, delle istruzioni $S = S + A[I]$ e $I = I + 1$. Queste istruzioni, espresse in termini di relazioni matematiche, corrispondono a $s_{k+1} = s_k + A[i_k]$ e $i_{k+1} = i_k + 1$, rispettivamente. Sostituendo le espressioni equivalenti per s_k e i_k derivanti dall'ipotesi induttiva, otteniamo:

$$s_{k+1} = \sum_{j=1}^{k-1} A[j] + A[k] \quad \text{e} \quad i_{k+1} = k + 1$$

Ma $s_{k+1} = \sum_{j=1}^{k-1} A[j] + A[k] = \sum_{j=1}^k A[j]$. Segue che, prima dell'inizio del $(k + 1)$ -esimo ciclo, la seguente proprietà sarà verificata:

$$s_{k+1} = \sum_{j=1}^k A[j] \quad \text{e} \quad i_{k+1} = k + 1$$

Questo assicura che la proprietà (1) è preservata ad ogni iterazione e che, quindi, essa è un invariante del ciclo **while**.

La dimostrazione appena svolta, insieme a quella fatta precedentemente sotto l'ipotesi di terminazione dell'algoritmo alla k_{fin} -esima iterazione, garantisce che se l'algoritmo, chiamato con un valore di N minore o uguale alla dimensione dell'array $A[]$, termina, allora la postcondizione richiesta è certamente soddisfatta.

Esercizi

[4.1] Si definisca la proprietà di correttezza del seguente algoritmo e la si dimostri utilizzando l'invariante di ciclo riportato:

Algorithm ALG-3(M)

1 $P = 1$

2 $I = 1$

3 **while** ($I < 2 * M$) **do** $INV: \{p_k = \prod_{j=1}^{k-1} (2 \cdot j - 1) \text{ e } i_k = 2 \cdot k - 1\}$

```

4   P = P * I
5   I = I + 2
6   return P

```

[4.2] Si definisca la proprietà di correttezza del seguente algoritmo e la si dimostri utilizzando l'invariante di ciclo riportato:

Algorithm BINSEARCH-ITER($X, A[], N$)

```

1   I = 1
2   J = N
3   while (I < J) do INV: { $j_k - i_k = n - k$  e se  $x \in A$ , allora  $(A[i_k] \leq x \leq A[j_k])$ }
4     P = [(I + J) / 2]
5     if (X > A[P]) then
6       I = P + 1
7     else
8       J = P
9   if X = A[I] then
10    return I
11  else
12    return NOT_FOUND

```

[4.3] Si consideri il seguente algoritmo che calcola il valore corrispondente al primo argomento A elevato all'esponente dato dal secondo argomento M. La preconditione è che M sia un valore intero non negativo e che A sia un valore reale. Si ne dimostri la correttezza, utilizzando l'invariante di ciclo riportato.

Algorithm EXP(A, M)

```

1   B = A
2   E = M
3   R = 1
4   while (E > 0) do INV: { $r_k \cdot b_k^{e_k} = a^m$ }
5     if (E è dispari) then
6       R = R * B
7       E = E - 1
8     else
9       B = B * B
10      E = E / 2
11  return R

```

[4.4] Si assuma che D sia un valore intero non negativo e che l'array A[] contenga i valori a_0, a_1, \dots, a_D nelle posizioni A[0], A[1], \dots , A[D] (con $D \leq \text{length}(A)$). Si definisca la proprietà di correttezza del seguente algoritmo e la si dimostri utilizzando l'invariante di ciclo riportato:

Algorithm POLY-ITER(A[], D, X)

```

1   T = 0
2   I = D
3   while (I ≥ 0) do INV: { $i_k = d - k + 1$  e  $T_k = \sum_{z=d-k+2}^d a_z x^{(z-(d-k+2))}$ }

```

```

4     T = A[I] + (X * T)
5     I = I - 1
6     return T

```

5 Terminazione

Nelle sezioni precedenti ci siamo concentrati sulla dimostrazione della correttezza parziale di algoritmi iterativi e ricorsivi. Al fine di garantire la correttezza totale è però necessario dimostrare la terminazione dell'algoritmo per ogni istanza di ingresso che soddisfi la precondizione. Iniziamo ricordando alcune proprietà che saranno utili allo scopo.

Proposizione 1: Ogni sequenza strettamente crescente di interi $a_1 < a_2 < \dots < a_n < \dots$ non possiede estremo superiore. Analogamente, ogni sequenza strettamente decrescente di interi $a_1 > a_2 > \dots > a_n > \dots$ non possiede estremo inferiore.

Si noti che un'immediata conseguenza della proposizione 1 è che, se la sequenza è strettamente crescente, per qualsiasi valore intero K esiste certamente un indice x tale che $a_x > K$.

Proposizione 2: Ogni sequenza di interi $a_1, a_2, \dots, a_n, \dots$ per la quale $a_{k+1} - a_k \geq \epsilon$ (con $\epsilon > 0$) non possiede estremo superiore. Analogamente, Ogni sequenza di interi $a_1, a_2, \dots, a_n, \dots$ per la quale $a_{k+1} - a_k \leq \epsilon$ (con $\epsilon < 0$) non possiede estremo inferiore.

Vediamo ora un'applicazione di queste proprietà nella dimostrazione di terminazione del semplice algoritmo iterativo ARRAYSUM della sezione precedente. L'algoritmo termina solamente se termina il ciclo **while** in esso contenuto e quest'ultimo termina quando la sua condizione $I \leq N$ risulta falsa, cioè non appena si verifica $I > N$. In termini di relazioni tra le variabili matematiche che abbiamo utilizzato per descrivere l'esecuzione dell'algoritmo, ciò si verifica se esiste un intero \hat{k} (che rappresenta il numero della prossima iterazione del ciclo) tale per cui $i_{\hat{k}} > n$. Dalla precondizione dell'algoritmo sappiamo che n è un numero intero compreso tra 1 e $length(A)$ e che $i_1 = 1$. L'invariante che abbiamo dimostrato nella sezione precedente, garantisce che all'inizio di ogni iterazione k -esima, varrà $i_k = k$. Abbiamo, inoltre, dimostrato che $i_{k+1} = i_k + 1$. Da questo possiamo concludere che le esecuzioni ciclo **while** generano, implicitamente, la sequenza di valori interi $i_1, i_2, \dots, i_k, \dots$ tale che:

$$0 < i_1 < i_2 < \dots < i_k < \dots$$

Questa è, appunto, una sequenza di interi strettamente crescente e, per la proposizione 1 non ammette estremo superiore. Segue che esisterà un valore \hat{k} tale che $i_{\hat{k}} > n$. Cioè esisterà

un'iterazione del ciclo (in particolare proprio la \hat{k} -esima) all'inizio della quale la condizione del **while** sarà falsa e obbligherà il ciclo (e in questo caso anche l'algoritmo) a terminare.

Se consideriamo \hat{k} il primo valore per cui $i_{\hat{k}} > n$, questo significa che $i_{\hat{k}-1} \leq n$. In altre parole, $i_{\hat{k}-1} \leq n < i_{\hat{k}} = i_{\hat{k}-1} + 1$. Poiché n è in valore intero, necessariamente deve essere $n = i_{\hat{k}-1}$, quindi $i_{\hat{k}} = n + 1$. Dall'invariante, otteniamo quindi che $\hat{k} = n + 1$. Quindi, l'algoritmo termina prima della $(n + 1)$ -esima iterazione, quindi dopo aver eseguito n iterazione del ciclo.

Consideriamo ora l'algoritmo dell'esercizio [4.2]. La condizione di terminazione dell'algoritmo corrisponde alla condizione di terminazione del ciclo **while** in esso contenuto ($I \geq J$). Il valore delle variabili I e J prima della prima iterazione sono rappresentati da $i_1 = 1$ e $j_1 = n$. È possibile dimostrare per induzione sulle iterazioni $k \geq 1$ del ciclo che ad ogni iterazione la differenza $j_k - i_k$ è sempre un valore intero e strettamente decresce ad ogni iterazione. In altre parole, detto $d_k = j_k - i_k$ (per ogni $k \geq 1$), possiamo dimostrare che la sequenza:

$$d_1, d_2, \dots, d_z, \dots$$

è una sequenza di interi strettamente decrescente. Per $k = 1$ è immediato verificare che $d_1 = j_1 - i_1 = n$ è un valore intero. Supponiamo ora che d_k sia un valore intero e che venga eseguita la k -esima iterazione del ciclo. Ciò avviene se è verificata la condizione del **while**, cioè se $i_k < j_k$. Il primo assegnamento determina $p_{k+1} = \lfloor \frac{i_k + j_k}{2} \rfloor$. Da $i_k < j_k$, sommando i_k ad entrambi i membri, possiamo dedurre che $i_k + i_k < i_k + j_k$. Dividendo entrambi i membri per 2 otteniamo che $i_k < \frac{i_k + j_k}{2}$. Poiché i_k è un valore intero per ipotesi induttiva, per la definizione di base di un numero, avremo che:

$$i_k \leq \left\lfloor \frac{i_k + j_k}{2} \right\rfloor$$

In maniera simile, partendo da $i_k < j_k$ e sommando j_k ad entrambi i membri e dividendo il risultato per 2, possiamo dedurre che $\frac{i_k + j_k}{2} < j_k$. Poiché $\lfloor \frac{i_k + j_k}{2} \rfloor \leq \frac{i_k + j_k}{2}$ per definizione di base di un numero, otteniamo che:

$$\left\lfloor \frac{i_k + j_k}{2} \right\rfloor < j_k$$

Possiamo, quindi, concludere che:

$$i_k \leq p_{k+1} < j_k$$

Si possono ora verificare due casi, a seconda del valore della condizione dell'**if**:

- nel primo caso ($x > A[p_{k+1}]$). In questo caso, viene eseguito l'assegnamento $I = P + 1$, che determina $i_{k+1} = p_{k+1} + 1$, mentre il valore di J rimane invariato, cioè $j_{k+1} = j_k$. Quindi, $d_{k+1} = j_{k+1} - i_{k+1} = j_{k+1} - (p_{k+1} + 1) = j_{k+1} - p_{k+1} + 1$. d_{k+1} è chiaramente

un valore intero. Poiché $j_{k+1} = j_k$, avremo anche che $d_{k+1} = j_k - p_{k+1} + 1$. Sappiamo, inoltre, che $i_k \leq p_{k+1}$, quindi $i_k < p_{k+1} + 1$. Da ciò segue immediatamente che:

$$d_{k+1} = j_k - p_{k+1} + 1 < j_k - i_k = d_k$$

- nel secondo caso ($x \leq A[p_{k+1}]$). In questo caso, viene eseguito l'assegnamento $J = P$, che determina $j_{k+1} = p_{k+1}$, mentre il valore di I rimane invariato, cioè $i_{k+1} = i_k$. In modo simile al caso precedente, avremo che $d_{k+1} = j_{k+1} - i_{k+1} = p_{k+1} - i_{k+1}$ è un valore intero. Poiché $i_{k+1} = i_k$, avremo anche che $d_{k+1} = p_{k+1} - i_k$. Essendo, però, $p_{k+1} < j_k$, possiamo concludere che

$$d_{k+1} = p_{k+1} - i_k < j_k - i_k = d_k$$

In entrambi i casi, possiamo concludere che d_{k+1} è un intero strettamente minore di d_k . Quindi, la sequenza $d_1, d_2, \dots, d_z, \dots$ è un sequenza strettamente decrescente di valore interi. Per la proposizione 2, questa sequenza non ha limite inferiore. Esisterà quindi un valore \hat{k} tale che $d_{\hat{k}} < 0$. Cioè, tale che $d_{\hat{k}} = j_{\hat{k}} - i_{\hat{k}} \leq 0$, che implica $j_{\hat{k}} \geq i_{\hat{k}}$. Se \hat{k} è il primo valore tale che $d_{\hat{k}} < 0$, segue che il ciclo terminerà proprio immediatamente prima dell'iterazione \hat{k} -esima. Concludiamo, quindi, che l'algoritmo termina sicuramente per ogni valore di N .

Consideriamo ora il problema della terminazione di algoritmi ricorsivo. Per la natura stessa della ricorsione, è necessario dimostrare che ogni chiamata all'algoritmo deve generare solo sequenze di chiamate ricorsive annidate che raggiungono uno dei casi basi della ricorsione. A tal fine, è spesso sufficiente dimostrare che la dimensione dell'input passato a tutte chiamate da effettuate da una qualsiasi chiamata all'algoritmo sia strettamente inferiore di quello ricevuto dalla chiamata. Sotto l'ipotesi di poter esprimere la dimensione dell'input tramite un valore intero, dimostrando questa proprietà per induzione su tale dimensione garantisce che tutte le sequenze di chiamate annidate generate da una qualche chiamata all'algoritmo determinano una sequenza di valori interi (le rispettive dimensioni degli input ricevuti) che è una sequenza di interi strettamente decrescente. Le proposizioni 1 e 2 sopra riportate possono quindi essere applicate per concludere che l'algoritmo termina.

Consideriamo l'algoritmo BINSEARCH della sezione 3. Nella sezione 3 è stata presentata una dimostrazione della correttezza parziale per induzione sulla dimensione della porzione dell'array contenuta tra il valore P e il valore R . In altre parole, la quantità intera su cui viene fatta induzione è $i = r - p + 1$. La dimostrazione stessa evidenzia che ogni chiamata all'algoritmo, per un qualche valore di $i \geq 0$ o termina immediatamente (per i casi base $i = 0, 1$) o genera una chiamata ricorsiva con parametri interi p e q , o una chiamata con parametri interi $q + 1$ e r . In entrambi i casi, abbiamo già osservato che le dimensioni delle sequenze individuate da questi valori interi sono strettamente minori di i ma sempre ≥ 0 (si veda la dimostrazione in sezione 3). La proposizione 1 garantisce che la sequenza di dimensioni passate alle chiamate annidate non ha limite inferiore, quindi dopo un numero

finito di chiamate annidate, l'input risulterà essere uguale a 1 o a 0, quindi garantendo la terminazione.

Esercizi

- [5.1] Si dimostri la terminazione degli algoritmi ricorsivi riportati agli esercizi [3.1], [3.2], [3.3] e [3.4]. Per la terminazione dell'esercizio [3.1], si consideri per dimensione dell'input l'altezza dell'albero in ingresso. Si noti, inoltre, che la condizione $T = \text{NIL}$, corrispondente ad uno dei casi base, si verifica quando l'altezza dell'albero T è 0.
- [5.2] Si dimostri la terminazione degli algoritmi iterativi riportati agli esercizi [4.1], [4.3] e [4.4].

6 Dimostrazione di correttezza dell'algorithm di Insertion Sort

In questa sezione finale, proveremo ad applicare le tecniche illustrate con semplici esempi nelle sezioni precedenti alla dimostrazione di correttezza dell'algorithm di ordinamento Insertion Sort. Di seguito è riportata la versione dell'algorithm che analizzeremo:

Algorithm INSERTIONSORT(A)

```

1  for j := 2 to length(A) do INV1:  $\{j_k = k + 1 \text{ e } A^k[1..k - 1] \text{ è permutazione ordinata di } A^1[1..k]\}$ 
2      key := A[j]
3      i := j-1
4      while (i > 0 and A[i] > key) do INV2:  $\{i_{k,z} = j_k - z \text{ e } A^{(k,z)}[j_k - z + 1..j_k] \geq \text{key}_k \text{ e } \quad \}$ 
            $\{A^{(k,z)}[j_k - z + 2..j_k] = A^{(k,1)}[j_k - z + 1..j_k - 1] \text{ e } \}$ 
            $\{A^{(k,z)}[1..j_k - z + 1] = A^{(k,1)}[1..j_k - z + 1] \quad \}$ 
5          A[i+1] := A[i]
6          i := i - 1
7      A[i+1] := key

```

dove $A^k[]$ si riferisce alla configurazione dell'array $A[]$ immediatamente prima della k -esima esecuzione del ciclo **for** ($A^1[]$ si riferisce alla configurazione iniziale dell'array, prima della prima esecuzione del ciclo **for**). Analogamente, j_k denota il valore della variabile J prima della k -esima esecuzione del ciclo **for**. $A^{(k,z)}[]$ si riferisce alla configurazione dell'array durante la k -esima esecuzione del ciclo **for** e prima della z -esima esecuzione del ciclo **while**, mentre $i_{k,z}$ denota il valore della variabile J durante la k -esima iterazione del **for** e prima della z -esima esecuzione del ciclo **while**. Per comodità, la notazione $A[p..r] \geq k$ verrà usata come abbreviazione per la proprietà $\forall p \leq x \leq r. A[x] \geq k$. Assumendo che $A^{fin}[]$ denoti la configurazione dell'array al termine dell'algorithm, possiamo esprimerla proprietà di correttezza come segue:

Correttezza $\left\{ \begin{array}{l} \textit{Precondizione: L'array } A \text{ contiene valori numerici (interi o reali).} \\ \textit{Postcondizione: Se } n = \textit{length}(A) \text{ e } A^{\textit{fin}} \text{ è la configurazione dell'array } A \\ \text{al termine dell'algorithm, allora } \forall 1 \leq x < n. A^{\textit{fin}}[x] \leq A^{\textit{fin}}[x + 1]. \end{array} \right.$

Come si può notare, l'algorithm contiene due cicli annidati uno nell'altro. Al fine di procedere alla dimostrazione di correttezza, dovremmo un'opportuna proprietà individuare per ciascun ciclo. L'idea di fondo dell'algorithm è quella che ad ogni iterazione del ciclo **for** la porzione ordinata dell'array viene estesa di una locazione, a partire dalla locazione 2 fino al termine dell'array ($\textit{length}(A)$). Inizialmente, tale porzione ordinata è quella che contiene la sola locazione 1. Questa proprietà è formalizzata dall'invariante INV1 riportato all'interno dell'algorithm.

Per quel che concerne il ciclo **while** interno, il suo ruolo è quello di traslare in avanti di una posizione tutte le locazioni della porzione ordinata dell'array che contengono elementi più grandi del valore *key* (originariamente contenuto nella locazione con la quale si vuole estendere la porzione ordinata nel nuovo ciclo **for**). L'invariante INV2 riportato nell'algorithm esprime formalmente tale proprietà.

Dimostreremo la correttezza dell'algorithm, dimostrando che gli invarianti di ciclo sono validi e infine trarremo la conclusione da ciò che essi implicano alla terminazione dei cicli.

La dimostrazione dell'invariante INV1 è, come prevedibile, per induzione sul valore $k \geq 1$ che indica la prossima esecuzione del ciclo. È immediato verificare che INV1 è verificata immediatamente prima della prima dell'esecuzione del ciclo (caso base), in quanto $k = 1$, $j_1 = 2 = k + 1$ e la porzione di array che va dalla locazione 1 alla locazione $j_1 - 1 = 1$ contiene un solo elemento ed è quindi ordinata.

Al fine di mostrare che l'invariante INV1 è mantenuto durante ogni esecuzione del ciclo **for** (caso induttivo), è necessario esaminare il corpo del ciclo. Deve valere che, all'inizio della k -esima esecuzione del **for**, $A^k[1..k]$ è una permutazione ordinata di $A^1[1..k]$. Per mostrare ciò sarà necessario esaminare prima l'invariante INV2. Dimostreremo questo invariante per induzione sul numero z di esecuzioni del ciclo **while**, fissando un arbitrario valore di $k \geq 1$.

L'invariante 2 è verificato ad ogni sua inizializzazione $z = 1$ (caso base), per qualsiasi $k \geq 1$. Infatti $i_{k,1} = j_k - 1 = j_k - z$, $key_k = A^{(k,1)}[j_k]$ e $A^{(k,1)}[i_{k,1}] > key_k$ (condizione del **while**). Quindi, ogni elemento in $A^{(k,1)}[j_k - z + 1..j_k]$ è $\geq key_k$. Poiché non è ancora stata eseguita alcuna iterazione del ciclo **while** per il k dato, il terzo e quarto congiunto dell'invariante INV 2 sono banalmente verificato. Infatti, per $z = 1$, gli intervalli $[j_k - z + 2..j_k]$ e $[j_k - z + 1..j_k - 1]$ sono vuoti, mentre il quarto congiunto corrisponde proprio a $A^{(k,1)}[1..j_k] = A^{(k,1)}[1..j_k]$.

Ogni iterazione $z > 1$ del **while** mantiene questo invariante (caso induttivo). Infatti:

- se viene eseguito il corpo del **for**, allora deve valere $A^{(k,z)}[j_{k,z}] > key_k$. Per ipotesi

induttiva, $i_{k,z} = j_k - z$, ergo $A^{(k,z)}[j_k - z] > key_k$.

- l'assegnamento $A[i+1] := A[i]$ sposta il valore di A in posizione $i_{k,z} = j_k - z$ (che si sa essere $> key_k$) in posizione $i_{k,z} + 1 = j_k - z + 1$, lasciando tutti gli altri elementi inalterati. Quindi, valgono le seguenti:

$$\begin{aligned} A^{(k,z+1)}[1..j_k - z] &= A^{(k,z)}[1..j_k - z] \\ A^{(k,z+1)}[j_k - z + 1] &= A^{(k,z)}[j_k - z] \\ A^{(k,z+1)}[j_k - z + 2..j_k] &= A^{(k,z)}[j_k - z + 2..j_k] \end{aligned}$$

- da $A^{(k,z+1)}[1..j_k - z] = A^{(k,z)}[1..j_k - z]$ e dall'ipotesi induttiva che garantisce $A^{(k,z)}[1..j_k - z] = A^{(k,1)}[1..j_k - z]$, otteniamo:

$$A^{(k,z+1)}[1..j_k - z] = A^{(k,1)}[1..j_k - z] \quad (2)$$

- da $A^{(k,z+1)}[j_k - z + 1] = A^{(k,z)}[j_k - z]$ e $A^{(k,z+1)}[j_k - z + 2..j_k] = A^{(k,z)}[j_k - z + 2..j_k]$, e dall'ipotesi induttiva che garantisce $A^{(k,z)}[j_k - z + 2..j_k] = A^{(k,1)}[j_k - z + 1..j_k - 1]$ e $A^{(k,z)}[j_k - z] = A^{(k,1)}[j_k - z]$, otteniamo:

$$A^{(k,z+1)}[j_k - z + 1..j_k] = A^{(k,1)}[j_k - z..j_k - 1] \quad (3)$$

- inoltre, sempre per ipotesi induttiva, $A^{(k,z+1)}[j_k - z + 2..j_k] = A^{(k,z)}[j_k - z + 2..j_k] \geq key_k$ che, insieme a $A^{(k,z+1)}[j_k - z] = A^{(k,z)}[j_k - z] > key_k$ (per la condizione del **while** e $A^{(k,z+1)}[j_k - z + 1] = A^{(k,z)}[j_k - z]$ (per effetto dell'assegnamento), otteniamo che:

$$A^{(k,z+1)}[j_k - z..j_k] \geq key_k \quad (4)$$

- infine, dopo l'assegnamento $i := i - 1$, avremmo che $i_{k,z+1} = i_{k,z} - 1 = j_k - z - 1 = j_k - (z + 1)$.

È immediato quindi verificare che (2), (3) e (4) sono equivalenti, rispettivamente, alle seguenti:

$$A^{(k,z+1)}[1..j_k - (z + 1) + 1] = A^{(k,1)}[1..j_k - (z + 1) + 1] \quad (5)$$

$$A^{(k,z+1)}[j_k - (z + 1) + 2] = A^{(k,1)}[j_k - (z + 1) + 1..j_k - 1] \quad (6)$$

$$A^{(k,z+1)}[j_k - (z + 1) + 1..j_k] \geq key_k \quad (7)$$

Queste tre, insieme al fatto che $i_{k,z+1} = j_k - (z + 1)$, garantiscono che l'invariante è ancora verificato immediatamente prima della $(z + 1)$ -esima iterazione del ciclo **while**.

Utilizzando le considerazioni della sezione 5, possiamo concludere che il ciclo **while** termina sicuramente per qualsiasi $k \geq 1$. Infatti, il valore iniziale di i , $i_{k,1}$ è posto a j_k che è

certamente un valore intero positivo. Ad ogni iterazione del ciclo, i viene decrementata di una unità, generando una sequenza strettamente decrescente di numeri interi. Quindi, se non termina prima a causa della condizione $A[i] \leq \text{key}$, il ciclo **while** termina poiché, per la proposizione 1, esiste almeno un valore di z per cui $i_{k,z} \leq 0$. Possiamo quindi concludere che, al termine del ciclo, i sarà certamente maggiore o uguale a 0. È facile convincersi che il primo valore di z per cui si verifica $i_{k,z} \leq 0$, è quello per cui $i_{k,z} = 0$.

Sia \hat{z} il valore di z per cui il ciclo termina. Avremo, quindi, che $i_{k,\hat{z}} \geq 0$. Ne segue che:

- $A^{k,\hat{z}}[j_k - \hat{z} + 2..j_k] = A^{k,1}[j_k - \hat{z} + 1..j_k - 1]$
- $A^{k,\hat{z}}[j_k - \hat{z} + 1..j_k] \geq \text{key}_k (= A^{k,1}[j_k])$
- $i = 0$ oppure $A^{k,\hat{z}}[j_k - \hat{z}] \leq \text{key}_k$ (condizione d'uscita del **while**)
- $A^{k,\hat{z}}[1..j_k - \hat{z}] = A^{k,1}[1..j_k - \hat{z}]$

Possiamo ora procedere a dimostrare il caso induttivo per l'invariante INV1 del ciclo **for**. Assumendo l'invariante vero immediatamente prima della k -esima iterazione del **for**, avremo che $A^k[1..j_k - 1] = A^{k,1}[1..j_k - 1]$ è una permutazione ordinata di $A^1[1..j_k - 1]$. La variabile key_k è uguale a $A^k[j_k]$ e $i_{k,z}$ a $j_k - 1$.

Al termine del ciclo **while**, viene eseguito l'assegnamento $A[i+1] = \text{key}$, che modifica solo la locazione $i_{k,\hat{z}} + 1 =$ dell'array $A^{k,\hat{z}}$. Ciò determina la configurazione A^{k+1} dell'array immediatamente prima dell'inizio dell'esecuzione $(k+1)$ -esima del ciclo **for**. Avremo quindi:

1. $A^{k+1}[j_k - \hat{z} + 2..j_k] = A^k[j_k - \hat{z} + 1..j_k - 1]$
2. $A^{k+1}[j_k - \hat{z} + 1] = \text{key}_k$ (assegnamento $A[i+1] = \text{key}$)
3. $A^{k+1}[j_k - \hat{z} + 2..j_k] \geq \text{key}_k (= A^1[j_k])$
4. $A^{k+1}[1..j_k - \hat{z}] = A^k[1..j_k - \hat{z}]$
5. $j_k - \hat{z} = 0$ oppure $j_k - \hat{z} > 0$ e $A^k[j_k - \hat{z}] \leq \text{key}_k$ (condizione di terminazione del **while**)

Da 1. e dall'ipotesi induttiva abbiamo che $A^{k+1}[j_k - \hat{z} + 2..j_k]$ è ordinata, da 4. e dall'ipotesi induttiva abbiamo che $A^{k+1}[1..j_k - \hat{z}]$ è ordinata. Consideriamo i due casi previsti dalla 5.:

- se $j_k - \hat{z} = 0$ allora $j_k - \hat{z} + 1 = 1$ e $j_k - \hat{z} + 2 = 2$. In questo caso, la 3. è equivalente a $A^{k+1}[2..j_k] \geq \text{key}_k$ e la 2. a $A^{k+1}[1] = \text{key}_k$. È immediato verificare che $A^{k+1}[1..j_k]$ è quindi una permutazione ordinata di $A^k[1..j_k]$ e, per ipotesi induttiva, anche di $A^1[1..j_k]$ come desiderato;

- se $j_k - \hat{z} > 0$ allora $A^k[j_k - \hat{z}] \leq key_k$. Quindi, vale che $A^k[j_k - \hat{z}] = A^{k+1}[j_k - \hat{z}] \leq key_k \leq A^{k+1}[j_k - \hat{z} + 2..j_k]$. Ancora una volta, possiamo concludere che $A^{k+1}[1..j_k]$ è quindi una permutazione ordinata di $A^k[1..j_k]$ e, per ipotesi induttiva, anche di $A^1[1..j_k]$ come desiderato;

Poichè $j_k = k + 1$ e j viene incrementato prima dell'inizio della prossima iterazione, in entrambi i casi otteniamo che all'inizio della $(k + 1)$ -esima iterazione del ciclo **for**, $j_{k+1} = (k+1)+1$ e $A^{k+1}[1..k+1]$ è una permutazione ordinata di $A^1[1..k+1]$, cioè proprio l'invariante INV1.

Questo completa la dimostrazione dell'invariante INV1.

Si noti che il ciclo **for** termina sicuramente per quel valore di $k = fin$ tale che $j_{fin} = n+1$. Per la validità dell'invariante avremo, quindi, che $fin = n$ e che " $A^{fin}[1..n]$ è una permutazione ordinata $A^1[1..n]$ ", che completa la dimostrazione di correttezza dell'algorithmo.