1 Iterazione, induzione e ricorsione

Iterazione, induzione e ricorsione sono concetti fondamentali che compaiono in varie forme nel modelli dei dati, nelle strutture dati e negli algoritmi.

1.1 Iterazione

Cosa si intende con iterazione?

Iterare significa eseguire in modo ripetitivo lo stesso compito, o versioni diverse dello stesso compito, fino al verificarsi di certe condizioni logiche.

Nell'informatica l'iterazione è un concetto che si trova in molte forme:

- **Nei modelli di dati**: concetti come le **liste** sono definiti in modo ripetitivo.
  Esempio: Una lista è vuota, oppure è un elemento seguito da un altro, seguito da un altro ancora...;
- **I programmi e gli algoritmi**: utilizzano l'iterazione per eseguire compiti ripetitivi senza speci care uno per uno un gran numero di singoli passi;
- **I linguaggi di programmazione**: utilizzano, nella realizzazione di algoritmi iterativi, costrutti ciclici, come ad esempio i comandi `while` e `for` del `c++`;

1.2 Induzione

L'**induzione** è un concetto strettamente collegato alla ricorsione, ma appartiene più al **mondo matematico** che a quello della programmazione.
Nell'induzione, si dimostra una proposizione per il **caso base**, e poi si mostra che, se vale per un caso generico, allora vale anche per il successivo.

Cosa si intende con induzione?

La dimostrazione per induzione è una tecnica utile per dimostrare la verità di un **asserto**.

Cos'è un *asserto*?

Si definisce **asserto** (o proposizione) un'affermazione dotata di valore di verità (ossia può essere vera o falsa).
Esempio: in una dimostrazione induttiva si tenta di dimostrare che $S(n)$ vale per tutti gli interi di n non negativi o, più in generale, per tutti gli interi maggiori di un certo limite inferiore.

Dimostrare la verità di un asserto permette di esprimere le proprietà di un programma.
In particolare, la dimostrazione per induzione si basa sulla **definizione di una classe di oggetti (o fatti) strettamente correlati tra loro**.
Esempio: sia $S(n)$ un asserto arbitrario su un intero n. Nella sua forma più semplice (**induzione semplice**), una dimostrazione induttiva dell'asserto $S(n)$ prevede **due passi**:

- **Caso base**: si occupa di costruire uno o più oggetti semplici.
  Nel caso dell'esempio, si dimostra che l'asserto $S(n)$ è vero per un valore particolare di n;
- **Passo induttivo**: costruisce oggetti più grandi che dipendono da quelli appena precedenti.
  Nel caso dell'esempio, si dimostra che per ogni $n \geqslant 0,$ se $S(n)$ è vero, lo è anche $S(n+1)$.

In una dimostrazione induttiva, **ogni istanza dell'asserto** $S(n)$ **dipende solo dall'asserto sul valore che precede** n. Se l'induzione parte da 0, per ogni intero n si deve dimostrare un asserto $S(n)$:

- La dimostrazione di $S(1)$ utilizza $S(0)$
- La dimostrazione di $S(2)$ utilizza $S(1)$
- e così via...

Ogni asserto dipende dal precedente in modo uniforme, e grazie alla dimostrazione del **passo induttivo** si garantisce la verità di tutti i passi successivi.

1.2.1 Induzione completa

Un induzione nella quale si dimostra la verità di $S(n+1)$ utilizzando come ipotesi induttiva soltanto $S(n)$ viene detta **induzione semplice**.

Cosa si intende con induzione completa?

Si parla di induzione completa (perfetta o forte) quando per dimostrare l'asserto S siamo autorizzati a utilizzare $S(i)$ per tutti i valori di i dalla base fino a n.

Quindi a differenza dell'induzione semplice, dove, per dimostrare $S(n+1)$ si usa solo $S(n)$ (ovvero **solo ciò** che si è **ottenuto nel passo precedente**), nell'induzione completa per dimostrare che $S(n+1)$ è vera, **si possono usare tutti i casi precedenti, non solo quello**

immediatamente precedente.
Anche in questo caso, per ottenere una dimostrazione induttiva dell'asserto $S(n)$ si utilizzano due passi:

- Si dimostra prima la base, $S(0)$;
- In seguito, si assumono le verità di $S(0), S(1), ..., S(n)$ e a partire da questi si dimostra $S(n+1)$.

Utilità dell'induzione completa

L'induzione completa, torna utile perché alcuni problemi non possono essere dimostrati basandosi solo sul passo precedente.
Ad esempio, se per calcolare $S(n)$ si ha bisogno sia di $S(n-1)$ che di $S(n-2)$ (come nella sequenza di fibonacci), allora serve induzione completa.

1.3 Ricorsione

Cosa si intende con ricorsione?

La **ricorsione** è una tecnica in un concetto viene definitio, direttamente o indirettamente, in termini di se stesso.

È molto simile all'iterazione, ma invece di ripetere istruzioni tramite un ciclo (`for`, `while`), **si ripete richiamando la stessa funzione**. Anche se all'**apparenza** possono sembrare **piu complessi** dei programmi iterativi, i programmi ricorsivi possono **risultare piu semplici** da

scrivere e **analizzare**.
Una funzione o procedura ricorsiva `P` può richiamare se stessa in due modi:

- **Direttamente**: quando dentro il corpo della funzione `P` c'è una chiamata a `P`;

```
void P() { P(); } // chiamata diretta
```

- **Indirettamente**: quando `P` chiama un'altra funzione `Q`, e poi `Q` chiama `P`, e così via ...

```
void P() { Q(); } // P chiama Q, che chiama P
void Q() { P(); }
```

Induttività della ricorsione

Utilizzando la ricorsione si costruisce, **implicitamente**, una **definizione induttiva** di come funziona l'algoritmo e di **quanto tempo impiega** per completarsi.
Infatti quando un algoritmo impiega la ricorsione, si fa uso di una formula detta **equazione di ricorrenza**: il tempo che serve per risolvere un problema più grande dipende dal tempo che serve per risolvere i problemi più piccoli.
Esempio: si immagini di voler calcolare il fattoriale di 4

```
fatt(4)
-> fatt(3)
-> fatt(2)
-> fatt(1)
-> fatt(0)
```

Quindi il tempo totale per `fatt(4)` è la somma del tempo per `fatt(3)`, più un piccolo tempo extra per l'operazione `n * ...`

In generale, la ricorsione è **induttiva** perché per dimostrare una proprietà di una procedura ricorsiva, abbiamo bisogno di dimostrare un asserto sull'effetto della chiamata di questa procedura, costruendo un **caso base** e i casi successivi a partire da esso.
In queste dimostrazioni si procede spesso per **induzione sulla dimensione dell'argomento**, cioè sulla grandezza del parametro su cui agisce la ricorsione, che diminuisce a ogni chiamata fino al caso base (nel caso di $n!$, il parametro è n).

1.4 Invarianti dei cicli

Cos'è un invariante?

Un **invariante** è una condizione sempre vera in un certo punto del programma.

Cos'è un invariante di ciclo (o asserzione induttiva)?

Invece, possiamo definire **invariante di ciclo** una condizione sempre vera all'inizio dell'iterazione di un ciclo. Gli invarianti di ciclo sono importanti per dimostrare la correttezza di **algoritmi iterativi**.
Piu formalmente possiamo dire che: *Un **invariante di ciclo** è un asserto S che è vero ogni volta che ci si trova in un particolare punto del ciclo.*

L'asserto S viene poi dimostrato per induzione su un **parametro** che costituisce una **misura del numero di volte che il ciclo viene intrapreso**. Questo parametro può essere:
- Il numero di volte che abbiamo raggiunto la guardia di un ciclo;
- Oppure, il valore della variabile indice utilizzata dal ciclo stesso.

La dimostrazione segue quindi gli stessi passaggi di una dimostrazione induttiva:
- **Inizializzazione** (caso base): la condizione è vera alla prima iterazione di un ciclo;
- **Conservazione** (passo induttivo): se la condizione è vera prima di un'iterazione del ciclo, allora rimane vera al termine (quindi, prima della successiva iterazione);
- **Conclusione**: tutto ciò ha senso se, al termine, l'invariante rappresenta quello che voglio ottenere, qundi la "correttezza" dell'algoritmo.

1.4.1 Invariante del ciclo `while` VS invariante del ciclo `for`

In genere le dimostrazioni di correttezza per induzione usano il numero di iterazioni del ciclo per cui l'invariante vale. Quando la condizione diviene falsa, possiamo quindi utilizzare contemporaneamente l'invariante del ciclo e la falsita della condizione per concludere qualcosa di interessante su ciò che vale al termine del ciclo.

Utilizzare una tipologia di costrutto iterativo, rispetto ad un'altra può rendere una dimostrazione di correttezza di un ciclo più complessa. Ad esempio:

```
for (i = 0; i < n; i++) {
...
...
}
```

Utilizzando un ciclo *for* il contatore è integrato nella struttura stessa del ciclo.
Il valore iniziale, la condizione di uscita e l'aggiornamento (`i++`) sono tutti dichiarati in modo esplicito e standard. Si sa con certezza che il ciclo verrà eseguito un numero finito di volte (n), a meno che il corpo del ciclo non modifichi `i` in modo anomalo. Quindi, la terminazione è ovvia: quando $i = n$, il ciclo si ferma.

Invece, la struttura del *while* è più **generale** e **felssibile**.
Nel primo caso si usa un contatore i come nel *for*, quindi il comportamento è analogo, **ma il linguaggio non obbliga a farlo!**

Primo caso

```
i = 0;
while (i < n) {
...
i++;
}
```

Secondo caso

```
x = 5;
while (x != 0) {
x = f(x);    // f potrebbe non
   diminuire x!
}
```

Nel secondo caso non c'è un contatore "ovvio" che cresce o diminuisce in modo regolare, e non è certo che il ciclo finirà (magari `f(x)` restituisce sempre un numero diverso da 0, o l'utente non scrive mai "exit").

Proprio per questo, **parte della dimostrazione di correttezza di un ciclo *while* consiste nel dimostrarne la terminazione**. Solitamente, la dimostrazione di terminazione viene fatta determinando **un'espressione *E***, che **coinvolge le variabili del programma stesso**, tale che:
- Il valore di *E* **diminuisce (o aumenta)** di almeno un'unità **a ogni iterazione del ciclo**;

- Quando il ciclo si ferma (quindi la **condizione è falsa**), il valore di *E* è **pari ad una costante minima (o massima) prefissata**.

Dunque, basandosi sulle caratteristiche appena descritte, è possibile determinare l'espressione *E*, ad esempio, con la seguente dimostrazione.

***Esempio*: dimostrazione della terminazione di un ciclo `while`**

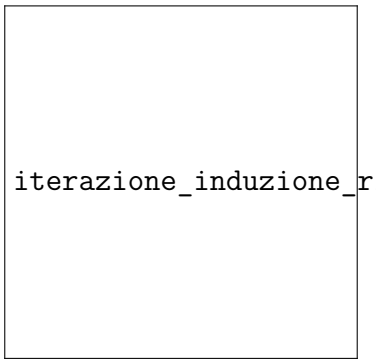
La seguente funzione "`Fattoriale`" calcola $n!$ passato da parametro, assumendo che $n \geqslant 1$. Lo scopo è quello di dimostrare che il ciclo `while` (*righe 4-6*) deve terminare.
Come detto precedentemente, per dimostrare la terminazione del ciclo bisogna determinare *E*.
Dunque, si cerca una grandezza che:

- parta positiva;
- diminuisca ad ogni iterazione;
- diventi negativa quando il ciclo finisce.

N.B: Per dimostrare la terminazione **si preferisce** trovare **una funzione *E* che decresce** verso un limite inferiore (è più facile da gestire nei teoremi di terminazione).
Una scelta che risulta naturale è $E = n - 1$, poiché:

- i è piccolo all'inizio, e dato che deve essere $i \leqslant n \Rightarrow n - 1$ è positivo;
- Ad ogni iterazione i aumenta di $1 \Rightarrow E$ diminuisce di 1;
- Quando $i = n + 1 \;\Rightarrow\; i > n \;\Rightarrow\; E = n - (n + 1) = -1$, cioè diventa negativo proprio quando il ciclo finisce.

A questo punto, dopo aver dimostrato la terminazione del ciclo `while`, è possibile dimostrare il funzionamento del codice tramite induzione.



Come definito in precedenza, l'invariante del ciclo è dato da un asserto, in questo caso $S(j)$.

Cosa afferma l'asserto?

Se si raggiunge il controllo del ciclo $i \leqslant n$ quando la variabile i ha un certo valore j, allora il valore della variabile `fatt` è `(j-1)!`

In altre parole, prima di ogni iterazione, `fatt` contiene il fattoriale del numero precedente a i. Quindi:

- quando $i = 2, fatt = 1!$;
- quando $i = 3, fatt = 2!$;
- ecc...

Dimostriamo che l'asserto vale sia per il caso base che per il passo induttivo:

1. **Caso base**: nel caso base, prima del ciclo, $fatt = 1$ e i ha un valore $j = 2$. Dunque $fatt = 1 = (j - 1) = 2 - 1 = 1$ e la **base è dimostrata**.
2. **Passo induttivo**: nel passo induttivo, assumendo che $S(j)$ sia vera per un generico valore $j \leqslant n$, è stato dimostrato che la variabile `fatt`, prima della nuova iterazione, sia $fatt =$

$(j-1)!$; Se si vuole dimostrare che anche $S(j+1)$, bisogna dimostrare che la variabile `fatt`, prima della nuova iterazione, sia $fatt = (j+1-1)! = j!$; Si distinguono due casi:

- Quando i ha valore $j > n$: il ciclo è già finito (la guardia non si attiva più), e dunque il valore di `fatt`, sarà il valore finale, ovvero j!;
- Quando i ha valore $j \leqslant n$: Dopo aver verificato che per $S(j)$ vale $fatt(j-1)!$ (con $i = j$) si esegue un'altra iterazione del ciclo.
  Dunque, in *riga 5* troviamo che $fatt = fatt \times i = (j-1)! \times j$ che equivale alla definizione di $j!$, mentre in *riga 6* si ha $i = i + 1 = j + 1$. Ora al prossimo controllo della prossima iterazione si avrà $fatt = (j-1)! \times j = j!$ e $i = j + 1$ che confermano l'asserzione $S(j+1)$.

1.4.2 Invarianti dei cicli - Esempio pratico con il `selectionSort()`

Obbiettivo del `selectionSort()`

Il `selectionSort()` prevede l'**ordinamento di un vettore** di n elementi i quali **vengono permutati** in modo che essi compaiano in un ordine non decrescente.

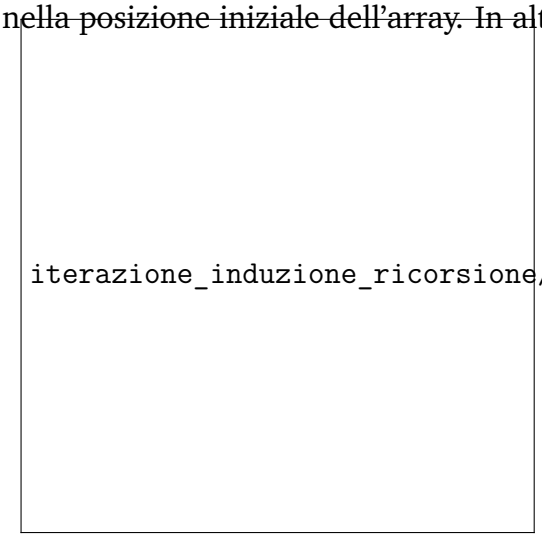
Possiamo descrivere il funzionamento del selection sort tramite il seguente pseudocodice:

iterazione_induzione_ricorsione/img5.png

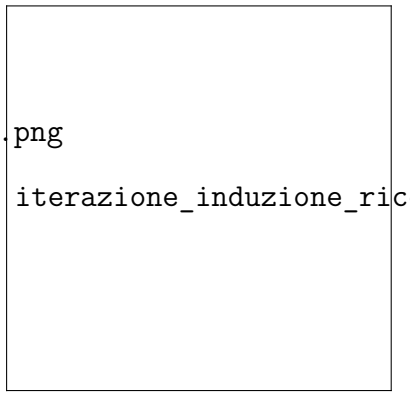
In particolare, come si può vedere in figura *(a)*, l'ordinamento avviene nel seguente modo:

1. Nella prima iterazione troviamo (selezioniamo) uno degli elementi piu piccoli tra tutti i valori in $A[1...n]$ e lo scambiamo con $A[1]$;
2. Nella seconda iterazione troviamo uno degli elementi piu piccoli tra tutti i valori in $A[2...n]$ e lo scambiamo con $A[2]$;
3. Dopo l'iterazione i-esima, $A[1...i]$ contiene gli i elementi piu piccoli ordinati in modo non decrescente;

Come si può vedere dall'immagine *(b)*, l'asserzione induttiva (o invariante di ciclo) è chiamata $T(m)$. Quest'ultima, come detto al capitolo **??**, è una condizione che deve essere vera immediatamente prima del controllo di terminazione del ciclo, cioè prima di controllare $i > n - 1$.
Quando i ha un certo valore m, sono già stati selezionati gli $m - 1$ elementi più piccoli e ordinati nella posizione iniziale dell'array. In altre parole:



(a)



(b)

- La parte sinistra dell'array $A[1...(m-1)]$ è già ordinata e contiene gli elementi più piccoli;
- La parte destra $A[m...n]$ è ancora da ordinare.

Questa è la proprietà $T(m)$, che **rimane vera ad ogni iterazione del ciclo** ($i = m$), quindi è proprio l'**invariante di ciclo**.

***Esempio*: dimostrazione tramite induzione**
Come detto al capitolo **??**, per poter dimostrare un asserto tramite induzione, possiamo utilizzare valore della variabile indice del ciclo stesso, in questo caso m.
Dunque, si dice che: ***si può dimostrare per induzione su*** m ***l'asserto*** $T(m)$.

Cosa afferma l'asserto?

Se si raggiunge il controllo del ciclo $i > n - 1$ alla (linea 2) con $i = m$, allora:
- Gli elementi $A[1...(m-1)]$ sono ordinati in senso non decrescente: $A[1] \leqslant A[2] \leqslant \ldots \leqslant A[m-1]$;
- Tutti gli elementi di $A[m...n]$ sono maggiori o uguali a ogni elemento di $A[1...(m-1)]$.

1. **Caso base**: quando $i = m = 1$ siamo all'inizio dell'algoritmo, e non abbiamo ancora ordinato nulla. Per vedere se $T(1)$ è vera si controlla che l'asserto sia corretto:
   - La parte (1) di $T(1)$ dice che $A[1...m-1] = A[1...0]$ è ordinato, ma $A[1...0]$ è vuoto, quindi è banalmente vero (una sequenza vuota è sempre ordinata).
   - La parte (2) di $T(2)$ dice che tutti gli elementi in $A[m...n] = A[1...n]$ sono maggiori o uguali a quelli in $A[1...m-1] = A[1...0]$, ma anche in questo caso gli elementi in $A[1...0]$ non esistono, quindi la condizione è automaticamente vera.

   Quindi T(1) è vera $\Rightarrow$ **caso base dimostrato**.

2. **Passo induttivo**: in questo passaggio si assume che $T(m)$ sia vera per un generico $m \leqslant n-1$, che verifica le condizioni dell'asserto, quindi:
   - La prima parte dell'array $A[1...(m-1)]$ è già ordinata;

- Tutti gli elementi compresi in $A[1...(m-1)]$ sono $\leqslant$ di ogni elemento compreso in $A[m...n]$.

Ora si vuole dimostrare che anche $T(m+1)$ sia vera. Durante l'iterazione con $i = m$:
- il ciclo interno (righe 3-9) **cerca il minimo** tra gli elementi non ordinati $A[m...n]$;
- poi **scambia** quel minimo con l'elemento $A[m]$ (quindi prima di tutti gli altri elementi non ordinati).

Anche dopo lo scambio rimane vero che:
- la porzione $A[1...m]$ è ora ordinata (perché il nuovo elemento in $A[m]$ è il più piccolo tra quelli rimanenti);
- tutti gli elementi in $A[(m+1)...n]$ rimangono $\geqslant$ di quelli precedenti.

Quindi $T(m+1)$ è vera perché subito dopo lo scambio il ciclo esterno incrementa la variabile i da m a $m+1$: *"Dato che ora il valore di i è $m+1$, l'asserzione $T(m+1)$ risulta vera."*

3. **Conclusione**: nell'ultima iterazione, quando $i = m = n$
   - i primi $n-1$ elementi $A[1...(n-1)]$ sono già ordinati e al posto giusto;
   - l'ultimo elemento $A[n]$ è automaticamente $\geqslant$ di tutti gli altri.

   Quando il programma termina, dunque, gli elementi di A sono in ordine non decrescente, cioè ordinati.