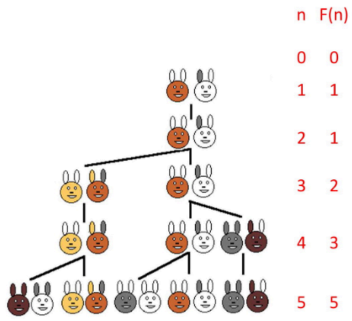
Lezione_05_DeA

2.5.1.3. Successione di Fibonacci - esempio

La *successione di Fibonacci* è definita così: $\begin{cases} F(0) = 0 \\ F(1) = 1 \\ F(n+1) = F(n) + F(n-1) \end{cases}$ $\forall n \geq 1$.

Si può pensare a $F(n)$ come il numero di coppie di conigli all'inizio del mese n se una coppia genera un'altra coppia ogni mese, a partire dal terzo mese, i conigli non muoiono, all'inizio del mese 1 c'è una coppia neonata.



Dimostrare che

$$F(n) = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n) \qquad (\implies F(n) \in \Theta(N\phi^n))$$

$\forall n \geq 0$, dove $\phi = \frac{1+\sqrt{5}}{2}$ (*golden ratio*) e $\hat{\phi} = \frac{1-\sqrt{5}}{2}$.

2.5.1.3.1. Induzione fallace

Dimostriamo $F(n) = 0$, $\forall n \geq 0$ $(n_0 = 0)$:

- $k = 0$
- Base: $F(0) = 0 \implies \checkmark$
- Passo induttivo: fissato $n \geq 0$ e assumendo $F(m) = 0$ pe rogni $0 \leq m \leq n$ (ipotesi induttiva), si ha $F(n+1) = F(n) + F(n-1) = 0 + 0$

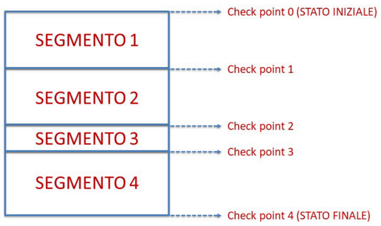
Il passo induttivo non è corretto quando $n = 0$, perché $F(n+1) = F(n) + F(n-1)$ vale solo per $n \geq 1$. Allora in questo caso devo usare una base più ampia: $n_0 = 0$, $k = 1$.

2.5.2. Correttezza

Per provare la correttezza di terminazione, cioè appunto la terminazione, è necessario assicurarsi che i cicli (inclusi i GOTO) e l'eventuale ricorsione abbiano termine.

Gli step dell'approccio generale alla soluzione del problema computazionale sono:

- *Definire lo stato iniziale* dell'algoritmo e quello *finale* che esso *deve raggiungere*;
- *Decomporre l'algoritmo in segmenti* e definire per ogni segmento lo stato in cui l'algoritmo si deve trovare del termine del segmento (*checkpoint*);
- *Dimostrare* che a partire dallo stato iniziale *si raggiungono* in successione *gli stati specificati* per la fine di ogni segmento. In particolare, lo stato che deve valere alla fine dell'ultimo segmento deve coincidere (o implicare) lo stato finale desiderato.
  I *segmenti notevoli* sono cicli (for, while, repeat-until).



> **Osservazione**
>
> I cicli sono i segmenti più difficili da analizzare (perché definiscono *implicitamente* molte operazioni) ma, insieme alla ricorsione, costituiscono gli strumenti essenziali per la scrittura di algoritmi o programmi interessanti.

2.5.3. Invariante

Per provare la correttezza di un ciclo (for, while, repeat-until, ...) bisogna dimostrare che al termine della sua esecuzione vale una certa *proprietà* $\mathcal{L}$ che rappresenta uno stato ed è *funzionale alla correttezza dell'algoritmo*. A tal fine si fa uso di un *invariante*.

> Un *invariante* per un ciclo è una *proprietà* espressa in funzione delle variabili usate nel ciclo, che descrive lo *stato* in cui si

> trova l'esecuzione alla fine di una generica iterazione del ciclo.

Nota bene

L'invariante deve essere scelto finalizzandolo alla correttezza e alla prova della proprietà $\mathcal{L}$.

2.5.4. Correttezza di un ciclo tramite invariante

Per dimostrare che alla fine del ciclo vale una certa proprietà $\mathcal{L}$, si individua un opportuno invariante e si dimostra che:

1. Esso *vale all'inizio del ciclo* (subito prima che il ciclo inizi);
2. Esso *vale alla fine di ciascuna iterazione*, che si dimostra induttivamente provando che se vale alla fine di una generica iterazione, vale alla fine della successiva;
3. *Alla fine dell'ultima iterazione, l'invariante implica la proprietà $\mathcal{L}$* (in alcuni casi coincide con essa).

Terminologia

L'esecuzione di un *ciclo* consiste di più di zero iterazioni delle istruzione in esso contenute (*corpo del ciclo*).
Ad esempio `for i<-1 to n do {corpo del ciclo}` è un ciclo che esegue sempre n iterazioni.

2.5.4.1. Esempio (`arrayMax`)

Algoritmo `arrayMax(A)`
Input: Array $A[0 \div n-1]$ di $n \geq 1$ interi.
Output: massimo intero di A.

```
currMax<-A[0];
for i<-1 to n-1 do{
    currMax<-max{currMax, A[i]};
}
return currMax;
```

Proprietà $\mathcal{L}$: alla fine del ciclo, $currMax$ è il massimo intero in A (quindi il valore restituito è corretto).

Invariante: Alla fine della iterazione $i \geq 0$, $currMax = max\{A[0], A[1], \ldots, A[i]\}$.

> **Osservazione**
>
> La fine dell'iterazione $i = 0$ corrisponde all'inizio del ciclo.

1. All'inizio del ciclo $(i = 0)$ l'invariante è reso vero dall'assegnamento $currMax < -A[0]$;
2. Disponiamo l'invariante uguale a vero a fine iterazione $i < n - 1 \implies currMax = max\{A[0], \ldots, A[i]\}$ e dimostriamo che vale anche alla fine della successiva iterazione.
   Nella iterazione $i + 1$ l'istruzione $currMax$ <- $max\{currMax, A[i+1]\}$ assicura che alla fine di tale iterazione $currMax$ <- $max\{A[0], \ldots, A[i+1]\}$;
3. Alla fine dell'ultima istruzione $(i = n - 1)$:
   Se vale'invariante, cioè se $currMax = max\{A[0], \ldots, A[n-1]\}$, allora $currMax$ è effettivamente il massimo intero, che è la proprietà $\mathcal{L}$.

2.5.4.2. Esempio (`arrayFind(A)`)

Algoritmo `arrayFind(A)`
Input: Elemento x, array $A[0 \div n-1]$ di n elementi.
Output: indice $i \in [0, n)$ tale che $A[i] = x$, se esiste, altrimenti -1.

```
i<-0;
while i < n do{
    if(x = A[i]) then return i;
    else i<-i+1;
}
return -1;
```

Proprietà $\mathcal{L}$: Il valore trovato è corretto.
Invariante: Alla fine di una generica iterazione $x \neq A[j] \ \forall 0 \leq j < i$ con i valore della variabile omonima alla fine dell'iterazione corrente.

1. All'inizio del ciclo $(i = 0)$ l'invariante vale per *vacuità* dato che il range $0 \div i - 1$ è vuoto;
2. Supponiamo l'invariante vero alla fine di una generica iterazione $\implies x \neq A[j] \ \forall 0 \leq j < i < n$.
   Alla fine della successiva iterazione:
   - Se $x = A[i] \implies i$ non cambia e si esce dal ciclo;

- Se $x \neq A[i] \implies i$ diventa $i+1$.
  In entrambi i casi l'invariante continua a valere;

3. Fine ultima iterazione, ci sono due possibili uscite:
   - $u_1 : x = A[i]$: In questo caso si restituisce i e chiaramente $\mathcal{L}$ vale;
   - $u_2 : i = n$ in questo caso si restituisce -1.
     Dato che l'invariante mi dice che $x \neq A[j]$ $\forall j : 0 \leq j < n = i$ so che x non è in A e quindi restituire -1 è corretto $\implies \mathcal{L}$ vale.

2.5.6. Ricorsione

> Un *algoritmo ricorsivo* è un algoritmo che invoca sé stesso (su istanze sempre più piccole) sfruttando la nozione di induzione.

La soluzione di un'istanza di taglia n è ottenuta *direttamente* se $n = n_0, n_0 + 1, \ldots, n_0 + k$ (casi base), altrimenti *riducendosi* alla soluzione di $r \geq 1$ istanze di taglia minore di n: se $n > n_0 + k$. Se $r = 1$ si parla di *linear recursion*.

2.5.6.1. Esempi

Algoritmo `linearSum(A,n)`
Input: array A, intero $n \geq 1$.
Output: $\sum_{i=0}^{n-1} A[i]$.

```
if (n = 1) then{
    return A[0];
}
else{
    return linearSum(A, n-1) + A[n-1];
}
```

Taglia dell'istanza: n; $n_0 = 1$ e $k = 0$.
Se voglio trovare la somma di tutti gli elementi di un array A, la prima invocazione sarà `linearSum(A,|A|)`.

Algoritmo `reverseArray(A,i,j)`
Input: array A, indici $i, j \geq 0$.
Output: array A con gli elementi in $A[i \div j]$ ribaltati.

```
if(i < j) then{
    swap(A[i], A[j]);
    reverseArray(A, i+1, j-1);
```

```
    }
    return
```

Taglia dell'istanza: $n = j - i + 1$.

Caso base: $n \leq 1$.

La prima invocazione per ribaltare tutto A è `reverseArray(A,0,|A|-1)`.