Esempi di Progetto di Algoritmi con Asserzione Logiche

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Lezione **2(b)**, **[29/9/23]**

SAPIENZA
UNIVERSITÀ DI ROMA

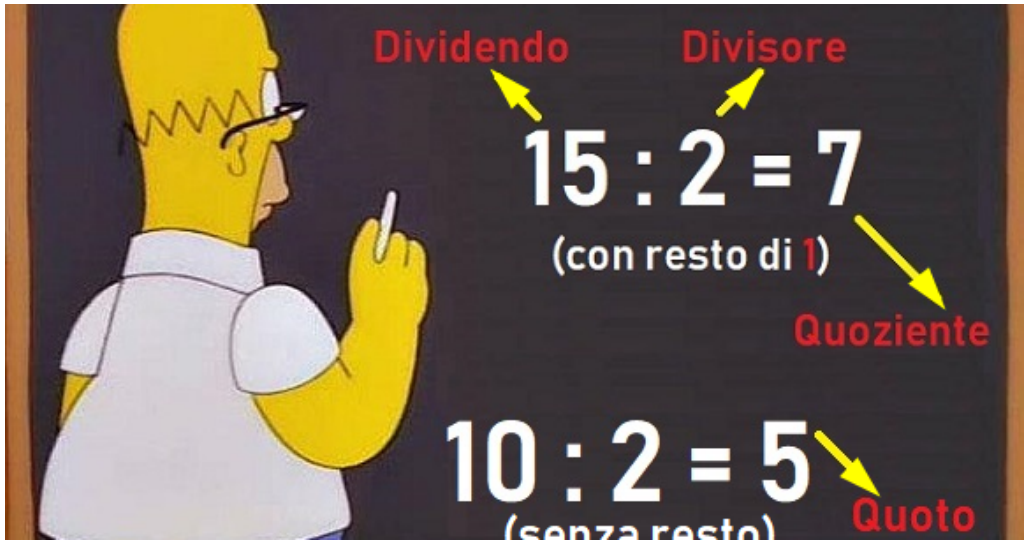
Asserzioni Logiche e progettazione

Vedremo che le **asserzioni logiche** possono risultare uno strumento molto utile non solo nella valutazione della correttezza dei programmi/algoritmi, ma anche come **guida nella progettazione di programmi/algoritmi corretti**.

L'idea è quella di cominciare dalle asserzioni finali e poi **a ritroso** stabilire i **comandi necessari** per soddisfarle, e **individuare** eventuali **precondizioni** necessarie.

Idea: Se devo soddisfare una certa proprietà ψ alla **fine di una computazione iterativa**, con guardia b, probabilmente **durante l'iterazione** posso soddisfare solo una proprietà φ **più debole** di ψ, ma tale che φ & not(b) $\Rightarrow \psi$ (**indebolimento**).

Ovviamente, come ogni metodologia… richiede comunque un po' di fantasia e intuizione…



Divisione Intera

Divisione Intera: specifica

Poniamoci il problema della **divisione intera** sui **naturali**.

a) Definizione della **specifica**: Dati $m \geq 0$, $n > 0$, occorre trovare due numeri: q (detto **quoziente**) ed r (detto **resto**) che soddisfano alla seguente relazione:

$$m = q \cdot n + r \quad \& \quad 0 \leq r < n$$

b) **Indebolimento** della specifica: La prima proprietà è soddisfatta da un numero infinito di coppie q, r (è di fatto, l'equazione di una retta). In particolare, ce n'è una banale quando $q = 0$, che suggerisce l'inizializzazione:

```
q, r = 0, m
```

Divisione Intera: invariante

c) Partendo da questi valori, possiamo pensare a un processo **iterativo di ricerca**, esplorando tutti i valori di q mantenendo l'invariante $m = q \cdot n + r$,

```
q, r = q+1, sub(r, n)
```

Infatti $q'n + r' = (q+1)n + r - n = qn + n + r - n = qn + r = m$

d) Sotto la precondizione $n > 0$, abbiamo che $r' = r - n < r$ e quindi r è una funzione decrescente e a valori positivi se $r > n$. E questo ci suggerisce anche la guardia del **while** che sarà:

```
magUg(r, n)
```

Importante ricordare che, per $r \geq n$ la chiamata `sub(r, n)` **soddisfa alle precondizioni** della funzione `sub`.

e) La negazione della guardia ($r < n$) e l'invariante implicano proprio la **postcondizione**: $m = q \cdot n + r \;\; \& \;\; 0 \leq r < n$

Divisione Intera: algoritmo

A questo punto, il programma si scrive da solo.

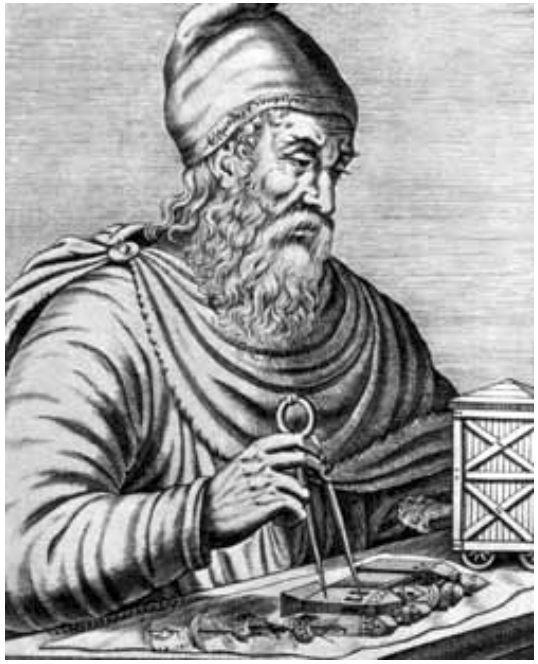
Osserviamo solo che questo programma è "*paghi uno, compri due*": infatti **calcola contemporaneamente quoziente** e **resto** della divisione intera, e infatti torniamo due valori.

Quante operazioni fa (complessità computazionale)?

Più spesso lo scriveremo come in **basso a destra**!

```
def div(m, n):
 # REQ: m≥0, n>0
 # ENS: ret q, r: m = q*n+r & 0<=r< n
   q, r = 0, m
   while magUg(n,r):
   # INV: m = q*n+r
      q, r = q+1, sub(r, n)
   return q, r
```

```
def div(m, n):
    q, r = 0, m
    while n ≥ r:
       q, r = q+1, r-n
    return q, r
```



Reinventiamo l'MCD di Euclide

Massimo comun divisore

Proviamo una strada alternativa per calcolare il **massimo comun divisore** di due numeri.

Chiaramente sappiamo **calcolare direttamente** il massimo comun divisore quando i due **numeri sono uguali**:

$$\mathrm{mcd}(n, n) = n$$

Il problema ammette delle **evidenti simmetrie**, infatti:

$$\mathrm{mcd}(m, n) = \mathrm{mcd}\ (n, m)$$

$$\mathrm{mcd}(m, n) = \mathrm{mcd}(-m, n) = \mathrm{mcd}(m, -n)$$

Infatti $-n$ ha gli stessi divisori di n.

Conosciamo con queste equazioni, il valore del mcd sulle due bisettrici del piano cartesiano (a coordinate intere, **esclusa l'origine**, visto che **mcd(0, 0) è indefinito**).

Proviamo a **estendere la nostra conoscenza** dei valori della funzione con **trasformazioni invarianti per mcd**.

Trasformazioni invarianti

Le **simmetrie** viste sopra sono da evitare in un processo di calcolo, in quanto essendo applicabili in tutte le direzioni, potrebbero portare a **computazioni non terminanti**.

Abbiamo però anche le seguenti uguaglianze:

$$\text{mcd}(m, n) = \text{mcd}\ (n + m, m) = \text{mcd}\ (n + m, n) \qquad (*)$$

$$\text{mcd}(m, n) = \text{mcd}(n - m, n) = \text{mcd}(n - m, m) \qquad (**)$$

e simili, ottenute per simmetria.

Ad esempio, se $d \mid m$ ("d divide m") e $d \mid n$ allora $m=k_1 d$ e $n=k_2 d$ per qualche k_1 e k_2 interi. E quindi $d \mid (m + n)$ e $d \mid (m - n)$ in quanto $m + n = d\ (k_1 + k_2)$, $m - n = d\ (k_1 - k_2)$.

Ciò significa che possiamo spostarci sul piano cartesiano, facendo **mosse invarianti per mcd**, applicando (*) e (**)

Resta da vedere se riusciamo a fare **mosse** che ci **avvicinano** verso i punti in cui il valore dell'mcd è noto in modo diretto (le **bisettrici**).

Terminazione: primo tentativo

Visto che l'**obiettivo** sono le coppie (n, n) è naturale pensare di usare i passi che fanno scendere la grandezza $m - n$ o $n - m$, che non fa differenza per simmetria ($\text{mcd}(m, n)=\text{mcd}(n, m)$), ad esempio col comando:

```
if m < n: n = n - m else: m = m - n
```

Una candidata funzione di terminazione è $|m - n|$. Tuttavia scopriamo che ciò **non funziona**, infatti:

- se $m > n > 0$, allora ho che $|m' - n'| = |(m - n) - n| = |m - 2n| < |m - n|$ solo se $n < {}^2/_3\, m$
- se $n > m > 0$, allora ho che $|m' - n'| = |(n - m) - m| = |n - 2m| < |m - n|$ solo se $m < {}^2/_3\, n$

Di conseguenza, la funzione $t(m, n) = |m - n|$ **non è una funzione di terminazione**, **strettamente decrescente** per un corpo di un ciclo contenente il comando visto sopra.

Terminazione: secondo tentativo

Tuttavia, col comando:

```
if m < n: n = n - m else: m = m - n
```
(*)

e assumendo $m, n > 0$ la quantità $m+n$ è strettamente decrescente.

D'altra parte, partendo con m, n strettamente positivi, il comando (*) azzererebbe m o n quando $m=n$, il che non ci interessa molto, perché in tal caso abbiamo finito e abbiamo trovato l'MCD.

Di conseguenza, la funzione $t(m, n) = m+n$ **è una funzione di terminazione, positiva** e **strettamente decrescente** nelle nostre ipotesi ($m, n > 0$) per un corpo di un ciclo contenente il comando (*).

MCD di Euclide/Dijkstra

Abbiamo riscoperto l'algoritmo di **Euclide** (~300 a.C.).
C'è una variante basata sui resti delle divisioni.

Questa presentazione è ispirata a uno scritto di **E. W. Dijkstra** (pubblicato in "*A Discipline of Programming*", 1978).

La versione coi resti **esegue meno iterazioni**, ma il calcolo del **resto** è **più complessa** di una **sottrazione**.

```
def mcd(m, n):
# REQ: m, n > 0
# ENS: return mcd(m, n)
    while m!=n:
    # INV: mcd(m,n)=mcd(m₀, n₀)
    # m, n ≥ 0, T = m + n
        if m < n: n = n-m
            else: m = m-n
    # H: m = n = mcd(m₀, n₀)
    return m
```

Addendum all'MCD

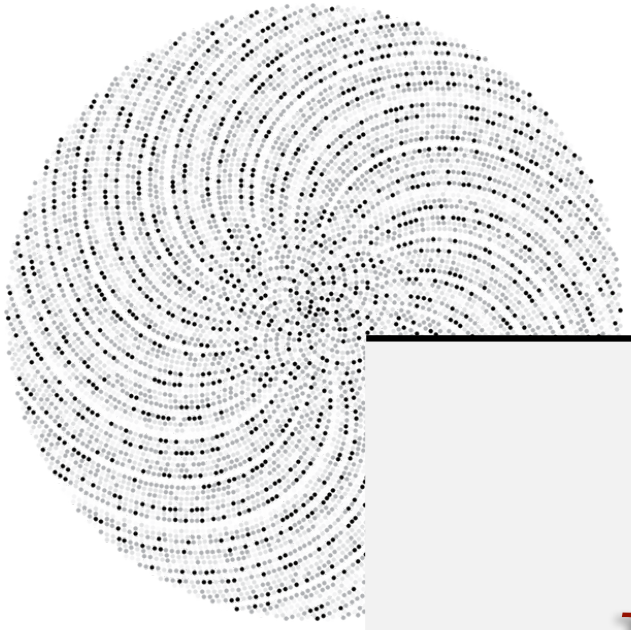
Sulle coppie (n, n), mcd(n, n) viene calcolato in **0** iterazioni. Rappresentando i razionali come coppie di interi, tutte le coppie (n, n) sono la classe di equivalenza del numero 1.

Basta **1** iterazione sulle coppie del tipo $(n, 2n)$ o $(2n, n)$. Queste sono le classi di equivalenza di 2 e 1/2.

Servono **2** iterazioni per le coppie del tipo $(n, 3n)$, $(3n, n)$, $(2n, 3n)$, $(3n, 2n)$. Queste sono le classi di equivalenza di 1/3, 3, 2/3 e 3/2.

Detto in altri termini, questo algoritmo **fa gli stessi passi di computazione** su tutte le **coppie di numeri** che rappresentano **lo stesso** numero **razionale**.

Esempio: mcd$(n, 1)$ fa n passi. Questo in qualche senso è il **caso pessimo**, in quanto servono $\sim m+n$ passi per ottenere il risultato ed $m+n$ è **il limite superiore** stabilito dalla **funzione di terminazione**.



Massimo Fattore Primo

Calcolare il massimo fattore primo

Problema: Calcolare il massimo fattore primo di un numero naturale n, usando la funzione `div`.

Idea 1: dovendo trovare il massimo, uno potrebbe scendere da n e cercare i divisori, ma occorrerebbe poi controllare che questi siano a loro volta primi.

Idea 2: generare i **fattori primi** di n in **ordine**, partendo da 2.

Possiamo evitare di trovare divisori composti, dividendo n ogni volta che si trova un divisore d (primo) e quindi evitando in seguito di avere un numero divisibile per i composti di d.

Seguiremo la seconda strada.

Calcolare il massimo fattore primo

Chiamando n_0 il valore iniziale di n possiamo cercare di mantenere l'invariante $\varphi(p, n) = \varphi_1(p, n)\ \&\ \varphi_2(p, n)$ dove:

$$\varphi_1(p, n) \equiv \forall\, d \in [2, p).\ d \nmid n$$

$$\varphi_2(p, n) \equiv \forall\, d \in [p, +\infty).\ d \text{ è primo } \&\ d \mid n_0 \Rightarrow d \mid n$$

La proprietà $\varphi_1(p, n)$ implica che se $p \mid n$, allora p è primo: se non lo fosse, sarebbe divisibile per qualche $d \in [2, p)$, ma allora d dividerebbe anche n.

La proprietà $\varphi_2(p, n)$ implica invece che troviamo **tutti i divisori primi** di n_0 (non ce li perdiamo dividendo per p)

Dovremo anche registrare l'ultimo divisore trovato in una variabile m, cosicché sarà vero $m = \max_{d \in [2, p)} d \nmid n_0$.

Calcolare il massimo fattore primo

Vediamo che φ_1 e φ_2 possono essere mantenute **invarianti**.

Inizializzazione:

$\varphi_1(2, n)$ vale banalmente perché l'intervallo $[2, 2)$ è vuoto.
$\varphi_2(2, n)$ vale banalmente perché $n = n_0$.

Conservazione

- Se $p \nmid n$ allora ho stabilito $\varphi_1(p+1, n)$ e anche $\varphi_2(p+1, n)$, perché p **non è** un **nuovo** divisore primo di n_0, altrimenti dividerebbe anche n (visto che vale $\varphi_2(p, n)$).

  Quindi è sufficiente incrementare p.

- Se ho $p \mid n$ è vero banalmente $\varphi_1(p, {}^{n}/_{p})$ perché ${}^{n}/_{p}$ ha al più gli stessi divisori di n, ma ${}^{n}/_{p}$ ha **gli stessi divisori primi** di n **maggiori o uguali** di p e quindi ho anche $\varphi_2(p, {}^{n}/_{p})$.

  Quindi aggiorno n con ${}^{n}/_{p}$.

Calcolare il massimo fattore primo

Terminazione: sotto l'ipotesi $p \geq 2$, la funzione $t = n - p$ è una funzione di terminazione, perché a ogni iterazione o incremento p, oppure divido n per p.

Questo suggerisce la **guardia** $p < n$, che implica $n - p > 0$, cioè la funzione di terminazione è a valori **positivi**.

Ecco il codice (senza asserzioni per motivi di spazio).

```
def maxPrimo(n):
    p = 2
    while p < n:
        q, r = div(n, p)
        if r == 0:    # n divisibile per p
            m = p     # registro un nuovo massimo
            n = q     # divido n
        else: p = p+1 # provo il prossimo divisore
    return m
```

Erratum

La **guardia** $p^*p < n$, garantisce come $p < n$ la terminazione.

Inoltre n è un composto di divisori di n_0 per la proprietà $\varphi_2(p, n)$.

Ma n è primo (non può avere altri divisori maggiori di p e gli eventuali divisori primi $d < p < n$, non sono fattori di n per la proprietà $\varphi_1(p, n)$).

Quindi ***n* è il massimo fattore primo di n_0**.

```
def maxPrimo(n):
    p = 2
    while p * p < n:
        q, r = div(n, p)
        if r == 0:    # n divisibile per p
            n = q     # divido n
        else: p = p+1 # provo il prossimo divisore
    return n
```

Esempio: generatore di fattori primi

Quali sono il caso ottimo e pessimo del **programma** che **genera** i **fattori primi**?

Caso ottimo: n è un numero "molto" composto, con molti fattori primi "piccoli". Ad esempio se $n = 2^k$, allora il programma genererà k volte il fattore primo 2, in sole k **iterazioni.** In questo caso, $k = \log_2 n$.

Caso pessimo: n è un numero primo. In tal caso, l'algoritmo testa la divisibilità per tutti i numeri tra 2 e n e quindi termina in n-2 iterazioni.

- **Esempio**: Per **36** si fanno **5** iterazioni, per **37** se ne fanno **35**.

Oppure: per $\mathbf{2^k}$ se ne fanno $\boldsymbol{k}$, se $\mathbf{2^k-1}$ è **primo** se ne fanno $\mathbf{2^k-3}$.

Caso medio: decisamente difficile da determinare, ma correlato al numero medio di fattori primi di un natura

- **Esercizio**: velocizzare l'algoritmo, fermando quando $p*p>n$ (attenzione). Quanto migliora

qui contiamo il numero delle iterazioni, poi entreremo in maggiori dettagli

That's all Folks!

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Lezione **2(b)** [**29/9/23**]

SAPIENZA
UNIVERSITÀ DI ROMA