Soluzioni degli esercizi

Queste soluzioni sono proposte soprattutto per favorire un'acquisizione progressiva delle conoscenze. Bisogna partire dall'assunto che esse *non* siano le uniche o le migliori soluzioni. Prima di studiare queste soluzioni, ognuno deve cercare in autonomia le *proprie*, che potranno anche essere molto diverse da quelle proposte. Alcune delle soluzioni seguenti potrebbero essere incomplete e presentare solo alcune idee per risolvere gli aspetti più critici del problema.

In queste proposte di soluzione noterete che i nomi delle variabili, i commenti ecc. sono in inglese. Un suggerimento è quello di provare a operare sul codice per esempio "*traducendolo*" in italiano in modo da riflettere sulla sua logica e il suo contenuto.

Esercizi capitolo 3 - Iterazioni

Cerchi in riga

Cerchiamo una relazione del tipo $x = m \cdot i + q$. Per il primo cerchio, $i = 0$; $x = q = r$. Ogni cerchio sarà distante $m = 2 \cdot r$ da quello precedente. Inoltre, nella larghezza L del canvas deve essere occupata da n diametri, quindi $L = 2 \cdot n \cdot r \implies r = \frac{L}{2 \cdot n}$.

▶️ https://fondinfo.github.io/play/?p13_bluerow.py

Cerchi concentrici

Dobbiamo stabilire di quanto dobbiamo modificare il raggio e il colore dei cerchi prima di iniziare le iterazioni. I cerchi vanno disegnati dal più grande al più piccolo, in caso contrario risulterebbe visibile solo l'ultimo disegnato. Possiamo invertire l'intervallo dei valori in modo che il raggio e la quantità di colore rosso crescano con i.

```
R = 250
g2d.init_canvas((2 * R, 2 * R))

n = int(g2d.prompt("Circles? "))
r = R / max(n, 1)  # radius: r_fst = r_m = r
c = 255 / max(n - 1, 1)  # color

for i in reversed(range(n)):
    g2d.set_color((c * i, 0, 0))
    g2d.draw_circle((R, R), r * i + r)

g2d.main_loop()
```

▶️ https://fondinfo.github.io/play/?p13_redcircles.py

Il colore e il raggio devono variare linearmente con $i \in [0, n-1]$. Quindi l'espressione sarà del tipo $m \cdot i + q$ e di questa dovremo individuare il valore dei parametri.

Ragioniamo sul raggio e in particolare sui suoi due valori estremi. Se il canvas è di 500px, allora il raggio più esterno sarà di 250px (per comodità, chiamiamolo R). Per dividere equamente lo spazio il raggio più interno sarà $\frac{R}{n}$. Calcoliamo dunque i coefficienti di linearità m e q.

$$i = 0; m \cdot i + q = q = \frac{R}{n}$$

$$i = n-1; m \cdot i + q = m \cdot (n-1) + \frac{R}{n} = R \implies m = \frac{R}{n}$$

$$radius = i \cdot \frac{R}{n} + \frac{R}{n}$$

Il colore invece parte dal nero, per $i = 0$, e arriva al rosso pieno, per $i = n - 1$.

$$i = 0; m \cdot i + q = q = 0$$

$$i = n-1; m \cdot i + q = m \cdot (n-1) = 255 \implies m = \frac{255}{n-1}$$

$$red = i \cdot \frac{255}{n-1}$$

Impostiamo l'intervallo del ciclo con `reversed(range(n))` per iniziare dal cerchio di dimensione maggiore. Per evitare un possibile denominatore nullo utilizziamo `max` per far in modo che questo risulti sempre ≥ 1.

Se invece, come ulteriore esercizio, volessimo invertire i colori dei cerchi, dovremmo considerare $i = 0; m \cdot i + q = q = 255$. Poi $i = n-1; m \cdot i + q = m \cdot (n-1) + 255 = 0 \implies m = -\frac{255}{n-1}$.

$$red_{inv} = 255 - i \cdot \frac{255}{n-1}$$

Quadrati in diagonale

Riprendiamo l'esempio già mostrato, per disporre dei quadrati lungo una linea diagonale. L'analisi del problema resta valida, tranne per il fatto che l è da determinare in funzione di n. Esattamente $n + 1$ semilati dei quadrati da disegnare devono coprire il lato L del canvas.

$$L = (n+1) \cdot \frac{l}{2} \implies l = \frac{2 \cdot L}{n+1}$$

▶️ https://fondinfo.github.io/play/?p13_squares.py

Resistenze con sentinella

```
total = 0
total_inv = 0

val = float(input("Value? "))
while val > 0:
    total += val
    total_inv += 1 / val
    val = float(input("Value? "))

if total_inv > 0:
    print(total, 1 / total_inv)
```

▶ https://fondinfo.github.io/play/?p13_resistors.py

Si tratta di un ciclo con sentinella. Sommiamo le resistenze per calcolare l'equivalente in serie e sommiamo l'inverso delle resistenze per calcolare l'equivalente in parallelo. Attenzione alla divisione per zero.

Somma di potenze di 2

```
n = int(input("n? "))
total = 0
for i in range(n):
    total += 2 ** i
print("The sum is", total)
print("Gauss' formula is", total == 2 ** n - 1)
```

Facciamo una prova sperimentale con $n = 4$:

$$2^0 + 2^1 + 2^2 + 2^3 = 15 = 2^4 - 1$$

In numerazione binaria, che presenteremo nel capitolo sulla rappresentazione dei dati, il risultato si può esprimere anche in questa forma:

$$1111_{(2)} = 10000_{(2)} - 1$$

Quadrato perfetto

Come primo passo possiamo testare i numeri da `1` a `n` e verificare se il quadrato di uno di questi è uguale al valore ricevuto in input.

```
n = int(input("n? "))
i = 1
while i <= n:
    if i * i == n:
        print("Perfect square of", i)
    i += 1
```

Qual è la condizione in cui stampare che *non* si tratta di un quadrato perfetto? Si potrebbe provare ad aggiungere un `else`, che però in quella posizione sarebbe sbagliato. Inoltre, possiamo notare che non ha senso continuare la ricerca quando `i * i > n`. Cambiamo quindi la condizione di permanenza nel ciclo, eliminando contestualmente anche la condizione di uguaglianza.

```
i = 1
while i * i < n:
    i += 1
```

A questo punto, ci sono due condizioni possibili di uscita dal ciclo:

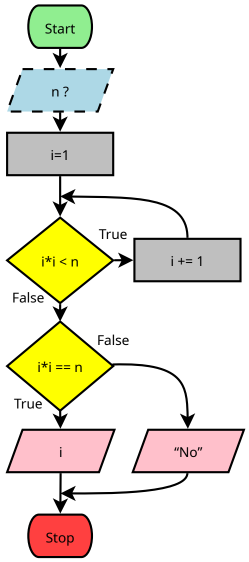
- `i * i == n` $\implies$ n è un quadrato prefetto
- `i * i > n` $\implies$ n non è un quadrato prefetto

Non ci resta quindi che distinguere tra questi due casi, all'uscita dal ciclo, con una istruzione `if-else`.

```
n = int(input("n? "))

i = 1
while i * i < n:
    i += 1

if i * i == n:
    print("Perfect square of", i)
else:
    print("Not a perfect square")
```



▶️ https://fondinfo.github.io/play/?p13_perfect.py

Numero segreto

```
from random import randint

MAX_SECRET = 90
MAX_TRIES = 10
guess = 0
tries = 0
secret = randint(1, MAX_SECRET)

while secret != guess and tries < MAX_TRIES:
    guess = int(input("your guess? "))
    tries += 1
```

```
    if secret < guess:
        print("The secret is smaller than", guess)
    elif secret > guess:
        print("The secret is larger than", guess)

if secret == guess:
    print("Congratulations, you guessed in", tries, "tries")
else:
    print("No luck! The secret was", secret)
```

▶ https://fondinfo.github.io/play/?p13_secret.py

La stanza del mostro

```
from random import randrange
W, H = 5, 5
player = monster = gold = (0, 0)
while monster == player:
    monster = (randrange(W), randrange(H))
while gold == player or gold == monster:
    gold = (randrange(W), randrange(H))

while player != monster and player != gold:
    direction = input(f"Position: {player}. Direction (w/a/s/d)? ")
    x, y = player  # unpacking
    if direction == "w" and y > 0:
        y -= 1
    elif direction == "a" and x > 0:
        x -= 1
    elif direction == "s" and y < H - 1:
        y += 1
    elif direction == "d" and x < W - 1:
        x += 1
    player = (x, y)

if player == gold:
    print(f"Position: {player}. Gold!")
else:
    print(f"Position: {player}. Monster!")
```

▶ https://fondinfo.github.io/play/?p13_monster.py

Due tuple risultano uguali se, a due a due, gli elementi in posizione corrispondente sono tutti uguali. Risultano diverse se almeno in un caso gli elementi in posizione corrispondente differiscono.

Provare a risolvere l'esercizio anche senza usare le tuple. Serve applicare con attenzione la proprietà di De Morgan:

$$\neg(a \wedge b) == (\neg a) \vee (\neg b)$$