

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 9 - Strutture dati bidimensionali

Incolonnamento dati

Per stampare i dati a console, bisogna procedere necessariamente riga per riga, da sinistra a destra (come su una macchina da scrivere). Però, in ogni posizione raggiunta, possiamo scegliere quale carattere scrivere. Nel codice sotto, vengono mostrate due formule, per mostrare la tabella ASCII per colonne o per righe.

```
FIRST, LAST = 32, 126
COLS, ROWS = 24, 4

for y in range(ROWS):
    for x in range(COLS):
        i = FIRST + x * ROWS + y
        # i = FIRST + x + y*COLS
        if i <= LAST:
            print(chr(i), end=" ")
    print()
```

 https://fondinfo.github.io/play/?p42_ascii.py

Funzione di smooth

Prima di tutto, ci concentriamo sulla parte *core* del problema: calcolare la media nell'intorno di una certa cella. Definiamo una funzione `avg` che svolga questo compito. Si può risolvere questo sottoproblema in molti modi diversi.

L'approccio qui proposto è basato sulla definizione a priori dell'intorno di una cella, chiamato `dirs`. Per ogni spostamento relativo, definito in `dirs`, accediamo alla cella vicina, se le sue coordinate (`x1`, `y1`) sono valide. Se la cella vicina esiste

effettivamente nella matrice, la contiamo e sommiamo il suo valore al totale. Alla fine, viene restituita la media ottenuta.

La funzione `smooth`, a questo punto, non deve fare altro che eseguire il calcolo `avg` per ogni cella della matrice originale. Il risultato, cella per cella, è aggiunto a una nuova matrice chiamata `result`. Volendo, si può fare tutto con una `comprehension`, su due `range`.

```
def avg(matrix: list, cols: int, rows: int, x0: int, y0: int) -> float:
    dirs = [(0, 0), (0, -1), (1, 0), (0, 1), (-1, 0)]
    count, total = 0, 0
    for dx, dy in dirs:
        x1, y1 = x0 + dx, y0 + dy
        if 0 <= x1 < cols and 0 <= y1 < rows:
            count += 1
            total += matrix[x1 + y1 * cols]
    return total / count

def smooth(matrix: list[float], cols: int, rows: int) -> list[float]:
    # return [avg(matrix, cols, rows, x, y)
    #         for y in range(rows) for x in range(cols)]
    result = []
    for y in range(rows):
        for x in range(cols):
            val = avg(matrix, cols, rows, x, y)
            result.append(val)
    return result
```

 https://fondinfo.github.io/play/?p42_smooth.py

Spirale

Anche questo problema si può risolvere in molti modi diversi. Il suggerimento è di mantenere come stato dell'applicazione sia la posizione attuale che la direzione di avanzamento: `x`, `y`, `dx`, `dy`. A ogni passaggio, aggiungeremo `dx` e `dy` a `x` e `y`, rispettivamente. Tutto questo è abbastanza semplice.

Bisogna prestare attenzione al punto in cui bisognerà cambiare direzione. A questo scopo, definiamo una funzione ausiliaria chiamata `available`. Essa valuta se una cella faccia parte della matrice e abbia valore 0 (no ancora visitata).

Per svoltare, bisogna cambiare i valori di `dx` e `dy`. Potremmo mantenere una lista di direzioni, partendo dalla prima (verso l'alto) e poi preseguiendo in senso orario, come è richiesto di ruotare.

```
[(0, -1), (1, 0), (0, 1), (-1, 0)]
```

Si può mantenere un indice su questa lista e farlo avanzare per selezionare la direzione successiva. Altrimenti, si può notare che i nuovi valori di `dx`, `dy` dipendono da quelli precedenti, secondo le seguenti formule. In effetti, queste sono le formule di rotazione di 90° rispetto all'origine, su un piano raster.

$$\begin{cases} dx' = -dy \\ dy' = dx \end{cases}$$

Vediamo dunque la soluzione complessiva.

```
def available(m: list, w: int, h: int, x: int, y: int) -> bool:
    return 0 <= x < w and 0 <= y < h and m[x + y * w] == 0

def spiral(w: int, h: int) -> list:
    m = [0] * (w * h)

    # initially: bottom-left cell, heading up
    x, y = 0, h - 1
    dx, dy = 0, -1

    for i in range(h * w):
        m[x + y * w] = i + 1
        # against border or visited cell?
        if not available(m, w, h, x + dx, y + dy):
            # turn 90° clockwise, raster: ('x', 'y') = (-y, x)
            dx, dy = -dy, dx
        x, y = x + dx, y + dy
    return m
```

 https://fondinfo.github.io/play/?p42_spiral.py

Lights out

Ci adeguiamo all'interfaccia BoardGame. Quindi, creiamo e inizializziamo una matrice nel metodo `__init__`. Poi accettiamo le mosse dell'utente con un metodo `move`. Mostriamo il contenuto di una cella con il metodo `read`.

Il modo più semplice e naturale, per rappresentare il gioco, è una matrice di booleani. Per generare la matrice di partenza, bisogna scegliere un numero di mosse casuali. La funzione `sample` permette di estrarre un numero desiderato di valori diversi, da un certo range. Nelle posizioni scelte (convertite in coordinate (x, y)) si esegue una normale mossa.

Eseguire una mossa significa cambiare il valore di cinque celle. Seguiamo l'impostazione usata già per l'esercizio sullo *smooth*, definendo il vicinato come una lista di tuple (dx, dy) .

La funzione `_get` ha l'istruzione `return` esplicita solo nel corpo di un `if`. Qualsiasi funzione però ha `None` come risultato implicito, se non indicato diversamente.

```
class LightsOut(BoardGame):
    """https://en.wikipedia.org/wiki/Lights_Out_(game)"""

    def __init__(self, w=5, h=5, level=4):
        self._w, self._h = w, h
        self._board = [False] * (w * h)
        for i in sample(range(w * h), level):
```

```

        self.play(i % w, i // w, "")
        self._solved = False
        # TODO: count available moves (= level)

    def _get(self, x, y) -> bool | None:
        if 0 <= x < self._w and 0 <= y < self._h:
            return self._board[y * self._w + x] # if outside, None

    def play(self, x: int, y: int, action: str):
        if self._get(x, y) != None: # is the cell within the board?
            for dx, dy in [(0, 0), (0, -1), (1, 0), (0, 1), (-1, 0)]:
                v = self._get(x + dx, y + dy)
                if v != None: # is this neighbor cell in the board?
                    self._board[(y + dy) * self._w + x + dx] = not v
            self._solved = not any(self._board)

```

 https://fondinfo.github.io/play/?p42_lightout.py

La stanza del mostro

Occorre disporre gli elementi in posizioni casuali, senza sovrapposizioni. La maniera più efficace è di disporli in ordine in una lista e poi mescolarla. Facciamo questa operazione su una lista di dimensione ridotta di una unità. In questo modo, possiamo inserire poi in testa una cella libera, come richiesto.

Per rispettare l'interfaccia BoardGame, facciamo queste operazioni nel costruttore. Segniamo in un campo `n` il numero di tesori che restano da scoprire.

Eseguiamo le mosse nel metodo `play`. Sono possibili solo le mosse nelle caselle a distanza 1 dalla posizione attuale. Se si trova un tesoro, si segna il tesoro nella cella come “scoperto” e si decrementa il contatore dei tesori da scoprire. Se si trova un orco, la partita è persa.

```

FREE, ORC, GOLD, FOUND, OUT = range(5) # 5 constants


class Quest(BoardGame):
    def __init__(self, w: int, h: int, golds: int, orcs: int):
        rest = w*h - golds - orcs - 1
        if rest < 0:
            raise ValueError("Too many golds and orcs")
        bd = [GOLD]*golds + [ORC]*orcs + [FREE]*rest
        shuffle(bd)
        self._bd, self._w, self._h = [FREE] + bd, w, h
        self._x = self._y = 0
        self._n, self._dead = golds, False

    def _get(self, x, y) -> int: # OUT if outside of board
        bd, w, h = self._bd, self._w, self._h
        return bd[x + y*w] if (0 <= x < w and 0 <= y < h) else OUT

    def play(self, x: int, y: int, action: str):
        v = self._get(x, y) # TODO : optionally allow diagonal moves
        if v != OUT and abs(x - self._x) + abs(y - self._y) == 1:
            if v == GOLD:

```

```
        self._bd[x + y*self._w] = FOUND
        self._n -= 1
    self._dead = v == ORC
    self._x, self._y = x, y
```

 https://fondinfo.github.io/play/?p42_quest.py