lezione16

November 16, 2023

1 Fondamenti di Programmazione

Andrea Sterbini

lezione 16 - 16 novembre 2023

[1]:
```
%load_ext nb_mypy
```

```
Version 1.0.5
```

2 RECAP:

- Colori come oggetti (con matematica dei colori)
- Immagini come matrici di Colori
- Filtri come oggetti che generano il colore del nuovo pixel
- Figure geometriche

3 METODOLOGIA di analisi Object Oriented

- Si individuano le strutture dati necessarie (**class**)
- con i loro **attributi** (informazioni "personali" di ciascun dato)
  - se una informazione è identica e comune a tutti gli individui la posso mettere come **attributo di classe**
- si definisce la loro inizializzazione (`__init__`) con gli argomenti necessari
- i **metodi** fondamentali (operazione sui dati o che calcolano valori a partire dagli attributi)
  - se vogliamo creare l'oggetto in altri modi possiamo creare un **@classmethod**

Ogni volta che vogliamo aggiungere una funzionalità ci dobbiamo chiedere quale oggetto deve essere "responsabile" (oppure "conosce le informazioni") per calcolarla

Metafora dell'ufficio - E' un po' come voler organizzare un ufficio con persone che hanno **tipi di mansioni** diverse - ciascuno ha le sue informazioni e si cerca di non assegnarle a più uffici (classi) - gli scambi tra impiegati devono essere minimi

4 Ereditarietà (Specializzazione/ Riuso e Ampliamento delle funzionalita)

Se si vede che diverse classi condividono dei comportamenti/attributi comuni - definiamo una superclasse che contiene l'implementazione comune dei metodi o gli attributi comuni - nelle sottoclassi

che ereditano da questa mettiamo solo gli attributi diversi ed i metodi diversi (se necessario c'è anche il modo di chiamare i metodi della superclasse)

Esempio: tutti gli animali visualizzati in un gioco hanno una posizione, una icona, un verso … l'insieme degli attributi e metodi comuni può essere messo nella classe Animale da cui ereditano Cane, Gatto, Cavallo, Ornitorinco ….

[1]:
```
# Tutte le Figure hanno:
# - posizione
# - colore
# - un metodo che le disegna (che per default non fa nulla)
# - un metodo che ne calcola l'area (per default 0)

# Ciascuna specifica figura
# - ha dei parametri specifici (raggio, lati, cateti, fuochi, retta e fuoco, ...
 ↪)
# - ha il suo metodo specifico che la disegna
# - ha il suo metodo specifico che ne calcola l'area

# definiamo la gerarchia di classi:
# Figura
#   Punto
#   Linea
#       Freccia
#   Rettangolo
#       Quadrato
#       TriangoloRettangolo
#   PoligonoRegolare
#       Triangolo
#       Pentagono
#   EllisseApprossimataDaCerchi
#       Cerchio
import turtle
t = turtle.Turtle()
turtle.colormode(255)
```

[2]:
```
class Figura:
    def __init__(self, x, y, colore, direzione=0, spessore=1, campitura=None):
        self._x = x
        self._y = y
        self._colore = colore
        self._direzione = direzione
        self._spessore = spessore
        self._campitura = campitura

    def area(self):
        raise NotImplementedError("Il metodo 'area' non è stato implementato")
```

```
    def draw(self, turtle):
        turtle.end_fill()
        turtle.up()
        turtle.goto(self._x,self._y)
        turtle.setheading(self._direzione)
        turtle.color(self._colore)
        turtle.pensize(self._spessore)
        turtle.down()
        if self._campitura:
            turtle.fillcolor(self._campitura)
            turtle.begin_fill()

    def move(self, x, y):
        self._x = x
        self._y = y
```

[9]:
```
#help(turtle)
```

[3]:
```
class Rettangolo(Figura):
    def __init__(self, x, y, colore, larghezza, altezza,
                 direzione=0, spessore=1, campitura=None):
        "creo un nuovo rettangolo"
        # riuso l'inizializzazione della mia superclasse per
        # gli attributi che già sa gestire
        super().__init__(x, y, colore, direzione, spessore, campitura)
        # gestisco qui gli attributi aggiuntivi
        self._larghezza = larghezza
        self._altezza = altezza

    def area(self):
        "calcolo l'area del rettangolo"
        return self._larghezza * self._altezza

    def draw(self, turtle):
        """
        Disegno il rettangolo son la tartaruga fornita come parametro.
        """
        super().draw(turtle)
        turtle.forward(self._larghezza)
        turtle.left(90)
        turtle.forward(self._altezza)
        turtle.left(90)
        turtle.forward(self._larghezza)
        turtle.left(90)
        turtle.forward(self._altezza)
        turtle.left(90)
```

```
        turtle.end_fill()
```

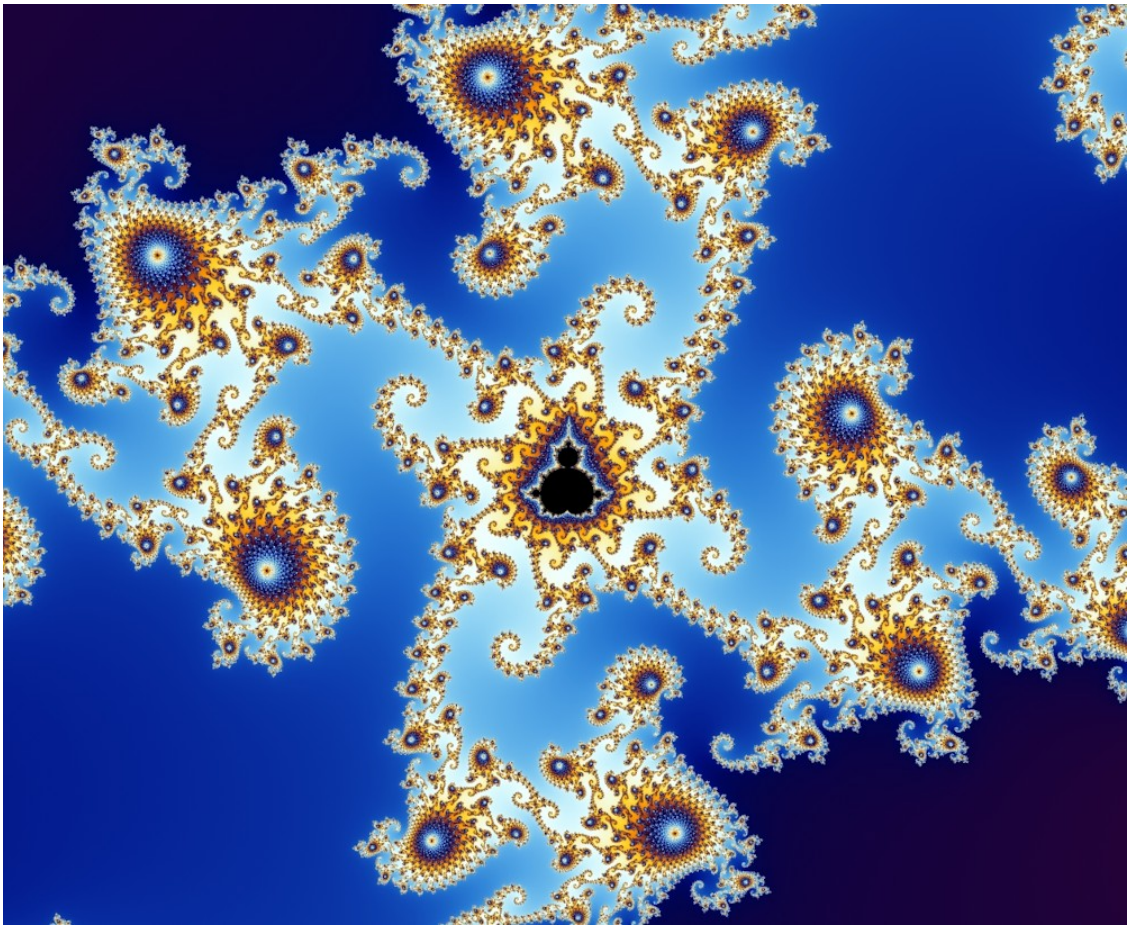
[4]:
```
class PoligonoRegolare(Figura):
    "Costruisco un poligono regolare"
    def __init__(self, x, y, lato, N, colore,
                 direzione=0, spessore=1, campitura=None):
        super().__init__(x, y, colore, direzione, spessore, campitura)
        self._N = N
        self._lato = lato

    def draw(self, turtle):
        super().draw(turtle)
        angolo_esterno = 360/self._N
        for _ in range(self._N):
            turtle.forward(self._lato)
            turtle.left(angolo_esterno)
        turtle.end_fill()


class Quadrato(Rettangolo):
    def __init__(self, x, y, lato, colore,
                 direzione=0, spessore=1, campitura=None):
        super().__init__(x, y, colore, lato, lato,
                         direzione, spessore, campitura )
```

[6]:
```
t.clear()

r = Rettangolo(50, 100, (255,0,0), 200, 100,
               direzione=45, spessore=5,
               campitura=(255,255,0))

p = PoligonoRegolare(-100, -100, 50, 5, (0,0,255),
                     20, 4, campitura=(255,0,0) )

q = Quadrato(100, 100, 90, (0,255,0), -30, 10)

r.draw(t)

r.move(-200,-100)

r.draw(t)

p.draw(t)

q.draw(t)

print(r.area(), q.area())
```

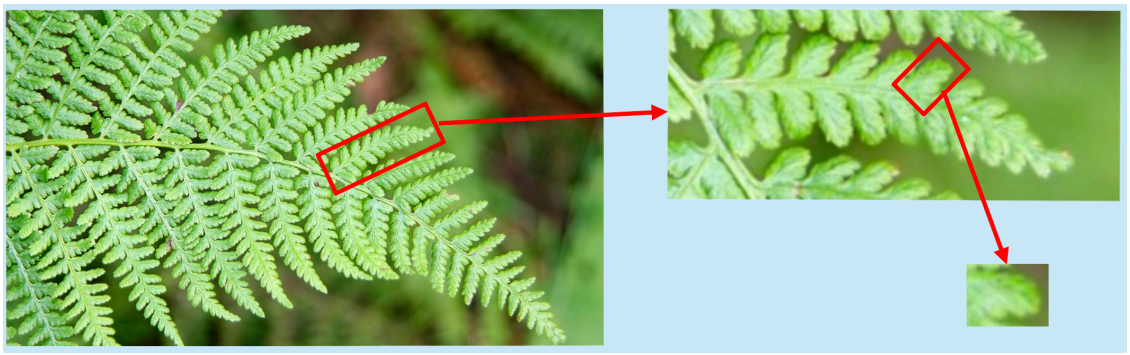
```
20000 8100
```

[10]:
```
t.clear()
t.hideturtle()
```

4.1 i Frattali (insieme di Mandelbrot)

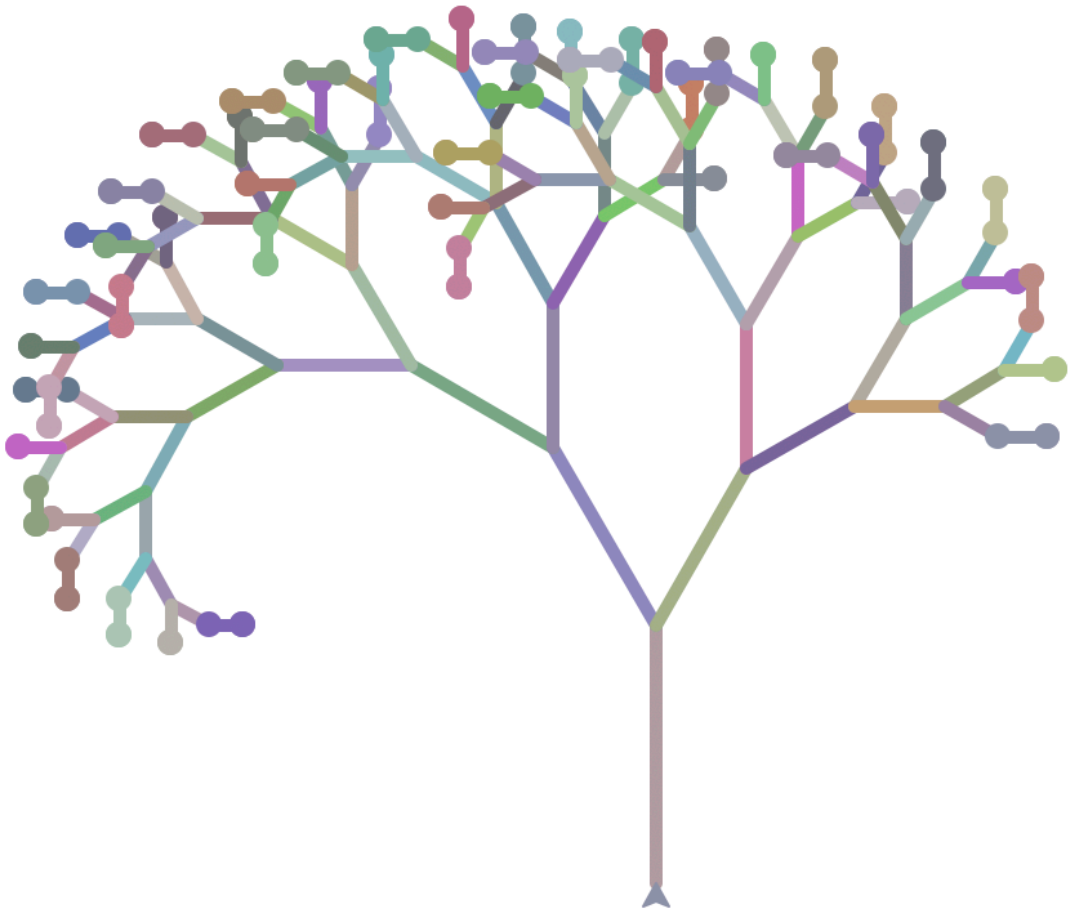


4.1.1 I frattali sono tra noi



Sono composti da **parti più piccole** che hanno la **stessa struttura del tutto**

4.1.2 Disegnamo un albero frattale



5 cos'è un albero frattale?

- ha un tronco che regge due rami
- i rami sono inclinati a sinistra e a destra del tronco
- ciascun ramo ha la **stressa struttura** di un albero ma leggermente **più piccolo**
- un albero di **livello <= zero è una foglia**

6 Problemi e Soluzioni ricorsive

Una funzione **è ricorsiva se chiama se' stessa** (principio di induzione).

6.1 Un problema ammette una soluzione ricorsiva se:

- SAPPIAMO COME rimpicciolire la dimensione del problema da risolvere (**riduzione**)
- esiste almeno un problema che ha una soluzione elementare (**caso base**)
- è sempre possibile, applicando ripetutamente la riduzione, arrivare ad uno dei casi base (**convergenza**)

- SAPPIAMO COME ottenere la soluzione del problema iniziale dalle soluzioni dei sottoproblemi (**composizione**)

6.2 cerchiamo le proprietà del problema ricorsivo

- Un albero di livello <= 0 è una foglia (cerchietto) (**caso base**)
- Un albero con N livelli è formato da: (**ricomposizione**)
  - un tronco lungo X
  - un **albero con N-1 livelli** inclinato a sinistra, con tronco 80% di X (**riduzione**)
  - un **albero con N-2 livelli** inclinato a destra, con tronco 70% di X (**riduzione**)
- sottraendo 1 o 2 ad N si arriva sempre ad un valore <= 0 (**convergenza**)

Questa è una definizione **ricorsiva**: i due rami sono due alberi! abbiamo **riduzione, caso base, convergenza e ricomposizione**!

Mi serve uno strumento di disegno: - col modulo **turtle** posso tracciare movimenti relativi alla tartaruga - col modulo **random** posso generare colori casuali

[11]:
```
# Per disegnare un albero frattale uso una tartaruga e dei numeri casuali
import turtle
from random import randint      # generatore di interi casuali
turtle.colormode(255)           # setto i colori in modalità RGB
t = turtle.Turtle()             # creo una tartaruga
t.penup()                       # alzo la penna
t.left(90)                      # giro verso l'alto
t.back(200)                     # mi posiziono in basso
t.pensize(5)                    # con penna cicciotta
t.speed(0)                      # e velocità alta
#help(turtle)                   # see also      (oppure 'pydoc turtle')
```

[12]:
```
def albero(t, tronco, angolo, livelli):
    '''disegno un albero con un certo tronco iniziale e # di livelli
    Argomenti:
        t:       la tartaruga a cui dare i comandi
        tronco:  lunghezza del tronco
        angolo:  inclinazione dei rami rispetto al tronco
        livelli: quanti livelli di rami disegnare
    '''
    if livelli <= 0:        # se caso base
        draw_leaf(t)        # disegno la "foglia"
    else:                   # altrimenti caso ricorsivo
        # disegno il tronco (e mi sposto alla sua fine)
        draw_trunk(t, tronco)
        # mi giro a sinistra
        t.left(angolo)
        # disegno il ramo sinistro, più piccolo 80% e con un livello di meno
        albero(t, tronco * 0.8, angolo, livelli-1)
        # mi giro a destra
        t.right(angolo*2)
```

```
        # disegno il ramo destro, più piccolo 70% e con due livelli in meno
        albero(t, tronco * 0.7, angolo, livelli-2)
        # torno nella direzione iniziale
        t.left(angolo)
        # torno alla base del tronco
        t.back(tronco)
```

[13]:
```
# devo definire come disegnare il tronco e la foglia
def draw_trunk(t, lunghezza):
    'Disegno un tratto di colore casuale'
    # cambio colore a caso
    R = randint(100, 200)
    G = randint(100, 200)
    B = randint(100, 200)
    t.color(R,G,B)
    # abbasso la penna
    t.pendown()
    # mi muovo in avanti di lunghezza pixel
    t.forward(lunghezza)
    # alzo la penna
    t.penup()
    # NOTA: ora sono all'estremo opposto del tronco
```

[14]:
```
def draw_leaf(t):
    'disegno una foglia col colore corrente'
    # abbasso la penna
    t.pendown()
    # disegno un pallino
    t.dot()
    # alzo la penna
    t.penup()
```

[16]:
```
## Vediamo se funziona :-)
t.clear() # pulisco il foglio
albero(t,100,30,10)
```

[15]:
```
## Vediamo se funziona :-)
t.clear() # pulisco il foglio
albero(t,100,20,9)
```

6.3 Esempio classico: il fattoriale

6.4 DEF: il fattoriale di un numero N intero positivo è il prodotto dei numeri da 1 a N positivo

- come ridurre il problema? **diminuisco N di 1**
- quale soluzione semplice conosco? **F(1) = 1**
- come ottengo F(N) da F(N-1)? **lo moltiplico per N**

- il passo di riduzione converge al caso base? **Sì, sottraendo 1 a N alla fine si arriva ad 1**

6.5 Esempio classico: fattoriale(N)

proprietà	esempio
caso base	`F(1) = 1`
riduzione	`F(N) -> F(N-1)`
convergenza	`N, N-1, ..... 1`
composizione	`F(N) = N*F(N-1)`

6.6 Schema di implementazione ricorsiva

```
def funzione(problema):
    if is_caso_base(problema):
        return soluzione nota
    else:
        sottoproblema  = riduzione(problema)
        sottosoluzione = funzione(sottoproblema)
        soluzione      = composizione(sottosoluzione)
        return soluzione
```

[17]:
```
from rtrace import trace

@trace(pause=True)
def fattoriale(N : int) -> int :
    if N==1:
        return 1
    else:
        return N*fattoriale(N-1)

fattoriale.trace(5)
```

```
------------------- Starting recursion -------------------

 entering       fattoriale(5,)
|-- entering    fattoriale(4,)
|--|-- entering fattoriale(3,)
|--|--|-- entering      fattoriale(2,)
|--|--|--|-- entering   fattoriale(1,)
|--|--|--|-- exiting    fattoriale(1,)  returns 1
|--|--|-- exiting       fattoriale(2,)  returns 2
|--|-- exiting  fattoriale(3,)  returns 6
|-- exiting     fattoriale(4,)  returns 24
 exiting        fattoriale(5,)  returns 120

-------------------- Ending recursion --------------------
Num calls: 5
```

[17]: 120

6.7 Altro caso classico: i coniglietti di Fibonacci

- Una coppia di conigli il primo mese è giovane e non prolifica
- dal secondo mese in poi fa una coppia di coniglietti ogni mese

Ad ogni mese il numero di coppie si ottiene sommando: - i nuovi coniglietti che nascono dalle coppie adulte (quelle di 2 mesi prima) - e le coppie correnti (1 mese prima)

6.7.1 Soluzione iterativa (simulando le coppie)

- F(0)=0
- F(1)=1
- ...
- F(N)=F(N-1)+F(N-2)

6.7.2 Soluzione ricorsiva

Se osserviamo il problema dal punto di vista dell'anno N: - **caso base**: 0 coppia se N==0 - **caso base**: 1 coppia se N==1 (erano troppo giovani) - **riduzione del problema**: Per calcolare F(N) ci servono F(N-1) ed F(N-2) - **composizione della soluzione**: F(N) = F(N-1) + F(N-2)

[18]:
```
@trace(pause=True)
def fibonacci(N : int) -> int :
    if N < 2:
        return 1
    else:
        return fibonacci(N-1) + fibonacci(N-2)

fibonacci.trace(4)
```

```
------------------- Starting recursion -------------------

 entering       fibonacci(4,)
|-- entering    fibonacci(3,)
|--|-- entering fibonacci(2,)
|--|--|-- entering      fibonacci(1,)
|--|--|-- exiting       fibonacci(1,)   returns 1
|--|--|-- entering      fibonacci(0,)
|--|--|-- exiting       fibonacci(0,)   returns 1
|--|-- exiting  fibonacci(2,)   returns 2
|--|-- entering fibonacci(1,)
|--|-- exiting  fibonacci(1,)   returns 1
|-- exiting     fibonacci(3,)   returns 3
|-- entering    fibonacci(2,)
|--|-- entering fibonacci(1,)
|--|-- exiting  fibonacci(1,)   returns 1
|--|-- entering fibonacci(0,)
|--|-- exiting  fibonacci(0,)   returns 1
```

```
|-- exiting     fibonacci(2,)   returns 2
 exiting        fibonacci(4,)   returns 5

-------------------- Ending recursion --------------------
Num calls: 9
```

[18]: 5

7 METODO: le 4 proprietà vi guidano per affrontare un problema ricorsivo sconosciuto:

- trovate il **caso base**
- confrontate **problemi** di dimensioni diverse per capire come fare la "riduzione"
- confrontate **soluzioni** di dimensioni diverse per capire come calcolare la "soluzione"
- **verificate che** applicando più volte il passo di riduzione **si arrivi sempre ad un caso base** (convergenza)
- altrimenti aumentate i casi base

7.1 E' sempre possibile simulare un ciclo con una ricorsione

7.2 E' sempre possibile simulare una ricorsione con uno o più cicli (e se necessario una pila/stack)

- Iacopini Bohm

[20]:
```python
#  0 1 1 2 3 5 8 13 21

def fibonacci_iter(N : int) -> int :
    coppie = [0, 1]
    for i in range(N):
        coppie.append(coppie[-1] + coppie[-2])
    print(coppie)
    return coppie[-1]

print(fibonacci_iter(70))
# MA ATTENZIONE!!!
# E' sufficiente ricordasi SOLO dei DUE mesi precedenti!
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584,
4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229,
832040, 1346269, 2178309, 3524578, 5702887, 9227465, 14930352, 24157817,
39088169, 63245986, 102334155, 165580141, 267914296, 433494437, 701408733,
1134903170, 1836311903, 2971215073, 4807526976, 7778742049, 12586269025,
20365011074, 32951280099, 53316291173, 86267571272, 139583862445, 225851433717,
365435296162, 591286729879, 956722026041, 1548008755920, 2504730781961,
4052739537881, 6557470319842, 10610209857723, 17167680177565, 27777890035288,
44945570212853, 72723460248141, 117669030460994, 190392490709135,
308061521170129]
308061521170129
```

[21]:
```python
# Mi ricordo SOLO dei DUE mesi precedenti
def fibonacci_iter2(N : int) -> int :
    corrente, precedente = 1, 0
    for i in range(N):
        corrente, precedente = corrente+precedente, corrente
    return corrente

print(fibonacci_iter2(70))
```

```
308061521170129
```

[22]:
```python
# simuliamo l'allevamento anno per anno da 1 ad N con la ricorsione
# - convertiamo le variabili di stato del ciclo in argomenti della funzione
@trace()
def _fibonacci_ric_efficiente(N : int, i : int,
                              corrente : int,
                             precedente : int):
    if i == N:
        return corrente
    else:
        corrente, precedente = corrente+precedente, corrente
        return _fibonacci_ric_efficiente(N, i+1, corrente, precedente)

# definiamo una funzione di appoggio che inizializza le variabili di stato
def fibonacci_ric_efficiente(N : int):
    return _fibonacci_ric_efficiente(N, 0, 1, 0)

# oppure potevamo dare dei valori di default appropriati alle "variabili di␣
 ↪stato"

_fibonacci_ric_efficiente.trace(7, 0, 1, 0)
```

```
------------------- Starting recursion -------------------
 entering       _fibonacci_ric_efficiente(7, 0, 1, 0)
|-- entering    _fibonacci_ric_efficiente(7, 1, 1, 1)
|--|-- entering _fibonacci_ric_efficiente(7, 2, 2, 1)
|--|--|-- entering      _fibonacci_ric_efficiente(7, 3, 3, 2)
|--|--|--|-- entering   _fibonacci_ric_efficiente(7, 4, 5, 3)
|--|--|--|--|-- entering        _fibonacci_ric_efficiente(7, 5, 8, 5)
|--|--|--|--|--|-- entering     _fibonacci_ric_efficiente(7, 6, 13, 8)
|--|--|--|--|--|--|-- entering  _fibonacci_ric_efficiente(7, 7, 21, 13)
|--|--|--|--|--|--|-- exiting   _fibonacci_ric_efficiente(7, 7, 21, 13) returns
21
|--|--|--|--|--|-- exiting      _fibonacci_ric_efficiente(7, 6, 13, 8)  returns
21
|--|--|--|--|-- exiting         _fibonacci_ric_efficiente(7, 5, 8, 5)   returns
21
|--|--|--|-- exiting    _fibonacci_ric_efficiente(7, 4, 5, 3)   returns 21
```

```
|--|--|-- exiting       _fibonacci_ric_efficiente(7, 3, 3, 2)   returns 21
|--|-- exiting  _fibonacci_ric_efficiente(7, 2, 2, 1)   returns 21
|-- exiting     _fibonacci_ric_efficiente(7, 1, 1, 1)   returns 21
 exiting        _fibonacci_ric_efficiente(7, 0, 1, 0)   returns 21
-------------------- Ending recursion --------------------
Num calls: 8
```

[22]: 21

[24]:
```python
# proviamo invece a lavorare in uscita dalla ricorsione
# - ciascuna chiamata ritorna la coppia *corrente, precedente*
# ovvero calcoliamo F(N) -> corrente, precedente
# - nel caso base la coppia è           1, 0
# - da corrente, precedente del mese prima (N-1) posso calcolare quelli di N
@trace()
def fibonacci_efficiente(N : int ) -> tuple[int,int] :
    if N == 0:
        return 1, 0        # all'inizio ci sono 0 ed 1 coppia
    else:
        # ottengo i due valori del mese precedente (F(N-1) e F(N-2))
        corrente, precedente = fibonacci_efficiente(N-1)
        # calcolo i due valori per questo mese     (F(N)   e F(N-1))
        return corrente+precedente, corrente

print(fibonacci_efficiente(5)[0])

fibonacci_efficiente.trace(5)
```

```
8
------------------- Starting recursion -------------------
 entering       fibonacci_efficiente(5,)
|-- entering    fibonacci_efficiente(4,)
|--|-- entering fibonacci_efficiente(3,)
|--|--|-- entering      fibonacci_efficiente(2,)
|--|--|--|-- entering   fibonacci_efficiente(1,)
|--|--|--|--|-- entering        fibonacci_efficiente(0,)
|--|--|--|--|-- exiting         fibonacci_efficiente(0,)        returns (1, 0)
|--|--|--|-- exiting    fibonacci_efficiente(1,)        returns (1, 1)
|--|--|-- exiting       fibonacci_efficiente(2,)        returns (2, 1)
|--|-- exiting  fibonacci_efficiente(3,)        returns (3, 2)
|-- exiting     fibonacci_efficiente(4,)        returns (5, 3)
 exiting        fibonacci_efficiente(5,)        returns (8, 5)
-------------------- Ending recursion --------------------
Num calls: 6
```

[24]: (8, 5)