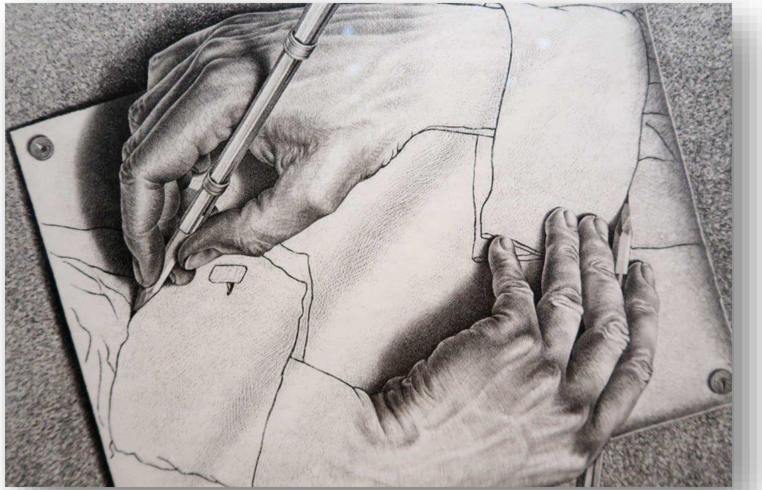
UNIVERSITÀ DI PARMA

ricorsione

informatica e laboratorio di programmazione



definizioni

- un oggetto si dice ***ricorsivo*** se è definito totalmente o parzialmente in termini di ***se stesso***
- la ricorsione è un mezzo molto potente per le definizioni e le dimostrazioni matematiche (***induzione***)
- si usano algoritmi ricorsivi quando il problema da risolvere presenta caratteristiche proprie di ricorsività (può essere risolto in termini di uno o più problemi analoghi ma di dimensioni inferiori)

immagine ricorsiva

definizioni ricorsive

- definizione dei ***numeri naturali***:
  1) 1 è un numero naturale
  2) il successore di un numero naturale è un numero naturale
- definizione di ***fattoriale*** di un numero intero positivo:
  1) 0! = 1
  2) n! = n * (n-1)!
- calcolo del ***MCD*** tra due numeri A e B (A>B)
  algoritmo di Euclide
  1) dividere A per B
  2) se il resto R è zero
     allora MCD(A,B)=B
     altrimenti MCD(A,B)=MCD(B,R)

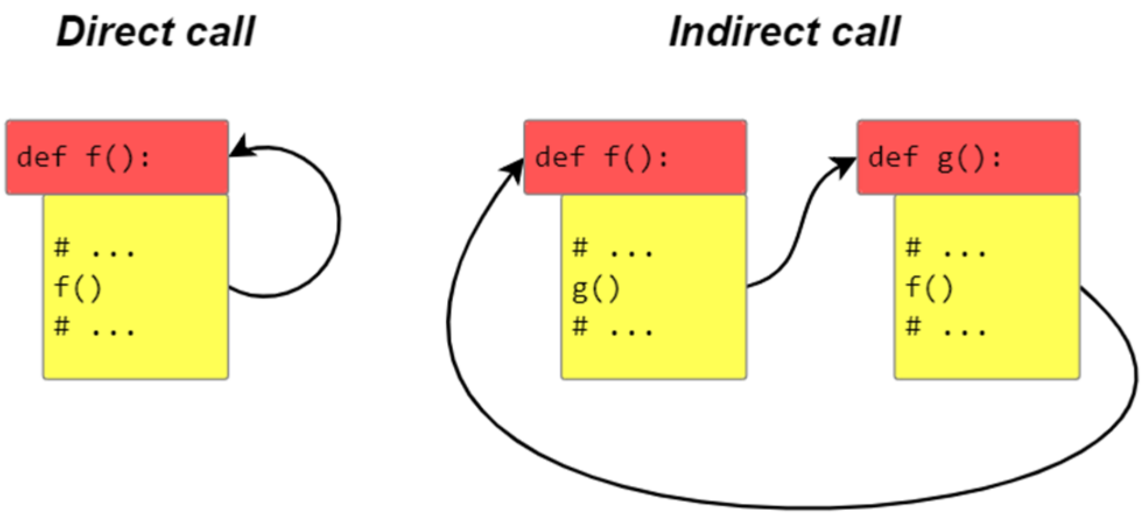
```
def mcd(a: int, b:int) -> int:
    r = a % b
    if (r==0):
        return b    #condizione di terminazione
    else:
        return(mcd(b,r))
```

terminazione

- il potere della ricorsività consiste nella possibilità di definire un insieme anche infinito di oggetti con un numero finito di comandi
- il problema principale quando si usano algoritmi ricorsivi è quello di garantire una ***terminazione*** (caso terminale, condizione di ***fine***, condizione iniziale)
- non è sufficiente inserire una condizione di terminazione, ma è necessario che le chiamate ricorsive siano tali da determinare il ***verificarsi*** di tale condizione in un numero finito di passi

procedure e funzioni ricorsive

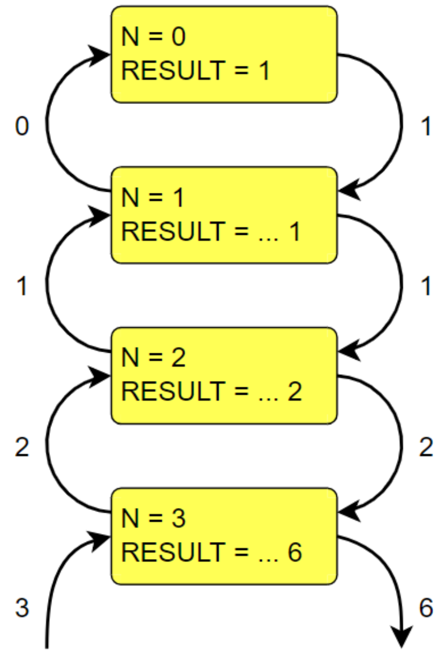
- un sottoprogramma ricorsivo è una procedura (o ***funzione***) all'interno della quale è presente una ***chiamata a se stessa*** o ad altro sottoprogramma che la richiama
- la ricorsione è ***diretta*** se la chiamata è interna al sottoprogramma altrimenti si dice indiretta
- molti linguaggi consentono ad una funzione (o procedura) di chiamare se stessa



fattoriale: ricorsione

- Ad ogni invocazione di una funzione, viene creato nello **stack** un nuovo record
- **Contesto locale** alla particolare attivazione della funzione stessa

```
def factorial(n: int) -> int:
    result = 1
    if n > 1:
        result = n * factorial(n - 1)
    return result
```



Ai primordi (Fortran 66 ecc.) solo allocazione statica
Spazio fisso ed unico per dati locali ad una funzione → no ricorsione

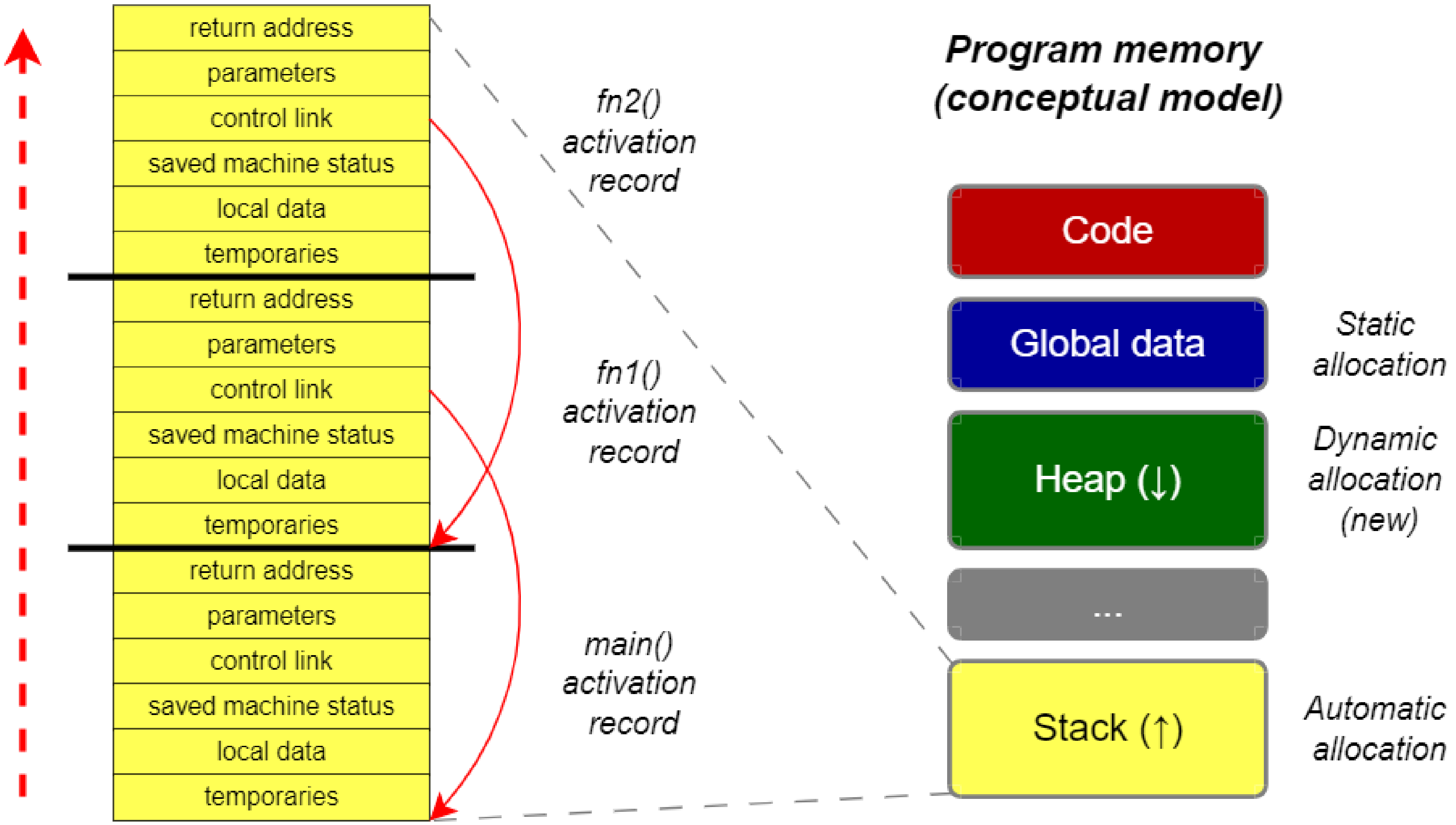
istanziazione

- ogni nuova chiamata di un sottoprogramma ricorsivo determina una ***nuova istanza*** dell'ambiente locale (distinto da quello precedente che comunque resta attivo)
- ad ogni chiamata si alloca ***nuova memoria*** e questo può determinare problemi di spazio
- i vari ambienti vengono salvati in una struttura di tipo ***LIFO*** (Stack o Pila) in modo che alla terminazione di una determinata istanza venga riattivata quella immediatamente precedente e così via

stack dell'applicazione

- ***pila***: memoria dinamica LIFO (Last In First Out)
  - dimensione massima prefissata
- il programma ci memorizza automaticamente:
  - ***indirizzo*** di ritorno per la funzione
    - inserito alla chiamata, estratto all'uscita
  - ***parametri*** della funzione
    - inseriti alla chiamata, eliminati all'uscita
  - ***variabili locali***, definite nella funzione
    - eliminate fuori dall'ambito di visibilità

record di attivazione

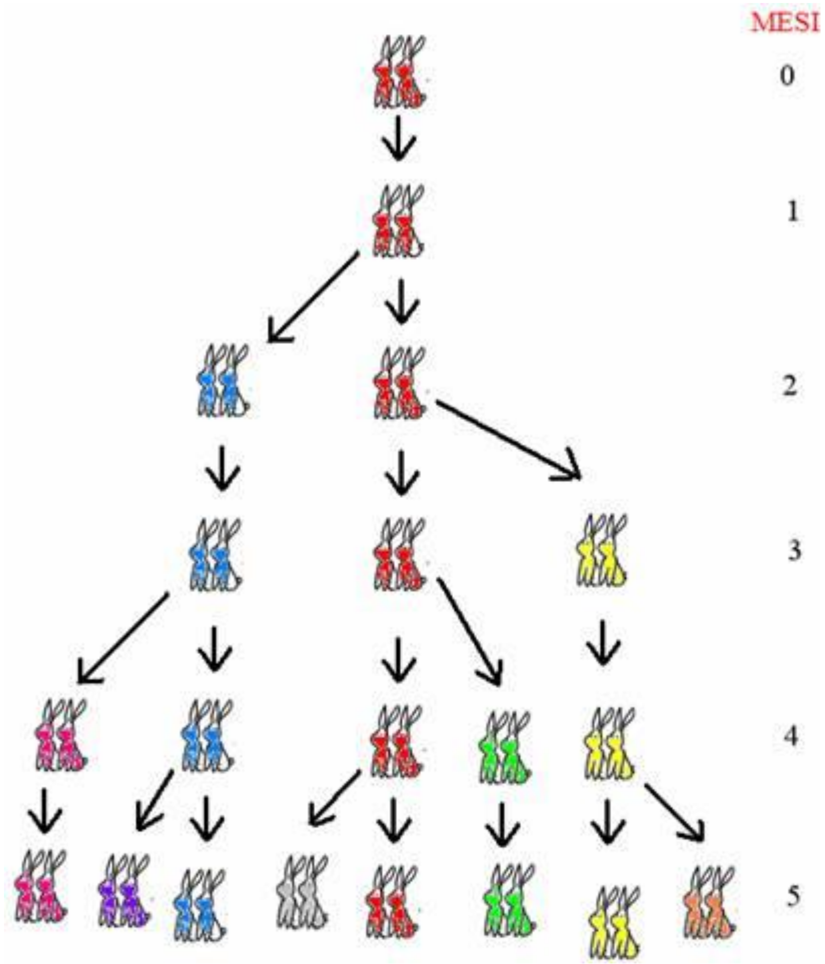


variabili: visibilità

- ***visibilità*** ⇒ insieme di istruzioni da cui è ***accessibile***
- ***ciclo di vita*** ⇒ ***esistenza*** in memoria della variabile (etichetta)
- i valori (oggetti) in Python sono tutti gestiti dinamicamente
- visibilità ***globale***
  - variabili fuori da ogni funzione - meglio ***evitare***!
  - allocazione statica in alcuni linguaggi
- visibilità ***locale alla funzione***
  - variabili locali e parametri
  - allocazione automatica di spazio in stack ad ogni attivazione della funzione (possibile la ricorsione)
- visibilità ***locale al blocco*** (es. if): non in Python!

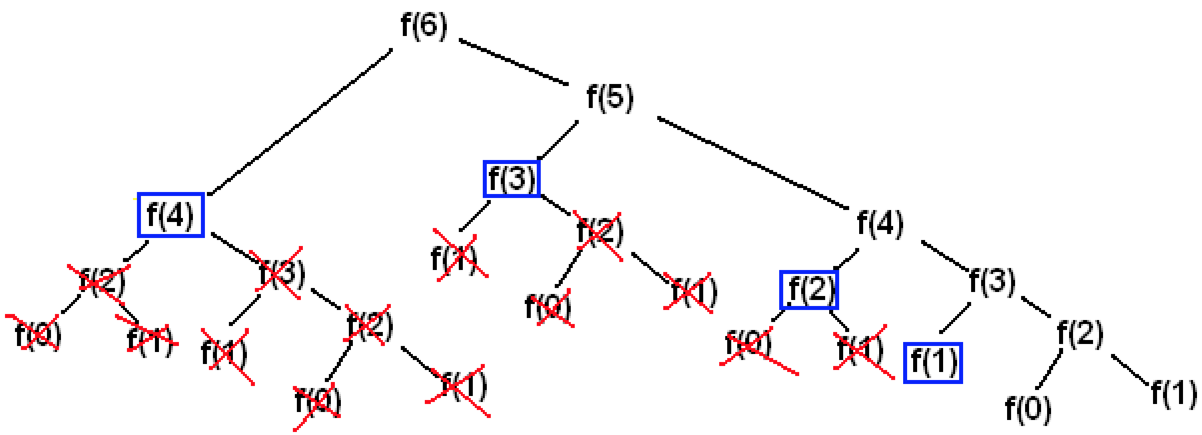
i conigli di Fibonacci

$$F_n = \begin{cases} 1 \text{ se } n = 1 \\ 1 \text{ se } n = 2 \\ F_{n-1} + F_{n-2} \text{ se } n \geq 3 \end{cases}$$



Fibonacci ricorsione


```
def fibonacci(n: int) -> int:
    '''
    n-esimo termine della successione
    '''
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```



Leonardo Pisano detto il Fibonacci
(Pisa, 1170 circa – Pisa, 1242 circa)

memoization

- ***memoization*** *is a term introduced by Donald Michie in 1968, which comes from the latin word memorandum* ***(to be remembered)***
- *memoization is a method used in computer science to* ***speed up calculations*** *by storing (remembering)* ***past calculations***
- *if repeated function calls are made with the same parameters, we can* ***store*** *the previous values instead of repeating unnecessary calculations*

fibonacci memoizzazione

- ***memoizzazione*** (*mettere in memoria*)
  - è una tecnica caratteristica della programmazione dinamica
  - nell'esempio ***memorizziamo in una lista*** i valori della successione che di volta in volta vengono ***calcolati***

```
_termini_calcolati = [0, 1] # termini noti per fib(0)* e fib(1)

def fibonacci(n: int) -> int:
    ''' ricorsione con memoizzazione mediante lista globale '''
    if n < len(_termini_calcolati): # già calcolato
        return _termini_calcolati[n]
    val = fibonacci(n-1) + fibonacci(n-2)
    _termini_calcolati.append(val)
    return val
```

decorators in Python

- in Python, functions are the ***first class objects***
  - ***functions are objects***
  - can be referenced to
  - passed to a variable
  - returned from other functions
- functions can be defined inside another function and can be passed as argument to another function
- ***decorators*** allows programmers to modify the behavior of function or class
- decorators can wrap another function in order to extend the behavior of wrapped function, without permanently modifying it
- in decorators, functions are taken as the ***argument*** into another function and then called inside the wrapper function

functools module

- ***lru_cache*** (least recently used cache) is a decorator in ***functools*** module

```
@functools.lru_cache()
def fibonacci(n: int) -> int:
    ''' decoratore per memoizzazione '''
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Fibonacci algoritmo iterativo


```
def fibonacci(n: int) -> int:
    ''' iterazione '''
    val = 1
    prec = 0
    for i in range(n-1):
        prec, val = val, val + prec
    return val
```

Fibonacci formula di Binet

$$x_n = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}$$

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1{,}6180339887$$

Jacques Philippe Marie Binet (1786 –1856)
matematico e astronomo francese

```python
def fibonacci (n: int) -> int:
    '''
       n-esimo termine calcolato con
       formula di Binet
    '''
    r5 = math.sqrt(5)
    fi  = (1+r5)/2
    fis = (1-r5)/2
    return int(round((1/r5)*(pow(fi,n)-pow(fis,n))))
```

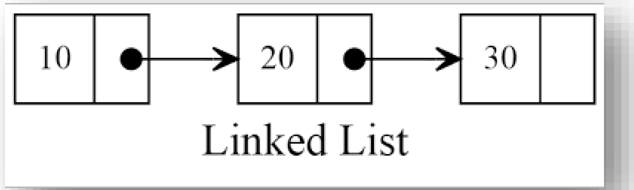
valori sperimentali

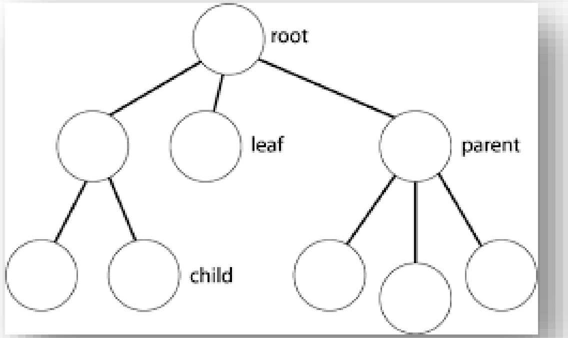

```
# senza memoizzazione: fib(40) = 102334155 time:  67.94661130
# memoizzazione lista: fib(40) = 102334155 time:  0.00009390
# iterazione         : fib(40) = 102334155 time:  0.00001080
# decorator          : fib(40) = 102334155 time:  0.00003360
# formula Binet      : fib(40) = 102334155 time:  0.00004190
```

https://github.com/albertoferrari/info_lab/blob/master/codice_lezioni/test_fibonacci.py

tipo di dato ricorsivo

- in un ***tipo di dato ricorsivo*** un valore può contenere valori dello stesso tipo
- ***lista*** collegata (linked list)
  - vuota, oppure...
    - nodo di testa, seguito da una lista collegata
- ***albero***
  - vuoto, oppure...
  - nodo di testa, seguito da più alberi





ricorsione

esercizi

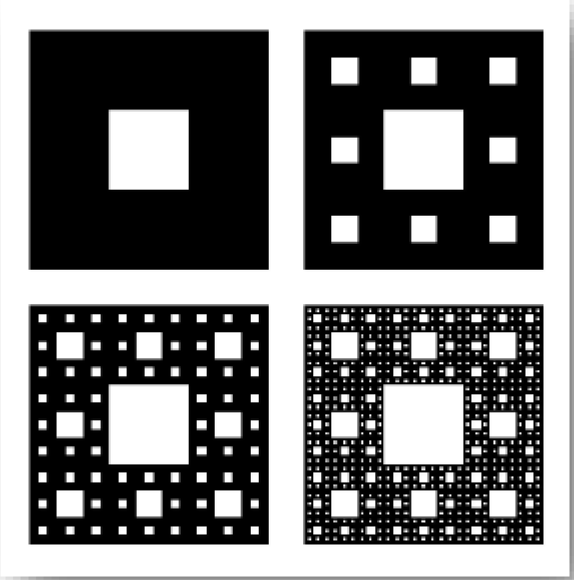
esercizi(1)

- ***8.1 palindromo***
  - *palindromo: testo che rimane uguale se letto al contrario*
  - scrivere una funzione ricorsiva per riconoscere i palindromi
    - parametro: testo da controllare
    - risultato: bool

stringa palindroma: se ha lunghezza 0 o 1, oppure...
prima lettera == ultima lettera e...
stringa rimanente (senza prima e ultima lettera) palindroma

[*https://github.com/albertoferrari/info_lab/blob/master/codice_lezioni/sl08_es01_stringa_palindroma.py*](https://github.com/albertoferrari/info_lab/blob/master/codice_lezioni/sl08_es01_stringa_palindroma.py)

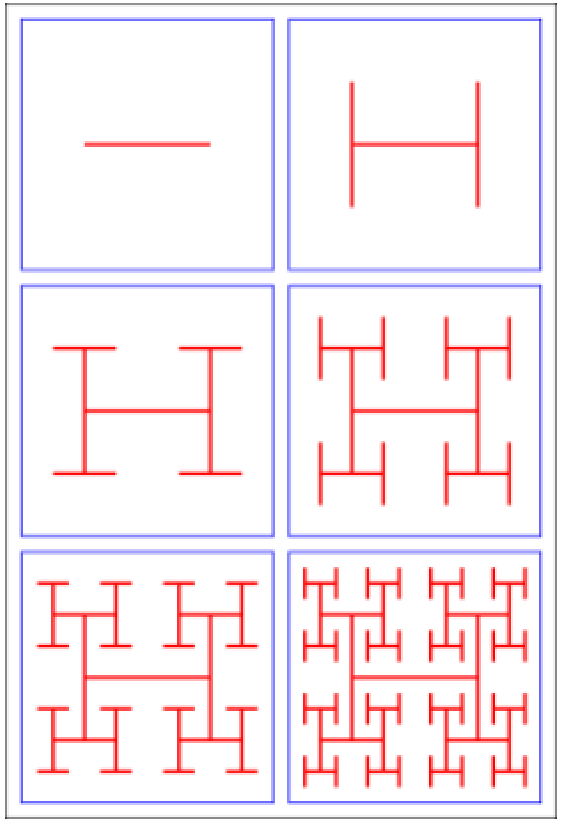
- ***8.2 Sierpinski carpet***
  - disegnare un ***frattale di Sierpinski***, di ordine n (scelto dall'utente)
    - dato un quadrato, dividerlo in 9 parti uguali
    - colorare la parte centrale
    - piapplicare l'algoritmo alle restanti 8 parti



https://github.com/albertoferrari/info_lab/blob/master/codice_lezioni/sl08_es02_sierpinski.py

esercizi(3)

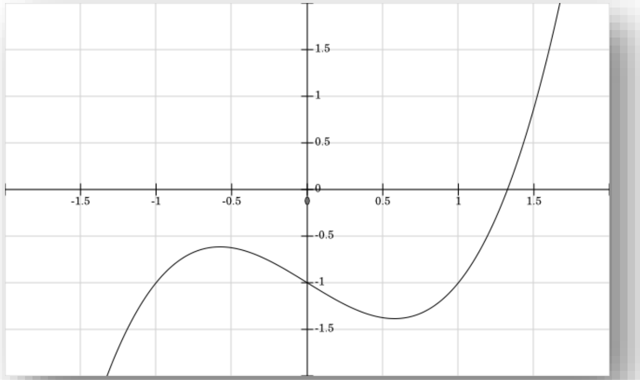
- ***8.3 Albero di H***
  - disegnare ricorsivamente un H-Tree
    - dividere l'area iniziale in due parti uguali
    - connettere con una linea i centri delle due aree
    - ripetere il procedimento per ciascuna delle due aree
    - alternare però la divisione delle aree in orizzontale e verticale
    - chiedere all'utente il livello di ricorsione desiderato



https://github.com/albertoferrari/info_lab/blob/master/codice_lezioni/sl08_es03_h.py

esercizi(4)

- ***8.4 bisezione, ricorsione***
  - trovare lo zero della seguente funzione matematica
    - $f(x) = x^3 - x - 1$, per $1 \leq x \leq 2$
    - trovare x t.c. $|f(x)| < 0.001$
  - definire una funzione ricorsiva di bisezione
    - parametri necessari: inizio intervallo di ricerca, fine intervallo di ricerca
    - invocare ad ogni livello la funzione su un intervallo dimezzato



https://github.com/albertoferrari/info_lab/blob/master/codice_lezioni/sl08_es04_bisezione.py

esercizi(6)

- ***8.5 anagrammi***
  - generare tutti gli anagrammi (***permutazioni***) di una stringa
  - risultato, una lista di stringhe
  - algoritmo:
    - stringa vuota: solo se stessa
    - altrimenti: per ogni carattere...
      - concatenarlo con tutte le permutazioni dei rimanenti caratteri (***ricorsione***)

[*https://github.com/albertoferrari/info_lab/blob/master/codice_lezioni/sl08_es05_anagrammi.py*](https://github.com/albertoferrari/info_lab/blob/master/codice_lezioni/sl08_es05_anagrammi.py)

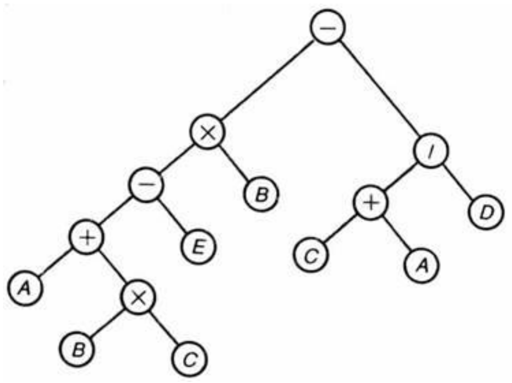
esercizi(7)

- ***8.6 torre di Hanoi***
  - tre paletti + N dischi di diametro decrescente
  - obiettivo ⇒ portare tutti i dischi dal primo all'ultimo paletto
  - si può spostare solo un disco alla volta
  - son si può mettere un disco su uno più piccolo
  - usare la ricorsione

Immediato spostare un solo disco.
N dischi: spostarne N-1 sul piolo né origine né dest.,
spostare l'ultimo disco sul piolo giusto,
spostare ancora gli altri N-1 dischi.

https://github.com/albertoferrari/info_lab/blob/master/codice_lezioni/sl08_es06_hanoi.py

- *8.7 notazione polacca*
  - leggere una riga di testo in una stringa
  - scrivere una funzione che valuti la stringa come una espressione, nella forma: "+ 2 7" (=9)
  - gli operandi possono essere a loro volta espressioni: "+ * 3 4 15" (=27)
  - scrivere una seconda funzione che trasformi l'espressione nell'abituale notazione infissa:
  - "((3 * 4) + 15)"
  - usare la ricorsione

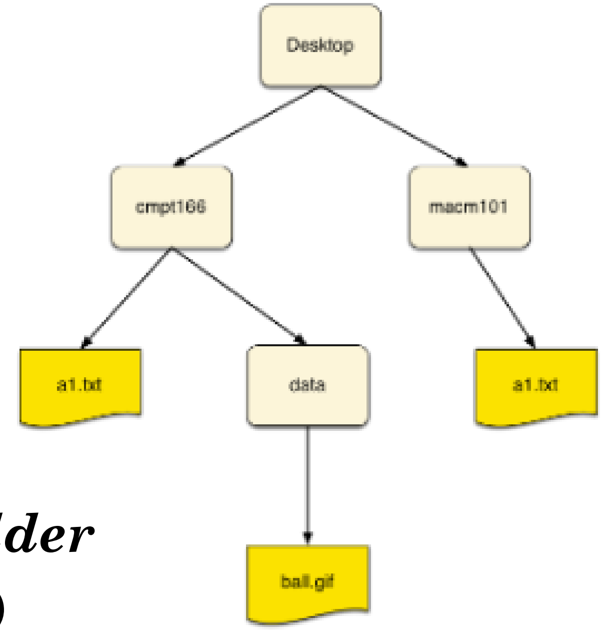


Supporre che i "token" siano tutti separati da spazio e che gli operatori abbiano tutti cardinalità fissa

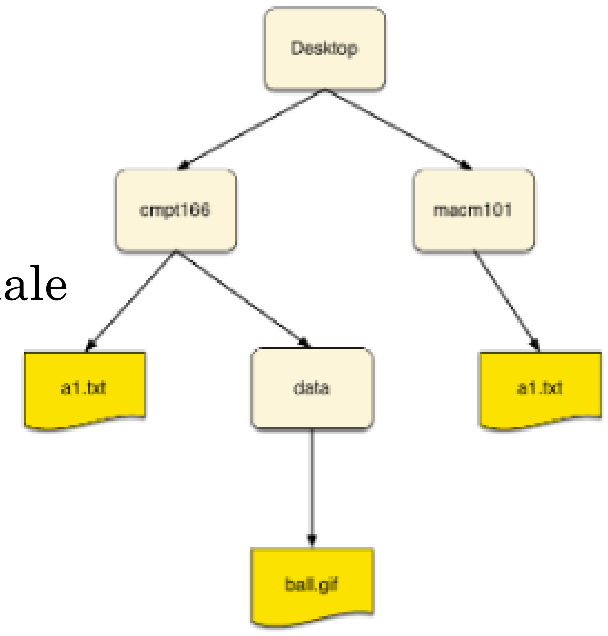
http://www.ce.unipr.it/brython/?p4_fun_polish.py

esercizi(9)

- ***8.9 documenti e cartelle***
  - un sistema *gerarchico* di gestione documenti è composto di due tipi di ***nodi*** (*classe base*)
    - i ***documenti***, caratterizzati da un nome e da un contenuto testuale (*classe derivata*)
    - le ***cartelle***, caratterizzate da un nome e da una lista di nodi contenuti (*classe derivata*)
  - creare una gerarchia delle tre classi: ***Node, Document, Folder***
  - le cartelle dovrebbero avere un metodo ***add_node(n: Node)***
  - nel corpo principale del programma, istanziare ed organizzare vari nodi
    - (senza input dell'utente)
  - ricreare con gli oggetti la struttura raffigurata a fianco



- ***8.10 dimensione delle cartelle***
  - aggiungere a tutti i nodi (es. precedente) un metodo ***size***
    - astratto nella classe base
    - per un documento, restituisce la lunghezza del contenuto testuale
    - per una cartella, restituisce la somma delle dimensioni dei nodi contenuti
  - calcolare la dimensione della struttura a fianco, inventando dei contenuti per i documenti presenti
  - aggiungere a tutti i nodi un metodo ***print(indent: int)*** per mostrare a terminale la struttura ad albero
    - astratto nella classe base
    - mostra il nome di documenti e cartelle
    - indenta opportunamente i nodi, rispetto alla cartella che li contiene



http://www.ce.unipr.it/brython/?p4_tree_nodes.py

- ***8.11 espressioni***
  - definire una gerarchia di classi per rappresentare espressioni matematiche
    - la classe base ***Expression*** ha un metodo astratto eval
      - senza parametri, restituisce il valore float dell'espressione
    - le sottoclassi concrete di una espressione sono:
      - ***Literal***, contenente un valore costante float
      - ***Sum***, contenente due operandi, entrambi espressioni
      - ***Product***, contenente due operandi, entrambi espr.
  - istanziare (senza fare parsing!) oggetti per rappresentare questa espressione:
    - 5 * (4 + 3 * 2)
  - calcolare il valore finale, chiamando eval sul nodo radice

http://www.ce.unipr.it/brython/?p4_tree_expression.py

- *8.12 espressioni prefisse*
  - aggiungere un metodo ***prefix*** a ***Expression*** (es. precedente)
  - genera una stringa in notazione prefissa (operatore seguito da operandi)

```
                                                  #     *   (prod2)
prod1 = Product(Literal(3), Literal(2))           #   / \
sum1 = Sum(Literal(4), prod1)                     # 5    +   (sum1)
prod2 = Product(Literal(5), sum1)                 #      / \
print(prod2.eval())                               #     4    *   (prod1)
print(prod2.prefix())                             #         / \
                                                  #        3   2
```

https://it.wikipedia.org/wiki/Notazione_polacca
http://www.ce.unipr.it/brython/?p4_tree_expression.py