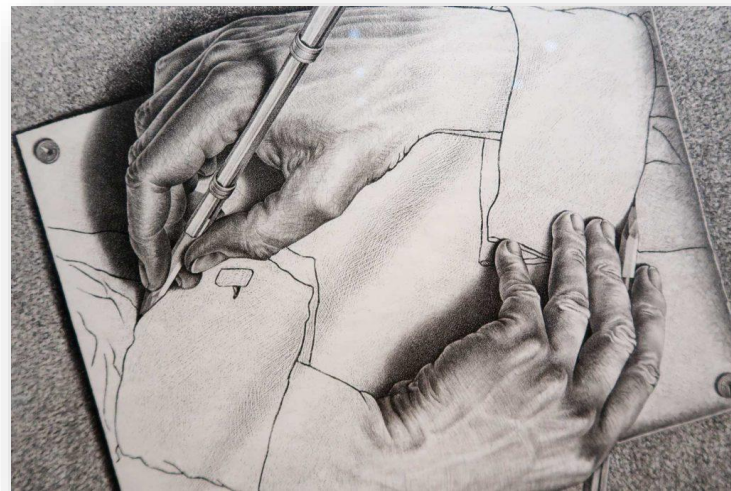




UNIVERSITÀ
DI PARMA

ricorsione

informatica e laboratorio di programmazione



- 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)



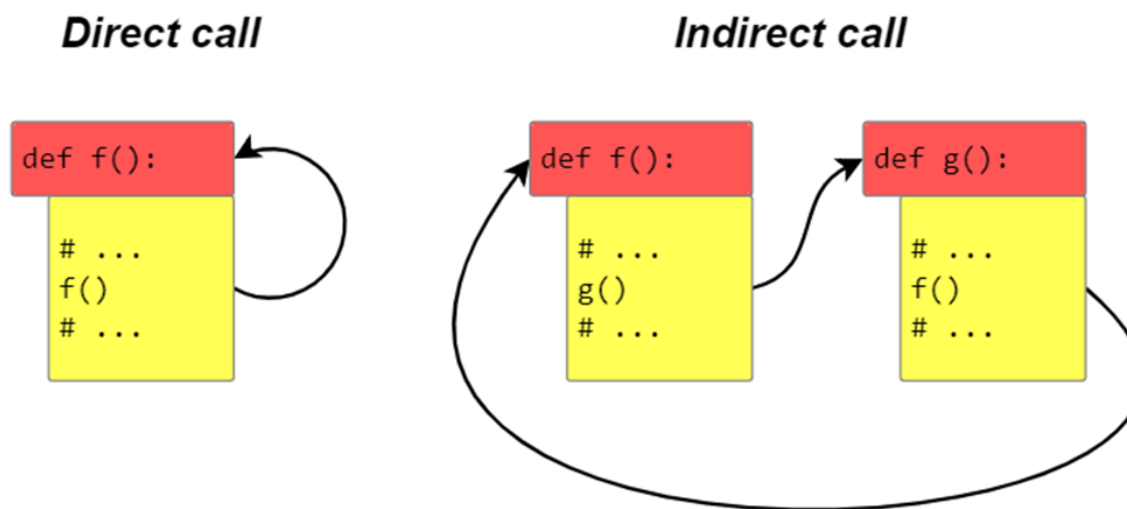
- definizione dei *numeri naturali*:
 - 1) 1 è un numero naturale
 - 2) il successore di un numero naturale è un numero naturale
- definizione di *fattoriale* di un numero intero positivo:
 - 1) $0! = 1$
 - 2) $n! = n * (n-1)!$
- calcolo del **MCD** tra due numeri A e B ($A > B$)
algoritmo di Euclide
 - 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))
```


- 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

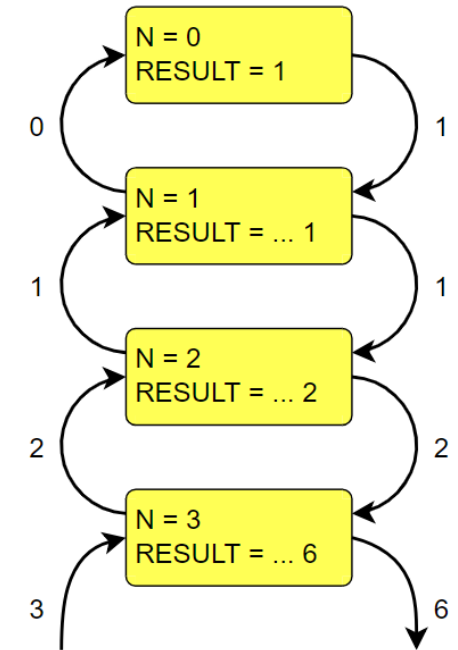
- un sottoprogramma ricorsivo è una procedura (o **funzione**) all'interno della quale è presente una **chiamata a se stessa** o ad altro sottoprogramma che la richiama
- la ricorsione è **diretta** se la chiamata è interna al sottoprogramma altrimenti si dice indiretta
- molti linguaggi consentono ad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



- 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
```

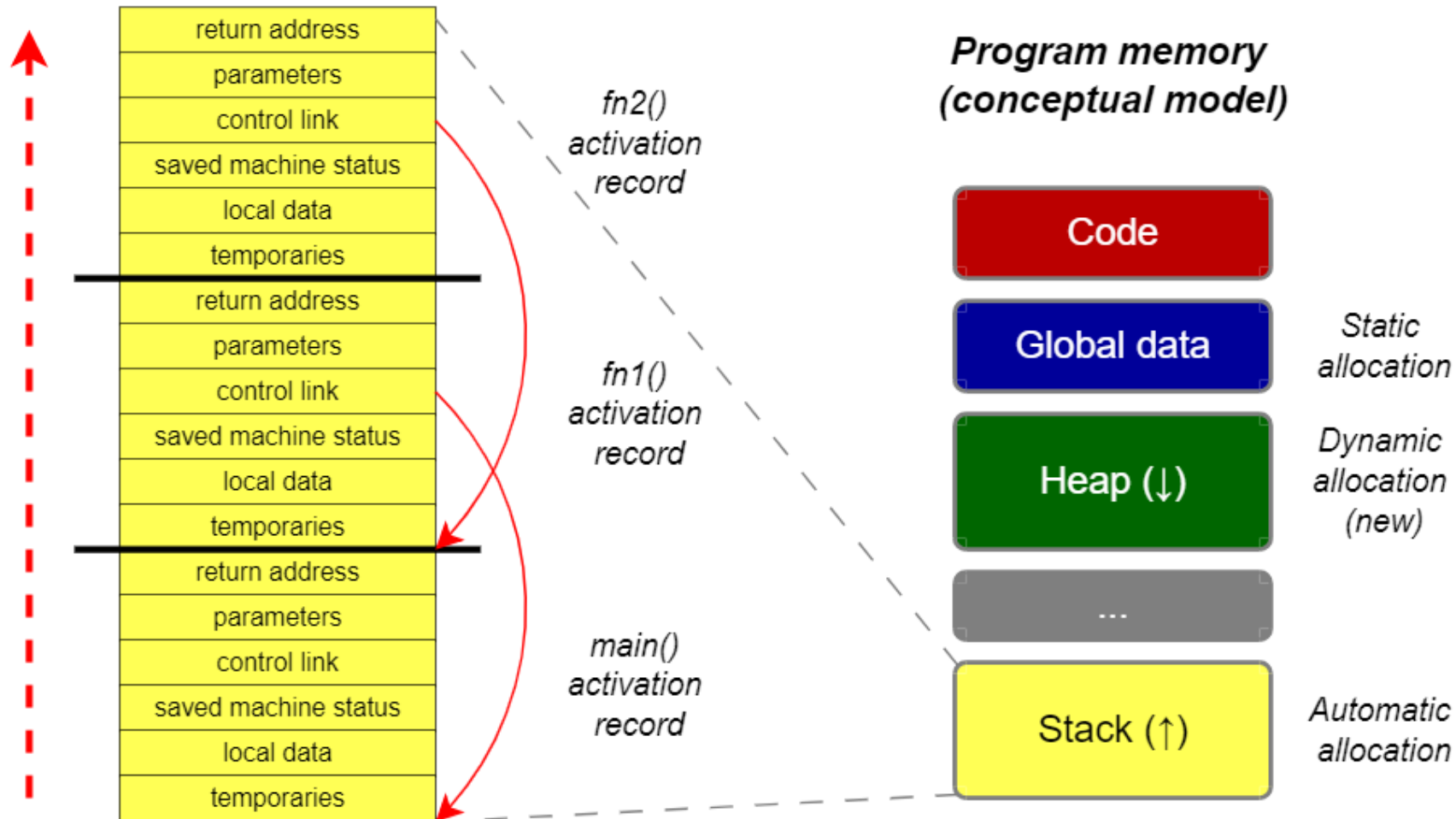
*Ai primordi (Fortran 66 ecc.) solo allocazione statica
Spazio fisso ed unico per dati locali ad una funzione → no ricorsione*



- 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

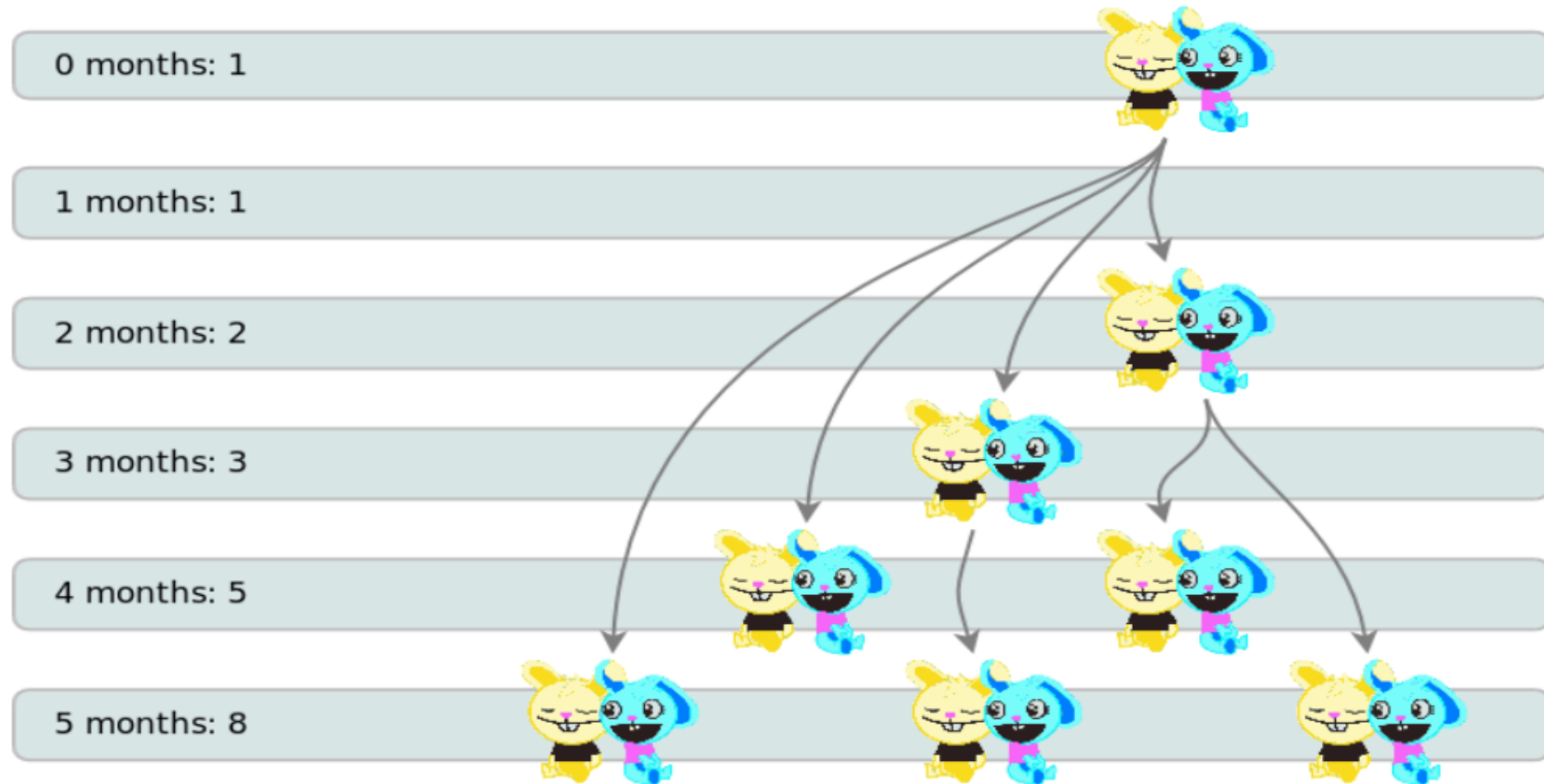
- **pila**: memoria dinamica LIFO (Last In First Out)
 - dimensione massima prefissata
- il programma ci memorizza automaticamente:
 - **indirizzo** di ritorno per la funzione
 - inserito alla chiamata, estratto all'uscita
 - **parametri** della funzione
 - inseriti alla chiamata, eliminati all'uscita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



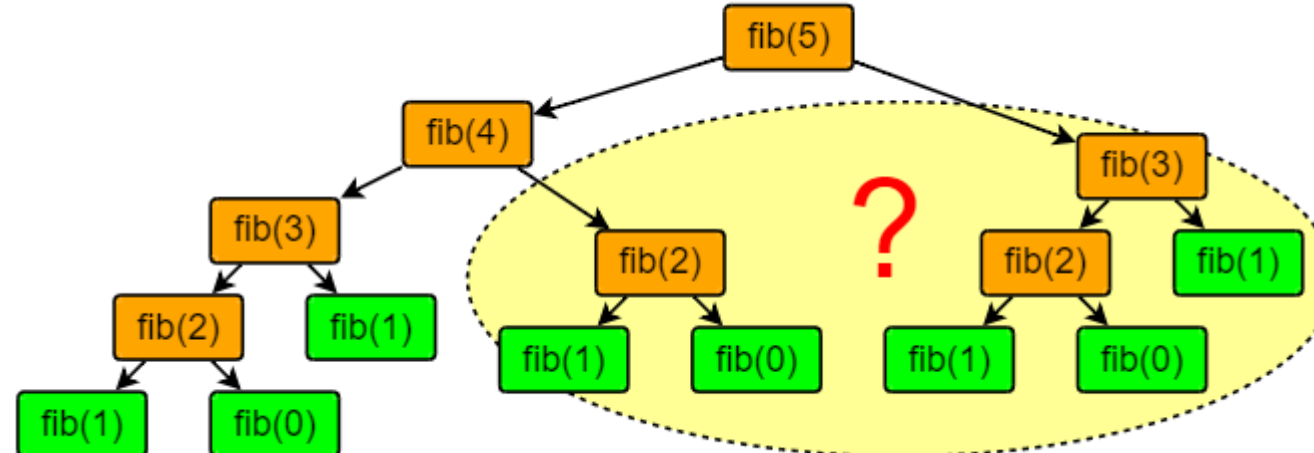


- **visibilità** \Rightarrow insieme di istruzioni da cui è **accessibile**
- **ciclo di vita** \Rightarrow **esistenza** in memoria della variabile (etichetta)
- i valori (oggetti) in Python sono tutti gestiti dinamicamente
- visibilità **globale**
 - variabili fuori da ogni funzione - meglio **evitare!**
 - 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!

$fib(0) = fib(1) = 1; fib(n) = fib(n-1) + fib(n-2);$



```
def fibonacci(n: int) -> int:  
    result = 1  
    if n > 1:  
        result = fibonacci(n-1) + fibonacci(n-2)  
    return result
```



- **memoizzazione** (*mettere in memoria*)
 - tecnica di programmazione che consiste nel salvare in memoria i valori restituiti da una funzione
 - in un riutilizzo successivo non è più necessario doverli ricalcolare.
 - è una tecnica caratteristica della programmazione dinamica

```
_fibonacci_lookup = [1, 1] # termini già calcolati

def fibonacci(n: int) -> int:
    if n < len(_fibonacci_lookup):
        return _fibonacci_lookup[n] # no ricorsione
    result = fibonacci(n - 1) + fibonacci(n - 2)
    _fibonacci_lookup.append(result) # memoizzazione
    return result
```



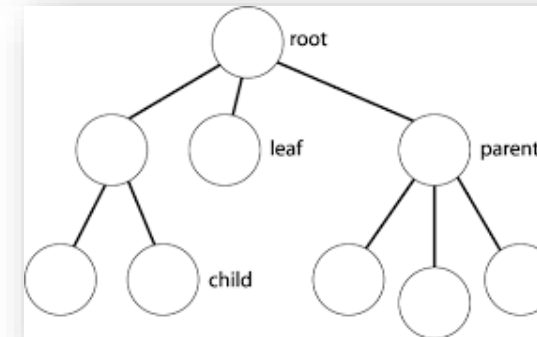
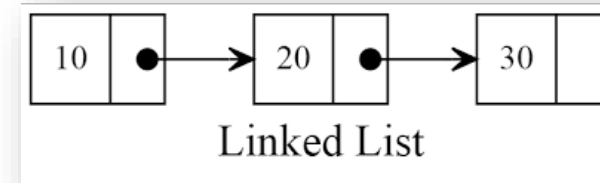
```
from functools import lru_cache
from time import sleep

@lru_cache()                                # function decoration
def fibonacci(n: int) -> int:
    result = 1
    if n > 1:
        result = fibonacci(n-1) + fibonacci(n-2)
    return result

@lru_cache()
def somma(a,b):
    sleep(3)
    return a + b
```

```
def fibonacci(n: int) -> int:  
    value = 1  
    previous = 0  
  
    for i in range(n):  
        value, previous = value + previous, value  
  
    return value
```

- 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



○ 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

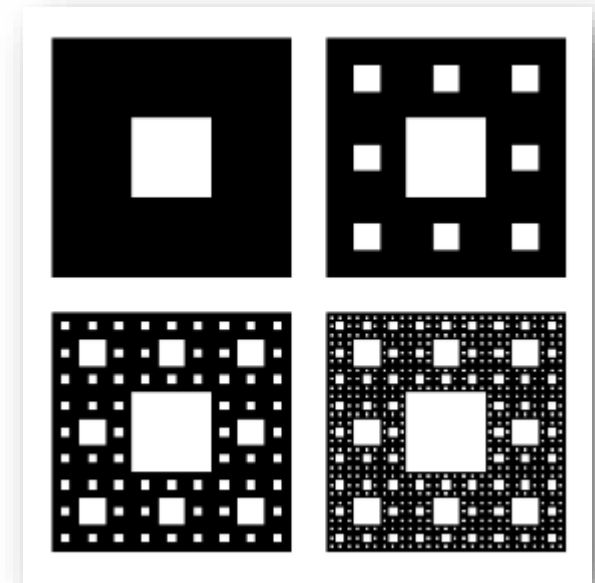


http://www.ce.unipr.it/brython/?p4_fun_palindrome.py

○ 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
 - riapplicare l'algoritmo alle restanti 8 parti

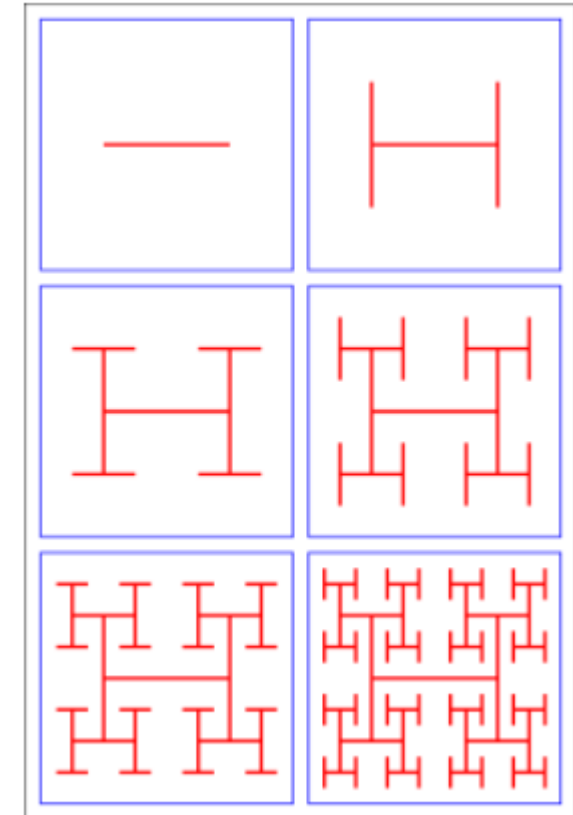
http://www.ce.unipr.it/brython/?p4_fun_sierpinski.py



○ 8.3 *Albero di H*

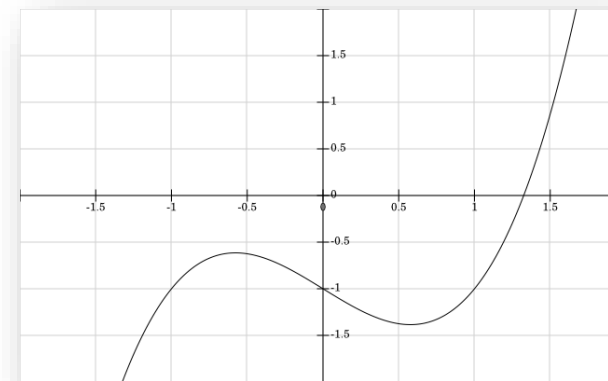
- disegnare ricorsivamente un H-Tree
 - dividere l'area iniziale in due parti uguali
 - connettere con una linea i centri delle due aree
 - ripetere il procedimento per ciascuna delle due aree
 - alternare però la divisione delle aree in orizzontale e verticale
 - chiedere all'utente il livello di ricorsione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

http://www.ce.unipr.it/brython/?p4_fun_htree.py



○ 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



http://www.ce.unipr.it/brython/?p4_fun_bisection.py

○ 8.5 combinazioni di simboli

- scrivere una funzione per generare tutte le combinazioni di lunghezza 3
 - parametri: str contenente l'insieme dei simboli validi (presenti su ogni ruota)
 - risultato: lista di str (combinazioni)
- possibile approccio
 - usare 3 cicli for annidati
 - ogni for scorre tutto l'insieme di simboli
 - una combinazione è una str, ottenuta concatenando i 3 simboli correnti



*Per test, fornire in input le vocali "AEIOU", per ottenere:
["AAA", "AAE", "AAI", "AAO", "AAU", "AEA", "AEE", "AEI", "AEO", "AEU", "AIA", ...*

○ 8.6 *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*)



http://www.ce.unipr.it/brython/?p4_fun_anagram.py

○ *8.7 torre di Hanoi*

- tre paletti + N dischi di diametro decrescente
- obiettivo \Rightarrow portare tutti i dischi dal primo all'ultimo paletto
- si può spostare solo un disco alla volta
- non si può mettere un disco su uno più piccolo
- usare la ricorsione

Immediato spostare un solo disco.

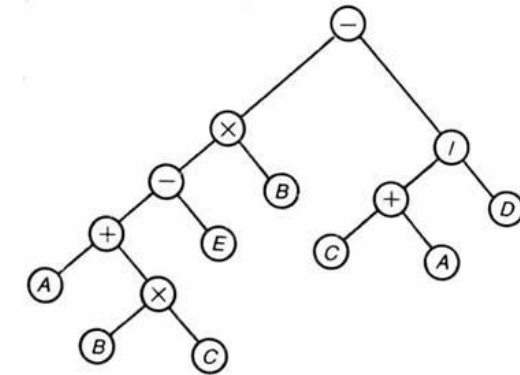
*N dischi: spostarne N-1 sul piolo né origine né dest.,
spostare l'ultimo disco sul piolo giusto,
spostare ancora gli altri N-1 dischi.*

http://www.ce.unipr.it/brython/?p4_fun_hanoi.py



○ 8.8 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

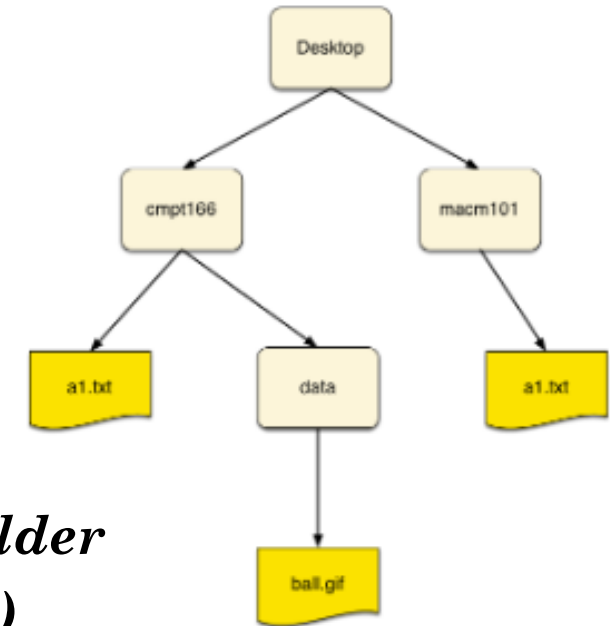


Supporre che i "token" siano tutti separati da spazio e che gli operatori abbiano tutti cardinalità fissa

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

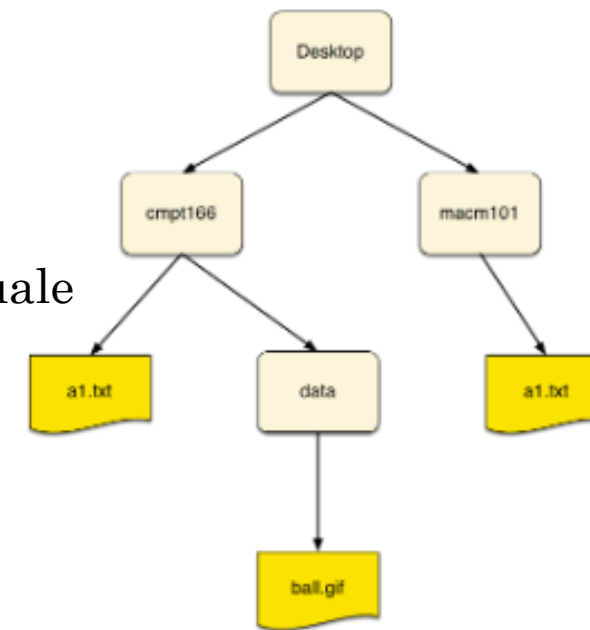
○ 8.9 *documenti e cartelle*

- 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)

```
prod1 = Product(Literal(3), Literal(2))
sum1 = Sum(Literal(4), prod1)
prod2 = Product(Literal(5), sum1)
print(prod2.eval())
print(prod2.prefix())
```

* (prod2)
/ \
5 + (sum1)
/ \
4 * (prod1)
/ \
3 2

https://it.wikipedia.org/wiki/Notazione_polacca

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