Progettazione di Programmi su Vettori

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Lezione **5(b)** **[10/10/23]**

SAPIENZA
UNIVERSITÀ DI ROMA

Vettori e ricorsione: stampa (1)

Il **costrutto di controllo più naturale**, quando occorre scandire sequenzialmente un **vettore** è il **ciclo `for`**.

È però possibile usare la **ricorsione**: una possibilità è mimare il **`for`**, passando tra i parametri l'indice `i`.

Stampiamo **prima l'elemento** all'indice *i*, **poi i successivi**.

➡Cosa accade se **invertiamo stampa** e **chiamate ricorsive**?

Prima stampa i successivi con la chiamata ricorsiva e poi l'elemento *v*[*i*]: ripetendo il ragionamento, **stamperei il vettore rovesciato!**

```
def stampaV(v)
    stampaVAux(v, 0, len(v))

def stampaVaux(v, i, n):
# ENS: stampa il vettore v[i,n)
    if i < n: #caso base
        print v[i]
        stampaVaux(v, i+1, n)
```

decresce
n - i

Vettori e ricorsione: stampa (2)

Una **seconda possibilità** è usare il valore n: è come se la funzione ricorsiva vedesse la porzione di vettore $v[0, n]$ (con n decrescente).

Siccome stampo l'ultimo, stavolta devo **prima stampare i precedenti**. Ma invertendo l'ordine di stampa e chiamata…

Morale: una scansione ricorsiva **percorre un vettore 2 volte**, una all'andata e una al ritorno dalle chiamate ricorsive.

È sempre **meglio pensare alle equazioni ricorsive** in astratto… ma occasionalmente è utile capire come effettivamente viene eseguita una chiamata ricorsiva.

```
def stampaV(v)
    stampaVAux(v, len(v)-1)

def stampaVaux(v, n):
# stampa il vettore v[0,n]
    if n > 0: #caso base
        stampaVaux(v, n-1) #stampa i precedenti
    print v[n]
```

Piccoli arsenali crescono

Nel **progetto di algoritmi** è bene pensare le soluzioni in termini di azioni astratte (metodologia **top-down**).

Nella **pratica della programmazione** è anche bene costruirsi piccole librerie di utilità da usare per evitare compiti noiosi (metodologia **bottom-up**).

Facciamo alcuni esempi nel mondo dei vettori.

```
def allocInit(x, n):
    v = alloca(n)
    for i=0 to n-1:
        v[i]==x
    return v
```

```
def allZero(n):
    return allocInit(0, n)
```

allocano e inizializzano un vettore

```
def inizializza(v, x):
    #REQ v allocato
    inizializza(v, x, 0, len(v))
```

```
def inizializza(v, x, inf, sup):
    #REQ v[inf, sup) allocato
    for i=inf to sup-1:
        v[i]==x
```

modificano un vettore parametro

```
def azzera(v):
    inizializza(v, 0)
```

Spazio e Tempo: memoization

Ricordiamo la versione inefficiente ricorsiva della funzione di Fibonacci: il suo problema è il **ri-calcolo degli stessi valori** e quindi la replicazione di interi sotto-alberi delle sue chiamate ricorsive.

Al prezzo di allocare un **vettore abbastanza grande**, è sempre possibile **evitare di ricalcolare** inutilmente **valori già calcolati**, memorizzandoli in un vettore.

Il controllo del **caso base** è sostituito dal **controllo** di aver **già calcolato** lo stesso valore ($f[n] \geq 0$).

Le **chiamate ricorsive** si **trasmettono** un riferimento al vettore **f**.

f viene **allocato** e inizializzato in ***allocInit***.

```
def fib(n):
    if n<2: return n
    return fib(n-1) + fib(n-2)
```

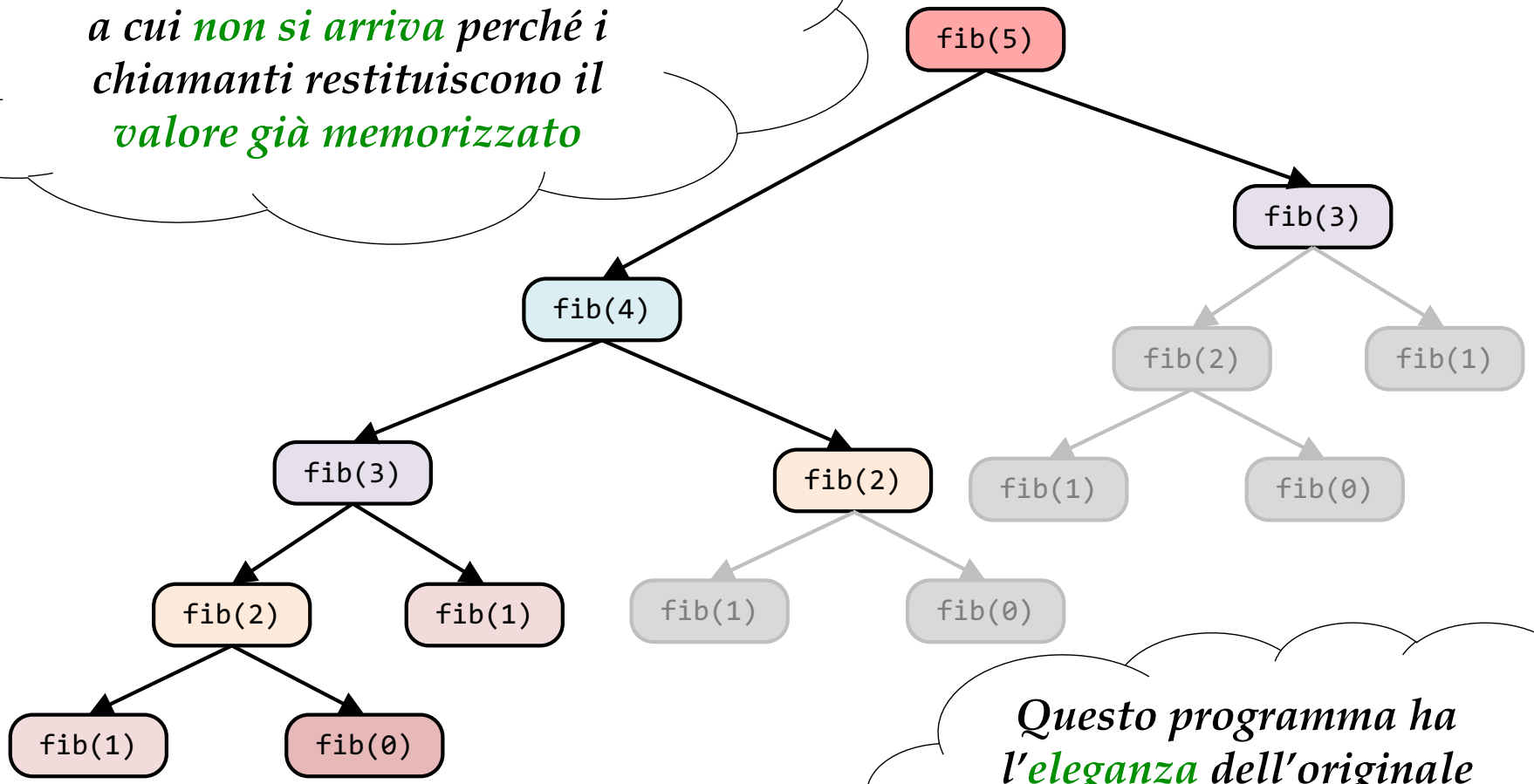
$\theta(n)$ tempo
$\theta(n)$ spazio

```
def fib(n):
    f = allocInit(n)
    f[0] = 0
    f[1] = 1
    return fibAux(n, f)

def fibAux(n, f):
    if f[n] ≥ 0: return f[n]
    f[n] = fibAux(n-1, f) +
                  fibAux(n-2, f)
    return f[n]
```

Fibonacci ricorsivo con memoization

In grigio le chiamate ricorsive a cui non si arriva perché i chiamanti restituiscono il valore già memorizzato



Questo programma ha l'eleganza dell'originale e complessità lineare. Spreca memoria

Problema con due vettori: bifronte

Problema: Dati due vettori u e v, determinare se u è un **bifronte** di v, cioè v è uguale a u rovesciato.

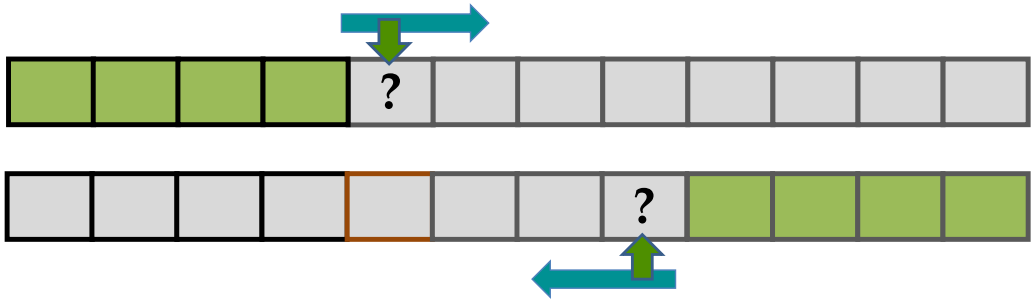
Soluzione: assumiamo che abbiamo la **stessa lunghezza** e chiamiamo v^R il vettore rovesciato, definito da $v^R[i] = v[n - i - 1]$.
Il problema è **verificare** se $u = v^R$. È un problema di **verifica**.

I **doni delle buone notazioni**: la formula **contiene già l'algoritmo**!

Usando i come indice, l'idea è mantenere la proprietà **invariante**:

$$\varphi(v, i, n) \equiv u[0, i) = v^R[0, i) = (v[n - i, n))^R$$

Se $u[i] = v[n - i - 1]$ allora **stabiliamo** $\varphi(v, i+1, n)$, altrimenti scopriamo un indice per cui $u[i] \neq v[n - i - 1]$ e quindi $u \neq v^R$.



1511. **Bifronte**

Vive in campagna
Nonna Xxxxxx nella sua cucina
il forno x xxxxx accende ogni mattina.
Prepara pane e buoni biscottini
per il ritorno dei suoi nipotini.

(A. Cozzolino)

Problema con due vettori: bifronte

Inizializzazione: È abbastanza ovvio che $\varphi(v, 0, n)$ è banalmente verificata perché **i due segmenti di vettore sono vuoti**.

Terminazione: Dovrebbe essere chiaro anche che $\varphi(v, n, n)$ implica proprio $u = v^R$. Quindi usciamo per $i = n$ e $t(n, i) = n - i$ è la "solita" **funzione di terminazione positiva** e strettamente **decrescente**.

Se $m \neq n$, allora u e v **non possono** essere **bifronti**, e quindi a poco prezzo possiamo **rilassare le precondizioni**.

```
def bifronte(u, v):
    m, n = len(u), len(v) #RE  m==n
    if (m!=n) return False;
    for i=0 to n-1:
    # INV: u[0,i) == v^R[n-i,n)
        if u[i]!=v[m-i-1]:
            return False
    return True
```

5911. **Bifronte con aggiunta d'estremi**

Meno male che era a dieta!

Per tutto il pranzo fu assai misurato,
ma non xxxxx alla vista del xxxxxxx:
ben tre fette di torta divorò.

(Nicole)

91148. **Bifronte**

A Yellowstone

Solitario nel parco cavalcava
il xxxxxx, contemplando la Natura.
Di xxxxxx gli sembrava su quel mondo
d'aquile, d'orsi e d'alci, ove un profondo
sentimento di pace dominava.

(D. Bottazzi)

I piaceri della composizionalità

Consideriamo un altro problema enigmistico: l'**antipodo**. Sono antipodi ad esempio *c-annolo* e *c-olonna* oppure *c-ane* e *c-ena* cioè due parole che hanno la stessa lettera iniziale, mentre le due "code" sono tra loro bifronti.

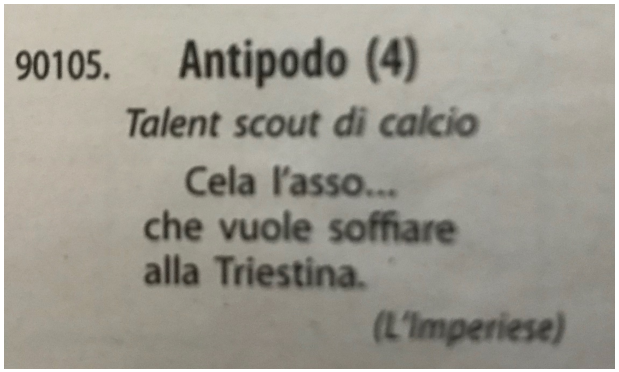
Se avessimo scritto la funzione precedente nella forma (⏵ **Esercizio**):

`bifronte(u, v, inf, sup)`

ecco il codice per verificare se due vettori sono antipodi.

Slogan: **Parametrizzare per comporre**!

```
def antipodo(u, v):
    if u[0]==v[0]:
        return bifronte(u, v, 1, len(u))
    return False
```


90105. **Antipodo (4)**

Talent scout di calcio

Cela l'asso...
che vuole soffiare
alla Triestina.

(L'Imperiese)

I rischi della composizionalità

Ripercorrendo il processo logico che ci ha portato alla funzione ***`bifronte`***, sembrerebbe appetibile il seguente programma.

`rovescia` e **`uguali`** sembrano due naturali funzioni sui vettori pronte all'uso nella nostra dispensa di funzioni.

Va benissimo, ma osservate:

- **`rovescia`** **modifica il vettore** *u*, questo potrebbe essere **indesiderato**;
- Sia **`rovescia`** che **`uguali`** fanno una scansione del vettore: quindi ne **fate due**, al posto dell'unica scansione di ***`bifronte`*** precedente (**confrontate i casi ottimi e medi**, ad esempio).

È un **fenomeno generale**: comporre funzioni è facile, **utilissimo nella progettazione** di una soluzione, ma a volte, l'efficienza richiede di "aprire le scatole"e modificare cosa accade dentro.

```
def bifronte(u, v):
    rovescia(u)
    return uguali(u, v)
```

That's all Folks!

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Lezione **5(b)** **[10/10/23]**

SAPIENZA
UNIVERSITÀ DI ROMA