Pile, Code & ... Cappanacci

corso di laurea in **Matematica**

Informatica Generale, Esercitazione **6**

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Esercitazione 6, Lezione 16

1. [**Scritto 21/6/22**] Si considerino i ***numeri di Cappanacci*** def. da:

$$C_k(n) = \begin{cases} 0 & se\ n < 1 \\ 1 & se\ n = 1 \\ \sum_{j=n-k}^{n-1} C_k(j) & altrimenti \end{cases}$$

Cioè, nella successione C_k, ogni elemento dopo il k-esimo è la somma dei k precedenti. Osserviamo che C_2 è Fibonacci, C_3 è nota come Tribonacci e $C_\infty(n) = 2^{n-1}$. Progettare un algoritmo che calcola $C_k(n)$.

2. Implementare il tipo di dato Coda usando un automa a pila, cioè un esecutore che ha solo pile come memoria (servono due pile).

Fare il contrario (pila con due code). Valutare la complessità delle operazioni in funzione delle sequenza di operazioni richieste.

3. [**Scritto 21/6/22 – dist.**] Disponendo di un automa a pila/code, scrivere un algoritmo che verifica se un'espressione (fornita in una coda di caratteri) ha le parentesi bilanciate. Considerare dapprima il caso che ci sia un solo tipo di parentesi, poi quello generale.

Assumere di avere funzioni che riconoscono le parentesi.

Cappanacci: idee balzane

Un'idea **balzana** era applicare la **definizione induttiva alla lettera** e fare un **programma ricorsivo** sullo stile del calcolo inefficiente dei numeri di Fibonacci.

- **Esercizio**: calcolare la complessità di questo programma.

Si può, ovviamente, renderlo efficiente usando memoization.

```
def cappanacciRec(n, k):
    if n ≤ 0: return 0
    if n == 1: return 1
    ck = 0
    for i=n-k-1 to n-1:
        c = c + cappanacciRec(i)
    return c
```

sempre meglio rispettare il prototipo

```
def knacciRecAux(n, k, ck):
    if n ≤ 0: return 0
        # valore già calcolato
    if ck[n] ≥ 0: return ck[n]
    for i=n-k-1 to n-1:
        ck[n] += knacciRecAux(i)
    return ck[n]

def knacciRecMem(n, k):
    ck = allZero(n)
    ck[1] = 0
    return knacciRecAux(n, k, ck)
```

Cappanacci da 10

La soluzione canonica era **caricare gli ultimi *k* valori calcolati in una coda**. Ogni nuovo numero calcolato, si fa uscire quello calcolato *k* iterazioni fa, che non serve più.

Per calcolare quello nuovo, era bene osservare:

$c_n = c_{n-1} + c_{n-2} + \ldots + c_{n-k+1} + c_{n-k}$ e $c_{n+1} = c_n + c_{n-1} + \ldots + c_{n-k+1}$

da cui si ha che: $c_{n+1} = c_n + c_n - c_{n-k} = 2\,c_n - c_{n-k}$, evitando di risommare k elementi, ottenendo una complessità $\Theta(nk)$.

Per rendere **uniforme il codice**, era opportuno caricare all'inizio nella coda *k*-1 zeri e un 1.

```
def knacci10(n, k):
    Q = newQ()
    for i=1 to k-1: enqueue(Q, 0)
    enqueue(Q, 1)
    ck = 0
    for i=2 to n:
        old = dequeue(Q)
        ck = 2*ck - old
        enqueue(Q, ck)
    return ck
```

Cappanacci da 11

A ben vedere, la coda viene usata in un modo molto particolare: ci sono sempre k elementi nella coda e a ogni iterazione **ne tolgo uno** e lo **sostituisco con quello nuovo** calcolato…

Si può fare molto meglio **con un vettore**! Ovviamente, occorre un po' di cura in più con gli indici.

Morale: usare le strutture dati "preconfezionate" rende i **programmi più semplici** da scrivere. "Specializzare" il comportamento alle caratteristiche del problema in esame permette di **scrivere programmi più efficienti** (anche **a parità di costo asintotico**) e…

… a volte **più belli**!

```
def knacci11(n, k):
    ck = allZeroV(k)
    ck[1] = 1
    for i=2 to n:
        ck[i % k] = 2*c[(i-1) % k] - c[i % k]
    return c[n % k]
```

in nuovo valore rimpiazza quello tolto