Dietro la Ricorsione

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Lezione **19(a)** [**28/11/23**]

SAPIENZA
UNIVERSITÀ DI ROMA

Chiamata di Funzione

L'esecuzione di una funzione necessita di alcuni dati per essere correttamente eseguita:

- memoria per le **variabili locali**, che devono essere **distinte** da eventuali **variabili globali** e **variabili locali di altre chiamate** (anche della stessa funzione) con **lo stesso nome**.
- memoria per i **parametri** passati alla funzione.
- memoria per il **punto di ritorno**, cioè il punto dove ricominciare ad eseguire quando la funzione termina la sua esecuzione. Siccome posso chiamare una funzione da più punti, questo dipende da **dove è avvenuta** la **chiamata corrente**.

Tutte queste informazioni vengono memorizzate in un record, detto **record di attivazione** (**activation record**) che viene allocato al momento della chiamata di una funzione e memorizzato sulla **pila di attivazione** (stack).

In **cima alla pila** di attivazione, c'è sempre la **funzione in esecuzione**, e sotto ci sono i record di attivazione delle **chiamate** precedenti e **non ancora chiuse** (al fondo c'è sempre `main` in **C**)

Implementazione della ricorsione

Vedremo con un paio di esempi, come si possa **trasformare ogni programma ricorsivo in iterativo**, avvalendosi di una struttura dati pila per mantenere informazione sulle chiamate ricorsive.

Vediamo prima il programma della Torre di Hanoi (Lezione **4**) e poi il programma massimo fattore primo (Esercitazione **2,** Lezione **4**).

Vediamo prima (in forma astratta) una **traduzione generale** per il programma di Hanoi.

I compilatori traducono la ricorsione, **generando codice per gestire sulla pila di attivazione, le chiamate di funzione** e in particolare per le **chiamate ricorsive**.

Noi useremo pile per:

- **simulare la ricorsione** con un programma iterativo;
- **mantenere lo stato delle torri** nel gioco della torre di Hanoi.

Vedremo poi traduzioni "**istanziate sul problema**" e **non sono** esattamente **la simulazione generale di ogni programma ricorsivo**, ma non sono poi tanto diverse.

Hanoi: traduzione sistematica

Individuiamo i punti da cui la computazione di Hanoi può **cominciare** o **riprendere** per effetto di:

- **chiamata** (**iniziale** o **ricorsiva** da altre attivazioni di Hanoi)
- **rientro da una chiamata** ricorsiva (**mezzo** e **fine**).

Per codificare questa informazione a noi basta semplicemente un intero nell'intervallo [0,2]. Nella "realtà" saranno **indirizzi di memoria delle istruzioni** del programma che sta eseguendo.

```
def hanoi(k, da, a, app):
    # punto 0: inizio
    if k==1: muovi(da, a)
    else:
        hanoi(k-1, da, app, a)
        # punto 1: mezzo
        muovi(da, a)
        hanoi(k-1, app, a, da)
        # punto 2: fine
```

Hanoi: ambiente locale e sua rappr.

L'**ambiente locale** di una singola chiamata è rappresentabile con i valori dei parametri **k**, **da**, **a**, **app**. Ci serve infine una variabile per sapere dove ricominciare a eseguire, il cosiddetto **punto di ritorno**: useremo una variabile **ret**, che avrà come possibili valori 0, 1 e 2.

Tutta questa informazione viene memorizzata in un record, che chiamiamo **HAR** ("**H**anoi **A**ctivation **R**ecord") e che verrà messo sulla pila di sistema.

Per mantenere il codice compatto, scriviamo le funzioni: **creaHAR** e **estraiHAR** per allocare e leggere un siffatto record.

```
def creaHAR(k, da, a, app, ret):
    H = allocaHAR()
    H->k = k
    H->da = da
    H->a = a
    H->app = app
    H->ret = ret
    return H
```

```
def estraiHAR(H):
    return H->k, H->da, H->a, H->app, H->ret
```

Traduzione Sistematica Ricorsione

Vediamo come **viene veramente eseguita** la funzione ricorsiva che risolve il rompicapo della Torre di Hanoi…

Ogni chiamata di funzione genera l'allocazione di un AR sulla pila

```
def hanoiIter(n):
    CS = newStack(); pc = inizio # program counter
    push(CS, creaHAR(n, 0, 1, 2, end)) # carico prima chiamata
    while not isEmpty(CS): # finchè non esaurisco le chiamate
        k, da, a, ap, ret = estraiHAR(top(CS)) # leggo lo stato locale
        if pc == inizio: # a seconda di dove eseguo…
            if k == 1: muovi(da, a); pc = ret; pop(CS)
            else: H = creaHAR(k-1, da, app, a, mezzo) # preparo e…
                  pc = inizio; push(CS, H) # carico prima chiamata
        else if pc == mezzo:
            muovi(da, a)
            H = creaHAR(k-1, app, a, da, fine) # preparo e…
            pc = inizio; push(CS, H) # carico seconda chiamata
        # pc == fine
        else: pop(CS) # scarico chiamata ricorsiva
              pc = ret  # ricomincio a eseguire dal return point
```

quando si chiama, si rimette pc a inizio. Si mette pc a ret al ritorno

Hanoi iterativo "fatto in casa"

Ma noi **non siamo compilatori** e il nostro problema non è tradurre sistematicamente tutti i programmi ricorsivi… forse possiamo trovare **soluzioni più semplici ritagliate** sul **problema** in esame…

Possiamo semplicemente usare una **pila** di "**mosse da eseguire**": alcune mosse sono **semplici** (quando $k = 1$) alcune sono **complesse** (quando $k > 1$) e necessitano per essere eseguite di essere rimosse dalla pila e **sostituite con mosse più semplici…**

```
def hanoiIterHomeMade(n):
    MS = newStack() # stack di mosse
    push(CS, creaMossa(n, 0, 1, 2))
    while not isEmpty(MS):
        k, da, a, app = estraiMossa(pop(CS))
        if k==1: muovi(da, a)
        else: # notare l'ordine inverso
            push(CS, creaMossa(k-1, app, a, da))
            push(CS, creaMossa(1, da, a, app})
            push(CS, creaMossa(k-1, da, app, a))
```

quale argomento usereste per dimostrare la terminazione?

creaMossa e estraiMossa come creaHAR…

E se volessimo lo stato del gioco?

Se volessimo **stampare lo stato del gioco**… o produrre una versione **grafica** che **sposta i dischi a video**?

Non ci basterebbe sapere la sequenza delle mosse, ma **dobbiamo sapere quali dischi** e a che **altezza si trovano**…

Facile, **uso tre pile**…

```
def initTorri(n):
    t = allocaVet(3)
    for i=0 to 2: t[i] = newStack(n)
        # possiamo dare la dim massima
    for i=n downto 1:
        # carico i dischi a diametro
        # decrescente sul primo piolo
        push(t[0], i)
    return t
```

```
def muovi(da, a, t):
    push(t[a], pop(t[da]))
```

Problema del Massimo Fattore Primo

Ricordiamo il **problema**: la funzione `scomponi(n)` restituisce sempre una coppia di numeri f_1 e f_2 tali che $f_1 \cdot f_2 = n$. Se n è **primo**, restituisce la coppia 1, n mentre se n è **composto** f_1 , $f_2 \neq 1$, ma non possiamo fare assunzioni su chi siano f_1 , f_2.

Determinare il massimo fattore primo di un naturale n, usando solo **`scomponi`** come abilità aritmetica.

Rivediamo il programma ricorsivo:

```
def maxPrimo(n):
    f1, f2 = scomponi(n)
    if f1 == 1 return f2
        # n è primo ed è il suo massimo
        # fattore primo
    # altrimenti
    return max(maxPrimo(f1), maxPrimo(f2))
```

Massimo Fattore Primo Iterativo

La versione iterativa **memorizza i numeri** ottenuti via via da **`scomponi`** fino a che `f1 == 1`, nel qual caso si confronta `f2` con il massimo trovato fino a quel momento.

A dire il vero, siccome **non è rilevante l'ordine** con cui esaminiamo i fattori prodotti (**max è commutativo**!), potevamo usare una **coda** o **altra struttura dati**.

Morale 1: anche se non siete compilatori, l'**eliminazione** della **ricorsione** passa per **una struttura dati**.

Morale 2: "**Think recursively**, act iteratively, only **when needed**".

```
def maxPrimoIter(n):
    F = newStack()
    push(F, n)
    maxP = 2
    while not isEmpty(F):
        f = pop(F)
        f1, f2 = scomponi(f)
        if f1 == 1:
            maxP = max(f2, maxP)
        else:  push(F, f1)
               push(F, f2)
    return maxP
```

```
F = newQueue
enqueue(F, n)
```

```
f = dequeue(F)
```

```
enqueue(F, f1)
enqueue(F, f1)
```

Heap e Code con Priorità

Code con Priorità

Le code con priorità sono un tipo di dato utile a disciplinare l'accesso a risorse seguendo una disciplina che dipende dalla priorità.

Le operazioni che ci aspettiamo dal **tipo di dato coda con priorità** sono:

- `P = newPQ(n)`: **crea** una nuova coda con priorità [con *n* posti, **opzionale**].
- `insert(P, x, p)`: **aggiunge** l'elemento *x* con priorità *p* alla coda
- `m = max(P)`: **legge** l'elemento di massima priorità nella coda e ne **restituisce** il valore [senza rimuoverlo]
- `m = extract_max(P)`: **estrae** l'elemento di massima priorità dalla coda e ne **restituisce** il valore
- `b = isFull(P)`: restituisce **TRUE se la coda è piena** (dà sempre FALSE se non è previsto un numero massimo di elementi nella coda)
- `b = isEmpty(P)`: restituisce **TRUE se la coda è vuota** (non contiene nessun elemento).

Implementazione Code con priorità

In una coda con priorità, l'**elemento** che viene **estratto** è quello con **maggiore priorità**.

Gli heap sono una struttura dati particolarmente adatta a rappresentare questo tipo di dato, in quanto **estrazioni e inserimenti** si possono fare in tempo $\boldsymbol{\theta(\log n)}$.

Vanno bene sia gli heap rappresentati a record & puntatori che quelli propriamente detti rappresentati con un vettore.

➧ **Esercizio**: e un albero binario di ricerca nella rappresentazione di code con priorità? quali sono le complessità delle operazioni ed eventuali punti deboli?

Nel seguito vediamo brevemente l'implementazione di alcune operazioni con heap su vettori.

Implementazione Code con Priorità /1

Conveniamo di rappresentare un heap con un record **`H`** in cui **`H->v`** è il vettore e **`H->heapSize`** è la parte occupata del vettore.

In un max-heap, **leggere il massimo** costa $\Theta(1)$, in quanto il massimo è sempre il primo elemento del vettore.

L'**estrazione** del massimo costa $\Theta(\log n)$ perché dopo l'estrazione occorre chiamare **`heapify`** per ripristinare la struttura dati heap.

```
def max(H):
    return H->v[0]
```

$\Theta(\log n)$

$\Theta(\log n)$

```
def extract-max(H):
    tmp = H->v[0]
    H->v[0], H->v[heapSize] =
            H->v[heapSize], H->v[0]
    H->heapSize = H->heapSize - 1
    heapify(H->v, 0)
    return tmp
```

Implementazione Code con Priorità /2

Analogamente, l'**inserimento** costa $\Theta(\log n)$ e richiede una funzione **`heapifyUp`** che ristabilisca l'ordinamento verticale verso l'alto, in modo del tutto analogo a **`heapify`** (che lo fa verso il basso).

Chiudiamo, osservando che sarebbe possibile migliorare l'efficienza evitando di fare gli scambi in **`heapify`** e **`heapifyUp`** e facendo scivolare gli elementi verso il basso/alto usando la tecnica tipica di **`insertionSort`**.

```
def up(i):
    return (i+1) div 2 - 1
```

$\Theta(\log n)$

```
def hepifyUp(v, i):
    p = up(i)
    if p > 0:
        if v[i] > v[p]:
            v[i], v[p] = v[p], v[i]
            heapifyUp(v, up(i))
```

$\Theta(\log n)$

```
def insert(H, x):
    H->v[H->heapSize] = x
        H->heapSize = H->heapSize + 1
    heapifyUp(H->v, H->heapSize)
```

$\Theta(\log n)$

Code con priorità: starvation & aging

- Segnalo un interessante problema sulle code con priorità, detta **starvation** (morte per fame): un elemento potrebbe **non uscire mai dalla coda** perché **entrano** sempre **elementi con maggiore priorità**.

Si usano tecniche di **aging** (gli elementi aumentano di priorità col tempo) per evitare queste situazioni usualmente indesiderate.

Usando **`heapifyUp`** si può facilmente aumentare il valore di una chiave mantenendo la struttura di heap.