Mergesort

Informatica@SEFA 2017/2018 - Lezione 13

Massimo Lauria <massimo.lauria@uniroma1.it>*

Mercoledì, 15 Novembre 2017

1 Struttura a pila (stack)

La **pila** è una delle **strutture dati** più semplici. Una struttura dati è un modo ragionato di disporre le dati in memoria, aggiungendo riferimenti incrociati e informazioni di supporto, che rendano efficienti alcuni operazioni.

Quale disposizione dei dati va adottata? Questo dipende dal tipo di operazioni che si vogliono fare velocemente:

- inserimenti, cancellazioni, aggiornamenti;
- ricerca di un elemento (e.g. lista ordinata o dizionario);
- inserimenti in coda o nel mezzo dei dati.

Non ci si dovrebbe sorprendere scoprendo che per rendere veloci certe operazioni si sacrificare l'efficienza di altre. E che le strutture dati con le caratteristiche migliori sono anche più complesse da organizzare o richiedono molte informazioni ausiliare.

La pila è una struttura dati che permette

- inserimento di dati (**push**)
- estrazione del dato più recente inserito (**pop**)

*http://massimolauria.net/courses/infosefa2017/

ovvero una struttura dati LAST-IN FIRST-OUT (**LIFO**).

Prima di discutere perché è interessante utilizzare una struttura dati di questo tipo è meglio chiarire con esempi il suo funzionamento.

In generale la pila è inclusa nella libreria del linguaggio oppure va implementata. In Python è possibile utilizzare una lista come pila.

```
pila = []    # Una pila vuota

pila.append(10)         # operazione PUSH
print(pila)

pila.append("gatto")    # operazione PUSH
print(pila)

pila.pop()              # operazione POP
print(pila)

pila.append([1,2,3])    # operazione PUSH
print(pila)

pila.pop()              # operazione POP
print(pila)

pila.pop()              # operazione POP
print(pila)

pila.pop()              # POP da pila vuota = un errore
```

```
[10]
[10, 'gatto']
'gatto'
[10]
[10, [1, 2, 3]]
[1, 2, 3]
[10]
10
[]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop from empty list
```

Il nome pila viene dal fatto che potete immaginare i dati come se fossero impilati uno sull'altro. Ogni dato nuovo va in cima alla pila. L'unico dato accessibile è quello in cima alla pila. In Python è possibile usare una lista, pensando che la cima della pila sia l'ultima posizione della lista stessa.

> Naturalmente se utilizzate altre operazioni di accesso ai dati diverse da append e pop allora non state più utilizzando la lista come se fosse una pila.

1.1 Usi della pila: contesti annidati

Fate conto che state leggendo un articolo di Economia, dove ad un certo punto viene utilizzato come argomento un'analisi statistica che utilizza un metodo che non conoscete. Allora potete andare a guardare un libro di statistica per chiarirvi le idee. Ma il formalismo matematico non vi confonde e andate a guardare una pagina wikipedia di matematica. Una volta chiarito il dubbio tornate al libro di statistica, e successivamente tornate all'articolo di economia.

Ogni volta che aprite un contesto nuovo in un certo senso **congelate** la situazione in cui eravate nel contesto precedente, e quando avete finito la riprendere da dove eravate rimasti.

1.2 Esempio: chiamate di funzioni

Quando richiamate una funzione da un vostro programma lo stato **chiamante**, ovvero delle variabili, del punto del programma a cui siete arrivati, ecc... viene **salvato** e la funzione **chiamata** inizia la sua esecuzione. Al termine di questa funzione (che a sua volta può richiamare altre funzioni) lo stato del programma chiamante viene ripristinato ed il chiamante può procedere.

```
def primo_livello():
    i = 5
    print("Primo Livello 1: i={}".format(i))
    secondo_livello()
    print("Primo Livello 2: i={}".format(i))
    i = 4
    print("Primo Livello 3: i={}".format(i))
    secondo_livello()
    print("Primo Livello 4: i={}".format(i))

def secondo_livello():
    i=-3
    print("Secondo Livello: i={}".format(i))


i=10
primo_livello()
```

```
Primo Livello 1: i=5
Secondo Livello: i=-3
Primo Livello 2: i=5
Primo Livello 3: i=4
Secondo Livello: i=-3
Primo Livello 4: i=4
```

La struttura dati nella quale questi contesti vengono salvati e ripristinati è una pila,

- l'entrata in una funzione è un **push**,
- l'uscita da una funzione è un **pop**.

1.3 Esempio: espressioni matematiche

Pensate ad un espressione matematica con le parentesi. All'apertura della parentesi l'espressione che si stava valutando in precedenza viene sospesa, fino a quando non si incontra una parentesi contraria che la bilanci. A quel punto la valutazione dell'espressione esterna viene ripresa.

Esempio: un piccolo programma che usa una pila per controllare che una serie di parentesi sia bilanciata.

```
def bilanciata(espressione):
    pila=[]
    for c in espressione:
        if c=='(':
            pila.append(c)
        elif c==')':
            if len(pila)==0:
                return False
            pila.pop()
    return len(pila)==0

print( bilanciata(" (()()) ") )
print( bilanciata(" )") )
print( bilanciata(" (") )
print( bilanciata(" ((()))(()())  ") )
print( bilanciata(" ( )  )(()())  ") )
```

```
True
False
False
True
False
```

1.4 Limite della ricorsione

Ogni volta che si chiama una funzione una certa quantità di informazioni vengono salvate sulla pila dedicata.

1. il punto in cui si trova l'esecuzione
2. il valore delle variabili visibili in quel punto

Questo ha un costo in termini di tempo e di memoria. In particolare è possibile che una ricorsione molto profonda (i.e., con tanti livelli di annidamento) possa richiedere troppa memoria. Vediamo un esempio di annidamento profondo.

```
def icounter(x=0):
    i=x
    while i<10000:
        i=i+1
    print("Iterazione terminata")

def rcounter(x=0):
    if x <= 10000:
        rcounter(x+1)
    else:
        print("Ricorsione terminata")

icounter()
rcounter()
```

```
Iterazione terminata
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in rcounter
  File "<stdin>", line 3, in rcounter
  File "<stdin>", line 3, in rcounter
  [Previous line repeated 994 more times]
  File "<stdin>", line 2, in rcounter
RecursionError: maximum recursion depth exceeded in comparison
```

2 Mergesort

La comprensione della struttura dati pila ci permette di capire più agevolmente algoritmi ricorsivi. Ora vediamo il **mergesort** un algoritmo ricorsivo di ordinamento per confronto e che opera in tempo $O(n \log n)$ e quindi è ottimale rispetto agli algoritmi di ordinamento per confronto.

2.1 Un approccio divide-et-impera

Un algoritmo può cercare di risolvere un problema

- dividendo l'input in parti
- risolvendo il problema su ogni parte
- combinando le soluzioni parziali

Naturalmente per risolvere le parti più piccole si riutilizza lo stesso metodo, e quindi si genera una gerarchia di applicazioni del metodo, annidate le une dentro le altre, su parti di input sempre più piccole, fino ad arrivare a parti così piccole che possono essere elaborate direttamente.

Lo schema divide-et-impera viene utilizzato spesso nella progettazione di algoritmi. Questo schema si presta molto ad una implementazione ricorsiva.

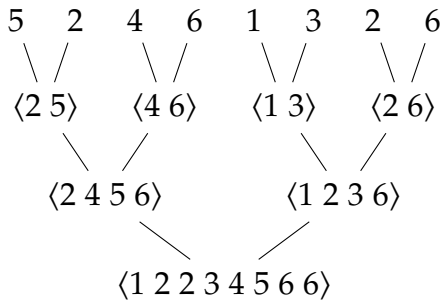
2.2 Schema principale del mergesort

1. dividere in due l'input
2. ordinare le due metà
3. fondere le due sequenze ordinate

Vediamo ad esempio come si comporta il mergesort sull'input

$$\langle 5\ 2\ 4\ 6\ 1\ 3\ 2\ 6 \rangle$$

La sequenza ordinata viene ottenuta attraverso questa serie di fusioni.



2.3 Implementazione

Lo scheletro principale del mergesort è abbastanza semplice, e non è altro che la trasposizione in codice dello schema descritto in linguaggio naturale.

```
def mergesort(S,start=0,end=None):
    if end is None:
        end=len(S)-1
    if start>=end:
        return
    mid=(end+start)//2
    mergesort(S,start,mid)
    mergesort(S,mid+1,end)
    merge(S,start,mid,end)
```

2.4 Fusione dei segmenti ordinati

Dobbiamo fondere due sequenze ordinate, poste peraltro in due segmenti adiacenti della stessa lista. L'osservazione principale è che il minimo della sequenza fusa è il più piccolo tra i minimi delle due sequenze. Quindi si mantengono due indici che tengono conto degli elementi ancora da fondere e si fa progredire quello che indicizza l'elemento più piccolo. Quando una delle due sottosequenze è esaurita, allora si mette in coda la parte rimanente dell'altra. `merge` usa una **lista aggiuntiva temporanea** per fare la fusione. I dati sulla lista temporanea devono essere copiati sulla lista iniziale.

```
def merge(S,low,mid,high):
    a=low
    b=mid+1
    temp=[]
    # Parte 1 - Inserisci in testa il pi  piccolo
    while a<=mid and b<=high:
        if S[a]<=S[b]:
            temp.append(S[a])
            a=a+1
        else:
            temp.append(S[b])
            b=b+1
    # Parte 2 -  Esattamente UNA sequenza    esaurita. Va aggiunta l'altra
    if a<=mid:
        residuo = range(a,mid+1)
    else:
        residuo = range(b,high+1)
    for i in residuo:
        temp.append(S[i])
    # Parte 3 - Va tutto copiato su S[start:end+1]
    for i,value in enumerate(temp,start=low):
        S[i] = value
```

Questo conclude l'algoritmo

```
dati=[5,2,4,6,1,3,2,6]
mergesort(dati)
print(dati)
```

```
[1, 2, 2, 3, 4, 5, 6, 6]
```

2.5 Running time

Per cominciare osserviamo che nelle prime due parti di `merge` un elemento viene inserito nella lista temporanea ad ogni passo, e poi questo elemento non viene più considerato. La terza parte ricopia tutti gli elementi passando solo una volta su ognuno di essi. Pertanto è chiaro che `merge` di due segmenti adiacenti di lunghezza n_1 e n_2 impiega $\Theta(n_1 + n_2)$ operazioni.

Definiamo come $T(n)$ il numero di operazioni necessarie per ordinare una lista di n elementi con `mergesort`. Allora

$$T(n) = 2T(n/2) + \Theta(n) \tag{1}$$

quando $n > 1$, altrimenti $T(1) = \Theta(1)$ e dobbiamo risolvere **l'equazione di ricorrenza** rispetto a T. Prima di tutto per farlo fissiamo una costante $c > 0$ abbastanza grande per cui

$$T(n) \leq 2T(n/2) + cn \qquad T(1) \leq c\ . \tag{2}$$

Espandendo otteniamo

$$T(n) \leq 2T(n/2) + cn \leq 4T(n/4) + 2c(n/2) + cn = 4T(n/4) + 2cn \tag{3}$$

Si vede facilmente, ripetendo l'espansione, che

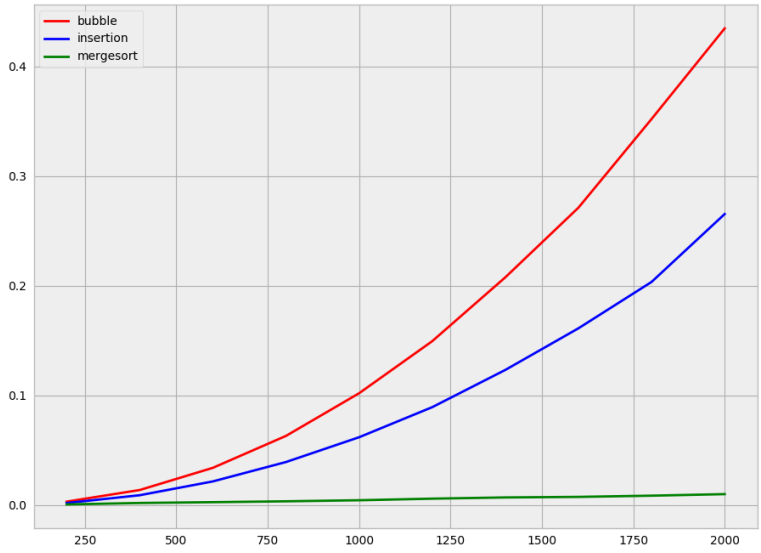
$$T(n) \leq 2^t T(n/2^t) + tcn \tag{4}$$

fino a che si arriva al passo t^* per cui $n/2^{t^*} \leq 1$, nel qual caso si ottiene $T(n) = c2^{t^*} + t^*cn \leq c(t^* + 1)n$.

Il più piccolo valore di t^* per cui $n/2^{t^*} \leq 1$ è $O(\log n)$, e quindi il running time totale è $O(n \log n)$.

Poichè il mergesort è un ordinamento per confronto il running time è $\Omega(n \log n)$, ed in ogni caso questo si può vedere anche direttamente dall'equazione di ricorrenza. Quindi il running time è in effetti $\Theta(n \log n)$.

2.6 Confronto sperimentale con insertion sort e bubblesort



2.7 Una piccola osservazione sulla memoria utilizzata

Mentre `bubblesort` e `insertionsort` non utilizzano molta memoria aggiuntiva oltre all'input stesso, `mergesort` produce una lista temporanea di dimensioni pari alla somma di quelle da fondere. E oltretutto deve ricopiarne il contenuto nella lista iniziale.

Con piccole modifiche al codice, che non vedremo, è possibile controllare meglio la gestione di queste liste temporanee e rendere il codice ancora più efficiente, dimezzando il tempo per le copie e riducendo quello per l'allocazione della memoria. In generale se nessuna di queste liste viene liberata prima della fine dell'algoritmo, la quantità di memoria aggiuntiva è $\Theta(n \log n)$, tuttavia se la memoria viene liberata in maniera più aggressiva allora quella aggiuntiva e $\Theta(n)$.