Informatica per Statistica
Riassunto della lezione dell'08/11/2013

Igor Melatti

Implementazione di *Binary-Search-Seq-Corr* e *Merge-Sort-Seq*

- Usare le sottosequenze è comodo, ma occorre prestare attenzione quando le si implementa in Python
- Per capire come mai, occorre capire cosa accade quando si effettuano assegnamenti che hanno a che vedere con le sottosequenze
- Si considerino i seguenti casi:

  1. assegnamento ad una variabile di una sequenza costante
     - qui non ci sono particolari problemi, l'effetto è quello che ci si aspetta
     - si provi ad eseguire il seguente esempio:

       ```
       A = [1, 2, 3]
       B = [1, 2, 3]
       A[2] = 6
       print A
       print B
       B[1] = 5
       print A
       print B
       ```

     - come si può notare, `A` e `B` sono indipendenti, ovvero si trovano in aree di memoria *diverse* (vedere Figura 1)
  2. assegnamento ad una variabile di un'altra variabile, di tipo sequenza
     - qui cominciano i risultati inattesi
     - si provi ad eseguire il seguente esempio (uguale a prima, ma cambia l'inizializzazione di `B`):

       ```
       A = [1, 2, 3]
       B = A
       A[2] = 6
       ```

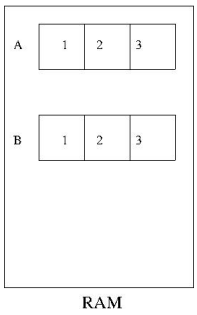

Figure 1: Assegnamento di liste costanti, o assegnamento tramite sottoliste

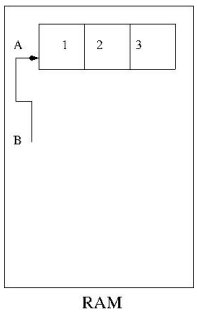

Figure 2: Assegnamento di liste a liste

```
print A
print B
B[1] = 5
print A
print B
```

- come si può notare, `A` e `B` sono in realtà la stessa sequenza: una modifica fatta all'uno si riflette anche sull'altra
- ovvero, `A` e `B` si trovano in aree di memoria *nella stessa area di memoria* (Figura 2)
- questo succede, per quanto riguarda questo corso, *solo per le sequenze*
- ad esempio, se `A` e `B` fossero stati numeri, questo non sarebbe successo:

```
A = 1
B = A
A = 6
print A
print B
B = 5
```

```
print A
print B
```

   - qui `A` e `B` non interferiscono tra loro: sono su aree di memoria *differenti*

3. assegnamento ad una variabile di una sottosequenza di un'altra variabile
   - qui si ritorna ai risultati "ragionevoli"
   - si provi ad eseguire il seguente esempio (uguale a prima, ma cambia l'inizializzazione di `B`):

```
A = [1, 2, 3]
B = A[0:]
A[2] = 6
print A
print B
B[1] = 5
print A
print B
```

   - come si può notare, `A` e `B` sono sequenze su aree di memoria *diverse*: una modifica fatta all'uno *non* si riflette anche sull'altra

- Dagli esempi precedenti si vede che c'è molta differenza tra gli assegnamenti `B = A` e `B = A[0:]` (o, equivalentemente a quest'ultim0, `B = A[:]`)
  - se si assegna una variabile di tipo sequenza, senza estrarre alcuna sottosequenza, si passa un *riferimento* o *puntatore*
  - è come se, nel fare `B = A`, anziché copiare tutta la sequenza, si copiasse solo una freccia che "punta" (nel senso di "va verso") alla zona di memoria che contiene `A`
  - invece, nel fare `B = A[:]`, si copia tutta la sequenza in una nuova zona di memoria

- Quando si passa una variabile come argomento di una funzione, si fa a tutti gli effetti un assegnamento
  - quindi, ecco cosa succede al momento della chiamata a funzione in un caso del genere:

```
B = [1, 2, 3]
def funzione(A): A[2] = 6
funzione(B)
print B
```

    1. si crea lo stack frame per `funzione`; dentro solo c'è la variabile `A`

2. si copia nell'`A` dello stack frame di `funzione` il `B` delle istruzioni principali; per farlo, si fa esattamente `A = B`
   * quindi, coerentemente con il caso 2 di cui sopra, si passa ad `A` il riferimento a `B`
   * quindi `A` e `B` puntano alla stessa area di memoria
3. nel fare `A[2] = 6`, la modifica si ripercuote anche su `B`, dato che l'area di memoria è la stessa
   * è di nuovo coerente con il caso 2 di cui sopra
   * è su questo che si basano le funzioni per realizzare l'ordinamento (come `insertion_sort`): le modifiche fatte all'argomento `A` hanno effetto anche sulla variabile usata nella chiamata

- Tenendo presente tutto ciò, si può procedere con le implementazioni Python di *Binary-Search-Seq-Corr* e *Merge-Sort-Seq*; si prenda inizialmente in considerazione la prima, riportata in Figura 3

```
1  Binary-Search-Seq-Corr(A, p, k)
2    n ← length(A)
3    if (n = 0) then
4      return 0
5    q ← ⌊n/2⌋
6    if (A[q] = k) then
7      return p + q
8    if (A[q] > k) then
9      return Binary-Search-Seq-Corr(A[1..q − 1], p, k)
10   else
11     return Binary-Search-Seq-Corr(A[q + 1..n], p + q + 1, k)
```

Figure 3: Ricerca binaria

  – dal momento che questa funzione non deve modificare la sequenza passatagli come argomento, non ci sono particolari problemi di implementazione: vedere Figura 4

- Le cose sono più complesse per il Merge Sort (riportato in Figura 5), dove appunto occorre modificare la sequenza data in input
- Pertanto, implementarlo "semplicemente" come in Figura 6 è sbagliato: la sequenza non viene effettivamente modificata
- Invece, occorre procedere con più attenzione, come in Figura 7

```
1  def binary_search(A, p, k):
2    n = len(A)
3    if (n == 0):
4      return 0
5    q = n/2
6    if (A[q] == k):
7      return p + q
8    if (A[q] > k):
9      return binary_search(A[0 : q], p, k)
10   else:
11     return binary_search(A[q + 1 : n], p + q + 1, k)
```

Figure 4: Ricerca binaria in Python

  - visto che la sequenza originale non viene modificata, ma viene modificata solo la copia locale, allora ci si può far *ritornare* questa sequenza locale (correttamente modificata) ed *assegnarla* alla sequenza originale
  - per far sì che solo la parte interessata venga modificata, basta usare nuovamente le sottosequenze: `A[ : q] = merge_sort(A[ : q])` e `A[q : ] = merge_sort(A[q : ])` fanno sì che la sequenza ritornata da `merge_sort` (parte sinistra dell'uguale) venga assegnata alla corrispettiva sottosequenza dell'originale (parte destra)

- **Esercizio:** cosa succede se si mette `A = B` al posto di `A[0:] = B` come ultima istruzione del `merge`? Come si può spiegare?

```
1  Merge-Sort-Seq(A)
2    n ← length(A)
3    if (n > 1) then
4      q ← ⌊n/2⌋
5      Merge-Sort-Seq(A[1..q])
6      Merge-Sort-Seq(A[q+1..n])
7      Merge-Seq(A, q)
8
9  Merge-Seq(A, q)
10   n ← length(A)
11   i ← 1
12   j ← 1
13   m ← q+1
14   while j ≤ q and m ≤ n do
15     if (A[j] < A[m]) then
16       B[i] ← A[j]
17       j ← j+1
18       i ← i+1
19     else
20       B[i] ← A[m]
21       m ← m+1
22       i ← i+1
23   for j ← j to q do
24     B[i] ← A[j]
25     i ← i+1
26   for m ← m to n do
27     B[i] ← A[m]
28     i ← i+1
29   for i ← 1 to n do
30     A[i] ← B[i]
```

Figure 5: Merge Sort con le sottosequenze

```
1  def merge_sort(A):
2    n = len(A)
3    if (n > 1):
4      q  = n/2
5      merge_sort(A[ : q])
6      merge_sort(A[q : ])
7      merge(A, q)
8
9  def merge(A, q):
10   n = len(A)
11   B = []
12   j = 0
13   m = q
14   while j < q and m < n:
15     if (A[j] < A[m]):
16       B = B + [A[j]]
17       j = j + 1
18     else:
19       B = B + [A[m]]
20       m = m + 1
21   for j in range(j, q):
22     B = B + [A[j]]
23   for m in range(m, n):
24     B = B + [A[m]]
25   for i in range(0, n):
26     A[i] = B[i]
```

Figure 6: Merge Sort in Python (sbagliato)

```
1  def merge_sort(A):
2    n = len(A)
3    if (n > 1):
4      q  = n/2
5      A[ : q] = merge_sort(A[ : q])
6      A[q : ] = merge_sort(A[q : ])
7      merge(A, q)
8    return A
9
10 def merge(A, q):
11   n = len(A)
12   B = []
13   j = 0
14   m = q
15   while j < q and m < n:
16     if (A[j] < A[m]):
17       B = B + [A[j]]
18       j = j + 1
19     else:
20       B = B + [A[m]]
21       m = m + 1
22   for j in range(j, q):
23     B = B + [A[j]]
24   for m in range(m, n):
25     B = B + [A[m]]
26   A[0:] = B
```

Figure 7: Merge Sort in Python (corretto)