Informatica per Statistica
Riassunto delle lezioni del 23/10/2013 e del 25/10/2013

Igor Melatti

Implementazione dell'*Insertion Sort* in Python

- Si consideri il programma Python di Figura 1

```
1 def insertion_sort(A):
2   for j in range(1, len(A)):
3     key = A[j]
4     i = j - 1
5     while (i >= 0 and A[i] > key):
6       A[i + 1] = A[i]
7       i = i - 1
8     A[i + 1] = key
```

Figure 1: Implementazione dell'*Insertion Sort* come funzione Python

```
1 Insertion-Sort(A)
2   for j ← 2 to length(A) do
3     key ← A[j]
4     /* Inserimento di A[j] nella sequenza ordinata A[1...j-1] */
5     i ← j-1
6     while i > 0 and A[i] > key do
7       A[i+1] ← A[i]
8       i ← i-1
9     A[i+1] ← key
```

Figure 2: Algoritmo *Insertion Sort*

  - è un'implementazione in Python dell'algoritmo *Insertion Sort* in Figura 2

- in Figura 1 è presente una *definizione di funzione*, ovvero la funzione `insertion_sort`, e null'altro
- quindi, eseguendolo come un programma, non ha apparentemente nessun effetto, ovvero non chiede dati in input (ad esempio da tastiera) e non stampa nulla su schermo
- nonostante questo, la funzione `insertion_sort` ha un input (una sequenza di numeri, o meglio una *lista* di numeri) e un output (sempre una sequenza o lista di numeri)
- la funzione `insertion_sort` ha l'intestazione alla riga 1 e il corpo dalla riga 2 alla riga 8
  - in particolare ha un solo parametro: `A` che viene usato nel corpo come una *sequenza* o *array* o *lista* di numeri
  - dalla sola intestazione il tipo non lo si può capire
- all'interno del corpo della funzione `insertion_sort` (righe 2-8):
  - il blocco di istruzioni implementa l'algoritmo *Insertion Sort*, ci si ritornerà più avanti
  - l'output della funzione `insertion_sort`, da un punto di vista strettamente sintattico, non c'è
  - concettualmente, l'output della funzione `insertion_sort` è l'array `A`, esattamente come accadeva all'algoritmo di riferimento *Insertion Sort*
  - le funzioni di questo tipo, ovvero che non fanno una `return`, sono dette anche *procedure*
    - sono usate per provocare i cosiddetti *effetti collaterali*
    - esempio tipico: per scrivere qualcosa su schermo con delle `print`
    - si vedrà meglio nelle lezioni a seguire
- come detto, questo programma, se eseguito, non fa niente, quindi sembrerebbe poco utile
- ma con un piccolo accorgimento si può fare sì che sia utile: si supponga di aver salvato il programma nel file `insertion_sort.py`
- si apra l'interprete interattivo (ma quanto detto lo si può anche scrivere su un secondo file, e far eseguire quello) e si scriva quanto riportato in Figura 3
- le righe 2–4 possono essere ripetute a piacere per ordinare diverse sequenze
- si consideri riga per riga quanto scritto in Figura 3
  - riga 1 esegue l'intero programma Python contenuto nel file `insertion_sort.py`; dato il contenuto di quest'ultimo, eseguirlo significa definire la funzione `insertion_sort` (senza chiamarla)

* in pratica, si sta usando `insertion_sort.py` come una libreria che definisce una sola funzione, `insertion_sort` per l'appunto
* riga 2 definisce una variabile `sequenza` di tipo lista
* per farlo, assegna a `sequenza` una costante di tipo lista
* sintatticamente, una costante di tipo lista è `[`<espressione1>`, ..., `<espressione$n$ >`]`
* n può anche essere zero, nel qual caso si ha la sequenza vuota
* come detto nelle lezioni iniziali, per accedere all'i-esimo elemento di una lista `sequenza` basta scrivere `sequenza`$[i-1]$
  - precisazione: nel seguito, quando si parla di indici di sequenze nella semantica, si intenderanno gli indici come nel linguaggio naturale
  - ovvero, il primo elemento è l'1-esimo, il secondo è il 2-esimo, il terzo è il 3-esimo, ... il ventunesimo è il 21-esimo, ...
  - questo coincide con la notazione dello pseudocodice, mentre per adattarsi a quello che fa Python occorre aggiungere 1 (quando si passa dall'indice Python all'indice "naturale") o sottrarre 1 (quando si fa il percorso inverso)
  - quindi `sequenza[0]` è il primo elemento di `sequenza`, ovvero l'1-esimo, ovvero il $(0+1)$-esimo elemento di `sequenza`
  - analogamente `sequenza[10]` è l'undicesimo elemento di `sequenza`, ovvero l'11-esimo, ovvero il $(10+1)$-esimo elemento di `sequenza`
  - al contrario, l'ottavo elemento di `sequenza` è quello all'indice $8-1$, quindi `sequenza[7]`
* in generale, sintatticamente si scrive <nome_var>`[`<espressione>`]` (operazione di *dereferenziamento*), e la semantica di $N[E]$ è, **se** $0 \leq v \leq |N|-1$**, il valore al posto** $v+1$ **nella sequenza** N**, con** v **risultante dalla valutazione di** E
* **se altrimenti** $-|N| \leq v < 0$, la semantica di $N[E]$ è **il valore al posto** $|N|+v+1$ **nella sequenza** N**, con** v **risultante dalla valutazione di** E (ad esempio, `sequenza[-1]` è equivalente a `sequenza[len(sequenza) - 1]`, quindi è l'ultimo elemento; `sequenza[-2]` è il penultimo, e `sequenza[-len(sequenza)]` è il primo)
* **altrimenti, se** $v \geq |N|$ **oppure** $v < -|N|$**, viene restituito un errore**
* è inoltre disponibile la funzione `len(sequenza)`, che ritorna il numero di elementi nella lista `sequenza`
* quindi il primo elemento di una lista `sequenza` è `sequenza[0]`, l'ultimo è `sequenza[len(sequenza) - 1]`

    * in realtà, Python sulle liste è molto ricco: permette ad esempio di estrarre sottoliste con la sintassi <nome_var>[<espressione1>:<espressione2>]
    * la semantica di $N[E_1 : E_2]$ è **la sottosequenza di N che va dall'indice $v_1 + 1$ all'indice v_2 compresi, con v_1 risultante dalla valutazione di E_1 e v_2 risultante dalla valutazione di E_2** (si possono usare anche indici negativi)
      · per quanto detto sopra, nella semantica si parla di indici "naturali"
      · volendo vederli ancora come indici del Python, allora la sottosequenza $N[E_1 : E_2]$ va dall'indice v_1 compreso a v_2 escluso (ovvero da v_1 a $v_2 - 1$)
    * il fatto di considerare diversamente i due estremi della sottosequenza (il primo indice è incluso, il secondo escluso) può sembrare controintuitivo, ma ha alcune belle proprietà, come ad esempio il fatto che $v_2 - v_1$ dà il numero di elementi nella sottosequenza (a meno che non si usino indici negativi)
    * inoltre, si possono "sommare" 2 liste o moltiplicare una lista per un intero: il risultato è lo stesso che si ottiene con le stringhe
    * quindi sommare 2 liste vuol dire concatenarle, moltiplicare una lista per un intero n vuol dire ripeterla n volte
    * si ricordi che invece, nello pseudocodice, gli indici delle sequenze vanno da 1 alla lunghezza della sequenza compresa

```
1 import insertion_sort
2 sequenza = [10, 20, 5, 15]
3 insertion_sort.insertion_sort(sequenza)
4 print sequenza
```

Figure 3: Un modo per invocare la funzione `insertion_sort` di Figura 1

  – concettualmente, una variabile di tipo lista non è altro che una scorciatoia per dichiarare n variabili contemporaneamente, con n lunghezza della sequenza
  – i vantaggi sono molteplici
    * se anziché `sequenza = [10, 20, 5, 15]` si fosse scritto
      `n1 = 10`
      `n2 = 20`
      `n3 = 5`
      `n4 = 15`
      non ci sarebbe stato modo di usare una variabile `i` e dire ad esempio che, quando `i` = 3, occorre considerare `n3`

    - invece, con gli array si può scrivere `sequenza[i]`, come detto, e questo è cruciale in moltissimi algoritmi
  - inoltre, se si dovessero dichiarare solo 5 o 10 variabili, le si potrebbe anche scrivere esplicitamente; ma se fossero 1000 o più, diventerebbe complicato per il programmatore
  - il fatto che gli indici delle sequenze di Python vadano da 0 a $n-1$ anziché da 1 a n come nello pseudocodice spiega perché nell'implementazione dell'algoritmo di Insertion Sort sono stati cambiati alcuni controlli
    - per la precisione, il ciclo `for` va da 1 a $n-1$ (c'è il minore stretto...) in Figura 1, mentre va da 2 ad n in Figura 2
    - e il ciclo while controlla che `i >= 0` anziché $i > 0$
  - esempi di uso della notazione di dereferenziamento con le parentesi quadre in Figura 1:
    - in riga 3, alla variabile `key` viene assegnato, tra quelli all'interno dell'array `A`, il `double` che si trova all'indice attualmente indicato dalla variabile `j`
    - in riga 6, il valore che all'interno dell'array `A` si trova all'indice attualmente indicato dalla variabile `i` aumentato di 1, viene sostituito con il valore che si trova all'indice attualmente indicato dalla variabile `i`
    - quindi, se `i` vale 3, l'effetto è che il valore attualmente dentro `A[4]` viene cancellato e al suo posto vi viene copiato `A[3]` (ovvero l'elemento immediatamente precedente)

- La sintassi (semplificata) del `for` è la seguente:

```
for <nome_var> in <espressione>:
        <blocco_istruzioni>
```

  - come nelle altre istruzioni di controllo di flusso, se <blocco_istruzioni> è costituito da più di una istruzione, va propriamente indentato; altrimenti, può anche essere scritto sulla stessa riga del for
  - per quanto riguarda la *semantica* dell'istruzione `for` N `in` E`:` B, vale quanto segue:
  - **l'effetto è quello di valutare per prima cosa E, che deve risultare in una sequenza S (altrimenti, dà errore)**
    - è semplificato ma ok per gli scopi di questo corso
  - **dopodiché si esegue B per $|S|$ volte; prima dell'esecuzione i-esima di B, si assegna l'i-esimo elemento di S a N**
    - quindi, dal punto di vista di Python, si assegna ad N il valore di $S[i-1]$

    - ovvero: questo vuol dire che prima della prima esecuzione, che è la 1-esima, si assegna ad N il primo valore di S, ovvero $S[1]$ in linguaggio naturale e $S[0]$ in linguaggio Python
    - se S è una variabile e B modifica S, questa modifica non si ripercuote sul for, dal momento che S è valutata una volta per tutte all'inizio
  - l'uso che del `for` verrà (almeno per ora) fatto sarà del tipo `for` N `in range(da, a):` B che vuol dire: *esegui* B *per* `a - da` *volte, con* N *che di volta in volta vale* $\texttt{da}, \texttt{da}+1, \texttt{da}+2, \ldots, \texttt{a}-1$
    - all'uscita dal `for`, `i` varrà `a`
    - ma B non verrà eseguito con `i` = `a`
    - `range` è una funzione predefinita (chiamabile senza importare nulla) che prende due interi a e b e restituisce la sequenza $[a, a+1, \ldots, b-1]$
    - così se occorre fare un ciclo 10000 di volte non occorre scrivere a mano una sequenza costante lunga 10000: basta usare `range`
      - per fare un esempio più facile, ma nello stesso ordine di idee: se si vuole ripetere un'istruzione I 20 volte, senza `range` occorre scrivere `for i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]:` I (oppure scrivere 20 volte I...)
      - oppure, si può scrivere in modo molto più compatto `for in in range(1, 21):` I
    - volendo, `range` prende anche un terzo argomento opzionale (ovvero, può prendere 2 o 3 argomenti): il terzo argomento, quello opzionale, vale sempre 1 quando viene omesso
    - altrimenti, `range(`a`, `b`, `c`)` restituisce la sequenza $[a, a+c, a+2c, \ldots, a+kc]$ tale che k è il massimo per cui $a + kc < b$

- La sintassi del `while` è la seguente:

```
while <condizione>:
          <blocco_istruzioni>
```

  - come nelle altre istruzioni di controllo di flusso, se <blocco_istruzioni> è costituito da più di una istruzione, va propriamente indentato; altrimenti, può anche essere scritto sulla stessa riga del for
  - <condizione> dev'essere un'espressione valutabile come un valore *booleano* (vero o falso; vedere l'`if` alla lezione 7)
  - per quanto riguarda la *semantica* dell'istruzione `while` C`:` B, vale quanto segue:
  - **l'effetto è quello di controllare se C è vera; se così è, viene eseguito B; dopodiché si controlla nuovamente se C è vera e**

così via (B, poi C, poi B, ...); se in qualsiasi momento (anche all'inizio) C è falsa, l'istruzione `while` termina e si passa alla successiva

  - è possibile mostrare che si può sempre scrivere il `for` con un `while`, ma non viceversa (in altri linguaggi di programmazione, ad esempio il C, sono invece equivalenti)
  - in realtà lo si potrebbe anche fare, ma occorrerebbe un ciclo precedente, quindi inefficiente

- Perché usare più funzioni?
  - da un punto di vista sintattico e semantico, qualsiasi programma Python può essere scritto senza definire una sola funzione, quindi usando solo le istruzioni principali
  - per esempio, il programma Python in Figura 4 è equivalente a quello di Figura 1

```
1 A = [10, 20, 5, 15]
2 for j in range(1, len(A)):
3   key = A[j]
4   i = j - 1
5   while (i >= 0 and A[i] > key):
6     A[i + 1] = A[i]
7     i = i - 1
8   A[i + 1] = key
9 print sequenza
```

Figure 4: Versione della Figura 1 con una sola definizione di funzione

  - tuttavia, messo così è meno *usabile*
  - se in seguito nasce la necessità di usare l'*Insertion Sort* in un altro programma, in cui ad esempio l'array da ordinare si chiama `B`, allora occorre riscrivere tutto il blocco di istruzioni che va dalla riga 2 alla riga 8
  - prestando attenzione al fatto di sostituire (a mano, ovvero lo deve fare il programmatore) `A` con `B`
  - si tratta di un'operazione facilmente soggetta ad errori
  - con la chiamata a funzione, invece, basta scrivere `insertion_sort(B);` e le sostituzioni le fa automaticamente il compilatore
  - la definizione della funzione `insertion_sort` resta così com'è, non sono necessarie modifiche
  - gli errori si riducono di molto

- **Esercizio 1:** il programma Python in Figura 3 è ad input fisso: per cambiare l'input, occorre modificare il programma stesso e rieseguirlo (o se usato interattivamente, occorre scrivere più volte le righe 2–4). In molti casi, questo non è desiderabile: si vuole un programma che possa accettare input *dopo* essere stato lanciato in esecuzione, ad esempio leggendoli da tastiera. Modificare il programma in Figura 3 in modo tale che:

  1. come prima cosa, venga letta da tastiera (chiamando opportunamente `input`) la dimensione dell'array
  2. dopodiché, tutti gli elementi di `A` vanno letti da tastiera (chiamando opportunamente `input`), usando un ciclo `for`
  3. dopodiché, si procede come è ora

- **Esercizio 2:** iterare l'esercizio 1 in modo tale che la sequenza in input sia letta più volte (e più volte riordinata), finché la dimensione letta da tastiera non è 0

- Si consideri nuovamente l'algoritmo *Total Sort* visto in lezione 2, riportato in Figura 5: si mostrerà ora un confronto empirico con l'*Insertion Sort*

```
1 Total-Sort(A)
2   for B in permutations(A)
3     ok ← true
4     for i ← 1 to length(B) - 1 do
5       if B[i+1] < B[i]
6         ok ← false
7    if ok
8      copy B into A
9      return
```

Figure 5: Algoritmo *Total Sort*

  - per farlo, si consideri un'implementazione dello *Total Sort* fatta come segue
  - la prima difficoltà sta nell'avere effettivamente l'oggetto che in pseudocodice è chiamato `permutations(A)`
  - in pratica, `permutations` dovrebbe essere una funzione in grado di restituire una sequenza contenente tutti i possibili riordinamenti della lista `A` in input
  - intuitivamente, come accade in Python, se in uno pseudocodice c'è scritto `for variabile in sequenza`, vuol dire che di volta in volta `variabile` assumerà tutti i valori all'interno di `sequenza`

- quindi, per poter eseguire questo algoritmo, occorre conoscere il contenuto di `sequenza`; dato che tale contenuto non è fisso, ma cambia al variare dell'input `A`, occorre *calcolarlo*
- servirebbe quindi un altro pseudocodice che dica come fare: si supponga di avere tale pseudocodice
- da notare che l'output della funzione `permutations` è una *sequenza di sequenze* (ovvero, una sequenza in cui ciascun elemento è a sua volta una sequenza)
- si supponga che la funzione `permutations`, scritta in Python in Figura 6, sia in grado di restituire una lista contenente tutti i possibili riordinamenti della lista `A` in input, ovvero sia un'implementazione dello pseudocodice (dato per buono) dell'omonima `permutations` di Figura 5

```
1  import math
2
3  def permutations(A):
4    res = []
5    for i in range(0, math.factorial(len(A))):
6      till_now = i
7      A_ord = A[0:]
8      for j in range(0, len(A)):
9        A_ord[j], A_ord[j + till_now%(len(A) - j)] = \
10         A_ord[j + till_now%(len(A) - j)], A_ord[j]
11       till_now = till_now/(len(A) - j)
12     res.append(A_ord)
13   return res
```

Figure 6: Funzione `permutations`

- questo significa che l'output della funzione `permutations` è una *lista di liste*
- a questo punto, è possibile implementare il *Total Sort* come mostrato in Figura 7
- pur nell'inefficienza di questo algoritmo, si può fare di meglio
- ovvero, l'algoritmo (e la sua implementazione), messi così, possono "esplodere" sia nella memoria che nel tempo
- se si dovessero ordinare 15 elementi, `permutations(A)` non entrerebbe in memoria, perché $15! \approx 10^{12} \approx 2^{40}$, quindi ci vorrebbe una RAM da svariati TB
- per contro, il tempo necessario per generare 15! liste è almeno all'interno della vita umana: supponendo di generare una decina di migliaia di permutazioni al secondo (il che è fattibile anche con un computer normale), si dovrebbero aspettare 3 anni

```
1  import permutations
2
3  for B in permutations.permutations(A):
4    ok = 1
5    for i in range(0, len(B) - 1):
6      if B[i] > B[i + 1]:
7        ok = 0
8        break
9    if (ok):
10     print B
11     exit(1)
```

Figure 7: Codice Python per *Total Sort*

- si può fare di meglio, e limitarsi ad esplodere nel tempo, usando pochissima memoria
- il trucco sta nello scrivere l'algoritmo (e relativa implementazione) in modo un po' più furbo: vedere l'algoritmo *Total-Sort-Better* in Figura 8

```
1  Total-Sort-Better(A)
2    n ← length(A)
3    for i ← 1 .. n!
4      B ← i_th_permutation(A, i)
5      ok ← true
6      for i ← 1 to length(B) - 1 do
7        if B[i + 1] < B[i]
8          ok ← false
9      if ok
10       copy B into A
11       return
```

Figure 8: Algoritmo *Total Sort Better*

- questa volta, non è necessario mantenere in RAM l'intero insieme delle permutazioni: basta avere un metodo per restituire l'i-esima permutazione (il che implica che le permutazioni devono essere effettivamente numerate)
- anche qui, viene mostrato direttamente il codice Python, Figura 9
- infine, la Figura 10 mostra come implementare l'algoritmo *Total Sort Better* in Figura 8
- **Esercizio:** in Figura 10 viene chiamata una funzione, `is_ord`,

```
import math

def i_th_permutation(A, i):
  till_now = i
  A_ord = A[0:]
  for j in range(0, len(A)):
    A_ord[j], A_ord[j + till_now%(len(A) - j)] = \
      A_ord[j + till_now%(len(A) - j)], A_ord[j]
    till_now = till_now/(len(A) - j)
  return A_ord
```

Figure 9: Funzione `i_th_permutation`

```
for i in range(0, math.factorial(len(A))):
  B = permutations.i_th_permutation(A, i)
  if is_ord(B):
    print B
```

Figure 10: Codice Python per *Total Sort Better*

che non è stata definita. Aiutandosi con la Figura 7, scrivere la definizione di `is_ord`

- Si consideri ora come effettivamente viene realizzata la funzione `permutations`: lo pseudocodice è riportato in Figura 11

```
permutations(A)
  res ← ∅
  n ← length(A)
  for i ← 0 to n! − 1
    k ← i
    B ← copy of A
    for j ← 0 to n − 1
      swap B[j] and B[j + (k mod (n − j))]
      k ← k div (n − j)
    res[i] ← B
  return res
```

Figure 11: Funzione `permutations`

  - l'idea è quella di dare un ordine alle permutazioni, ovvero di stabilire come si fa a generare la prima, come si fa a generare la seconda... fino alla $n!$-esima

- si può procedere così: se ci sono n elementi, le permutazioni che hanno il primo elemento al primo posto (ovvero, il primo elemento non si sposta) saranno $(n-1)!$ (si sarà fissato un elemento, quindi restano da scegliere i restanti $n-1$...)
- lo stesso si può dire per quelle che hanno al primo posto il secondo elemento (ovvero, si scambia il primo elemento con il secondo), quelle che hanno al primo posto il terzo elemento (si scambia il terzo con il primo), fino all'ultimo
- questo per il primo posto; e per il secondo? occorre stare attenti a non scegliere nuovamente l'elemento già fissato in prima posizione
- ma questo è in realtà semplice: basta ripetere lo stesso ragionamento di prima, operando però sulla sequenza ottenuta al passo precedente (in cui si è effettuato al più un scambio) e ragionando su cosa mettere al secondo posto, a partire dal secondo elemento della sequenza già modificata
- infatti, ci saranno ora $(n-1)$ elementi da sistemare, e le permutazioni che hanno il secondo elemento al secondo posto saranno $(n-2)!$...
- in generale, all'iterazione j della perutazione i-esima, ci saranno $n-j+1$ elementi da sistemare, e le permutazioni che avranno il j-esimo elemento all'i-esimo posto saranno $(n-j)!$
- quindi, ogni permutazione viene generata facendo un certo numero di scambi (*swap*)

- La traduzione dallo pseudocodice in Figura 11 al codice Python in Figura 6 è immediata, tranne che per un punto: come si fa lo *swap* a riga 8 di Figura 11?
  - in generale, per scambiare i valori di due variabili `a` e `b`, si potrebbe fare
    ```
    a = b
    b = a
    ```
  - non funziona: perché?
  - soluzione (sostanzialmente valida per tutti i linguaggi di programmazione):
    ```
    tmp = a
    a = b
    b = tmp
    ```
  - soluzione specifica per Python:
    ```
    a, b = b, a
    ```
  - nel caso in esame, si ha quanto riportato alle righe 9 e 10 di Figura 6

- L'assegnamento nella sua forma più generale ha sintassi `<nome_var1>, ..., <nome_varn> = <espressione1>, ..., <espressionen>`, con $n \geq 1$

- La semantica di $N_1, \ldots, N_n$ = $E_1, \ldots, E_n$ è **prima si valutano** $E_1, \ldots, E_n$**; siano** $v_1, \ldots, v_n$ **i rispettivi valori. Dopodiché, si procede con l'assegnare** v_1 **ad** N_1**, . . . ,** v_n **ad** N_n