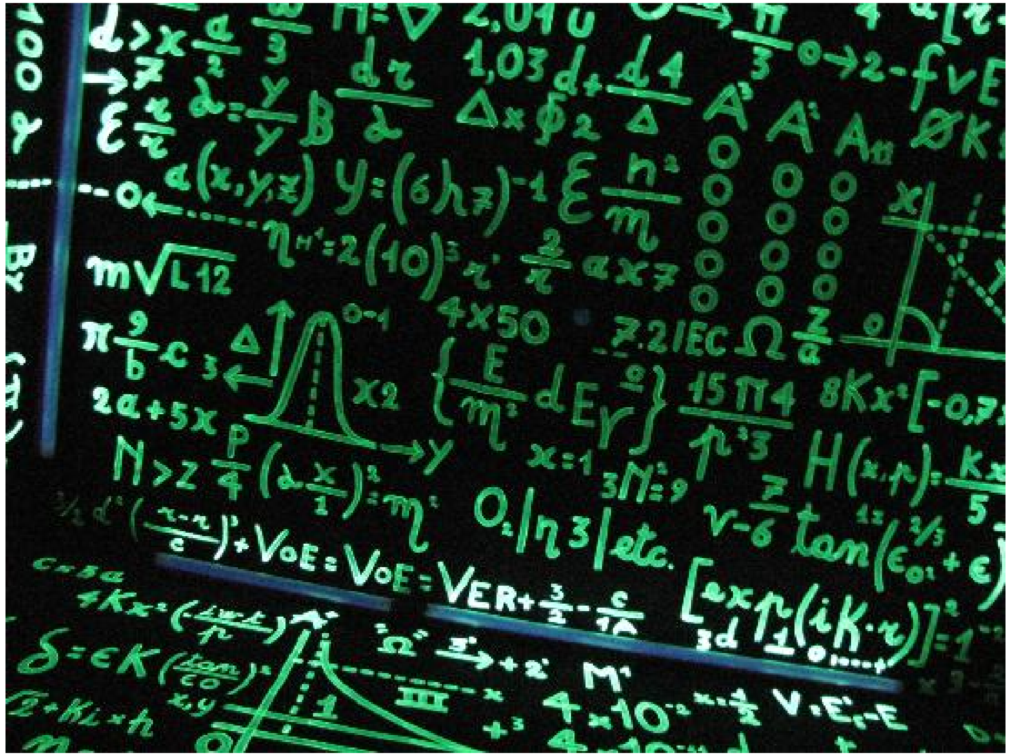
Dall'Algoritmo al Codice



Il Pensiero Computazionale

Percorso Formativo per i Docenti della Scuola Secondaria di II Grado

Lezione 1 - Parte 2 (per coder "principianti")

Stefano Forti and Davide Neri

Sommario

Risolveremo insieme i seguenti problemi in Python:

1. Un Problema dalle Olimpiadi della Matematica (2015)
2. QuickSort
3. Merge Sort

Un problema dalle Olimpiadi della Matematica (2015)

Camilla ha una scatola che contiene 2015 graffette. Ne prende un numero positivo n e le mette sul banco di Federica, sfidandola al seguente gioco. Federica ha a disposizione due tipi di mosse: può togliere 3 graffette dal mucchio che ha sul proprio banco (se il mucchio contiene almeno 3 graffette), oppure togliere metà delle graffette presenti (se il mucchio ne contiene un numero pari). Federica vince se, con una sequenza di mosse dei tipi sopra descritti, riesce a togliere tutte le graffette dal proprio banco.

(a) Per quanti dei 2015 possibili valori di n Federica può vincere?

(b) Le ragazze cambiano le regole del gioco e decidono di assegnare la vittoria a Federica nel caso riesca a lasciare sul banco una singola graffetta. Per quanti dei 2015 valori di n Federica può vincere con le nuove regole?

Lavoro di Squadra!

Provate a risolvere il problema 😄

Suggerimento

Federica può applicare al mucchio di graffette ciascuna di queste due funzioni ripetutamente:

$$f(n) = n - 3 \;\; if \; n \geqslant 3$$

$$g(n) = n/2 \;\; if \; n = 2k$$

Federica vince se rimangono 0 graffette. Ciò è possibile se e solo se, alla penultima mossa, ci sono 3 graffette sul tavolo.

Si possono costruire le soluzioni vincenti usando f^{-1} e g^{-1} a partire da un caso base vincente?

Soluzione

I numeri buoni sono dunque tutti quelli nell'intervallo $[3, 2015]$ che portano a 3 graffette residue con una certa sequenza di applicazioni di f e g.

Le possiamo generare (e contare!) semplicemente col seguente codice Python.

```
soluzione = [0] * 2016 # crea una lista con tutti zeri da 0 a 2015 (incluso)
soluzione[3] = 1

for i in range(3, 2016):
   if (soluzione[i] == 1):
      solUno = i * 2 # un numero a cui applicare g
      solDue = i + 3 # un numero a cui applicare f

      if (solUno < 2016):
         soluzione[solUno] = 1
      if (solDue < 2016):
         soluzione[solDue] = 1

print(sum(soluzione))
```

Come si risolve (b)?

```
soluzione = [0] * 2016 # crea una lista con tutti zeri da 0 a 2015 (incluso)
soluzione[1] = 1

for i in range(1, 2016):
   if (soluzione[i] == 1):
      solUno = i * 2 # un numero a cui applicare g
      solDue = i + 3 # un numero a cui applicare f
      if (solUno < 2016):
         soluzione[solUno] = 1
      if (solDue < 2016):
         soluzione[solDue] = 1

print(sum(soluzione))
```

Una soluzione per il caso generale?

Scriviamo una **funzione** Python per risolvere il caso generale, specificando il totale delle graffette `numGraffette` e le graffette che da lasciare per vincere `numVittoria`.

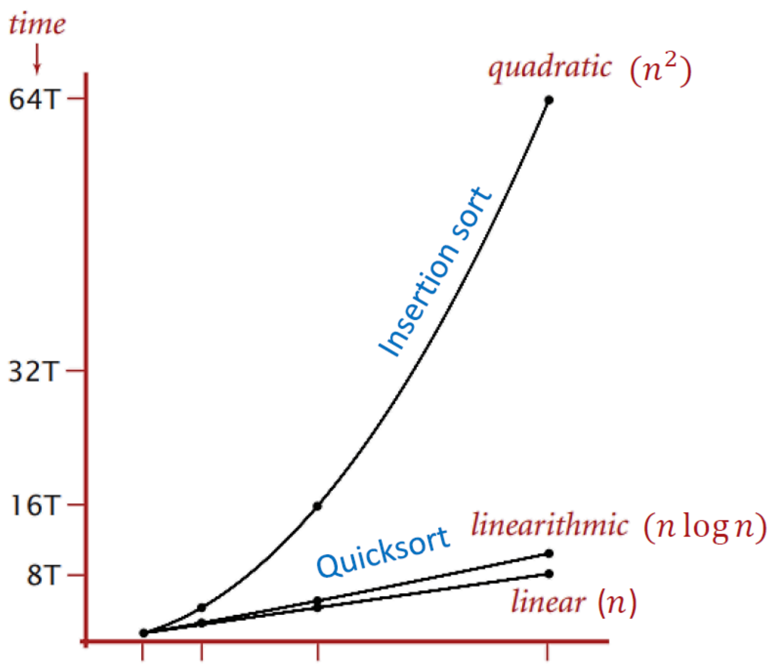
```
def trovaVittorie(numGraffette, numVittoria):
    numGraffette = numGraffette + 1
    soluzione = [0] * numGraffette
    soluzione[numVittoria] = 1

    for i in range(1, numGraffette):
        if (soluzione[i] == 1):
            solUno = i * 2 # un numero a cui applicare g
            solDue = i + 3 # un numero a cui applicare f
            if (solUno < 2015):
                soluzione[solUno] = 1
            if (solDue < 2015):
                soluzione[solDue] = 1

    return sum(soluzione)

print(trovaVittorie(2015, 3)) # soluzione di (a)
print(trovaVittorie(2015, 1)) # soluzione di (b)
```

Il Problema dell'Ordinamento

Ordinare significa riposizionare n elementi di una certa collezione secondo un dato ordine.

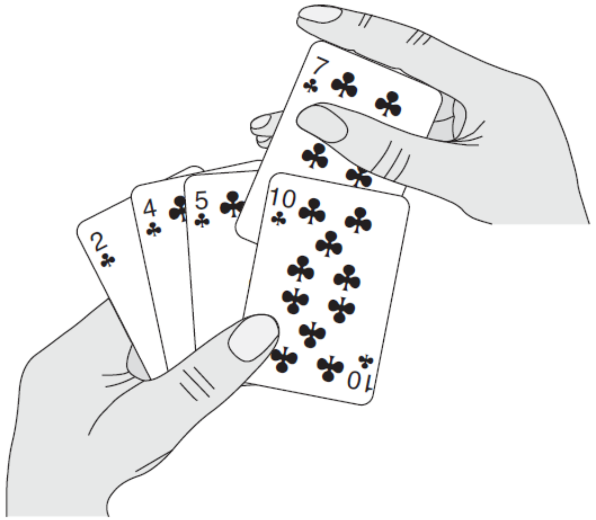
Vedremo e confronteremo due algoritmi di ordinamento:

- *Insertion Sort* (o ordinamento per inserimento) con complessità quadratica $O(n^2)$
- *Quick Sort* (o ordinamento veloce) con complessità linearitmica $O(n \lg n)$



Insertion Sort (Idea)

💡 *E' l'algoritmo con cui si riordina una mano di carte.*



Consta di tre passi:

1. Rimuovi un elemento dalla collezione.
2. Confrontalo coi successivi finché non trovi il suo posto nell'attuale configurazione.
3. Ripeti finchè non sono finiti gli elementi.

Insertion Sort (Demo)

Insert-sort with Romanian folk dance

Insertion Sort (Pseudo-codice)

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

📚 Provate voi a eseguire il codice sulla lista `[54,26,93,17,77,31,44,55,20]` e implementate la funzione Python `insertionSort(A)`.

Insertion Sort (Codice)

```
def insertionSort(A):
   for j in range(1,len(A)):

     key = A[j]
     i = j - 1

     while i >= 0 and A[i] > key: # rispetto allo pseudocodice gli indici partono da 0
         A[i+1]=A[i]
         i = i - 1

     A[i+1] = key
```

Esempio:

```
unaLista = [54,26,93,17,77,31,44,55,20]
insertionSort(unaLista)
print(unaLista)
```

Prendere il tempo!

⌛ In Python possiamo cronometrare la durata di esecuzione di un programma. Basta importare la libreria `time` e usare la funzione `time.time()` come nell'esempio qui sotto:

```
import random as rnd
import time

unaLista = []

# generiamo una lista di 10000 interi a caso
for i in range(10000):
   unaLista.append(rnd.randint(1,100000))

start = time.time() # segna il tempo di inizio nella variabile start
insertionSort(unaLista)
stop = time.time() # segna il tempo di fine nella variabile start
print('Insertion sort per', stop-start, "secondi.")
```

Quick Sort (Idea)

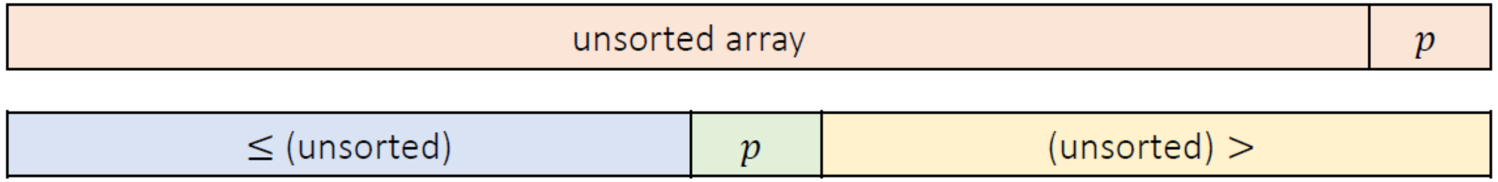
Quick Sort è tra gli algoritmi più usati per l'ordinamento.

Utilizza un approccio *divide et impera*.

💡 *L'algoritmo divide la collezione in due parti, poi le ordina indipendentemente.*

L'idea di base segue due passi:

1. Sceglie un elemento p (*pivot*) nella collezione e la divide in due sotto-collezioni, una con gli elementi $e \leqslant p$, l'altra con gli elementi $e > p$. Complessità: $O(n)$.



2. Ripete (1) sulle due metà.

Quick Sort (Demo)

Quick-sort with Hungarian folk dance

Quick Sort (Pseudo-codice)

Quick Sort

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

Partizionamento lineare

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Quick Sort (Codice)

```
def quickSort(alist):
   quickSortHelper(alist,0,len(alist)-1)

def quickSortHelper(alist,first,last):
   if first<last:
       splitpoint = partition(alist,first,last)
       quickSortHelper(alist,first,splitpoint-1)
       quickSortHelper(alist,splitpoint+1,last)
```

```python
def partition(alist,first,last):
   pivotvalue = alist[first]
   leftmark = first+1
   rightmark = last
   done = False

   while not done:
       while leftmark <= rightmark and alist[leftmark] <= pivotvalue:
           leftmark = leftmark + 1
       while alist[rightmark] >= pivotvalue and rightmark >= leftmark:
           rightmark = rightmark -1
       if rightmark < leftmark:
           done = True
       else:
           temp = alist[leftmark]
           alist[leftmark] = alist[rightmark]
           alist[rightmark] = temp

   temp = alist[first]
   alist[first] = alist[rightmark]
   alist[rightmark] = temp

   return rightmark
```

Quick Sort (Esempio)

```
alist = [54,26,93,17,77,31,44,55,20]
quickSort(alist)
print(alist)
```

Merge Sort

La funzione di libreria `sorted()` di Python implementa una variante del Merge Sort visto in classe.

Provate a prendere i tempi di questa variante del Merge Sort!

```
unaLista = []
for i in range(10000):
   unaLista.append(rnd.randint(1,100000))

start = time.time() # segna il tempo di inizio nella variabile start
sorted(unaLista)
stop = time.time() # segna il tempo di fine nella variabile start
print('Merge Sort per', stop-start, "secondi.")
```

Prendiamo i tempi!

```python
import random as rnd
import time

unaLista = []
for i in range(10000):
   unaLista.append(rnd.randint(1,100000))

start = time.time() # segna il tempo di inizio nella variabile start
insertionSort(unaLista)
stop = time.time() # segna il tempo di fine nella variabile start
print('Insertion Sort per', stop-start, "secondi.")

unaLista = []
for i in range(10000):
   unaLista.append(rnd.randint(1,100000))

start = time.time() # segna il tempo di inizio nella variabile start
quickSort(unaLista)
stop = time.time() # segna il tempo di fine nella variabile start
print('Quick Sort per', stop-start, "secondi.")
```

Prendere il tempo!

Tempo di esecuzione dei tre algoritmi di ordinamento con `20000` elementi sulla "nostra" macchina.

```
Insertion sort per 33.475194454193115 secondi.
Quick Sort per 0.09186649322509766 secondi.
Merge Sort per 0.007998466491699219 secondi.
```

Esercizio

Si può ordinare in tempo lineare $O(n)$ conoscendo l'intervallo $[0, M]$ in cui si trovano i numeri da ordinare?

La morale è ancora quella...

> **Algoritmi efficienti** sono da considerarsi migliori di **computer potenti**.
> O, ancora, l'accoppiata vincente è **algoritmi efficienti** su **computer potenti**.

	insertion sort (n^2)			mergesort (n log n)			quicksort (n log n)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Esercizi

1. Data una stringa `s`, come si può stabilire se è palindroma? Scrivere lo pseudocodice e il codice della funzione `def palindromo(s):` che restituisce `True` se la parola è palindroma e `False` altrimenti.
2. Date due stringhe `A` e `B`, come si può stabilire se una è l'anagramma dell'altra? Esistono almeno quattro soluzioni rispettivamente di complessità esponenziale $O(2^n)$, quadratica $O(n^2)$, linearitmica $O(n \lg n)$ e lineare $O(n)$. Scrivere lo pseudocodice di almeno due soluzioni e implementare la più efficiente.
3. Data una lista D di n interi (positivi e negativi), come si può stabilire la sottolista di somma massima, i.e. come possiamo decidere a e b tali da ottenere il massimo $\max\limits_{a,b \in \mathbb{N}_n} \{\sum_{i=a}^{b} D[i]\}$. Esistono almeno tre soluzioni rispettivamente di complessità cubica $O(n^3)$, quadratica $O(n^2)$ e lineare $O(n)$. Scrivere lo pseudocodice di almeno due soluzioni e implementare la più efficiente.