Complessità e algoritmi

In questa lezione:

- La complessità degli algoritmi
- L'ordinamento per selezione (SelectionSort)
- Profiling del SelectionSort
- Analisi del SelectionSort
- La notazione O-grande
- Il MergeSort
- Profiling del MergeSort
- Analisi del MergeSort

La complessità degli algoritmi

Motivazione

L'analisi delle prestazioni degli algoritmi è generalmente basata sul numero di istruzioni di codice macchina che vengono eseguite per risolvere un problema.
Nel caso di un programma per la macchina URM queste possono essere seplicemente contate in funzione dell'ingresso.
Per calcolare $f(x, y) = x + y$ dalla configurazione iniziale dei registri $x, y, 0, 0, \ldots$, il programma aggiunge 1 a r_0 e r_2 y volte:

$$\begin{array}{ll} I_1 & J(2,1,5) \\ I_2 & S(0) \\ I_3 & S(2) \\ I_4 & J(0,0,1) \end{array}$$

Il calcolo terminerà quando $r_2 = r_1$, lasciando $x + y$ in R_0.
Al termine saranno eseguite esattamente $4y + 1$ istruzioni.

La complessità degli algoritmi

Motivazione

In generale però non possiamo sapere esattamente quali sono le istruzioni di linguaggio macchina che corrispondono ad una istruzione di un linguaggio ad alto livello.

Come fare per misurare le prestazioni di un algoritmo?

Un approccio è lanciare il relativo programma e misurare il tempo trascorso dall'inizio dell'esecuzione alla fine. Parleremo in questo caso di misurazione delle prestazioni (profiling).
Un altro approccio è quello di misurare le visite alle variabili (lettura o scrittura) o altre operazioni basilari come l'incremento di indici, le operazioni aritmetiche, o il confronto tra valori. Tutte queste operazioni richiedono più o meno la stessa quantità di lavoro a livello di linguaggio macchina. Parleremo in questo caso di analisi delle prestazioni.

Applicheremo questi approcci a due algoritmi di ordinamento.

L'ordinamento per selezione (SelectionSort)

Un algoritmo di ordinamento sposta gli elementi di una lista di dati in modo tale che al termine siano memorizzati in qualche ordine specifico.

Ci poniamo il problema di ordinare in modo crescente una lista di interi.

Il SelectionSort *considera la parte di lista non ordinata, seleziona l'elemento minimo e lo mette nella posizione di testa*.

11	9	17	5	12
0	1	2	3	4

All'inizio la "parte di lista non ordinata" è tutta la lista e l'elemento minimo è 5.
Questo viene messo al posto di 11 e l'11 viene messo al posto del 5: si scambiano le posizioni.

L'ordinamento per selezione (SelectionSort)

Dopo lo scambio di 5 e 11:

5	9	17	11	12
0	1	2	3	4

Il 9 è nella posizione giusta:

5	9	17	11	12
0	1	2	3	4

L'11 e il 17 si scambiano posto:

5	9	11	17	12
0	1	2	3	4

Il 12 e il 17 si scambiano posto:

5	9	11	12	17
0	1	2	3	4

L'ordinamento per selezione (SelectionSort)

Codice Python – file: selectionsort.py

```python
## Sorts a list, using selection sort.
#  @param values the list to sort
#
def selectionSort(values) :
   for i in range(len(values)) :
      minPos = minimumPosition(values, i)
      temp = values[minPos]  # swap the two elements
      values[minPos] = values[i]
      values[i] = temp

## Finds the smallest element in a tail range of the list.
#  @param values the list to sort
#  @param start the first position in values to compare
#  @return the position of the smallest element in the
#  range values[start] . . . values[len(values) - 1]
#
def minimumPosition(values, start) :
   minPos = start
   for i in range(start + 1, len(values)) :
      if values[i] < values[minPos] :
         minPos = i

   return minPos
```

L'ordinamento per selezione (SelectionSort)

Codice Python – file: selectiondemo.py

Esempio di uso della procedura selectionSort:

```
##
#  This program demonstrates the selection sort algorithm by sorting a
#  list that is filled with random numbers.

from random import randint
from selectionsort import selectionSort

n = 20
values = []
for i in range(n) :
   values.append(randint(1, 100))

print(values)
selectionSort(values)
print(values)
```

Output (esempio):

[65, 46, 14, 52, 38, 2, 96, 39, 14, 33, 13, 4, 24, 99, 89, 77, 73, 87, 36, 81]

[2, 4, 13, 14, 14, 24, 33, 36, 38, 39, 46, 52, 65, 73, 77, 81, 87, 89, 96, 99]

Profiling del SelectionSort

Per eseguire il profiling del SelectionSort potremmo utilizzare un cronometro, ma in realtà vogliamo misurare il tempo che l'algoritmo impiega per essere eseguito

- escludendo il tempo per caricare in memoria il programma;
- escludendo il tempo per stampare l'output.

Per misurare il tempo di esecuzione:

- usiamo la funzione di libreria `time()` dal modulo `time`.
- Questa restituisce il numero di secondi trascorsi dalla mezzanotte del 1 gennaio 1970.
- La differenza tra i valori restituiti da due chiamate alla funzione dà il tempo trascorso.

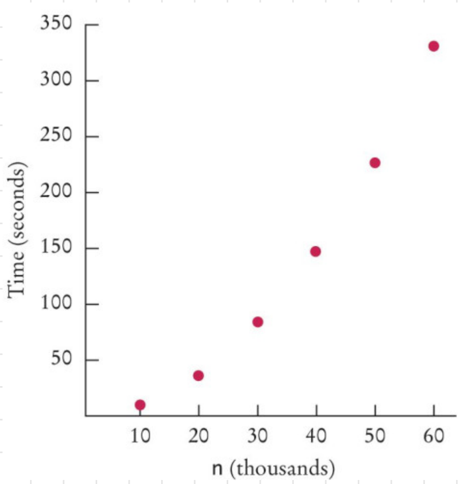
Profiling del SelectionSort

Codice Python – file: selectiontimer.py

```
##
#  This program measures how long it takes to sort a list of a
#  user-specified size with the selection sort algorithm.
#

from random import randint
from selectionsort import selectionSort
from time import time

# Prompt the user for the list size.
n = int(input("Enter list size: "))

# Construct random list.
values = []
for i in range(n) :
   values.append(randint(1, 100))

startTime = time()
selectionSort(values)
endTime = time()

print("Elapsed time: %.3f seconds" % (endTime - startTime))
```


Profiling del SelectionSort

Di seguito, il tempo impiegato da SelectionSort in funzione del numero degli elementi *n* nella lista.



List size (n)	Seconds
10,000	9
20,000	38
30,000	85
40,000	147
50,000	228
60,000	332

I tempi dipendono anche dalla versione del linguaggio, la velocità del processore e dal sistema operativo utilizzato, ma l'andamento rimane invariato.

Analisi del SelectionSort

Il grafico che misura le prestazioni del SelectionSort mostra un andamento che ricorda una parabola. Possiamo fare una analisi per dimostrarlo?
Possiamo contare il numero di *visite* agli elementi della lista.
La prima volta, per trovare l'elemento minimo visitiamo n elementi.
La seconda visitiamo $n-1$ elementi
La terza $n-2$ elementi
Ci fermiamo quando restano 2 elementi.
Quindi le visite per la ricerca dei minimi sono:

$$n+(n-1)+(n-2)+\ldots+3+2=\sum_{i=2}^{n} i=\frac{n(n+1)}{2}-1$$

Inoltre, per ogni volta facciamo 2 visite per scambiare gli elementi, per un totale di $2(n-1)$ visite. Di conseguenza in tutto le visite sono:

$$\frac{n(n+1)}{2}-1+2(n-1)=\frac{1}{2}n^2+\frac{5}{2}n-3$$

Analisi del SelectionSort

Quindi l'andamento è quello di una parabola. Semplificando ancora osserviamo che il contributo maggiore al crescere di n lo dà il termine $\frac{1}{2}n^2$.
Infatti questo per $n=2000$ vale 2000000 mentre gli altri termini contribuiscono per 4997: un valore insignificante rispetto a 2000000.
Il numero delle visite cresce, con ottima approssimazione, come $\frac{1}{2}n^2$.
Cosa succede se raddoppiamo la lunghezza della lista, per esempio, da $n=1000$ a $n=2000$? Facendo il rapporto del numero delle visite otteniamo :

$$\left(\frac{\frac{1}{2}\cdot 2000^2}{\frac{1}{2}\cdot 1000^2}\right)=4$$

Quindi, al raddoppio della lunghezza degli elementi le visite quadruplicano e per una lunghezza tripla le visite diventano nove volte maggiori.
Il coefficiente $\frac{1}{2}$ si semplifica sempre: dunque, possiamo dire che le visite sono "dell'ordine di n^2".

La notazione O-grande

Per il SelectionSort, il tempo $T(n)$ per elaborare una lista di n elementi, è proporzionale al numero di visite, cioè $T(n) = c \cdot \left(\frac{1}{2}n^2 + \frac{5}{2}n - 3\right)$, dove c rappresenta il tempo per eseguire una visita.

Possiamo quindi dire che anche il tempo $T(n)$ è "dell'ordine di n^2" e scriviamo $O(n^2)$.

In generale, la funzione $T(n)$ *rappresenta il tempo di elaborazione richiesto da un algoritmo per risolvere un determinato problema di dimensione* n.

$T(n)$ può rappresentare il tempo medio di esecuzione, ma generalmente indica il tempo per il caso peggiore (analisi worst-case).

La notazione O-grande

Data un'altra funzione $f(n)$ (solitamente "semplice") scriviamo:

$$T(n) = O(f(n))$$

se
$T(n)$ cresce al crescere di n con andamento limitato superiormente da $f(n)$

Nel caso del SelectionSort abbiamo usato come $f(n) = n^2$.

Cosa vuol dire esattamente "andamento limitato superiormente da $f(n)$"?

La notazione O-grande

Formalmente:

$T(n) = O(f(n))$ ($T(n)$ è un O-grande di $f(n)$) se per ogni valore di n superiore ad una certa soglia

$$T(n) \leq C \cdot f(n)$$

per qualche valore costante C.

Si può facilmente dimostrare che se $T(n)$ è un polinomio di grado k, allora

$$T(n) = O(n^k)$$

La notazione O-grande

La tabella seguente mostra espressioni O-grande frequentemente utilizzate:

Espressione	**Nome**	**Esempio**
$O(1)$	Costante	Accesso ad un elemento di una lista
$O(\log(n))$	Logaritmica	Ricerca di un elemento in una lista ordinata
$O(n)$	Lineare	Ricerca elemento massimo in una lista
$O(n \log(n))$	Log-lineare	MergeSort
$O(n^2)$	Quadratica	SelectionSort
$O(n^3)$	Cubica	Moltiplicazione ingenua di matrici $n \times n$
$O(2^n)$	Esponenziale	Alg. per problema del commesso viaggiatore
$O(n!)$	Fattoriale	Alg. "forza bruta" per commesso viaggiatore

La notazione Ω

L'espressine $T(n) = O(f(n))$ significa che *T non cresce più velocemente di f*, ma è possibile che T cresca molto più lentamente!

Se $T(n) = 5n^2 - n + 7$ è corretto dire che $T(n)$ è $O(n^2)$, ma anche $O(n^3)$ o $O(n^{10})$. Per dire che T cresce almeno quanto f si usa l'espressione:

$$T(n) = \Omega(f(n))$$

$T(n) = \Omega(f(n))$ ($T(n)$ è Omega-grande di $f(n)$) se per ogni valore di n superiore ad una certa soglia

$$T(n) \geq C \cdot f(n)$$

per qualche valore costante C.

Ad esempio $T(n) = 5n^2 - n + 7$ è $\Omega(n^2)$, ma anche $\Omega(n)$

La notazione Θ

Per dire che T ed f crescono con la stessa velocità si usa l'espressione:

$$T(n) = \Theta(f(n))$$

$T(n) = \Theta(f(n))$ ($T(n)$ è Theta di $f(n)$) se valgono contemporaneamente:

$$T(n) = O(f(n)) \text{ e } T(n) = \Omega(f(n))$$

Ad esempio $T(n) = 5n^2 - n + 7$ è $\Theta(n^2)$, ma non $\Theta(n)$ o $\Theta(n^3)$.

Le notazioni Θ e Ω sono importanti per un'analisi accurata degli algoritmi, ma è pratica comune usare solo la notazione O-grande con la stima più precisa possibile.

L'ordinamento per fusione (MergeSort)

Si può dimostrare che ogni algoritmo di ordinamento è $\Omega(n \log n)$. Esistono algoritmi di ordinamento più efficienti del SelectionSort, che è $O(n^2)$?

Sì, tra questi il MergeSort e il QuickSort, entrambi $O(n \log n)$, quindi ottimi.

Analizziamo il MergeSort che si basa sulla seguente idea.

Supponiamo di avere le due metà della lista già ordinate allora è sufficiente fondere (da cui il nome ordinamento per fusione, *merge*) le due metà in un'unica lista ordinata.

Se le due metà non sono ordinate si applica ricorsivamente l'algoritmo a ciascuna metà fino ad arrivare a liste di un elemento o nessun elemento, chiaramente già ordinate.

Il MergeSort

fusione

Divisione di una lista in due metà ordinate:

5	9	10	12	17	1	8	11	20	32

Fusione delle due metà:

5	9	10	12	17	1	8	11	20	32	1									
5	9	10	12	17	1	8	11	20	32	1	5								
5	9	10	12	17	1	8	11	20	32	1	5	8							
5	9	10	12	17	1	8	11	20	32	1	5	8	9						
5	9	10	12	17	1	8	11	20	32	1	5	8	9	10					
5	9	10	12	17	1	8	11	20	32	1	5	8	9	10	11				
5	9	10	12	17	1	8	11	20	32	1	5	8	9	10	11	12			
5	9	10	12	17	1	8	11	20	32	1	5	8	9	10	11	12	17		
5	9	10	12	17	1	8	11	20	32	1	5	8	9	10	11	12	17	20	
5	9	10	12	17	1	8	11	20	32	1	5	8	9	10	11	12	17	20	32

Il MergeSort

Codice Python – file: mergesort.py

Vediamo il codice Python per l'algoritmo MergeSort assumendo l'esistenza di una procedura `mergeLists()` che fonde due liste ordinate.

```
## Sorts a list, using merge sort.
#  @param values the list to sort
#
def mergeSort(values) :
   if len(values) <= 1 : return
   mid = len(values) // 2
   first = values[ : mid]
   second = values[mid : ]
   mergeSort(first)
   mergeSort(second)
   mergeLists(first, second, values)
```


La procedura mergeLists()

Codice Python – file: mergesort.py

```
def mergeLists(first, second, values) :
   iFirst = 0   # Next element to consider in the first list.
   iSecond = 0  # Next element to consider in the second list
   j = 0        # Next open position in values.

   while iFirst < len(first) and iSecond < len(second) :
      if first[iFirst] < second[iSecond] :
         values[j] = first[iFirst]
         iFirst = iFirst + 1
      else :
         values[j] = second[iSecond]
         iSecond = iSecond + 1
      j = j + 1

   while iFirst < len(first) :  # Copy any remaining entries of first.
      values[j] = first[iFirst]
      iFirst = iFirst + 1
      j = j + 1

   while iSecond < len(second) : # Copy any remaining entries of second.
      values[j] = second[iSecond]
      iSecond = iSecond + 1
      j = j + 1
```

Profiling del MergeSort

Codice Python – file: mergetimer.py

Analogamente al SelectionSort, utilizziamo il seguente programma per il profiling del MergeSort.

```python
## This program measures how long it takes to sort a list of a
#  user-specified size with the merge sort algorithm.

from random import randint
from mergesort import mergeSort
from time import time

# Prompt the user for the list size.
n = int(input("Enter list size: "))

# Construct random list.
values = []
for i in range(n) :
   values.append(randint(1, 100))

startTime = time()
mergeSort(values)
endTime = time()

print("Elapsed time: %.3f seconds" % (endTime - startTime))
```


Profiling del MergeSort

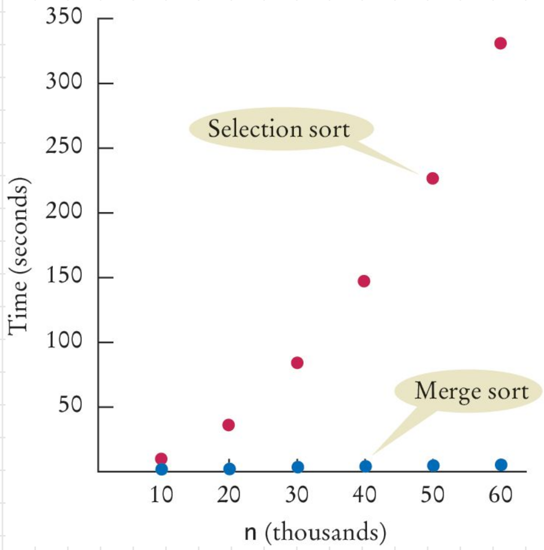
Confronto tra tempi impiegati da MergeSort e SelectionSort.

List size (n)	Merge Sort (seconds)	Selection Sort (seconds)
10,000	0.105	9
20,000	0.223	38
30,000	0.344	85
40,000	0.470	147
50,000	0.599	228
60,000	0.729	332

Notare come i tempi del MergeSort si mantengono sotto il secondo, mentre il SelectionSort richiede 5 minuti e mezzo per ordinare una lista di 60000 elementi.

Profiling del MergeSort

Confronto tra tempi impiegati da MergeSort e SelectionSort.



Analisi del MergeSort

Il grafico mostra che le prestazioni del MergeSort sono migliori del SelectionSort. Possiamo dimostrarlo?
Contiamo il numero di *visite* agli n elementi della lista assumendo $n = 2^m$, cioè che sia una potenza di 2.
Analizziamo le visite del processo di fusione.
Bisogna aggiungere un elemento alla lista `values` da `first` o da `second`, facendo un confronto. Quindi abbiamo una scrittura su `values`, una lettura da `first` e una da `second`. In tutto $3n$ visite.
Prima di poter fare la fusione dobbiamo copiare gli elementi in `first` e `second`, in tutto $2n$ visite.
Il totale delle visite per la fusione è $5n$.
Quindi il tempo $T(n)$ per l'algoritmo MergeSort è:

$$T(n) = T(n/2) + T(n/2) + 5n = 2T(n/2) + 5n$$

Analisi del MergeSort

Quanto vale $T(n) = 2T(n/2) + 5n$?

Ricorsivamente:

$$T(n/2) = 2T(n/4) + 5n/2$$

Quindi $T(n) = 2 \times 2T(n/4) + 5n + 5n$

Ricorsivamente:

$$T(n/4) = 2T(n/8) + 5n/4$$

Quindi $T(n) = 2 \times 2 \times 2T(n/8) + 5n + 5n + 5n$

Generalizzando:

$$T(n) = 2^k T(n/2^k) + 5nk$$

Analisi del MergeSort

Avendo ipotizzato $n = 2^k$, per $k = m$ si ha:

$$\begin{aligned} T(n) &= 2^k T(n/2^k) + 5nk \\ &= 2^m T(2^m/2^m) + 5nm \\ &= nT(1) + 5n\log_2 n \\ &= n + 5n\log_2 n \end{aligned}$$

Il termine che cresce di più è $5n\log_2 n$ ma possiamo trascurare il fattore 5. Inoltre, la base del logaritmo può essere trascurata perché i logaritmi sono correlati tramite un fattore costante. Es:

$$\log_2 n = \log_{10} n / \log_{10} 2 \approx 3.32193 \log_{10} n$$

Quindi

$$T(n) = O(n\log n)$$

Analisi della Complessità di un problema

Utilizzando la teoria della complessità degli algoritmi, si può studiare la complessità di un problema.

In particolare possiamo dire che un problema ha complessità $O(f(h))$ se questa è la complessità del miglior algoritmo che risolve il problema.

Diciamo che un problema ha complessità $\Omega(f(h))$ se dimostriamo che nessun algoritmo può risolvere il problema in tempo inferiore a $\Omega(f(h))$.