Laboratorio di Algoritmi e Strutture Dati

Laboratorio, esercizio 1

"THE PROBABILITY OF SOMEONE WATCHING YOU IS PROPORTIONAL TO THE STUPIDITY OF YOUR ACTIONS."

Programma concettuale e videolezioni

	video	argomento
	1	Intuizione algoritmica
	2	Correttezza e complessitá algoritmi iterativi
	3	Correttezza e complessitá algoritmi ricorsivi
	4	Notazione asintotica
	5	Ricorrenze e loro soluzioni
	6	Esercizi su correttezza, complessitá, e ricorrenze
array	7	*MergeSort*
	8	Laboratorio: esercizio 0 (introduzione al laboratorio)
	9	Laboratorio: esercizio 1
	10	*QuickSort*
	11	*CountingSort* e *RadixSort*
	12	Esercizi sull'ordinamento

Verso l'efficienza

Quando un'implementazione è **efficiente**? Assumendo che il codice scritto sia corretto e perfettamente funzionante, gli elementi che contribuiscono all'efficienza di una soluzione sono almeno due:

- Un'idea algoritmica corretta, completa, terminante, e asintoticamente efficiente (quando possibile, ottima), e
- Un'implementazione che sfrutti le caratteristiche del linguaggio, delle strutture dati, e ottimizzazioni locali che non influenzano il comportamento asintotico (almeno non in generale).

Gli esercizi di laaboratorio tendono a formarci con la vista al secondo punto.

Primo esercizio

In questo primo esercizio vogliamo costruire un nuovo algoritmo di ordinamento basato sui confronti. L'obbiettivo è quello di sfruttare i punti forti di due algoritmi che abbiamo visto: *InsertionSort* e *MergeSort*. Il risultato sará un algoritmo misto, per così dire, che si comporti meglio, dal punto di vista sperimentale, di entrambi.

Primo esercizio: idea

Da un punto vista analitico, il tempo di esecuzione di *MergeSort* è $\Theta(n \cdot lg(n))$ (in tutti i casi), e quello di *InsertionSort* è $\Theta(n^2)$ (nel caso peggiore). Prendiamo unicamente il limite superiore nel caso peggiore. Abbiamo, per *MergeSort*:

$$T(n) \leq c_1 \cdot n \cdot lg(n) \Rightarrow O(n \cdot lg(n))$$

e per *InsertionSort*:

$$T(n) \leq c_2 \cdot n^2 \Rightarrow O(n^2)$$

Primo esercizio: idea

Prima del punto di incontro, InsertionSort è meglio

Possiamo mostrare che, per n **sufficientemente piccolo**, succede che:

$$\boxed{c_2 \cdot n^2 < c_1 \cdot n \cdot lg(n).}$$

Questa disequazione è vera per tutti i valori di n **piú piccoli** di un certo valore k che bisogna trovare. Dimostrarlo formalmente richiede, primo, trovare le costanti c_1, c_2 (che dipendono dall'implementazione e dalla macchina), e, poi, risolvere un'equazione esponenziale, che implica, a sua volta, usare la funzione W di Lambert. Noi, invece, cercheremo di mostrarlo in maniera sperimentale.

Primo esercizio (prima parte)

Esercizio 1 (prima parte)

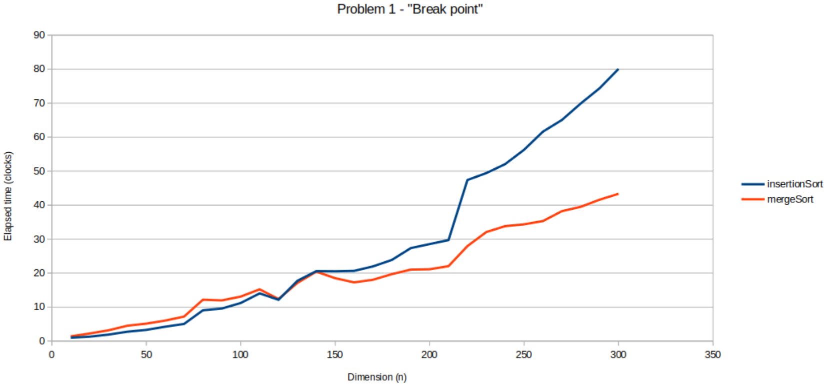
Realizzare un esperimento implementando sia InsertionSort *che* MergeSort, *misurando il tempo di esecuzione sperimentale per input di lunghezza crescente, con multiple ripetizioni per la stessa lunghezza, su entrambi gli algoritmi. Dedurre il valore di k (punto massimo di incrocio tra le curve) per la propria macchina e implementazione.*

Il risultato richesto prevede:

- Una rappresentazione grafica delle curve di tempo.
- L'identificazione del valore massimo k oltre il quale *MergeSort* è sempre piú efficiente di *InsertionSort*.

Primo esercizio (prima parte)

Un esempio di possibile risultato è:



Primo esercizio: una versione piú efficiente di *MergeSort*

Come usiamo questa informazione per costruire una versione piú efficiente (sperimentalmente) di *MergeSort*? Poichè l'idea dell'algoritmo è quella di ordinare, ricorsivamente, array sempre piú piccoli (i pezzi dell'array originale), ad un certo punto, durante la ricorsione, si arriverá ad avere un array di dimensione inferiore a k. In quel momento non è piú conveniente richiamare *MergeSort*, ma conviene chiamare *InsertionSort*.

proc ***HybridSort*** (A, p, r)

$$
\begin{cases}
\textbf{if } (r - p + 1 > k) \\
\quad \textbf{then} \\
\quad \begin{cases}
q = [(p + r)/2] \\
\mathit{HybridSort}(A, p, q) \\
\mathit{HybridSort}(A, q + 1, r) \\
\mathit{Merge}(A, p, q, r)
\end{cases} \\
\quad \textbf{else} \quad \mathit{InsertionSort}(A, p, r)
\end{cases}
$$

Primo esercizio (seconda parte)

Esercizio 1 (seconda parte)

Realizzare un esperimento implementando sia MergeSort *che* HybridSort*, quest'ultimo implementato utilizzando la costante k precedentemente trovata, misurando il tempo di esecuzione sperimentale per input di lunghezza crescente, con multiple ripetizioni per la stessa lunghezza, su entrambi gli algoritmi.*

Il risultato richesto prevede:

- Una rappresentazione grafica delle curve di tempo, dove si vede che *HybridSort* migliora sempre il tempo di esecuzione di *MergeSort*;
- Una dimostrazione sperimentale di correttezza attraverso test randomizzati e funzioni antagoniste.

Primo esercizio (seconda parte)

Una bozza dello pseudo-codice della soluzione:

proc Experiment ()

$$
\begin{cases}
\dots \\
\textbf{for } (experiment = 1 \textbf{ to } max\_experiment) \\
t\_tot\_merge = 0 \\
t\_tot\_hybrid = 0 \\
\textbf{for } (instance = 1 \textbf{ to } max\_instances) \\
\begin{cases}
A = GenerateRandom(current\_length) \\
B = Copy(A) \\
t\_start = clock() \\
MergeSort(A) \\
t\_end = clock() \\
t\_elapsed = t\_end - t\_start \\
t\_tot\_merge = t\_tot\_merge + t\_elapsed \\
t\_start = clock() \\
HybridSort(B) \\
t\_end = clock() \\
t\_elapsed = t\_end - t\_start \\
t\_tot\_hybrid = t\_tot\_hybrid + t\_elapsed
\end{cases} \\
t\_final = t\_tot/max\_instances \\
current\_lenght = current\_length + step
\end{cases}
$$

Laboratorio: esercizio richiesto (seconda parte)

Da un punto di vista asintotico, la soluzione proposta **non** è piú efficiente di *MergeSort*:

$$T(n) = \begin{cases} 2 \cdot T(\frac{n}{2}) + \Theta(n) & \text{se } n > k \\ k^2 & \text{altrimenti} \end{cases}$$

Si puó utilizzare il metodo dello sviluppo per convincersi che è ancora vero che:

$$T(n) = \Theta(n \cdot lg(n))$$

Laboratorio: esercizio richiesto (seconda parte)

Da un punto di vista sperimentale, invece, ci aspettiamo un risultato del genere (in questo esempio, $k = 120$):