Ordinamento

Introduzione

- L'ordinamento di dati è una primitiva fondamentale utilizzata praticamente ovunque, in applicazioni scientifiche e commerciali.
- La sua importanza ha motivato un'intensa attività di ricerca per più di mezzo secolo che ne ha studiato a fondo gli aspetti computazionali sviluppando un gran numero di soluzioni algoritmiche efficienti in contesti generali e non.
- Noi (ri)vedremo:
  1. **Algoritmi basati su confronti:** MergeSort, QuickSort (design pattern *divide-and-conquer*), InsertionSort, e un lower bound che si applica a qualsiasi algoritmo basato su confronti.
  2. **Algoritmi non-basati su confronti:** BucketSort, RadixSort.

Ordinamento basato su confronti

Specifica Input-Output

Input: sequenza $S = S[0]S[1] \ldots S[n-1]$ di $n \geq 0$ chiavi da un universo ordinato

Output: sequenza S ordinata in senso crescente

Definizione

Algoritmo di ordinamento basato su confronti (comparison-based):
Per ogni istanza di input S l'ordinamento delle chiavi è determinato ESCLUSIVAMENTE da confronti tra coppie di chiavi $S[i]$ e $S[j]$.

N.B.: Rappresentiamo S come array, assumendo di avere accesso diretto a ciascun $S[i]$. Gli algoritmi che presenteremo possono anche essere implementati su liste, ma risulterebbero meno efficienti in pratica.

Design Pattern: Divide-and-conquer

Un algoritmo *divide-and-conquer* per un problema computazionale Π è un algoritmo ricorsivo che risolve ogni istanza i di Π di taglia n in 3 fasi:

1. **Divide:**
   - Se $n \leq n_0 \Rightarrow$ risolvi i direttamente
   - Se $n > n_0 \Rightarrow$ suddividi i in 2 o più istanze di Π, $i_1, i_2, \ldots$, ciascuna taglia $< n$
2. **Recur:** Risolvi ricorsivamente ciascuna istanza i_j, con $j \geq 1$
3. **Conquer:** Determina la soluzione di i combinando opportunamente le soluzioni delle istanze i_j, con $j \geq 1$.

Osservazioni

- Il valore n_0 definisce il caso base dell'algoritmo
- La locuzione *divide-and-conquer* (o *divide-et-impera* in latino) fa riferimeno alla tecnica politica di frammentare l'opposizione per tenerla sotto controllo. L'uso della locuzione pare risalire a Filippo il Macedone (382-336 a.C.).

MergeSort

Strategia Divide-and-Conquer: caso base $n_0 = 1$

1. **Divide:**
   - $S_1 \leftarrow$ sottosequenza con la prima metà di S
   - $S_2 \leftarrow$ sottosequenza con la seconda metà di S
2. **Recur**: ordinamento ricorsivo delle due sottosequenze
   - `MergeSort(`S_1`)`
   - `MergeSort(`S_2`)`
3. **Conquer:** $S \leftarrow$ "fusione" di S_1 e S_2 ordinate

MergeSort (continua)

Algoritmo `MergeSort(`S`)`

Input: come da specifica generale
Output: come da specifica generale

if $n \leq 1$ **then return** S;
/* **Divide** */
$S_1 \leftarrow S[0 \div \lceil n/2 \rceil - 1]$; // $\lceil n/2 \rceil$ chiavi
$S_2 \leftarrow S[\lceil n/2 \rceil \div n - 1]$; // $\lfloor n/2 \rfloor$ chiavi;
/* **Recur** */
`MergeSort(`S_1`);`
`MergeSort(`S_2`);`
/* **Conquer** */
`Merge(`S_1`,`S_2`,`S`);`
return S;

Osservazione: `Merge(`S_1`,`S_2`,`S`)` fonde le due sequenze ordinate S_1 e S_2 in un'unica sequenza ordinata S che usa lo stesso spazio dalla sequenza di input, dato che le chiavi sono state copiate in S_1 e S_2. (Lo pseudocodice, riportato nel prossimo lucido, può essere sostituito direttamente al posto dell'invocazione `Merge(`S_1`,`S_2`,`S`)`)

Algoritmo `Merge`

Algoritmo `Merge`(S_1,S_2,S)

Input: sequenze S_1,S_2 ordinate in senso crescente, sequenza vuota S
Output: sequenza $S = S_1 \cup S_2$ ordinata in senso crescente

$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$;
while (($i < S_1$.size()) and ($j < S_2$.size())) **do**
  if $S_1[i] \leq S_2[j]$ **then** $S[k++] \leftarrow S_1[i++]$;
  else $S[k++] \leftarrow S_2[j++]$;
while ($i < S_1$.size()) **do** $S[k++] \leftarrow S_1[i++]$;
while ($j < S_2$.size()) **do** $S[k++] \leftarrow S_2[j++]$;

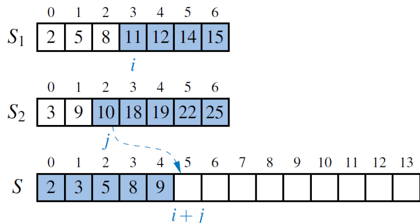
Correttezza: esercizio

Osservazione: L'algoritmo funziona correttamente anche se S_1 e S_2 hanno lunghezze diverse.

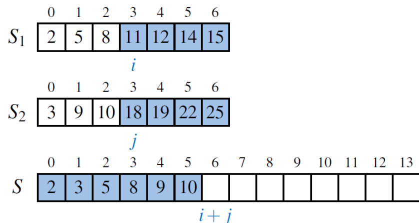
Esempio: generica iterazione del I while di Merge

(a): inizio iterazione

(b): fine iterazione



(a)



(b)

Analisi di Merge

Proposizione

La complessità di `Merge`(S_1,S_2,S) è $\Theta(n)$, dove n è il numero di chiavi in S_1 più il numero di chiavi in S_2.

Dimostrazione

La dimostrazione discende immediatamente dalle seguenti osservazioni:

- Ogni iterazione di ciascun dei tre cicli while richiede un numero costante di operazioni
- In ogni iterazione di ciascun dei tre cicli while una chiave viene memorizzata definitivamente nella sequenza di output S, e quindi vengono eseguite n iterazioni tra tutti i tre cicli while
- Le operazioni fatte al di fuori dei cicli while sono in numero costante

Analisi di MergeSort

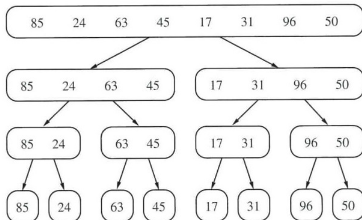
- Utilizziamo l'*albero della ricorsione*
- Distinguiamo 2 casi

Caso 1: $n = 2^d$ per un qualche intero $d \geq 0$.

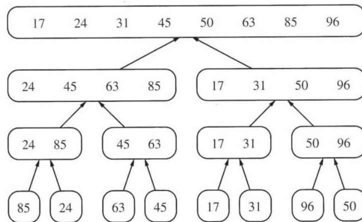
Caso 2: $2^d < n < 2^{d+1}$, per un qualche intero $d \geq 0$.

Analisi di MergeSort. Caso 1: $n = 2^d$

Il seguente esempio ([GTG14, Figura 13.1]) fa vedere l'albero della ricorsione per una sequenza di $n = 8$ chiavi, riportando l'input (a) e l'output (b) di ciascuna invocazione ricorsiva.



(a)



(b)

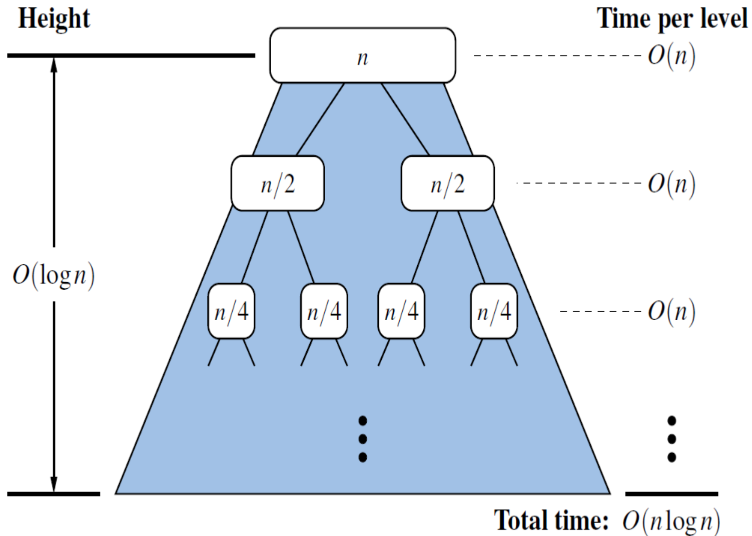
Analisi di MergeSort. Caso 1: $n = 2^d$

È facile generalizzare l'esempio del lucido precedente provando che se $n = 2^d$ (e quindi $d = \log_2 n$) l'albero della ricorsione ha le seguenti caratteristiche.

- Per $0 \leq i \leq d$, il livello i dell'albero ha 2^i nodi corrispondenti a invocazioni su istanze di taglia $n/2^i$.
- Il costo associato a un nodo corrispondente a un'invocazione su un'istanza di taglia $n/2^i$, è $\Theta\left(n/2^i\right)$. Per i nodi interni il costo è determinato dal Merge, mentre per i nodi foglia il costo è costante dato che corrispondono a invocazioni su istanze di taglia 1.

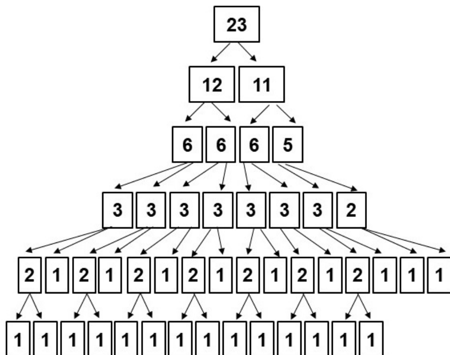
$\Rightarrow$ La complessità dell'algoritmo è

$$\Theta\left(\sum_{i=0}^{d} 2^i \frac{n}{2^i}\right) = \Theta\left(n \cdot d\right) = \Theta\left(n \log n\right).$$


Height
Time per level
n
O(n)
n/2
n/2
O(n)
O(log n)
n/4
n/4
n/4
n/4
O(n)
Total time: O(n log n)

Analisi di MergeSort. Caso 2: $2^d < n < 2^{d+1}$

A titolo d'esempio, riportiamo di seguito l'albero della ricorsione di MergeSort relativo a un'istanza di taglia $n = 23$, facendo vedere le taglie delle istanze associate ai nodi dell'albero.



Analisi di MergeSort. Caso 2: $2^d < n < 2^{d+1}$

Esercizio

Si consideri l'esecuzione di `MergeSort` su un'istanza di taglia $2^d < n < 2^{d+1}$, per un qualche intero $d \geq 0$. Usando l'intuizione ricavata dall'esercizio precedente ($n = 23$) dimostrare che:

1. Per $0 \leq i \leq d$, il livello i dell'albero della ricorsione ha 2^i nodi corrispondenti a invocazioni dell'algoritmo su istanze con taglia in $[2^d/2^i, 2^{d+1}/2^i]$ (induzione su i).
2. Il livello $d+1$ ha $x \leq n$ nodi corrispondenti a invocazioni dell'algoritmo su istanze di taglia 1.

Dall'esercizio discende immediatamente che se $2^d < n < 2^{d+1}$ (e quindi $d \in \Theta(\log n)$), la complessità è

$$\Theta\left(\left(\sum_{i=0}^{d} 2^i \frac{2^d}{2^i}\right) + x\right) = \Theta(n \cdot d) = \Theta(n \log n).$$

Analisi di MergeSort: conclusione

Dall'analisi nei lucidi precedenti segue immediatamente la seguente proposizione.

Proposizione

La complessità di `MergeSort`(S) è $\Theta(n \log n)$, dove $n \geq 1$ è il numero di chiavi in S.

Esercizio

Sia $k = 2^d$, per un qualche intero $d > 0$, e siano $S_1, S_2, \ldots, S_k$ sequenze ordinate di taglia m ciascuna. Progettare e analizzare un algoritmo efficiente per fondere le k sequenze in un'unica sequenza ordinata di taglia $n = k \cdot m$.

Svolgimento

Strategia banale: fondere prima S_1 con S_2, poi $S_1 \cup S_2$ con S_3, poi $S_1 \cup S_2 \cup S_3$ con S_4 ecc.

$\Rightarrow$ Complessità: $\Theta\left(\sum_{i=2}^{k} im\right) = \Theta\left(k^2 m\right) = \Theta\left(kn\right)$.

Strategia più efficiente: fondere le sequenze iniziali a coppie, poi le fusioni delle varie coppie ancora a coppie, ecc.

L'algoritmo basato sulla strategia più efficiente è il seguente:

Algoritmo `kMerge`$(S_1, S_2, \ldots, S_k)$

Input: sequenze ordinate $S_1, S_2, \ldots, S_k$ di taglia m ciascuna
Output: sequenza $\cup_{i=1}^{k} S_i$ ordinata

Sia $\mathrm{S}_i^0 = \mathrm{S}_i$, per $1 \leq i \leq k$;
for $j \longleftarrow 1$ **to** d **do**
  for $i \longleftarrow 1$ **to** 2^{d-j} **do**
    $\mathrm{S}_i^j \leftarrow$ sequenza vuota;
    `Merge`$(\mathrm{S}_{2i-1}^{j-1}, \mathrm{S}_{2i}^{j-1}; \mathrm{S}_i^j)$
return S_1^d

È facile vedere che ciascuna iterazione del ciclo **for** esterno richiede tempo $\Theta(n)$. Quindi la complessità totale risulta essere $\Theta(nd) = \Theta(n \log k)$. Si noti che se $k = n$ e $m = 1$ l'algoritmo è equivalente a MergeSort.

Esercizio (da Esempio di Scritto Completo (4))

Siano $S_1 = S_1[0]S_1[1]\ldots S_1[n_1 - 1]$ e $S_2 = S_2[0]S_2[1]\ldots S_2[n_2 - 1]$ due sequenze ordinate di chiavi intere. In ogni sequenza le chiavi sono distinte, ma una stessa chiave può comparire sia in S_1 che in S_2. Progettare una modifica dell'algoritmo `Merge` che restituisca il numero di chiavi che compaiono sia in S_1 che in S_2. L'algoritmo deve usare spazio aggiuntivo costante oltre alle due sequenze di input.

Esercizio C-13.42 [GTG14]

Sia S una sequenza di n elementi sui quali è definito un ordine totale. Si ricordi che una *inversione* in S è una coppia di elementi $S[i], S[j]$ tali che $j < i$ ma $S[j] > S[i]$. Descrivere un algoritmo che determina il numero di inversioni in S in tempo $O(n \log n)$. (**Suggerimento:** adattare MergeSort.)

QuickSort

Strategia Divide-and-Conquer: caso base $n_0 = 1$

1. **Divide:** $p \leftarrow S[n-1]$ (*pivot*)

$$\begin{aligned} L &\leftarrow \{\text{chiavi} < p \text{ in } S\} \\ E &\leftarrow \{\text{chiavi} = p \text{ in } S\} \\ G &\leftarrow \{\text{chiavi} > p \text{ in } S\} \end{aligned}$$

2. **Recur**: ordinamento ricorsivo delle due sottosequenze
   - `QuickSort(`L`)`
   - `QuickSort(`G`)`
3. **Conquer:** $S \leftarrow L \circ E \circ G$, dove $\circ =$ "concatenazione".

Implementazione in-place di QuickSort

Algoritmo `QuickSortInPlace`(S, a, b)

Input: sequenza S di n chiavi, indici $0 \leq a, b < n$
Output: sequenza S ordinata in senso crescente tra $S[a]$ e $S[b]$

if $a \geq b$ **then return** S;
/* **Divide** */
$\ell \leftarrow \text{Partition}(S, a, b)$;
/* **Recur** */
`QuickSortInPlace`($S, a, \ell - 1$);
`QuickSortInPlace`($S, \ell + 1, b$);
/* **Conquer** */
return S;

Osservazioni:

- `QuickSortInPlace`($S, 0, n - 1$) ordina tutta la sequenza.
- L'algoritmo è in-place (spazio aggiuntivo costante oltre a quello per l'input) se si ignora gestione della ricorsione, che in ogni momento deve mantenere lo stato delle invocazioni ricorsiva aperte e non ancora terminate. Per ciascuna di tali invocazioni (che possono essere anche $\Theta(n)$) l'algoritmo deve mantenere i valori di a, b ed ℓ.

Implementazione in-place di QuickSort

Partition(S, a, b) riorganizza le chiavi in $S[a \div b]$ intorno a un pivot p, separando quelle $\leq p$ da quelle $\geq p$, e restituendo la posizione finale ℓ del pivot. Come pivot viene scelto il valore inizialmente in $S[b]$

Algoritmo `Partition`(S, a, b)

Input: sequenza S di n chiavi, indici $0 \leq a < b < n$
Output: sequenza S riorganizzata tra a e b, e indice $\ell \in [a, b]$ tale che $S[i] \leq S[\ell] \leq S[j]$ per ogni i, j con $a \leq i \leq \ell \leq j \leq b$

$p \leftarrow S[b]$;
$\ell \leftarrow a$; $r \leftarrow b - 1$;
while $(\ell \leq r)$ **do**
  while $((\ell \leq r)$ AND $(S[\ell] \leq p))$ **do** $\ell \leftarrow \ell + 1$;
  while $((\ell \leq r)$ AND $(S[r] \geq p))$ **do** $r \leftarrow r - 1$;
  if $(\ell < r)$ **then** swap$(S[\ell], S[r])$;
swap$(S[\ell], S[b])$;
return ℓ

Esempio (Partition)

$\ell = a$						r	b
85	24	63	45	17	31	96	50
ℓ					r		
85	24	63	45	17	31	96	50
ℓ					r		
31	24	63	45	17	85	96	50
		ℓ		r			
31	24	63	45	17	85	96	50
		ℓ		r			
31	24	17	45	63	85	96	50
			r	ℓ			
31	24	17	45	63	85	96	50

Correttezza di QuickSortInPlace

La correttezza di `QuickSortInPlace` discende immediatamente da quella di `Partition`(S, a, b).

La correttezza di `Partition`(S, a, b) si dimostra tramite il seguente invariante che vale alla fine di ciascuna iterazione del ciclo while più esterno:

- $S[j] \leq p$, per ogni $a \leq j < \ell$
- $S[j] \geq p$, per ogni $r < j \leq b - 1$
- $\ell \leq r + 1$

Esercizio

Dimostrare la correttezza di `Partition`(S, a, b) tramite l'invariante indicato sopra.

Complessità di Partition

Proposizione

La complessità di `Partition`(S, a, b) è $\Theta(b - a + 1)$, ovvero lineare nel numero di chiavi presenti tra gli indici a e b.

Dimostrazione

La complessità discende dalle seguenti osservazioni:

- Ogni due iterazioni del while esterno:
  - l'indice ℓ cresce, oppure l'indice r decresce, oppure entrambe le cose
  - il numero di operazioni eseguito è proporzionale al numero di incrementi/decrementi dei due indici ℓ e r.
- Il numero totale di incrementi/decrementi dei due indici ℓ e r è porporzionale alla lunghezza della sottosequenza: $b - a + 1$.

Complessità di QuickSortInPlace

A titolo d'esempio disegnare (per esercizio) l'albero della ricorsione relativo alla esecuzione di `QuickSortInPlace` usando le due sequenze S indicate qui sotto

- $S = 1, 4, 2, 3, 9, 8, 7, 5, 10, 6$,
- $S = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$,

riportando per ogni nodo dell'albero la sottosequenza in input alla corrispondente chiamata dell'algoritmo.

Complessità di QuickSortInPlace

Proposizione

La complessità di `QuickSortInPlace`$(S, 0, n-1)$ è $\Theta\left(n^2\right)$.

Dimostrazione

Utilizziamo l'albero della ricorsione:

- Ogni nodo dell'albero corrisponde a una chiamata `QuickSortInPlace`(S, a, b) (alla radice: $a = 0, b = n-1$).
- Per $i \geq 0$, la somma delle taglie delle istanze del livello i dell'albero è $\leq n - i$. Infatti, a ogni livello almeno una chiave (ovvero un pivot) viene sistemata definitivamente. ($\Rightarrow$ il numero di livelli è $\leq n$.)
- Il costo di un nodo è lineare nella taglia dell'istanza associata $\Rightarrow$ il costo totale contribuito dai nodi del livello i è $O\left(n - i\right)$.

$\Rightarrow$ la complessità di `QuickSortInPlace`$(S, 0, n-1)$ è

$$O\left(\sum_{i=0}^{n-1} n - i\right) = O\left(n^2\right)$$

Complessità di QuickSortInPlace (continua)

Dimostrazione (continua)

La sequenza ordinata rappresenta un'istanza cattiva per QuickSort in quanto genera un albero della ricorsione con n livelli e costo totale $\Theta\left(n^2\right)$. Di consequenza la complessità dell'algoritmo è quindi $\Theta\left(n^2\right)$. □

Randomized QuickSort

`Randomized QuickSort`: variante di `QuickSortInPlace` che

Evita il caso pessimo probabilisticamente

Sostituisce l'istruzione
$p \leftarrow S[b]$

con le seguenti tre istruzioni

$i \leftarrow$ intero random in $[a, b]$;
swap$(S[i], S[b])$;
$p \leftarrow S[b]$

Osservazione: Per ogni istanza S esistono molte esecuzioni possibili e quindi molti diversi tempi di esecuzione, funzione delle scelte probabilistiche (scelta del pivot) fatte dall'algoritmo.

Algoritmi Deterministici vs Algoritmi Probabilistici

Definizione

Un algoritmo si dice deterministico se per ogni istanza di input esiste *una sola possibile esecuzione* e quindi una sola sequenza di operazioni eseguite.

Tutti gli algoritmi visti sin'ora (tranne Randomized QuickSort) sono deterministici.

Di un algoritmo deterministico A si analizza di solito la complessità al caso pessimo rappresentata da una funzione $t_A(n)$ che per ogni n taglia dell'istanza restituisce il massimo numero di operazioni eseguite da A su istanze di taglia n.

Algoritmi Deterministici vs Algoritmi Probabilistici

Definizione

Un algoritmo si dice probabilistico (randomized) se per ogni istanza di input esistono *diverse possibili esecuzioni*, che dipendono dalle scelte casuali fatte dall'algoritmo, e quindi diverse sequenze di operazioni.

L'obiettivo di un algoritmo probabilistico è di confinare il caso pessimo a una frazione trascurabile di esecuzioni per ogni istanza

Sia A un algoritmo probabilistico che risolve un problema computazionale Π, e sia $t_{A,i}$ il numero di operazioni eseguite da A per risolvere l'istanza i di Π. Dato che per i esistono diverse esecuzioni di A dipendenti dalle scelte casuali fatte dall'algoritmo, è opportuno considerare $t_{A,i}$ come una variabile aleatoria di cui si può stimare la media o la coda della distribuzione.

Analisi di Randomized QuickSort

Randomized QuickSort è un algoritmo probabilistico in quanto esistono diverse possibili esecuzioni per ogni istanza, che sono determinate dalle scelte dei pivot.

Sia $t_{i,\mathrm{RQS}}$ la variabile aleatoria che rappresenta il numero di operazioni eseguite da Randomized QuickSort per risolvere l'istanza i.

Proposizione (senza prova)

Per ogni istanza i di taglia n si ha che

$$\begin{aligned} \Pr(t_{i,\mathrm{RQS}} \in O(n \log n)) &\geq 1 - \frac{1}{n} \quad (\textit{complessità in alta probabilità}) \\ E[t_{i,\mathrm{RQS}}] &\in \Theta(n \log n). \quad (\textit{complessità in media}) \end{aligned}$$

La proposizione rende quantitativamente evidente il fatto che, in pratica, la performance di Randomized QuickSort *per una qualsiasi istanza di input* è paragonabile a quella di MergeSort e dei miglior algoritmi basati su confronti.

Esercizio

Svolgere i seguenti punti:

1. Dire per quali istanze `QuickSortInPlace` esegue fedelmente la strategia L-E-G descritta all'inizio.
2. Dire come si comporta `QuickSortInPlace` su un'istanza con tutte chiavi uguali.
3. Modificare `QuickSortInPlace` in modo che implementi fedelmente la strategia L-E-G *per tutte le possibili istanze*, e dire come si comporta la modifica su un'istanza con tutte chiavi uguali.

Esercizio C-13.48 [GTG14]

Sia dato un insieme A di n viti e un insieme B di n dadi. Ogni vite va bene per un solo dado. Progettare e analizzare un algoritmo per trovare tutte le coppie (vite,dado) giuste avendo a disposizione una primitiva che data una vite a e un dado b decide in tempo costante se a è *piccola*, *giusta*, o *grande* per b.

Svolgimento

(Adattamento di `QuickSort`.) Si prende un dado X arbitrario e lo si confronta con tutte le viti, trovando la vite giusta Y e separando le rimanenti tra quelle piccole per X e quelle grandi per X. Poi, si confronta la vite Y con tutti i dadi (escluso X), separando i dadi piccoli per Y da quelli grandi per Y. Siano V_L e V_G gli insiemi delle viti rispettivamente piccole e grandi per X, e siano D_L e D_G gli insiemi dei dadi rispettivamente piccoli e grandi per Y. L'algoritmo viene chiamato ricorsivamente sulle coppie (V_L, D_L) e (V_G, D_G). L'analisi è analoga a quella fatta per `QuickSort` ed è lasciata come esercizio.

InsertionSort

Algoritmo `InsertionSort(`S`)`

Input: sequenza S di n chiavi
Output: sequenza S ordinata in senso crescente

```
for i ← 1 to n − 1 do
    curr ← S[i];
    j ← i − 1;
    while ((j ≥ 0) AND (S[j] > curr)) do
        S[j + 1] ← S[j];
        j ← j − 1;
    S[j + 1] ← curr;
return S
```

Osservazione: `InsertionSort` è un algoritmo in-place

Complessità di InsertionSort

Analisi standard:

- Il ciclo while all'interno dell'i-esima iterazione del ciclo for esegue $O(i)$ iterazioni ciascuna con un numero costante di operazioni.

  $\Rightarrow$ La complessità totale risulta quindi essere $O\left(\sum_{i=1}^{n-1} i\right) = O\left(n^2\right)$.

- La complessità è anche $\Omega\left(n^2\right)$. Lo si vede facilmente considerando l'istanza costitutita da una sequenza di chiavi distinte inizialmente ordinata in senso decrescente.

Complessità di InsertionSort (continua)

Analisi più fine:

Definizione

Data una sequenza di n chiavi S, una inversione è una coppia di indici (i, j) con $0 \leq i \neq j < n$ tale che $j < i$ e $S[j] > S[i]$.

Osservazione: Sia K il numero di inversioni in una sequenza S di n chiavi. È immediato vedere che

$$0 \leq K \leq \binom{n}{2} = n(n-1)/2.$$

Goal: Analisi della complessità di InsertionSort in funzione sia della taglia che del numero di inversioni della sequenza S.

Complessità di InsertionSort (continua)

Proposizione

La complessità di `InsertionSort`(S) è $\Theta(n + K)$, dove n è il numero di chiavi e K il numero di inversioni in S.

Dimostrazione

La complessità deriva dalle seguenti constatazioni:

- Nell'i-esima iterazione del for, il numero di iterazioni eseguite dal ciclo while è esattamente il numero di inversioni (i, j) della sequenza S iniziale, con i fissato.
- Il totale delle iterazioni di while, aggregato su tutte le iterazioni del for, è quindi pari a K.
- Fuori dai cicli while, `InsertionSort` esegue $\Theta(n)$ operazioni

Complessità di InsertionSort (continua)

Osservazioni:

- L'analisi non contraddice quella standard, dato che si può avere $K = \Theta\left(n^2\right)$, ma evidenzia il fatto che InsertionSort è molto efficiente quando la sequenza di input è quasi ordinata (ovvero, ha poche inversioni)
- L'introduzione di K permette una partizione più fine delle istanze e quindi una più precisa analisi della complessità.

Lower Bound per Algoritmi Basati su Confronti

Teorema (senza prova)

Per istanze di taglia n, un algoritmo di ordinamento A basato su confronti esegue $\Omega(n \log n)$ confronti al caso pessimo. Tale lower bound vale anche per il numero di confronti eseguiti in alta probabilità o in media nel caso A sia probabilistico.

Dato che MergeSort, HeapSort, (Randomized) QuickSort e InsertionSort sono basati su confronti, dal lower bound consegue che

- MergeSort, HeapSort e Randomized QuickSort hanno complessità asintoticamente ottima.
- QuickSort (deterministico) e InsertionSort non hanno complessità asintoticamente ottima ma per InsertionSort si può definire una famiglia abbastanza ampia di istanze per cui esso risulta molto efficiente.

Ordinamento non basato su confronti

Assumiamo ora che la sequenza S da ordinare sia costituita dal $n \geq 0$ entry e che l'ordinamento debba produrre in putput la sequenza di entry ordinate per chiave (in senso crescente).

Gli algoritmi non basati su confronti che vedremo sono:

- BucketSort (chiavi in $[0, N-1]$): crea N code (bucket) associate alle chiavi ed esegue 2 scansioni della sequenza S
  - *I scansione:* memorizza le entry nei bucket in base al valore della chiave
  - *II scansione:* estrae le entry dai bucket ordinate.
- RadixtSort (chiavi composte da d cifre in $[0, N-1]$): esegue d volte `BucketSort`. È una generalizzazione di `BucketSort`.

BucketSort

Algoritmo `BucketSort(`S`)`

Input: sequenza S di n entry con chiavi intere in $[0, N-1]$
Output: sequenza S con le entry ordinate per chiave crescente

Crea un array B di N code $B[0], B[1], \ldots, B[N-1]$;
for $i \leftarrow 0$ *to* $n-1$ **do**
  $k \leftarrow S[i]$.getKey();
  $B[k]$.enqueue($S[i]$);

$i \leftarrow 0$;
for $k \leftarrow 0$ *to* $N-1$ **do**
  while `!(`$B[k]$`.isEmpty())` **do** $S[i{+}{+}] \leftarrow B[k]$.dequeue();

Correttezza: banale.

Esempio: per esercizio, applicare `BucketSort` alla sequenza
$(3, A)(7, D)(4, E)(3, K)(3, P)(8, J)$.

Complessità di BucketSort

Implementazione FIFO dei bucket: (doubly-)linked list con inserimento (*enqueue*) in coda e rimozione (*dequeue*) in testa.

Proposizione

La complessità di `BacketSort`(S) è $\Theta(n+N)$, dove n è il numero di entry in S e $[0, N-1]$ è il range delle chiavi.

Dimostrazione

Primo ciclo for: complessità $\Theta(n)$.

Secondo ciclo for: sia n_k il numero di entry in $B[k]$, per $0 \leq k < N$.

L'iterazione k del for esegue $\Theta(n_k + 1)$ operazioni (N.B. Nel caso $n_k = 0$ si esegue un numero costante di operazioni, e da qui il $+1$.

Se ne deduce che la complessità totale è

$$\Theta\left(n + \sum_{k=0}^{N-1}(n_k + 1)\right) = \Theta(n + N).$$

Osservazioni

- Se $N = o(n \log n)$, `BucketSort` è asintoticamente più veloce di `MergeSort`, `HeapSort`, `QuickSort`.
- $N = O(n) \Rightarrow$ complessità $\Theta(n)$!

Definizione

Un algoritmo di ordinamento per sequenze di entry si dice stabile (stable sorting) se coppie di entry con la stessa chiave mantengono in output lo stesso ordine relativo che avevano in input

Proposizione

`BucketSort` è stabile

Dimostrazione

Conseguenza immediata della politica FIFO di accesso ai bucket. □

RadixSort

- Generalizzazione di BucketSort (estende il range di linearità)
- Molto usato in applicazioni scientifiche
- **Input:** n entry con chiavi costituite da d cifre in $[0, N-1]$, da ordinare *lessicograficamente*.

  N.B.: Le chiavi possono essere visti come interi in $[0, N^d - 1]$, rappresentati in base N
- **Idea:**
  - d fasi
  - Ogni fase: ordinamento stabile su una cifra diversa
  - L'ordine in cui vengono scelte le cifre è importante!

Qual è l'ordine giusto?

Esempio ($d = 2$, $N = 10$)

$$S = (29, A)(72, F)(63, H)(27, K)$$

- Partendo dalla cifra più significativa:
  - Dopo I fase: $(29, A)(27, K)(63, H)(72, F)$
  - Dopo II fase: $(72, F)(63, H)(27, K)(29, A)$ **:(:(**

- Partendo dalla cifra meno significativa:
  - Dopo I fase: $(72, F)(63, H)(27, K)(29, A)$
  - Dopo II fase: $(27, K)(29, A)(63, H)(72, F)$ **:):)**

N.B.: È importante che l'ordinamento sia stabile

RadixSort (continua)

Algoritmo `RadixSort`(S)

Input: sequenza S di n entry con chiavi:
$(c_{d-1}, c_{d-2}, \ldots, c_1, c_0)$, dove $c_i \in [0, N-1]$, $\forall i$
Output: sequenza S con le entry ordinate lessicograficamente per chiave

for $i \leftarrow 0$ *to* $d-1$ **do** ordina S in modo stabile rispetto a c_i ;

Esercizio

Sia S una sequenza di 5 entry con le seguenti chiavi: $513, 116, 273, 506, 018$. Far vedere l'ordine in cui compaiono le chiavi dopo ogni iterazione del ciclo for di RadixSort. Sfruttando l'intuizione ottenuta, dire, nel caso generale, che ordinamento sussiste tra le entry alla fine della Iterazione i del for

Svolgimento

- Dopo iterazione 0: $513, 273, 116, 506, 018$
- Dopo iterazione 1: $506, 513, 116, 018, 273$
- Dopo iterazione 2: $018, 116, 273, 506, 513$

Alla fine della Iterazione i del for le entry sono ordinate lessicograficamente in base alle $i + 1$ cifre meno significative.

Correttezza di RadixSort

La correttezza di RadixSort può essere dimostrata tramite il seguente invariante che vale alla fine di ogni iterazione del for:

le entry di S sono ordinate in base ai valori definiti dalle $i+1$ cifre meno significative delle chiavi: $(c_i, \ldots, c_1, c_0)$

- All'inizio ($i = -1$) l'invariante è banalmente vero
- Iterazione $i \Rightarrow$ Iterazione $i+1$: alla fine dell'iterazione $i+1$ le entry sono ordinate rispetto a c_{i+1} in modo stabile. La stabilità assicura che entry con lo stesso valore di c_{i+1} mantengono lo stesso ordinamento che avevano all'inizio dell'iterazione, ovvero l'ordinamento in base a $(c_i, \ldots, c_1, c_0)$ (grazie all'invariante per la precedente iterazione)
- Alla fine del ciclo $i = d-1$ l'invariante implica immediatamente che la sequenza ha le entry ordinate per chiave.

Complessità di RadixSort

Supponendo di usare `BucketSort` per ordinare S in ciascuna iterazione del for, la prova della seguente proposizione risulta immediata.

Proposizione

La complessità di `RadixSort`(S) è $\Theta(d(n+N))$, dove n è il numero di entry in S e le chiavi sono costituite da d cifre in $[0, N-1]$.

Osservazione: $d = O(1)$ e $N = O(n) \Rightarrow$ complessità $\Theta(n)$

Esempio

Stimiamo le complessità relative di MergeSort e RadixSort per ordinare tutti i cittadini italiani per codice fiscale

- Numero di cittadini $n \simeq 6 \cdot 10^7 \simeq 2^{26}$
- Codice Fiscale: stringa di 16 caratteri da un alfabeto di 36 (che associamo con gli interi in $[0, 35]$)

MergeSort: $\simeq 26 \cdot n$

RadixSort: Vediamo il CF composto da d cifre in $[0, N-1]$. Consideriamo due casi:

1. $d = 16$, $N = 36$: la complessità è $\simeq 16 \cdot (n + N) \simeq 16 \cdot n$
2. $d = 4$, $N = 36^4$: la complessità diventa $\simeq d \cdot (n + N) < 5 \cdot n$

IN OGNI CASO VINCE RADIXSORT!

Esercizio C-13.39 [GTG14]

Dato un array A di n interi nell'intervallo $[0, n^2 - 1]$, descrivere un metodo semplice per ordinare A in tempo $O(n)$.

Svolgimento

Consideriamo le diverse alternative:

- **Ordinamento basato su confronti:** complessità $\Omega(n \log n)$
- **BucketSort**: complessità $\Theta(n + n^2) = \Theta(n^2)$
- **RadixSort**: complessità $\Theta(d(n + N))$ ma bisogna scegliere un'opportuna base N di rappresentazione per le chiavi che a sua volta determina il numero di cifre d.

  Ad es. base $N = n \Rightarrow$ ogni chiave $k \in [0, n^2 - 1]$ può essere rappresentata tramite $d = 2$ cifre (c_1, c_0) in $[0, N - 1]$, dove

  $$c_i = \left\lfloor \frac{k}{N^i} \right\rfloor \bmod N, \quad i = 0, 1.$$

  Quindi, con $N = n$ e $d = 2$ la complessità di RadixSort è $\Theta(n)$.

Osservazione

In generale, per ordinare una sequenza S di n chiavi nel range di interi $[0, M-1]$, con $M > n$, conviene scegliere $N = n$ come base delle chiavi. Vediamo il motivo.

Supponiamo di scegliere una base $N \leq M$ generica ($N > M$ non ha senso!). È facile vedere che una qualsiasi chiave $k \in [0, M-1]$ può essere rappresentata con $d = \lceil \log_N M \rceil$ cifre in base N, ovvero $k = \sum_{i=0}^{d-1} c_i \cdot N^i$, dove

$$c_i = \left(\left\lfloor \frac{k}{N^i} \right\rfloor \bmod N \right) \in [0, N-1], \qquad \text{per } 0 \leq i < d.$$

Applicando RadixSort con tale rappresentazione la complessità è

$$\Theta\left(d(n+N)\right) = \Theta\left((\log_N M)(n+N)\right) = \Theta\left(\frac{\log M}{\log N}(n+N)\right),$$

e si vede che tale formula è minimizzata scegliendo $N = n$.

Esercizio C-13.40 [GTG14]

Siano $S_1, S_2, \ldots, S_k$ k sequenze non vuote di entry con chiavi nel range $[0, N-1]$, per un qualche $N \geq 2$. Descrivere un algoritmo che ordini separatamente tutte le sequenze in tempo $O(n+N)$, dove n è la somma delle taglie delle sequenze. (In output le sequenze devono rimanere distinte e quindi l'esercizio non chiede di ordinare l'unione delle sequenze, che sarebbe banale.)

Svolgimento

Idea: applicare `RadixSort` all'unione delle entry, usando come chiave per ciascuna entry la coppia (c_1, c_0) dove c_0 è la chiave originale della entry e c_1 è l'indice della sequenza a cui essa appartiene (e quindi: $c_0 \in [0, N-1]$ e $c_1 \in [1, k]$).

Dopo l'ordinamento gli elementi di ciascuna sequenza risulteranno contigui e ordinati in base alla loro chiave originale.

L'algoritmo si compone dei seguenti passi:

1. Trasferisci le n entry in un'unica sequenza S, assegnando a ciascuna una chiave (c_1, c_0) come indicato sopra.
2. Ordina S tramite `RadixSort`
3. Estrai da S le k sequenze $S_1, S_2, \ldots, S_k$ ordinate.

Correttezza: banale

Complessità. Sia $M = \max\{N, k+1\}$.

- Passi (1) e (3): eseguibili banalmente in tempo $O(n)$.
- Passo (2): applicazione di `RadixSort` a una sequenza di n entry con chiavi formate da $d = 2$ cifre in $[0, M-1] \Rightarrow$ tempo $O(d(n+M)) = O(n+N)$ in quanto $k \leq n$, e quindi $M = O(n+N)$.

$\Rightarrow$ La complessità dell'algoritmo è $O(n+N)$, come richiesto.

Esercizio (da Esempio di II Compitino/Scritto Completo (3))

Sia S una sequenza di n chiavi intere. Non si fanno ipotesi sul range delle chiavi, che può essere arbitrariamente grande, ma si sa che in S ci sono solo $k = O(\log n)$ chiavi distinte. Una chiave si dice *frequente* se compare almeno 10 volte in S. Progettare un algoritmo di complessità $o(n \log n)$ che determini quante sono le chiavi frequenti in S. Far vedere che la complessità è $o(n \log n)$.

Esempio domande prima parte

- Il seguente algoritmo ordina una sequenza S di n interi.

  ```
  S1 ← interi pari di S; S2 ← interi dispari di S;
  Ordina S1 con MergeSort;
  Ordina S2 con QuickSort deterministico;
  Merge(S1,S2;S);
  ```

  Osservando che la separazione di pari e dispari può essere fatta in tempo $\Theta(n)$, determinare la complessità al caso pessimo dell'algoritmo esprimendola con $\Theta(\cdot)$.
- Spiegare cosa significa che Randomized QuickSort ha complessità $O(n \log n)$ in alta probabilità.
- Sia S una sequenza di n chiavi intere da ordinare. Dire quale tra gli algoritmi di ordinamento che conoscete usereste nei seguenti casi, specificando la complessità risultante: (i) le chiavi da ordinare appartengono all'intervallo $[1, n^2]$; (ii) esistono 3 chiavi, tolte le quali S risulta già ordinata. Motivare le risposte.
- Definire cosa si intende per ordinamento stabile di una sequenza di entry, e argomentare che, con un'opportuna implementazione, l'algoritmo BucketSort produce un ordinamento stabile.

Riepilogo

- Algoritmi di ordinamento basati su confronti
  - Paradigma Divide-and-Conquer
  - MergeSort: specifica, complessità con albero della ricorsione
  - QuickSort deterministico: specifica (quasi) in-place, complessità con albero della ricorsione
  - InsertionSort: specifica, complessità in funzione del numero di chiavi e del numero di inversioni
  - Algoritmi deterministici vs algoritmi probabilistici
  - Randomized QuickSort: specifica, complessità in alta probabilità e in media
  - Lower bound per algoritmi di ordinamento basati su confronti
- Algoritmi di ordinamento non basati su confronti
  - BucketSort
  - Ordinamento Stabile
  - RadixSort

Errata

Cambiamenti rispetto alla prima versione dei lucidi:

- Lucido 23: migliorata la specifica input-output di Partition
- Lucidi 31-35 (31-33 nella nuova versione): semplificati e resi aderenti a quanto fatto a lezione.
- Lucido 58 (56 nella nuova versione): aggiunta la specifica che le sequenze di input sono non vuote.