

# Problemi di Algoritmica 2

Jacopo Notarstefano

8 dicembre 2012

If you're having girl problems  
I feel bad for you son  
I got 99 problems  
but a bitch ain't one.

---

Jay-Z

# Problema 1

## Ordinamento in memoria esterna

**Problema.** *Nel modello EMM (external memory model), mostrate come implementare il  $k$ -way merge, ossia la fusione di  $n$  sequenze individualmente ordinate e di lunghezza totale  $N$ , con costo I/O di  $O(\frac{N}{B})$  dove  $B$  è la dimensione del blocco. Minimizzare e valutare il costo di CPU. Analizzare il costo del merge (I/O complexity, CPU complexity) che utilizza tale  $k$ -way merge.*



## Problema 2

### Limite inferiore per la permutazione

**Problema.** *Estendere l'argomentazione usata per il limite inferiore del problema dell'ordinamento in memoria esterna a quello della permutazione: dati  $N$  elementi  $e_1, e_2, \dots, e_N$  e un array  $\pi$  contenente una permutazione degli interi in  $[1, 2, \dots, N]$ , disporre gli elementi secondo la permutazione in  $\pi$ . Dopo tale operazione, la memoria esterna deve contenerli nell'ordine  $e_{\pi[1]}, e_{\pi[2]}, \dots, e_{\pi[N]}$ .*



## Problema 3

### Permutazione in memoria esterna

**Problema.** *Dati due array  $A$  e  $C$  di  $N$  elementi, dove  $A$  è l'input e  $C$  una permutazione di  $\{0, 1, \dots, n-1\}$ , descrivere e analizzare nel modello EMM un algoritmo ottimo per costruire  $A[C[i]]$  per  $0 \leq i \leq n-1$ .*





## Problema 4

# Multi-selezione in memoria esterna

**Problema.** *Scrivere tutti i passaggi dell'analisi del costo e della correttezza dell'algoritmo di multi-selezione visto a lezione.*

Vogliamo esibire un algoritmo che selezioni un certo numero di pivot da un insieme  $S$  di cardinalità  $N$  in modo tale che la distanza fra pivot consecutivi sia piccola. Il nostro scopo sarà usare questo algoritmo per costruire un analogo del QUICKSORT in memoria esterna, così come la  $k$ -way merge ci ha permesso di costruire l'algoritmo di MERGE SORT in memoria esterna.

Ci potremmo aspettare di dover trovare  $m$  pivot, in analogia a quanto facciamo per la Merge. In realtà è sufficiente determinarne  $\sqrt{m}$ . Diamo di seguito l'algoritmo e due lemmi. Nel primo dimostreremo il costo lineare, nel secondo la correttezza dell'algoritmo.

---

**Algoritmo 1** Multi-selezione in memoria esterna

---

- 1: Carico e ordino in memoria principale  $\frac{N}{M}$  run di  $M$  elementi ciascuno.
  - 2: Da ogni run seleziono un elemento ogni  $\frac{\sqrt{m}}{4}$  e chiamo  $G$  (elementi verdi) l'insieme degli elementi selezionati.
  - 3: Uso l'algoritmo dei cinque autori  $\sqrt{m}$  volte per selezionare in  $G$  un elemento ogni  $\frac{4N}{m}$  e chiamo  $R$  (elementi rossi) l'insieme degli elementi selezionati.
  - 4: Ritorno  $R$ .
- 

**Lemma 1** (Costo). *L'algoritmo compie  $O(n)$  I/O.*

**Dimostrazione.** *La prima riga dell'algoritmo comporta soltanto di scandire tutti gli elementi: l'ordinamento di ogni run viene infatti svolto in memoria principale, e non comporta ulteriori I/O. Anche la seconda riga consiste in una scansione di tutti gli elementi. Per stimare il numero di I/O della terza riga sfruttiamo invece il fatto che ogni esecuzione dell'algoritmo dei cinque autori comporta una scansione di tutti gli elementi. Abbiamo dunque  $\sqrt{m}$  scansioni di  $|G|$  elementi, perciò:*

$$\sqrt{m} \cdot O\left(\frac{|G|}{B}\right) = \sqrt{m} \cdot O\left(\frac{4N}{B\sqrt{m}}\right) = O\left(\frac{4N}{B}\right) = O(n),$$

dove la prima eguaglianza discende dal fatto che, avendo selezionato un elemento ogni  $\frac{\sqrt{m}}{4}$ , la cardinalità di  $G$  è  $\frac{4N}{\sqrt{m}}$ . Ogni riga contribuisce quindi  $O(n)$  I/O, da cui la tesi.



Figura 4.1: Ogni riga orizzontale rappresenta un run ordinato, e i cerchietti gli elementi di ogni run. Cerchietti rossi e verdi rappresentano rispettivamente gli elementi di  $R$  e  $G$ , cerchietti neri i restanti elementi senza colore. Sono inoltre raffigurati in giallo i bordi definiti dalla posizione degli elementi rossi nei quali possiamo avere elementi senza colore compresi fra un rosso e un verde.

**Lemma 2** (Correttezza). *Il numero di elementi di  $S$  compresi fra due elementi di  $R$  è minore di  $\frac{3}{2} \frac{N}{\sqrt{m}}$ .*

**Dimostrazione.** *Vogliamo dunque stimare il numero di elementi di  $S$  compresi fra due generici elementi rossi  $r_1$  e  $r_2$ . Possiamo dividerli in tre categorie:*

- *Gli elementi verdi compresi fra i due elementi rossi  $r_1$  e  $r_2$ .*
- *Gli elementi senza colore compresi fra due elementi verdi.*
- *Gli elementi senza colore compresi fra un rosso e un verde.*

*I primi sono facilmente maggiorati da  $X = \frac{4N}{m}$ : nella terza riga dell'algoritmo abbiamo infatti scelto un rosso ogni  $\frac{4N}{m}$  elementi verdi.*

*I secondi sono invece maggiorati da  $Y = \frac{N}{\sqrt{m}} - \frac{4N}{m}$ . Per la seconda riga dell'algoritmo abbiamo infatti  $\frac{\sqrt{m}}{4} - 1$  elementi senza colore fra due verdi consecutivi appartenenti allo stesso run, avendo scelto un verde ogni  $\frac{\sqrt{m}}{4}$  elementi. Per la terza riga abbiamo al più  $\frac{4N}{m}$  elementi verdi fra  $r_1$  e  $r_2$ , dunque al più  $\frac{4N}{m}$  coppie di elementi verdi consecutivi nello stesso run. Ma allora il numero cercato è stimato dal prodotto, e quindi da:*

$$\frac{4N}{m} \left( \frac{\sqrt{m}}{4} - 1 \right) = \frac{N}{\sqrt{m}} - \frac{4N}{m}.$$

*I terzi sono invece maggiorati da  $Z = \frac{n}{2\sqrt{m}} - \frac{2n}{m}$ . Per vedere questo abbiamo bisogno della figura. Osserviamo infatti che la posizione dei due elementi rossi definisce un paio di "bordi" (raffigurati in giallo in figura) in cui è possibile trovare gli elementi del terzo tipo. Ognuno di questi bordi può contenere al più  $\frac{\sqrt{m}}{4} - 1$  elementi, perché se ne contenesse di più conterrebbe sicuramente due verdi, quindi elementi compresi fra due verdi. Per ognuno degli  $\frac{N}{M}$  run abbiamo insomma 2 bordi contenenti al più  $\frac{\sqrt{m}}{4} - 1$  elementi, perciò possiamo stimare il numero di elementi del terzo tipo con il loro prodotto, cioè:*

$$2 \cdot \frac{n}{m} \cdot \left( \frac{\sqrt{m}}{4} - 1 \right) = \frac{n}{2\sqrt{m}} - \frac{2n}{m},$$

dove abbiamo usato che  $\frac{N}{M} = \frac{n}{m}$  essendo  $n = \frac{N}{B}$  e  $m = \frac{M}{B}$ .

Di conseguenza il numero totale degli elementi compresi fra  $r_1$  e  $r_2$  è maggiorato da:

$$\begin{aligned} X + Y + Z &= \frac{4N}{m} + \frac{N}{\sqrt{m}} - \frac{4N}{m} + \frac{n}{2\sqrt{m}} - \frac{2n}{m} \\ &\leq \frac{N}{\sqrt{m}} + \frac{n}{2\sqrt{m}} \end{aligned}$$

nella quale abbiamo cancellato i termini uguali di segno opposto e un termine negativo, ottenendo un'ulteriore maggiorazione. Abbiamo inoltre:

$$\begin{aligned} \frac{N}{\sqrt{m}} + \frac{n}{2\sqrt{m}} &\leq \frac{N}{\sqrt{m}} + \frac{N}{2\sqrt{m}} \\ &= \frac{3}{2} \frac{N}{\sqrt{m}} \end{aligned}$$

essendo  $n = \frac{N}{B}$ , da cui la tesi.



# Problema 5

## MapReduce

**Problema.** *Utilizzare il paradigma Scan & Sort mediante la MapReduce per calcolare la distribuzione dei gradi in ingresso delle pagine Web. In particolare, specificare quanti passi di tipo MapReduce sono necessari e quali sono le funzioni Map e Reduce impiegate. Ipotizzare di avere già tali pagine a disposizione.*



## Problema 6

### Navigazione implicita in vEB

**Problema.** *Dato un albero completo memorizzato secondo il layout di van Emde Boas (vEB) in modo implicito, ossia senza l'ausilio di puntatori (come succede nello heap binario implicito), trovare la regola per navigare in tale albero senza usare puntatori espliciti.*





# Problema 7

## Layout di alberi binari

**Problema.** *Proporre una paginazione di alberi binari in blocchi di dimensione  $B$  per realizzare un loro layout in memoria esterna: valutare se un qualunque cammino minimo radice-nodo di lunghezza  $l$  attraversa sempre  $O(\frac{l}{\log B})$  pagine. NOTA: per chi vuole, esiste una versione più impegnativa di questo esercizio, basta contattarmi per averla.*



## Problema 8

### Suffix array in memoria esterna

**Problema.** *Utilizzando la costruzione del suffix array basata sul MERGE SORT e la tecnica DC3 vista a lezione, progettare un algoritmo per EMM per costruire il suffix array di un testo che abbia la stessa complessità del MERGE SORT in EMM.*



## Problema 9

### Famiglia di funzioni hash uniformi

**Problema.** *Mostrare che la famiglia di funzioni hash  $H = \{h(x) = ((ax + b) \bmod p) \bmod m\}$  è (quasi) uniforme, dove  $a, b \in [m]$  con  $a \neq 0$  e  $p$  è un numero primo sufficientemente grande.*



## Problema 10

### Count-min sketch: estensione

**Problema.** *Estendere l'analisi vista a lezione permettendo di incrementare e decrementare i contatori con valori arbitrari.*





# Problema 11

## Count-min sketch: prodotto scalare

**Problema.** *Mostrare come utilizzare il paradigma del count-min sketch per approssimare il prodotto scalare (i.e., approssimare  $\sum_{k=1}^n F_a[k] \cdot F_b[k]$ ).*



## Problema 12

### Count-min sketch: interval query

**Problema.** *Mostrare come utilizzare il paradigma del count-min sketch per rispondere alle interval query (i.e., approssimare  $\sum_{k=i}^j F[k]$ ).*



## Problema 13

### Elementi distinti

**Problema.** *Progettare e analizzare un algoritmo di data streaming che permetta di approssimare il numero di elementi distinti.*



# Problema 14

## Cuckoo hashing

**Problema.** *Scrivere tutti i passaggi dell'analisi del costo dell'inserimento di un elemento in una tabella di cuckoo hashing. Discutere anche della cancellazione e della sua complessità.*

Dobbiamo fare hashing da un insieme di  $n$  elementi ad un insieme di  $r$  elementi. Invece di procedere nel modo classico usando le liste di adiacenza (che nel caso pessimo potrebbero contenere tutti gli elementi), vogliamo un modo che ci permetta di avere esattamente un elemento in associato ad ognuno degli  $r$  valori (quindi che sia possibile salvare gli  $n$  elementi su un array di dimensione  $r$ ). Per farlo, ci avvaliamo di due funzioni hash, 2-wise indipendenti,  $h_1(x)$  e  $h_2(x)$ , in questo modo:

---

**Algoritmo 2** Inserimento in Cuckoo hashing

---

- 1: provo ad inserire  $x$  in  $h_1(x)$
  - 2: se la cella è libera lo inserisco semplicemente
  - 3: se la cella non è libera tolgo l'elemento  $y$  dalla cella  $h_1(x)$  e ripeto il procedimento per  $y$ , provando a inserirlo in  $h_1(y)$  se  $h_1(x) = h_2(y)$  o in  $h_2(y)$  se  $h_1(x) = h_1(y)$ , per un massimo di  $n$  volte.
  - 4: se il valore non è stato ancora inserito si cambiano le funzioni di hashing, e si riprova ad inserire il valore che si stava cercando di inserire alla  $n$ -esima iterazione.
- 

Si nota, come vedremo, che il numero di iterazioni massimo nel punto 3 serve per evitare di andare in loop quando si tenta di inserire un valore.

Per fare l'analisi è necessario introdurre i concetti di grafo Cuckoo e di bucket, più un lemma:

**Definizione** (Grafo Cuckoo). *Il grafo Cuckoo è un grafo che ha per nodi le celle dell'array, con un arco uscente in ogni cella contenente un valore, che punta alla cella alternativa secondo le funzioni  $h_1(x)$ ,  $h_2(x)$ . Ovvero, se il nodo  $i$  contiene il valore  $x$ ,  $i$  avrà un arco uscente verso  $h_2(x)$  se  $i = h_1(x)$  o viceversa un arco verso  $h_1(x)$  se  $i = h_2(x)$ .*

**Definizione** (Bucket). *Si dice bucket di un valore  $x$  l'insieme dei nodi raggiungibili dai nodi  $\{h_1(x), h_2(x)\}$  nel grafo Cuckoo. Ossia tutti i nodi con cui potremmo avere a che fare nel caso volessimo inserire  $x$ .*

**Lemma 3.** Per ogni nodo  $i$  e  $j$ , e ogni  $c > 1$ , se  $r \geq 2cn$ , la probabilità che esista un cammino tra  $i$  e  $j$  di lunghezza  $l$  è al più  $\frac{c^{-l}}{r} = \frac{1}{c^l r}$ . Ovvero, se il numero di celle nell'array è sufficientemente più grande del numero di valori salvati, la probabilità che esista un cammino di lunghezza  $l$  tra due nodi è  $O(\frac{1}{r})$ , e decresce esponenzialmente.

**Dimostrazione.** Procediamo per induzione sulla lunghezza del percorso:

- per  $l = 1$ : un percorso di lunghezza 1 tra due nodi  $i$  e  $j$  esiste sse per qualche  $x$   $h_1(x) = i \wedge h_2(x) = j$  oppure  $h_1(x) = j \wedge h_2(x) = i$ , si ha:

$$\begin{aligned} Pr[(h_1(x) = i \wedge h_2(x) = j) \vee (h_1(x) = j \wedge h_2(x) = i)] &= \\ &= Pr[h_1(x) = i \wedge h_2(x) = j] + Pr[h_1(x) = j \wedge h_2(x) = i] = \\ &= 2Pr[h_1(x) = i] Pr[h_2(x) = j] = \\ &= 2 \frac{1}{r} \frac{1}{r} = \frac{2}{r^2} \end{aligned}$$

Siccome il numero di elementi per cui vale la proprietà vista sopra è al più  $n$ , si ha:

$$\begin{aligned} Pr[\exists \text{ percorso di lunghezza 1 tra } i \text{ e } j] &\leq n \frac{2}{r^2} = \frac{2n}{r} \frac{1}{r} \\ &\left\{ r \geq 2cn \text{ per ipotesi} \Rightarrow c \leq \frac{r}{2n} \Rightarrow \frac{1}{c} \geq \frac{2n}{r} \right\} \\ &\leq \frac{1}{cr} = \frac{c^{-1}}{r} \end{aligned}$$

- per  $l > 1$  è necessario che:
  1. Esista un percorso ottimo lungo  $l - 1$  da  $i$  a  $k$ .
  2. Esista un arco tra  $k$  e  $j$ .

Abbiamo

$$Pr[(1)] = \frac{c^{1-l}}{r}$$

per ipotesi induttiva. Inoltre, usando lo stesso ragionamento di prima otteniamo che

$$Pr[(2)] = \frac{c^{-1}}{r}.$$

Notiamo che i valori possibili di  $k$  sono  $r$  e che quindi la probabilità totale è data da:

$$r Pr[(1)] Pr[(2)] = r \frac{c^{-l}}{r^2} = \frac{c^{-l}}{r}$$

□



La probabilità che al punto 3 dell'inserimento avvenga un reashing, è maggiorata dalla probabilità che per qualche elemento esista un ciclo. Se notiamo che *esiste un ciclo di lunghezza  $l \Leftrightarrow$  esiste un percorso di lunghezza  $l$  tra  $i$  e  $i$*  otteniamo:

$$\begin{aligned}
& Pr [\exists \text{un ciclo per il nodo } i \text{ nel grafo Cuckoo}] \\
&= \sum_{l=1}^{\infty} Pr [\exists \text{ciclo di lunghezza } l \text{ nel grafo Cuckoo}] \\
&= \sum_{l=1}^{\infty} Pr [\exists \text{percorso di lunghezza } l \text{ tra } i \text{ e } i] \\
&\leq \sum_{l=1}^{\infty} \frac{c^{-l}}{r} = \frac{1}{r(c-1)}
\end{aligned}$$

da cui:

$$\begin{aligned}
& Pr [\text{rehashing}] \\
&= \sum_{i=1}^r Pr [\exists \text{un ciclo per il nodo } i \text{ nel grafo cuckoo}] \\
&= r * \frac{1}{r(c-1)} = \frac{1}{c-1}
\end{aligned}$$

Ponendo  $c = 3$  la probabilità di un rehash è  $\frac{1}{2}$ , e di  $n$  rehash è  $\frac{1}{2^n}$ , quindi il numero atteso di rehash ad ogni inserimento è

$$\sum_{i=1}^n i * \frac{1}{2^i} = 2.$$

Il costo medio di un inserimento quindi è dato dal costo di due rehashing, ognuno da  $\Theta(n)$ , quindi a sua volta  $\Theta(n)$ , mentre il costo ammortizzato per ogni inserimento è  $O(1)$ .

La cancellazione avviene in  $O(1)$ , cercando il valore da cancellare nelle sole due celle possibili ed eliminandolo, è possibile osservare infatti che la struttura che si ottiene è ancora un cuckoo hashing dove la funzione  $h_1(x)$  è quella che mette tutti gli elementi esattamente dove sono e la funzione  $h_2(x)$  è una qualunque (o varianti equivalenti).



# Problema 15

## Random search tree

**Problema.** *Scrivere l'algoritmo per inserire una chiave in un random search tree con una sola discesa dalla radice (i.e., senza dover risalire poi dalla foglia appena inserita verso la radice mediante le rotazioni).*



# Problema 16

## Lista invertita compressa

**Problema.** Prendiamo una sequenza ordinata crescente di  $n$  interi  $i_1, i_2, \dots, i_n$ , come per esempio una lista invertita. La rappresentazione compressa differenziale è la sequenza  $S$  di  $|S|$  bit ottenuti concatenando  $\gamma(i_1), \gamma(i_2 - i_1), \dots, \gamma(i_n - i_{n-1})$ , dove  $\gamma(x)$  rappresenta il gamma code di Elias per la codifica dell'intero  $x \geq 1$  in  $2\lfloor \log_2 x \rfloor + 1$  bit. Mostrare come aggiungere un'opportuna directory di spazio  $O(|S|)$  bit (meglio ancora, di  $o(|S|)$  bit) per poter accedere velocemente, dato  $j \in [2 \dots n]$ , alla codifica  $\gamma(i_j - i_{j-1})$ . Estendere tale approccio per accedere velocemente a  $i_j$  (e quindi poter eseguire una ricerca binaria sugli interi della lista invertita compressa).



# Problema 17

## Prefix tree del codice di Huffman

**Problema.** *Impostare un algoritmo per costruire il prefix tree del codice di Huffman. Dimostrare l'ottimalità di tale albero in termini di numero di bit utilizzati per codici prefix free dei simboli.*





## Problema 18

### Applicazioni di LZ77

**Problema.** *Sfruttando le caratteristiche dell'algoritmo LZ77 di Lempel e Ziv per suddividere un testo in una sequenza di frasi, mostrare come utilizzare LZ77 per*

- (a) nascondere dei bit all'interno del file compresso risultante,*
- (b) trovare la più lunga sottostringa che si ripete, ossia che appare almeno due volte nel testo.*



# Problema 19

## Dizionario di LZ78

**Problema.** *Progettare una struttura dati per memorizzare e interrogare velocemente il dizionario delle frasi ottenute incrementalmente con l'algoritmo LZ78. Valutare il costo delle soluzioni proposte.*