

Data Stream

Ing. Francesco Silvestri
silvest1@dei.unipd.it

Grandi Moli di Dati, Prof. A. Pietracaprina
A.A. 2009-2010

Indice

1	Introduzione	1
2	Modelli di Data Stream	2
3	Esempi	4
3.1	Missing Number	4
3.2	Elemento di Maggioranza	5
3.3	Estrazione degli Elementi Frequenti	8
4	Approfondimenti	11

1 Introduzione

L'avanzare della tecnologia ha creato sistemi per l'acquisizione di dati sempre più efficienti e veloci, che ha portato alla creazione di enormi archivi a cui vengono applicate tecniche di elaborazione dati o di estrazione di nuove informazioni (*Data Mining*). Per limitazioni tecnologiche o per efficienza, spesso i dati vengono forniti all'unità di calcolo come un flusso (*stream*) che non può essere interamente contenuto nella memoria a disposizione dell'unità. I problemi su *Data Stream* sono quelli in cui i dati da analizzare vengono forniti da una sorgente esterna come un flusso continuo. Alcuni campi in cui si possono trovare problemi su Data Stream sono: applicazioni finanziarie, monitoraggio di reti, sicurezza, reti di telecomunicazioni, applicazioni web, reti di sensori. Alcuni scenari in cui i dati sono forniti come stream sono i seguenti:

- In alcune applicazioni la distinzione tra acquisizione ed elaborazione non è possibile per vari motivi: la velocità di generazione dei dati è maggiore del tempo necessario per memorizzarli su supporti a lunga durata; le informazioni ricavate da un'analisi futura

non sono più utilizzabili; il costo economico della memorizzazione di tutti i dati supera i benefici dovuti alla loro analisi.

Ad esempio, si supponga si voglia calcolare quanti indirizzi IP distinti utilizzano maggiormente una data connessione. Per stimare il carico di lavoro di tale problema, si consideri una connessione a 2 Gbps con pacchetti da 50 byte ciascuno: questo implica che transitano circa $5 \cdot 10^6$ pacchetti al secondo (cioè un pacchetto ogni $0.2 \mu\text{s} = 2 \cdot 10^{-7} \text{s}$). Se venissero mantenuti solo gli indirizzi IP delle sorgenti (128 bit con IPv6), verrebbero utilizzati 76MB di nuova memoria al secondo, ovvero 6.3TB al giorno. L'informazione utile, ovvero il numero di indirizzi unici frequenti, occupa però solo pochi byte. Una soluzione esatta richiede praticamente l'allocazione dell'intero input, ma si possono ottenere algoritmi approssimati che occupano un numero di bit logaritmico nella lunghezza dello stream. Un altro esempio che verrà descritto in Sezione 3.3 è riconoscere gli utenti web tendenzialmente favorevoli a cliccare su link pubblicitari.

- Gli algoritmi per External Memory sono progettati per dispositivi con due livelli di memoria (una memoria veloce e un disco esterno) in cui i dati vengono trasferiti in blocchi di dimensione fissa: il modello premia l'accesso a dati contenuti nello stesso blocco, ma permette un accesso casuale ai blocchi di memoria. Alcuni sistemi di memorizzazione, come i dischi magnetici o i sistemi a nastro, raggiungono però migliori prestazioni quando i dati vengono letti sequenzialmente dal disco, ovvero quando i dati sono visti come uno stream. Diversamente dal precedente esempio, uno stream può essere riletto più volte dall'unità di calcolo.

Le funzioni che devono essere calcolate sugli stream di dati sono solitamente complicate e non risolubili in maniera esatta e deterministica senza il salvataggio dell'intero input. Infatti, molti algoritmi su Data Stream utilizzano algoritmi randomizzati o approssimati.

2 Modelli di Data Stream

Nel modello classico di data stream, l'unità di calcolo riceve uno stream $\Sigma = (x_1, \dots, x_n)$ di lunghezza n (x_i viene ricevuto prima di x_{i+1} per ogni $1 \leq i < n$) e deve calcolare la funzione $f(x_1, \dots, x_n)$. L'unità di calcolo è dotata di una memoria di lavoro in cui poter salvare variabili di supporto (vedi Figura 1), ma la dimensione della memoria è piccola rispetto alla lunghezza n dello stream, tipicamente $O(\text{polylog } n)^1$, e quindi lo stream non può essere completamente salvato nella memoria. Un algoritmo per Data Stream è definito dai metodi `update(i, x_i)` e `query()`: alla ricezione dell'elemento x_i viene invocato il metodo `update(i, x_i)` che aggiorna lo stato della memoria; il valore della funzione $f(x_1, \dots, x_n)$ è restituito dal metodo `query()`, che viene invocato al termine dello stream. Un algoritmo può effettuare più letture (*pass*) dello stream Σ : in questo caso `query()` viene invocato solo al termine dell'ultima lettura.

¹polylog $n = \log^c n$, dove $c > 0$ è una costante arbitraria.

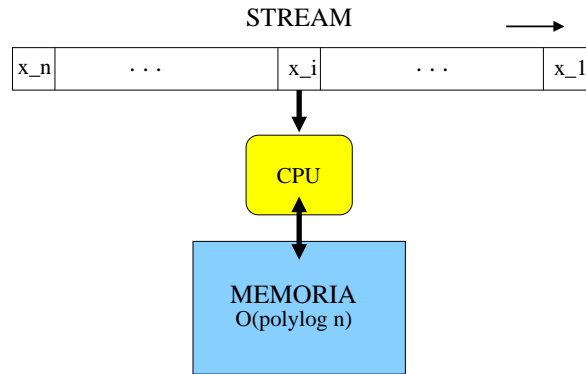


Figura 1: Esempio di sistema di calcolo per il Data Stream.

Gli algoritmi per risolvere problemi su Data Stream vengono valutati rispetto alle seguenti metriche:

- *Update Time* $T_u(n)$: il massimo numero di operazioni richieste per aggiornare la memoria in seguito all'arrivo di un elemento dello stream. Sia $t_u(i)$ il massimo numero di operazioni eseguite da `update(i, x_i)`, allora $T_u(n) = \max_{1 \leq i \leq n} \{t_u(i)\}$.
- *Query time* $T_q(n)$: il massimo numero di operazioni richieste da `query()` per restituire il valore della funzione richiesta.
- *Bit number* $S(n)$. Il massimo numero di bit di memoria utilizzati dall'algoritmo.
- *Pass number* $P(n)$: il massimo numero di letture dello stream effettuate dall'algoritmo.

Un algoritmo su Data Stream è *efficiente* se l'update time, il query time e lo storage sono $O(\text{polylog}(n))$ e il pass number è $O(1)$. La scelta del logaritmo è una conseguenza del fatto che sono necessari $\lceil \log n \rceil + 1$ bit per mantenere traccia del numero di elementi dello stream ricevuti.

È stato dimostrato che per alcuni problemi (ad esempio quelli su grafo) si ha $S(n)P(n) = \Omega(n)$, e quindi non è possibile sviluppare algoritmi efficienti per questi problemi sul modello classico di streaming. Sono stati quindi introdotti dei nuovi modelli di Data Stream più potenti:

- *Semi-Streaming*: In questo modello si richiede che $S(n) = O(n \cdot \text{polylog } n)$, ovvero si rilassa l'ipotesi sulla dimensione della memoria. Questo modello viene tipicamente usato per problemi su grafi con n nodi e $\omega(n)$ archi (e.g., grafi densi): in questo caso il modello contiene $\Omega(1)$ parole di memoria per nodo.
- *Streaming with sorting*: Questo modello è pensato per stream provenienti da dischi esterni per cui esistono primitive di ordinamento molto efficienti. Diversamente dal modello

classico, l'unità di calcolo può scrivere uno stream di output temporaneo, il quale viene successivamente ordinato rispetto a qualche chiave e poi dato in input all'algoritmo.

3 Esempi

3.1 Missing Number

Problema. Alice invia $n - 1$ numeri interi tra 1 ed n a Bob, uno dopo l'altro e in ordine casuale. Lo scopo di Bob è di individuare il *missing number* minimizzando il numero di bit necessari.



Bob is behind the door...

Esempio. Sia $n = 5$. Alice invia $(3, 2, 1, 5)$ a Bob. Il missing number che Bob deve indovinare è 4.

Formalizzazione. Sia $\Sigma = (x_1, \dots, x_{n-1})$ lo stream di numeri interi inviato da Alice a Bob, dove $1 \leq x_i \leq n$ e $x_i \neq x_j$ per ogni $i \neq j$ e $1 \leq i, j \leq n - 1$. Trovare un algoritmo che restituisca il missing number, ovvero il valore intero k tale che $1 \leq k \leq n$ e $k \neq x_i$ per ogni $1 \leq i \leq n - 1$.

Algoritmo. Una soluzione banale è la seguente: Bob alloca un vettore B di n bit a 0, e per ogni x_i impone $B[x_i] = 1$; terminato lo stream, il missing number è data dalla posizione dell'unica cella di B contenente 0. Questa soluzione richiede una sola lettura dello stream e update time costante, ma $\Theta(n)$ bit di memoria e query time $O(n)$ nel caso peggiore (ovvero quando Bob deve leggere tutto il vettore B alla ricerca del bit a 0).

Un algoritmo più efficiente è dato dalla seguente intuizione. Sia k il missing number. Poiché la somma dei primi n numeri è data dalla sommatoria notevole $\frac{n(n+1)}{2}$, è banale vedere che:

$$k = \frac{n(n+1)}{2} - \sum_{1 \leq i \leq n-1} x_i.$$

Quindi Bob deve mantenere solo le somme parziali degli x_i . Lo pseudocodice dei metodi `update(i, x_i)` e `query()` è il seguente:

Algorithm 1: `update(i, x_i)`

```
1 if  $i = 1$  then  $s = 0$ ;
2  $s \leftarrow s + x_i$ ;
```

Algorithm 2: `query()`

```
1 return  $\frac{n(n+1)}{2} - s$ ;
```

Teorema 1. *L'algoritmo usato da Bob per calcolare il missing number tra n numeri è ottimo e richiede update e query time costanti², $\Theta(\log n)$ bit di memoria e una sola lettura dello stream.*

Dim.: L'update e query time richiedono solo $O(1)$ operazioni e l'algoritmo termina dopo una sola lettura dello stream: questi bound sono ovviamente ottimi. L'algoritmo richiede al massimo $2 \log n$ bit di memoria per mantenere s : poiché sono necessari almeno $\log n$ bit per rappresentare il missing number, l'algoritmo è chiaramente ottimo a meno di un fattore costante. \square

Esercizio 1. *Alice invia a Bob $n - 2$ numeri interi compresi tra 1 ed n , uno dopo l'altro e in ordine casuale; Bob deve determinare i due missing number. Si formalizzi il problema e si descriva un algoritmo su Data Stream (ovvero le funzioni **update** e **query**) che risolva il problema usando $\Theta(\log n)$ bit. (Suggerimento: la somma dei quadrati dei primi n numeri è una sommatoria notevole...)*

3.2 Elemento di Maggioranza

Problema. Dato uno stream di lunghezza n trovare, se esiste, l'elemento che compare almeno $\lfloor n/2 \rfloor + 1$ volte (*elemento di maggioranza*).

Esempio. L'elemento di maggioranza nello stream $(A, A, A, C, C, B, B, C, C, C, B, C, C)$ è C , mentre non è presente nello stream (A, A, A, B, B, B, C) .

Formalizzazione. Sia $\Sigma = (x_1, \dots, x_n)$ uno stream con elementi appartenenti ad un alfabeto S . Trovare un algoritmo per Data Stream che restituisca l'elemento di maggioranza, se questo esiste.

Algoritmo. Proponiamo ora un algoritmo per Data Stream [BM91] che restituisce un elemento *candidate* dopo una prima lettura dello stream: se l'elemento di maggioranza esiste, allora questo è *candidate*, se non esiste, *candidate* contiene un generico elemento dello stream. Se non si hanno informazioni sull'esistenza dell'elemento di maggioranza, è necessaria una seconda lettura dello stream per verificare se *candidate* appare almeno $\lfloor n/2 \rfloor + 1$ volte.

L'algoritmo utilizza due variabili: *candidate* e *count*, quest'ultima inizializzata a 0. Alla ricezione di x_i , per ogni $1 \leq i \leq n$, l'algoritmo controlla se *count* = 0. In caso affermativo imposta *candidate* a x_i ; altrimenti controlla se *candidate* = x_i : in caso affermativo incrementa *count* di 1, altrimenti decrementa *count* di 1. Al termine dello stream, se l'elemento di maggioranza esiste, questo è contenuto in *candidate*; se invece l'elemento di maggioranza non esiste *candidate* contiene un generico elemento dello stream. Lo pseudocodice dei metodi

²Si suppone che le operazioni fondamentali (+, -, ×, /) su numeri con al più $O(\log n)$ bit richieda tempo costante.

`update(i, x_i)` e `query()` è il seguente:

Algorithm 3: `update(i, x_i)`

```

1 if  $count = 0$  then  $candidate \leftarrow x_i$ ;
2 if  $candidate = x_i$  then
3    $count \leftarrow count + 1$ 
4 else
5    $count \leftarrow count - 1$ 

```

Algorithm 4: `query()`

```

1 return  $candidate$ ;

```

Teorema 2. *L' algoritmo restituisce l'elemento di maggioranza, se esiste, di uno stream di lunghezza n e richiede $update$ e $query$ time costanti, $\Theta(\log n)$ bit di memoria e una sola lettura dello stream.*

Dim.: Le complessità dell'algoritmo sono evidenti. Dimostriamo ora la sua correttezza.

Siano $candidate_i$ e $count_i$ rispettivamente i valori delle variabili $candidate$ e $count$ dopo l'invocazione di `update(i, x_i)`, con $1 \leq i \leq n$. Dimostriamo per induzione il seguente invariante: per ogni $1 \leq i \leq n$, gli elementi dello stream (x_1, \dots, x_i) possono essere partizionati in due multinsiemi U_i e P_i ,³ dove U_i contiene $count_i$ elementi che coincidono con $candidate_i$, mentre P_i contenente $i - count_i$ elementi che possono essere raggruppati in coppie di elementi distinti. Sia $i = 1$: in questo caso $candidate_1 = x_1$ e $count_1 = 1$ e la partizione cercata è data da $U_1 = \{x_1\}$ e $P_1 = \emptyset$. Ora, supponendo vera l'ipotesi per $i \geq 1$, dimostriamo che l'invariante vale per $i + 1$. Abbiamo tre casi:

1. $count_i = 0$. Per l'ipotesi induttiva abbiamo che $U_i = \emptyset$ e $P_i = \{x_1, \dots, x_i\}$. Dopo la lettura di x_{i+1} si ha che $candidate_{i+1} = x_{i+1}$ e $count_{i+1} = 1$, quindi imponendo $U_{i+1} = \{x_{i+1}\}$ e $P_{i+1} = P_i$ si ottiene la partizione cercata.
2. $count_i > 0$ e $candidate_i = x_{i+1}$. Dopo la lettura di x_{i+1} si ha $candidate_{i+1} = candidate_i$ e $count_{i+1} = count_i + 1$, quindi imponendo $U_{i+1} = U_i \cup \{x_{i+1}\}$ e $P_{i+1} = P_i$ si ottiene la partizione cercata per l'ipotesi induttiva.
3. $count_i > 0$ e $candidate_i \neq x_{i+1}$. Dopo la lettura di x_{i+1} si ha $candidate_{i+1} = candidate_i$ e $count_{i+1} = count_i - 1$, quindi imponendo $U_{i+1} = U_i - \{x_j\}$ e $P_{i+1} = P_i \cup \{x_{i+1}, x_j\}$, dove x_j è un elemento arbitrario di U_i , si ottiene la partizione cercata: infatti gli elementi in P_i si possono appaiare per l'ipotesi induttiva, mentre x_{i+1} può essere appaiato con x_j essendo $x_{i+1} \neq x_j$.

³Intuitivamente, un multinsieme è una generalizzazione di insieme che permette elementi ripetuti.

Avendo provato che il precedente invariante vale per ogni $1 \leq i \leq n$, dimostriamo che se esiste l'elemento di maggioranza in Σ , allora questo è $candidate_n$, ovvero il valore ritornato da `query()`. Sia k l'elemento di maggioranza e supponiamo per assurdo che $candidate_n \neq k$. Visto che k non appare in U_n per definizione e che compare al più $(n - count_n)/2$ volte in P_n (altrimenti non sarebbe possibile creare coppie di elementi distinti), abbiamo che k compare al più $(n - count_n)/2 \leq n/2$ volte, che contraddice l'ipotesi che k sia l'elemento di maggioranza. \square

Esempio Il funzionamento dell'algoritmo è spiegato bene dal seguente esempio. Sia $S = (A, B, C)$ e si consideri lo stream $(A, A, A, C, C, B, B, C, C, C, B, C, C)$. Dopo la prima lettura abbiamo (il carattere in grassetto indica l'ultimo elemento ricevuto):

Stream	i	$candidate$	$count$
$(A, A, A, C, C, B, B, C, C, C, B, C, C)$	1	A	1

Dopo la lettura dei primi tre elementi si ha:

Stream	i	$candidate$	$count$
$(A, A, A, C, C, B, B, C, C, C, B, C, C)$	3	A	3

Dopo la lettura dei primi sei elementi, $count$ viene riportato a 0 non essendoci altre A . Si ha quindi:

Stream	i	$candidate$	$count$
$(A, A, A, C, C, B, B, C, C, C, B, C, C)$	6	A	0

Nell'iterazione successiva, il nuovo candidato diventa B e si ha:

Stream	i	$candidate$	$count$
$(A, A, A, C, C, B, B, C, C, C, B, C, C)$	7	B	1

Dopo la lettura del nono simbolo (C), il nuovo candidato diventa C e non cambierà fino alla fine dello stream. Infatti, dopo aver letto l'ultimo elemento dello stream si ha:

Stream	i	$candidate$	$count$
$(A, A, A, C, C, B, B, C, C, C, B, C, C)$	13	C	3

Si verifica facilmente che C è effettivamente l'elemento di maggioranza.

Un esempio di stream senza elemento di maggioranza è dato da (A, A, A, B, B, B, C) . In questo caso, al termine dello stream si ha:

Stream	i	$candidate$	$count$
(A, A, A, B, B, B, C)	7	C	1

Non è possibile verificare se C sia un elemento di maggioranza senza una seconda lettura dello stream.

3.3 Estrazione degli Elementi Frequenti

Problema. Un problema frequente su Data Stream è l'estrazione di elementi frequenti in uno stream di lunghezza n , ovvero elementi che compaiono più di αn volte, con $0 < \alpha < 1$.

Un'applicazione di questo problema si ha nella gestione degli annunci pubblicitari su siti web [MAA05]. Uno scenario semplificato è composto dalle seguenti figure:

- uno o più *inserzionisti* che vogliono pubblicizzare dei loro annunci su siti web;
- l'*editore* che offre siti web in cui pubblicizzare annunci;
- gli *utenti web* che sono i destinatari ultimi degli annunci; nel presente caso sono i frequentatori dei siti web dell'editore.

Un annuncio web è composto da una breve descrizione di un servizio o prodotto e da un link che rinvia ad un sito web gestito dall'inserzionista. Quando un utente richiede una pagina web dell'editore con annunci, questa viene generata dinamicamente con annunci specializzati per quel utente.

L'editore può stringere accordi commerciali con più inserzionisti, i quali possono pagare lo spazio pubblicitario principalmente in due modi:

- *Pay-Per-Click (PPC)*: l'inserzionista paga un contributo C_c per ogni annuncio visualizzato che riceve un click dall'utente.
- *Pay-Per-Impression (PPI)*: l'inserzionista paga un contributo C_i per ogni visualizzazione (*impressione*) di un annuncio, indipendentemente dal fatto che l'utente clicchi il link associato all'annuncio.

La scelta della modalità di pagamento è scelta dall'inserzionista e dipende da vari fattori commerciali (il tipo di prodotto commercializzato, politiche aziendali, analisi di mercato, ...). L'editore gestisce contemporaneamente sia accordi PPC che PPI e supponiamo per semplicità che abbia a disposizione un numero sufficientemente grande di PPC e PPI.

Lo scopo dell'editore è massimizzare gli introiti dovuti alle pubblicità. Solitamente $C_c \gg C_i$, ma la soluzione ottimale per massimizzare i guadagni non è l'utilizzo di soli annunci PPC. Infatti, molti utenti tendenzialmente non sono interessati agli annunci web e quindi non cliccano il link associato: in questo caso è preferibile un annuncio PPI perché garantisce all'editore un introito, sebbene inferiore a quello di un annuncio PPC. Se però un utente è propenso a cliccare su annunci web, allora è preferibile il PPC perché garantisce, con buona probabilità, un guadagno superiore. È necessario quindi individuare i "cliccatori" frequenti: se un utente è un "cliccatore" frequente allora verrà visualizzata una pubblicità PPC nella

pagina richiesta, altrimenti si sceglierà una PPI. Successivamente alla creazione della pagina web, le strutture dati necessarie per memorizzare gli utenti frequenti devono essere aggiornate sia nel caso l'utente clicchi sull'annuncio che in caso contrario.

Come stima del carico, si pensi che un editore gestisce mensilmente $150 \cdot 10^6$ navigatori unici, $50 \cdot 10^3$ siti Web e $30 \cdot 10^3$ inserzionisti. Poiché ogni utente viene identificato con un identificatore tra i 128 ai 512 bit, sono necessari dai 2 agli 8 GB solo per mantenere il loro elenco. Il numero di volte in cui viene controllato se un utente è un “cliccatore” frequente è $\gg 150 \cdot 10^6$, perché lo stesso utente può accedere a più pagine web. Inoltre i tempi di risposta devono essere molto brevi perché, successivamente al riconoscimento di un “cliccatore” frequente, sono necessarie altre operazioni (come l'individuazione del contenuto degli annunci).

Formalizzazione. Sia $S = \{s_1, \dots, s_m\}$ un insieme di elementi e sia $\Sigma = (x_1, \dots, x_n)$ lo stream, dove $x_i \in S$ per ogni $1 \leq i \leq n$. Diciamo che s_j , con $s_j \in S$, è α -frequente in (x_1, \dots, x_n) se $F(j) \geq \alpha n$, dove $F(j) = |\{k : x_k = s_j, 1 \leq k \leq n\}|$ denota il numero di occorrenze di s_j in (x_1, \dots, x_n) . Definiamo con $I(\alpha)$ il seguente insieme:

$$I(\alpha) = \{s_j \in S \mid F(j) \geq \alpha n\},$$

ovvero $I(\alpha)$ contiene tutti e soli gli elementi α -frequenti dello stream Σ .

Fatto 1. $I(\alpha)$ contiene al più $\lfloor 1/\alpha \rfloor$ elementi

Dim.: Se $|I(\alpha)| > \lfloor 1/\alpha \rfloor$, allora, per la definizione di elemento frequente, il numero di elementi ricevuti dello stream è non minore di $|I(\alpha)|(\alpha n) \geq (\lfloor 1/\alpha \rfloor + 1)\alpha n > (1/\alpha)\alpha n = n$ elementi, ma questa è una contraddizione. Quindi $|I(\alpha)| \leq \lfloor 1/\alpha \rfloor$. \square

Trovare un algoritmo che restituisca gli elementi $I(\alpha)$, ovvero gli elementi α -frequenti dello stream Σ .

Una possibile interpretazione dell'esempio descritto precedentemente è la seguente: n è il numero di click totale effettuato dagli utenti web, lo stream contiene gli identificatori degli utenti web che hanno cliccato su un annuncio web, mentre i “cliccatori” frequenti sono quelli che hanno effettuato più di αn click sugli annunci.

Algoritmo. Il seguente algoritmo è descritto in [KSP03] e l'invocazione di `query()` al termine dello stream restituisce un insieme di al più $\lfloor 1/\alpha \rfloor$ elementi che include tutti gli elementi di $I(\alpha)$. Lo pseudocodice dei metodi `update(i, x_i)` e `query()` è il seguente:

Algorithm 5: `update(i, x_i)`

```
1 if  $i = 1$  then
2   Sia  $K$  un dizionario vuoto che contiene elementi  $(s, c)$ , dove  $c$  è un intero e  $s \in S$  è
   la chiave della coppia;
3 Sia  $x_i = s_j$  con  $s_j \in S$ ;
4 if  $s_j \in K$  then Sostituisci  $(s_j, c)$  con  $(s_j, c + 1)$ ;           /* Incremento di  $c$  */
5 else Inserisci  $(s_j, 1)$  in  $K$ ;
6 if  $|K| > \lfloor 1/\alpha \rfloor$  then
7   foreach  $(s, c) \in K$  do
8     if  $c = 1$  then Rimuovi  $(s, c)$  da  $K$ ;           /* Decremento di  $c$  */
9     else Sostituisci  $(s, c)$  con  $(s, c - 1)$ ;
```

Algorithm 6: `query()`

```
1 return tutte le chiavi contenute in  $K$ ;
```

Teorema 3. *L'insieme K contiene al più $\lfloor 1/\alpha \rfloor$ elementi al termine di ogni chiamata a `update(i, x_i)` e `query()` restituisce un insieme contenente $I(\alpha)$.*

Dim.: Dimostriamo che il dizionario K contiene al più $\lfloor 1/\alpha \rfloor$ elementi alla fine di ogni chiamata a `update(i, x_i)`. Chiaramente questo è vero al termine della prima chiamata `update($1, x_1$)`. Si consideri ora una chiamata generica `update(i, x_i)`, con $i > 1$: per induzione, K contiene al più $\lfloor 1/\alpha \rfloor$ elementi all'inizio della chiamata; in ogni chiamata solo una nuova coppia (s, c) può essere inserita in K (Linea 5), portando $|K|$ ad al più $\lfloor 1/\alpha \rfloor + 1$; ma nel caso $|K| = \lfloor 1/\alpha \rfloor + 1$, la Linea 8 viene eseguita e la taglia di K riportata a $\lfloor 1/\alpha \rfloor$ perché almeno la variabile c dell'ultima coppia inserita vale uno.

Sia $s_j \in I(\alpha)$ e $F(j)$ la sua frequenza in (x_1, \dots, x_n) ; essendo s_j α -frequente, $F(j) \geq \alpha n$. Si supponga per assurdo che s_j non venga restituito da `query()`. È evidente che s_j è stato inserito in K (o la variabile associata c incrementata) $F(j)$ volte; poiché s_j non è presente in K durante l'esecuzione di `query()`, s_j è stato rimosso da K (o la variabile associata c decrementata) $F(j)$ volte. Una cancellazione/decremento avviene solo quando K contiene $\lfloor 1/\alpha \rfloor + 1$ elementi, quindi ogni cancellazione/decremento di s_j implica altre $\lfloor 1/\alpha \rfloor$ cancellazioni/decrementi di elementi in K ma diversi da s_j . Quindi tra la prima chiamata a `query($1, x_1$)` all'ultima a `query(n, x_n)`, vi sono state $(\lfloor 1/\alpha \rfloor + 1) F(j) \geq (\lfloor 1/\alpha \rfloor + 1) \alpha n > n$ cancellazioni/decrementi. Ma questo è impossibile perché il numero totale di inserzioni/incrementi è esattamente n e il numero di cancellazioni/decrementi non può essere maggiore di n . Dunque si può concludere che s_j è presente nell'insieme restituito da `query()`. \square

Teorema 4. *Un insieme che contiene $I(\alpha)$ può essere calcolato utilizzando $O((\log n)/\alpha)$ bit di memoria e $O(1/\alpha)$ query time nel caso peggiore. L'update time richiede $O(1)$ operazioni in senso medio e ammortizzato.*

Dim.: Il dizionario può essere implementato tramite una *hash table* con $O(1/\alpha)$ bucket, dove inserzione e cancellazione richiedono tempo $O(1)$ in senso medio⁴. La sostituzione richiesta nelle Linee 4 e 9 si può eseguire in tempo medio costante. Poiché il numero di inserzioni/incrementi non può essere minore del numero di cancellazioni/decrementi, il costo di ogni cancellazione/decremento può essere addebitato ad una inserzione/incremento. Quindi, essendoci al più un inserimento/incremento per ogni invocazione di `update(i, x_i)`, il costo totale delle chiamate a `update(i, x_i)` durante la lettura di (x_1, \dots, x_n) è $O(n)$ in senso medio, e quindi il costo di ogni chiamata a `update(i, x_i)` è $O(1)$ in senso ammortizzato e medio.

Poiché il dizionario contiene al più $\lfloor 1/\alpha \rfloor$, il query time richiede $O(1/\alpha)$ operazioni per estrarre le chiavi da K .

L'hash table contiene $O(1/\alpha)$ bucket e al più $1/\alpha$ coppie, ognuna delle quali richiede al più $O(\log n)$ bit. Quindi l'algoritmo utilizza $O((\log n)/\alpha)$ bit. \square

Come nel caso della majority, l'insieme ritornato da `query()` contiene tutti gli elementi α -frequenti di Σ più alcuni elementi non α -frequenti (falsi positivi). Per ottenere tutti e soli gli elementi α -frequenti è necessaria una seconda lettura dello stream in cui si calcolano le frequenze di tutti gli elementi ritornati dall'algoritmo.

Teorema 5. *L'insieme $I(\alpha)$ può essere calcolato esattamente con due letture dello stream di input, utilizzando $O(\log n/\alpha)$ bit e $O(1/\alpha)$ query time nel caso peggiore. L'update time richiede $O(1)$ operazioni in senso medio e ammortizzato.*

Dim.: Nella prima passata si applica l'algoritmo descritto precedentemente, e sia $\tilde{I}(\alpha)$ l'insieme restituito da `query()`. Nella seconda passata vengono calcolate le frequenze degli elementi in $\tilde{I}(\alpha)$ e si ricava $I(\alpha)$ eliminando gli elementi non α -frequenti. Per mantenere il conteggio delle frequenze si può utilizzare nuovamente un dizionario implementato tramite hash table. Poiché gli elementi di $I(\alpha)$ sono tutti contenuti in $\tilde{I}(\alpha)$, la correttezza dell'algoritmo segue. \square

4 Approfondimenti

Una semplice introduzione al Data Stream si trova in [Hay08]. Eventuali approfondimenti sul Data Stream possono essere trovati in [DF07, Mut05].

Bibliografia

[BM91] Robert S. Boyer and J. Strother Moore. Mjrtj: A fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 105–118, 1991.

⁴Per la definizione di complessità in senso medio e ammortizzato e per la definizione di hash table si rinvia a [GT05].

- [DF07] Camil Demetrescu and Irene Finocchi. *Handbook of Applied Algorithms: Solving Scientific, Engineering, and Practical Problems*, A. Nayak ed I. Stojmenovic eds, chapter 8, Algorithms for Data Streams. John Wiley and Sons, 2007.
- [GT05] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, Inc., New York, NY, USA, 2005.
- [Hay08] Brian Hayes. The Britney Spears problem. *American Scientist*, 96(4):274–279, 2008.
- [KSP03] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems*, 28(1):51–55, 2003.
- [MAA05] Ahmed Metwally, Divyakant Agrawal, and Amr Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proc. of 10th International Conference of Database Theory*, 2005.
- [Mut05] S. Muthukrishnan. Data streams: algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2):117–236, 2005.