Problemi di Algoritmica 2

Alessandro Ambrosano Jacopo Notarstefano Francesco Salvatori

16 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.*

Date k sequenze individualmente ordinate, il k-way merge mantiene in memoria principale k buffer di input e un buffer di output, tutti di dimensione B.

Algoritmo 1 k-way merge in memoria esterna

1: Inizializza ciascun buffer di input a contenere il primo blocco della corrispondente run. Marca quel buffer come attivo.
2: Tra gli elementi ancora inutilizzati dei buffer di input, trova il più piccolo.
3: Sposta tale elemento nella prima posizione libera del buffer di output.
4: Se il buffer di output risulta pieno, svuotalo in memoria esterna e reinizializzalo.
5: Se il buffer di input da cui ho prelevato l'elemento è giunto alla fine, leggi il prossimo blocco della run corrispondente. Se anche la run è terminata, marca quel buffer come inattivo. Se il numero di buffer ancora attivi è > 1, torna al punto 2.
6: Quando resta un solo buffer attivo, corrispondente a un'unica run attiva, copia quest'ultima in memoria esterna.

Il costo, in termini di operazioni di I/O, è chiaramente $O(\frac{N}{B})$; l'algoritmo ovviamente richiede che in memoria principale ci sia abbastanza spazio per tutte le run, ossia $k \leq \frac{M}{B} - 1$. Il costo di CPU dipende interamente dal passo 2 dell'algoritmo. Usando una selezione lineare per trovare il minimo si pagherebbe $O(k)$ ad ogni iterazione, per un totale di $O(k \cdot N)$; è preferibile allora usare una heap di minimo, con inserimento ed estrazione in tempo logaritmico. Così facendo ogni iterazione viene a costare $O(\log k)$, per un costo di CPU totale di $O(N \log k)$.

Il k-way merge-sort prevede che ciascuna run abbia dimensione M, in modo che possa essere interamente ordinata in memoria principale: questo porta a scegliere $k = \lceil \frac{N}{M} \rceil$. Ovviamente questo è possibile solo se

$$\left\lceil \frac{N}{M} \right\rceil + 1 \leq \frac{M}{B},$$

il che, dati i valori attualmente comuni per M e B, risulta verificato per file di dimensioni fino a qualche TeraByte.
Per quanto riguarda gli I/O, ciascun elemento è interessato da due operazioni di input (una per l'ordinamento della run corrispondente, una per il k-way merge) e due di output (quando la run ordinata viene scritta in memoria esterna, e quando il buffer di output del k-way merge viene svuotato), il che porta ad un costo in termini di I/O di

$$O\left(\frac{N}{B}\right).$$

Il costo di CPU è dato da $\frac{N}{M}$ ordinamenti di run, dal costo $O(M \log M)$ ciascuno[1], a cui va sommato il costo del k-way merge, che è $O(N \log \frac{N}{M})$. In totale si ottiene un costo medio CPU di

$$O\left(N \log M + N \log \frac{N}{M}\right) = O\left(\max\left\{N \log M, N \log \frac{N}{M}\right\}\right),$$

che al caso pessimo può risultare $O\left(\max\left\{NM, N \log \frac{N}{M}\right\}\right)$.

[1]Non riesco a usare mergesort a causa delle sue richieste di memoria, dunque usando quicksort (in versione eventualmente randomizzata) ottengo un costo *al caso medio* di $O(M \log M)$, mentre il caso pessimo risulta ovviamente quadratico

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 $e_1, e_2, \ldots, e_N$ e un array π contenente una permutazione degli interi in $[1, 2, \ldots, N]$, disporre gli elementi secondo la permutazione in π. Dopo tale operazione, la memoria esterna deve contenerli nell'ordine $e_{\pi[1]}, e_{\pi[2]}, \ldots, e_{\pi[N]}$.*

Mostreremo che il problema presenta un lower-bound di $\Omega\left(\min\left\{N,\ \frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right\}\right)$. L'idea è la seguente: consideriamo una sequenza di t operazioni di input, e contiamo quante permutazioni diverse possiamo ottenere. Trovando il minimo t che ci garantisca la possibilità di ottenere *tutte* le permutazioni distinte avremo allora il lower-bound desiderato.

Consideriamo una generica operazione di input: possiamo scegliere tra al più N pagine in memoria (in generale se ho effettuato k operazioni di I/O le pagine tra cui scegliere sarebbero $\frac{N}{B} + k$; in ogni caso se ho più di N pagine già scritte allora ho già permutato gli elementi nell'ordine corretto, quindi è lecito usare N come sovrastima). Distinguiamo ora due casi:

- Se la pagina che ho scelto viene caricata in memoria per la prima volta (e questo avviene $\binom{N}{B}$ volte), posso permutare i suoi elementi in $B!$ modi, e disporli tra gli elementi già in memoria in al più $\binom{M}{B}$ modi.
- Se la pagina era già stata caricata in memoria in precedenza, i suoi elementi sono già permutati nell'ordine desirderato, dunque devo solo scegliere gli $\binom{M}{B}$ modi in cui disporli rispetto agli elementi già in memoria.

Unendo i due casi, dopo una sequenza di t I/O posso produrre al più

$$N^t \cdot (B!)^{\frac{N}{B}} \cdot \binom{M}{B}^t$$

permutazioni distinte; ovviamente vogliamo che questo numero risulti maggiore di $N!$. Passando ai logaritmi, dev'essere

$$t \log N + \frac{N}{B} \log B! + t \log \binom{M}{B} \geq \log N!,$$

da cui, usando le approssimazioni $\log x! \sim x \log x$ e $\log \binom{n}{k} \sim k \log \frac{n}{k}$,

$$t \log N + N \log B + tB \log \frac{M}{B} \geq N \log N$$

$$t \left(\log N + B \log \frac{M}{B} \right) \geq N(\log N - \log B)$$

$$t \geq \frac{N \log \frac{N}{B}}{\log N + B \log \frac{M}{B}}$$

Distinguiamo ancora due casi:

- Se $\log N \leq B \log \frac{M}{B}$ dev'essere

$$t \geq \frac{N \log \frac{N}{B}}{2B \log \frac{M}{B}} = \Omega \left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} \right).$$

- Se invece $\log N \geq B \log \frac{M}{B}$ allora

$$t \geq \frac{N \log \frac{N}{B}}{2 \log N} \geq \frac{N \log N - N \log B}{2 \log N} =$$

$$= \frac{1}{2} \left(N - N \frac{\log B}{\log N} \right) \geq \frac{1}{2}(N - \frac{1}{2}N) = \Omega(N),$$

  il che ovviamente vale purché $\frac{\log B}{\log N} \leq \frac{1}{2}$, ossia $B \leq \sqrt{N}$, relazione vera in tutti i casi di interesse.

Dall'unione dei due casi segue il lower-bound desiderato.

Problema 3

Permutazione in memoria esterna

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

Dall'analisi svolta sul Lower Bound della permutazione, sappiamo che il costo di I/O è dato da $\min\{n, \text{sort}(n)\}$, dove $\text{sort}(n) = O\left(\frac{N}{B} log_{\frac{N}{M}} \frac{N}{B}\right)$.

Algoritmo 2 Permutazione in memoria esterna

1: Calcola il minimo fra $\{n, \text{sort}(n)\}$
2: Se il minimo è n si utilizza un algoritmo banale che semplicemente esegue un'iterazione sugli elementi e scrive la permutazione.
3: Nel caso opposto invece si utilizza tecnica MapReduce.

Si può notare che l'algoritmo banale è eseguito solo in poche circostanze, ad esempio quando la dimensione B del blocco è molto piccola. Di seguito è mostrato soltanto l'algoritmo con tecnica MapReduce.

Algoritmo 3 Permutazione in memoria esterna con tecnica MapReduce

1: Creare le coppie $\langle i, C[i] \rangle$, ovvero le coppie che mettono in relazione una posizione del vettore C con l'indice di permutazione.
2: Ordina le coppie in base alla seconda componente (l'indice di permutazione).
3: Sostituisci nelle coppie $C[i]$ con $A[C[i]]$.
4: Ordina le coppie per la prima componente.
5: Scrivi nel vettore D le seconde componenti delle coppie.

Analizziamo il costo dei singoli passi dell'algoritmo:

- Il *primo* passo dell'algoritmo esegue una lettura di N elementi e una scrittura di N coppie. Da qui si deduce che il costo di I/O è $O\left(\frac{N}{B}\right)$.
- Il costo di I/O del *secondo* passo è uguale a quello di un ordinamento che, utilizzando ad esempio il k-way merge-sort è $O\left(\frac{N}{B} log_{\frac{N}{M}} \frac{N}{B}\right)$.

- Nel *terzo* passo, poiché le coppie sono ordinate per $C[i]$, si accede al vettore A in modo sequenziale, quindi il costo di I/O è $O\left(\frac{N}{B}\right)$.
- Anche per il *quarto* il costo di I/O è $O\left(\frac{N}{B} log_{\frac{N}{M}} \frac{N}{B}\right)$.
- Per l'*ultimo* passo il costo di I/O è $O\left(\frac{N}{B}\right)$ in quanto sono letti e scritti N sequenzialmente.

In conclusione il costo di I/O dell'algoritmo appena esposto è $O\left(\frac{N}{B} log_{\frac{N}{M}} \frac{N}{B}\right)$.

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 4 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.

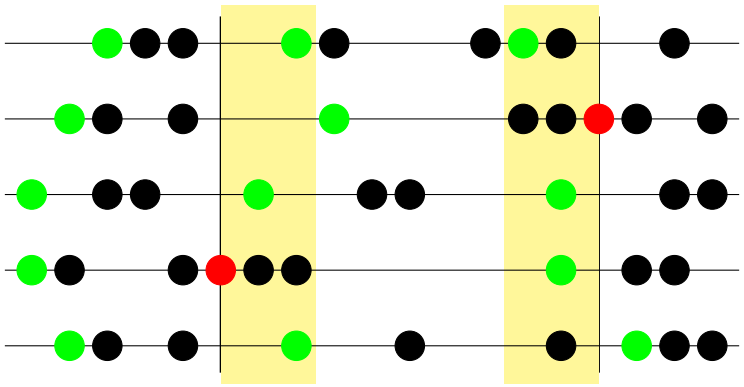

Figura 4.1: Ogni riga orizzontale rappresenta un run ordinato, e i cerchietti gli elementi di ogni run. Cerchietti rossi e verdi rappresentano rispettivamente gli elementi di R e G, cerchietti neri i restanti elementi senza colore. Sono inoltre raffigurati in giallo i bordi definiti dalla posizione degli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

La prima cosa da fare è estrarre tutti i link nelle pagine Web. Definiamo una funzione Parser che presa una pagina web restituisce tutti i link presenti in essa.

Algoritmo 5 Funzione Map - Creazione coppie dei link

```
1: function Map(W)
2:     for each p in W do
3:         links ← Parser (p)
4:         for each l in links do
5:             create ⟨l, 1⟩
```

La funzione Map prende la lista delle pagine Web, estrae da ogni pagina tutti i suoi link e crea una coppia per ognuno di essi con associata la costante 1, la quale serve a indicare che è presente un link con quel identificativo.

Prima di richiamare la funzione Reduce, per il paradigma MapReduce, le coppie sono ordinate e suddivise per il campo chiave in modo tale da distribuire al meglio il carico di lavoro sulle macchine.

Algoritmo 6 Funzione Reduce - Creazione coppie numero link

```
1: function Reduce(key, list)
2:     sum ← 0
3:     for each p in list do
4:         sum++
5:     create ⟨sum, 1⟩
```

Poiché le coppie sono state raggruppate per link, quando passiamo un gruppo alla funzione Reduce questa calcola il numero di link che ha quella pagina in ingresso. La funzione restituisce una coppia che indica che c'è una pagina che ha un certo numero di link in ingresso.

Per ottenere la distribuzione dei gradi in ingresso delle pagine Web, applichiamo una nuova funzione REDUCE alle coppie precedentemente create (prima della funzione le coppie sono ordinate e suddivise in gruppi come al passo precedente).

Algoritmo 7 Funzione Reduce - Creazione gradi in ingresso

```
1: function REDUCE(key, list)
2:     sum ← 0
3:     for each p in list do
4:         sum++
5:     create ⟨key,sum⟩
```

In conclusione bisogna eseguire una MAP per trovare i link presenti nelle pagine Web e due funzione REDUCE, una per contare il numero di link che ha in ingresso una pagina e una per calcolare la distribuzione dei gradi in ingresso.

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.*

Per consentire la navigazione nel layout implicito è necessaro costruire, durante la fase di archiviazione dell'albero, una tabella di dimensione $O(\log n)$, in cui per ogni profondità d, memorizziamo:

- B[d]: la dimensione di ogni bottom tree avente radice a profondità d;
- T[d]: la dimensione del corrispondente top tree;
- D[d]: la profondità della radice di tale top tree.

Inoltre quando effettuiamo una ricerca (a partire dalla radice) di un nodo v di profondità d, teniamo traccia di:

- i: la posizione che v occuperebbe in una memorizzazione BFS;
- Pos[j]: la posizione nella memorizzazione vEB del particolare nodo z a profondità j che ho incontrato durante la ricerca.

Consideriamo la rappresentazione binaria di i: letti da sinistra verso destra, i suoi bit rappresentano le svolte effettuate nella ricerca di v nella rappresentazione BFS (secondo la regola: 0 svolta a sinistra nella BFS, 1 svolta a destra; nota: il bit più a sinistra ovviamente vale sempre 1, e indica il fatto che si parte con la radice).
Consideriamo una suddivisione in cui il nodo v cercato è radice di un bottom tree. Allora se il corrispondente top tree è alto k, cioè $\mathrm{T}[d] = 2^k - 1$, i k bit più a destra di i rappresentano l'indice del bottom tree cercato tra tutti quelli della suddivisione in questione. Poiché tutti i bit di T[d] (in totale k) valgono 1, posso calcolare questo indice come i & T[d].
Vale allora

$$\mathrm{Pos}[d] = \mathrm{Pos}(\mathrm{D}[d]) + \mathrm{T}[d] + (i \ \& \ \mathrm{T}[d])\mathrm{B}[d],$$

ossia:

- Parto da Pos(D[d]), posizione della radice del top tree; dopodiché avanzo di:
- Tanti nodi quanti ve ne sono nel top tree, cioé T[d];
- Tanti nodi quandi ve ne sono in ogni bottom tree, cioé B[d], e faccio questo tante volte quanti sono i bottom tree che precedono quello di radice v ossia, poiché gli indici partono da 0, i & T[d].

Usando ripetutamente questa formula è dunque possibile effettuare la ricerca di un qualsiasi nodo v.

Problema 7

Layout di alberi binari

Problema. *Proporre una paginazione di un generico albero binario in blocchi di dimensione B per ottenere un layout in memoria esterna in cui un cammino radice-foglia attraversi $O(h \log B)$ pagine, dove h è l'altezza dell'albero. Opzionale: per chi vuole, esiste una versione più impegnativa di questo esercizio dove un cammino radice-nodo di lunghezza l attraversa $O(\frac{l}{\log B})$ pagine; contattarmi per discutere questa opzione.*

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.*

Diamo di seguito un esempio di applicazione dell'algoritmo Dc3 alla stringa `"abracadabra"`. Per ragioni tecniche aggiungiamo tre caratteri speciali `$` in fondo alla stringa, ottenendo `"abracadabra$$$"`. Scandiamo la stringa e sostituiamo ogni carattere con il proprio rango nell'ordine alfabetico dei caratteri della stringa stessa, con la convenzione che il carattere `$` precede ogni altro carattere. Otteniamo dunque:

Carattere	a	b	r	a	c	a	d	a	b	r	a	$	$	$
Rango del carattere	1	2	5	1	3	1	4	1	2	5	1	0	0	0
Indice del suffisso	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Dividiamo i caratteri in 3 gruppi secondo la congruenza modulo 3 della posizione nella stringa. Assegnamo rispettivamente i colori verde, rosso e giallo alle 3 classi di congruenza:

Carattere	a	b	r	a	c	a	d	a	b	r	a	$	$	$
Rango del carattere	1	2	5	1	3	1	4	1	2	5	1	0	0	0
Indice del suffisso	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Trascuriamo per il momento il gruppo verde e sostituiamo ogni carattere di indice i dei gruppi giallo e rosso con la tripla dei caratteri di indice $i, i+1$ e $i+2$. Riordiniamo inoltre la tabella secondo la divisione in colori, tralasciando le triple che cominciano per il simbolo `$`:

	Gruppo 0	Gruppo 1			Gruppo 2			
Caratteri								
Indice del suffisso		1	4	7	10	2	5	8	11

Problema 9

Famiglia di funzioni hash uniformi

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

Dobbiamo mostrare che, preso $h \in H$ e dati $k \neq l$ chiavi, u, v valori, vale

$$\Pr[h(k) = u \wedge h(l) = v] \approx \Pr[h(k) = u] \cdot \Pr[h(l) = v.]$$

Mostreremo che $\Pr[h(k) = u] \approx \Pr[h(l) = v] \approx \frac{1}{m}$ e che $\Pr[h(k) = u \wedge h(l) = v] \approx \frac{1}{m^2}$.

Contiamo anzitutto quante sono le chiavi k tali che $h(k) = u$ per un certo valore u.
Dato un generico r, l'equazione $ax + b = r \pmod p$ ha un'unica soluzione x in $[p]$ in quando $a \neq 0$ è sicuramente invertibile. Questo significa che esiste un unico k tale che $ak + b = r \pmod p$.
Se vogliamo inoltre che $r = u \pmod m$, le scelte possibili per r sono almeno $\left\lfloor \frac{p}{m} \right\rfloor$ e al più $\left\lceil \frac{p}{m} \right\rceil$.
Pertanto, dato u, se chiamiamo t il numero di possibili valori per k tali che $h(k) = u$, abbiamo $\left\lfloor \frac{p}{m} \right\rfloor \leq t \leq \left\lceil \frac{p}{m} \right\rceil$, ossia

$$\frac{\left\lfloor \frac{p}{m} \right\rfloor}{p} \leq \Pr[h(k) = u] \leq \frac{\left\lceil \frac{p}{m} \right\rceil}{p},$$

da cui

$$\Pr[h(k) = u] \approx \frac{1}{m}.$$

Consideriamo ora $\Pr[h(k) = u \wedge h(l) = v]$. Se vogliamo che $h(k) = u$ sono possibili per k t valori distinti, con t come sopra, e altrettanti per l affinché sia $h(l) = v$. In totale le coppie (k, l) possibili sono $p(p-1)$, essendo $h \neq k$, da cui

$$\frac{\left(\left\lfloor \frac{p}{m} \right\rfloor\right)^2}{p(p-1)} \leq \Pr[h(k) = u \wedge h(l) = v] \leq \frac{\left(\left\lceil \frac{p}{m} \right\rceil\right)^2}{p(p-1)},$$

da cui

$$\Pr[h(k) = u \wedge h(l) = v] \approx \frac{1}{m^2},$$

come volevasi dimostrare.

Problema 10

Count-min sketch: estensione

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

Lemma 3 (Approssimazione)**.** *Per il count min sketch a valori arbitrari vale:*

$$Pr\left[F[i] - 3\varepsilon\,||F|| \leq \tilde{F}[i] \leq F[i] + 3\varepsilon\,||F||\right] \geq 1 - \delta^{\frac{1}{4}}$$

con $\tilde{F}[i] = median_j\{T[j, h_j(i)]\}$.

Dimostrazione. *Osserviamo innanzi tutto che* $\tilde{F}[i] = T[\hat{i}, h_{\hat{i}}(i)]$ *per qualche* $\hat{i}$*, e vale*

$$\tilde{F}[i] = F[i] + \sum_{k=1}^{n} I_{\hat{i},i,k} F[k]$$

dove $I_{j,i,k}$ *è la variabile indicatrice così definita:*

$$I_{j,i,k} = \begin{cases} 1 & se\ i \neq k \wedge h_j(i) = h_j(k) \\ 0 & altrimenti \end{cases}$$

Ponendo $X_{j,i} = \sum_{k=1}^{n} I_{j,i,k} F[k]$ *abbiamo:*

$$\begin{aligned}
&Pr\left[F[i] - 3\varepsilon\,||F|| \leq \tilde{F}[i] \leq F[i] + 3\varepsilon\,||F||\right] \\
&= Pr\left[F[i] - 3\varepsilon\,||F|| \leq F[i] + \sum_{k=1}^{n} I_{\hat{i},i,k} F[k] \leq F[i] + 3\varepsilon\,||F||\right] \\
&= Pr\left[F[i] - 3\varepsilon\,||F|| \leq F[i] + X_{\hat{i},i} \leq F[i] + 3\varepsilon\,||F||\right] \\
&= Pr\left[-3\varepsilon\,||F|| \leq X_{\hat{i},i} \leq 3\varepsilon\,||F||\right] \\
&= Pr\left[|X_{\hat{i},i}| \leq 3\varepsilon\,||F||\right]
\end{aligned}$$

$$
\begin{aligned}
E[I_{j,i,k}] &= Pr\left[i \neq k \wedge h_j(i) = h_j(k)\right] \\
&= Pr\left[\cup_{a=1}^{c} i \neq k \wedge h_j(i) = a \wedge h_j(k) = a\right] \\
&= \sum_{a=1}^{c} \underbrace{Pr\left[i \neq k \wedge h_j(i) = a \wedge h_j(k) = a\right]}_{\textit{Sono funzioni di hash uniformi}} \\
&\leq \sum_{a=1}^{c} \frac{1}{c} = \frac{c}{c^2} = \frac{1}{c} = \frac{\varepsilon}{e}
\end{aligned}
$$

da cui

$$
\begin{aligned}
E[|X_{j,i}|] &= E\left[|\sum_{k=1}^{n} I_{j,i,k} F[k]|\right] \\
&\leq \sum_{k=1}^{n} E[|I_{j,i,k}|]|F[k]| = \frac{\varepsilon}{e} \sum_{k=1}^{n} |F[k]|] = \frac{\varepsilon}{e} ||F||
\end{aligned}
$$

e quindi per ogni j, usando la disuguaglianza di Markov

$$
Pr\left[|X_{j,i}| \geq 3\varepsilon ||F||\right] \leq \frac{E[|X_{j,i}|]}{3\varepsilon ||F||} = \frac{\frac{\varepsilon}{e} ||F||}{3\varepsilon ||F||} = \frac{1}{3e}
$$

Notiamo che perché valga $|X_{\hat{i},i}| \geq 3\varepsilon ||F||$, deve valere $|X_{j,i}| \geq 3\varepsilon ||F||$, per questo scopo introduciamo le seguenti variabili casuali:

$$
Y_j = \begin{cases} 1 & \textit{se } |X_{j,i}| \geq 3\varepsilon ||F|| \\ 0 & \textit{altrimenti} \end{cases}
$$

Se poniamo $p = E[Y_j] = Pr[|X_{j,i}| \geq 3\varepsilon ||F||\,|]$, e $Y = Y_1 + \cdots + Y_r$, abbiamo $E[Y] = rp$. Noi vogliamo $Y \geq \frac{r}{2}$, per calcolarne la probabilità dobbiamo introdurre la disuguaglianza di Chernoff.

Definizione (Chernoff bound)**.** *Se $X_1, \ldots, X_n$ sono n prove di Poisson indipendenti, identicamente distribuite, ossia $\forall i. P[X_i = 1] = p, P[X_i = 0] = 1 - p$, se prendiamo $X = \sum_{i=1}^{n} X_i$ e $\mu = E[X]$, per ogni $\lambda > 0$ si ha:*

$$
Pr\left[X \geq (1+\lambda)\mu\right] < \left(\frac{e^{\lambda}}{(1+\lambda)^{1+\lambda}}\right)^{\mu}
$$

Quindi ponendo $\mu = E[Y] = rp$, $(1+\lambda)\mu = \frac{r}{2} \Rightarrow 1 + \lambda = \frac{1}{2p}$, abbiamo

$$
\begin{aligned}
Pr\left[Y > \frac{r}{2}\right] &< \left(\frac{e^{\lambda}}{(1+\lambda)^{1+\lambda}}\right)^{\mu} = \frac{1}{e^{\mu}} \left(\frac{e}{1+\lambda}\right)^{(1+\lambda)\mu} \\
&= \frac{1}{e^{rp}} \left(\frac{e}{\frac{1}{2p}}\right)^{\frac{r}{2}} = \frac{1}{e^{rp}} (2pe)^{\frac{r}{2}}
\end{aligned}
$$

Quindi se $\frac{1}{e^{rp}}(2pe)^{\frac{r}{2}} \leq \frac{1}{2^{\frac{r}{4}}} (= \delta^{\frac{1}{4}})$ *abbiamo concluso:*

$$\frac{1}{e^{rp}}(2pe)^{\frac{r}{2}} \leq \frac{1}{2^{\frac{r}{4}}} \equiv \underbrace{2^{\frac{r}{4}} \leq e^{rp}\frac{1}{(2pe)^{\frac{r}{2}}}}_{rp \geq 0 \Rightarrow e^{rp} \geq 1} \Leftarrow 2^{\frac{r}{4}} \leq \frac{1}{(2pe)^{\frac{r}{2}}}$$
$$\equiv 2^{\frac{1}{2}} \leq \frac{1}{2pe} \equiv p \leq \frac{1}{2\sqrt{2}e}$$

che vale in quanto $2\sqrt{2}e \sim 7.668$ *e* $p < \frac{1}{8}$. □

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* $\sum_{k=1}^{n} F_a[k] \cdot F_b[k]$*).*

Lemma 4 (Approssimazione)**.** *Per il prodotto scalare, con valori non negativi per* F_a *e* F_b *valgono*

- $a \cdot b \leq a \tilde{\cdot} b$
- $Pr[a \tilde{\cdot} b \leq a \cdot b + \varepsilon \, ||a|| \, ||b||] \geq 1 - \delta$

con $a \tilde{\cdot} b = \min_j \left\{ \sum_{i=1}^{c} T_a[j, h_j(i)] T_b[j, h_j(i)] \right\}$.

Dimostrazione. *Osserviamo innanzi tutto che per un certo* $\hat{i}$ *vale*

$$a \tilde{\cdot} b = \sum_{i=1}^{c} T_a[\hat{i}, h_{\hat{i}}(i)] T_b[\hat{i}, h_{\hat{i}}(i)]$$

e se definiamo la variabile $I_{j,i,k}$ *come*

$$I_{j,i,k} = \begin{cases} 1 & \text{se } i \neq k \wedge h_j(i) = h_j(k) \\ 0 & \text{altrimenti} \end{cases}$$

otteniamo la seguente uguaglianza:

$$a \tilde{\cdot} b = a \cdot b + \sum_{p,q} I_{\hat{i},p,q} F_a[p] F_b[q]$$

pertanto, vista la non negatività degli elementi di F_a *e* F_b *la prima disuguaglianza vale banalmente.*

Per la seconda, ragioniamo col complementare e cerchiamo di calcolare

$$Pr[a \tilde{\cdot} b \geq a \cdot b + \varepsilon \, ||a|| \, ||b||]$$

Che, se poniamo $X_{j,i} = \sum_{p,q} I_{j,p,q} F_a[p] F_b[q]$ *equivale a calcolare*

$$Pr[X_{\hat{i},i} \geq \varepsilon \, ||a|| \, ||b||]$$

Per poterla calcolare ci servono innanzi tutto le speranze di $E[I_{j,i,k}]$ e $E[X_{j,i}]$, date da:

$$\begin{aligned} E[I_{j,i,k}] &= Pr\left[i \neq k \wedge h_j(i) = h_j(k)\right] \\ &= Pr\left[\cup_{a=1}^{c} i \neq k \wedge h_j(i) = a \wedge h_j(k) = a\right] \\ &= \sum_{a=1}^{c} \underbrace{Pr\left[i \neq k \wedge h_j(i) = a \wedge h_j(k) = a\right]}_{\textit{Sono funzioni di hash uniformi}} \\ &\leq \sum_{a=1}^{c} \frac{1}{c} = \frac{c}{c^2} = \frac{1}{c} = \frac{\varepsilon}{e} \end{aligned}$$

$$\begin{aligned} E[X_{j,i}] = E[\sum_{i,k} I_{j,i,k} F_a[i] F_b[k]] &= \sum_{i,k} E[I_{j,i,k}] F_a[i] F_b[k] \\ = \frac{\varepsilon}{e} \sum_{i,k} F_a[i] F_b[k] &\leq \frac{\varepsilon}{e} \left|\left|a\right|\right| \left|\left|b\right|\right| \end{aligned}$$

Si ha quindi, per la disuguaglianza di Markov:

$$Pr[X_{j,i} \geq \varepsilon \left|\left|a\right|\right| \left|\left|b\right|\right|] \leq \frac{E[X_{j,i}]}{\varepsilon \left|\left|a\right|\right| \left|\left|b\right|\right|} = \frac{\frac{\varepsilon}{e} \left|\left|a\right|\right| \left|\left|b\right|\right|}{\varepsilon \left|\left|a\right|\right| \left|\left|b\right|\right|} = \frac{1}{e}$$

Osserviamo che affinché valga $X_{\hat{i},i} \geq \varepsilon \left|\left|a\right|\right| \left|\left|b\right|\right|$, è necessario che valga per tutte le r $X_{j,i}$, in quanto $X_{\hat{i},i}$ è la minima tra queste, trattandosi le $X_{j,i}$ di variabili casuali indipendenti, abbiamo che:

$$Pr[X_{\hat{i},i} \geq \varepsilon \left|\left|a\right|\right| \left|\left|b\right|\right|] < \frac{1}{2^r} = \delta,$$

da cui

$$Pr[X_{\hat{i},i} \leq \varepsilon \left|\left|a\right|\right| \left|\left|b\right|\right|] \geq 1 - \delta.$$

□

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* $\sum_{k=i}^{j} F[k]$*).*

Per risolvere questo problema suddividiamo ogni range in modo canonico in sottorange di dimensione 2^k, e usiamo $\log n$ count min sketch per memorizzare separatamente i sottorange di dimensione diversa.

La suddivisione avviene in intervalli di tipo $[x2^k + 1, (x + 1)2^k]$ (*diadici*) per $k \in [0, \log n - 1]$, e ogni volta che ci arriva un dato andiamo ad incrementare tutti i count min sketch negli intervalli corrispondenti.

Lemma 5 (Suddivisione canonica)**.** *Qualunque intervallo può essere suddiviso in al più* $2 \log n$ *intervalli diadici.*

Dimostrazione. *Sia* $[l^*, r^*]$ *l'intervallo diadico più grande che può stare dentro il nostro intervallo* $[l, r]$*. Evidentemente vale* $|[l^*, r^*]| < n$ *e quindi l'intervallo diadico è al più della forma* $[x2^{\log n - 1} + 1, (x + 1)2^{\log n - 1}]$*, cioè* $k < \log n$*.*

Consideriamo i due intervalli indotti $[l, x2^k + 1]$ *e* $[(x + 1)2^k, r]$*, ci sono due casi possibili:*

- *uno dei due ha dimensione* $\geq 2^k$*, questo non è un assurdo perché i due intervalli di dimensione* 2^k *potrebbero essere spostati di* 2^k *rispetto all'intervallo diadico canonico più vicino di dimensione* 2^{k+1}*.*

  In questo caso ci limitiamo a togliere l'intervallo di dimensione 2^k *o dal fondo del primo intervallo indotto o dall'inizio del secondo (è semplice immaginare perché si deve trovare proprio lì), e ci riconduciamo al secondo caso.*

- *entrambi gli intervalli indotti hanno dimensione* $< 2^k$*, questo vuol dire che ogni intervallo indotto può contenere al più un intervallo di dimensione* 2^{k-1}*, altrimenti sarebbe di dimensione* 2^k *e quindi assurdo. Inoltre l'intervallo di dimensione* 2^{k-1} *può essere solo in fondo al primo intervallo o in testa al secondo (altrimenti potrebbero essercene due). Quindi nel caso non si trovasse nessun intervallo di dimensione* 2^{k-1} *contenuto vorrebbe dire che lo stesso intervallo indotto ha dimensione* $< 2^{k-1}$*, quindi ripetiamo l'osservazione con* $k - 2$*, e così via.*

Il numero totale di intervalli è quindi dato da $2 \log n$*, in quanto per* k_{max} *ce n'è al più due, in accordo al primo caso, e per valori di k minori ce ne può essere al più uno*

per ogni intervallo indotto, siccome andiamo a togliere sempre dalla testa o dalla coda, siccome $k_{max} < \log n$ abbiamo la tesi.

Lemma 6 (Approssimazione)**.** *Per la range query con valori non negativi valgono le seguenti proprietà:*

- $\mathcal{Q}(l,r) \leq \tilde{\mathcal{Q}}(l,r)$
- $Pr\left[\tilde{\mathcal{Q}}(l,r) \leq \mathcal{Q}(l,r) + 2\varepsilon \log n \, ||F||\right] \geq 1 - \delta.$

Dimostrazione. *Abbiamo*

$$\tilde{\mathcal{Q}}(l,r) = \sum_{k=0}^{\log n - 1} \tilde{\mathcal{Q}}(d_{2^k,i_{k,1}}) + \tilde{\mathcal{Q}}(d_{2^k,i_{k,2}})$$

dove $d_{2^k,i_{k,1}}$ e $d_{2^k,i_{k,1}}$ sono i due intervalli diadici di dimensione 2^k in cui abbiamo scomposto il nostro intervallo $[l,r]$, supponendo per tutti gli intervalli non esistenti di avere $d_{a,b} = []$ e $\mathcal{Q}(d_{a,b}) = 0$.

Inoltre, per ogni intervallo diadico, dalle proprietà dei count min sketch per valori non negativi si ha:

$$\tilde{\mathcal{Q}}(d_{2^k,i}) = \mathcal{Q}(d_{2^k,i}) + \sum_{l=1}^{\frac{n}{2^k}} I_{\hat{i},i,l} \mathcal{Q}(d_{2^k,l})$$

da cui segue la prima disuguaglianza.

Per la seconda disuguaglianza abbiamo, ponendo $X_{j,i} = \sum_{l=1}^{\frac{n}{2^k}} I_{\hat{i},i,l} \mathcal{Q}(d_{2^k,l})$

$$E[X_{j,i}] \leq \frac{\varepsilon}{e} ||F||$$

Quindi, su tutti gli intervalli

$$E[\tilde{\mathcal{Q}}(l,r) - \mathcal{Q}(l,r)] \leq 2\frac{\varepsilon}{e} \log n \, ||F||$$

Da cui, per la disuguaglianza di Markov:

$$\begin{aligned}
&Pr\left[\tilde{\mathcal{Q}}(l,r) \geq \mathcal{Q}(l,r) + 2\varepsilon \log n \, ||F||\right] \\
&= Pr\left[\mathcal{Q}(l,r) + \tilde{\mathcal{Q}}(l,r) - \mathcal{Q}(l,r) \geq \mathcal{Q}(l,r) + 2\varepsilon \log n \, ||F||\right] \\
&= Pr\left[\tilde{\mathcal{Q}}(l,r) - \mathcal{Q}(l,r) \geq 2\varepsilon \log n \, ||F||\right] \\
&\leq \left(\frac{E[\tilde{\mathcal{Q}}(l,r) - \mathcal{Q}(l,r)]}{2\varepsilon \log n \, ||F||}\right)^r \\
&= \left(\frac{2\frac{\varepsilon}{e} \log n}{2\varepsilon \log n \, ||F||}\right)^r = \left(\frac{1}{e}\right)^r < \left(\frac{1}{2}\right)^r = \delta
\end{aligned}$$

da cui segue la tesi.

Attenzione: l'ultimo passo della dimostrazione è falso.

Problema 13

Elementi distinti

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

Cominciamo fissando le notazioni usate nel seguito. Sia $\chi = x_1 x_2 \dots x_n$ uno stream di n caratteri scelti da un insieme di cardinalità $m \leq n$ di simboli distinti. Denotiamo con $D(\chi)$ il numero di elementi distinti effettivamente presenti nello stream.

Per prima cosa risolviamo un problema più semplice: dato un numero t vogliamo decidere con alta probabilià se $D(\chi) \gg t$ oppure $D(\chi) \ll t$. Più precisamente vogliamo risolvere il seguente

Problema (Semplificato)**.** *Data una soglia t e uno stream χ, rispondere*

- *"Sì" se $D(\chi) \geq t$,*
- *"No" se $D(\chi) < \frac{t}{2}$,*
- *indifferentemente "Sì" o "No" altrimenti,*

con probabilità maggiore di $1 - \delta$ di essere corretto.

Supponiamo di avere una funzione hash $h : \chi \to [1, t]$ ideale, tale cioè che $\forall i,\ \Pr[h(x) = i] = \frac{1}{t}$. In termini di questa funzione siamo quasi in grado di risolvere il precedente problema. Abbiamo infatti il seguente

Algoritmo 8 Contatore con rumore

```
1: function NoisyCounter(h)
2:     for each x_i ∈ X do
3:         if h(x_i) = t then
4:             return "Sì"
5:     return "No"
```

Lemma 7. *Supponiamo che $D(\chi) \geq t$ o $D(\chi) < \frac{t}{2}$. Allora il precedente algoritmo è corretto con probabilità maggiore di* 0.6*.*

Dimostrazione. Osserviamo che basta stimare la probabilità di rispondere "No". Infatti questa sarà la probabilità d'errore qualora $D(\chi)$ fosse maggiore di t e la probabilità di successo nell'altro caso.

- Supponiamo $D(\chi) \geq t$. Allora la probabilità di fallimento è la probabilità che $D(\chi)$ volte la funzione hash sia diversa da t, il che accade con probabilità $\Pr[h(x_i) \neq t] = 1 - \frac{1}{t}$. Dunque possiamo scrivere

$$\begin{aligned}\Pr[\text{fallimento}] &= \overbrace{\left(1-\frac{1}{t}\right)\left(1-\frac{1}{t}\right)\cdots\left(1-\frac{1}{t}\right)}^{D(\chi) \text{ volte}} \\ &= \left(1-\frac{1}{t}\right)^{D(\chi)} \leq \left(1-\frac{1}{t}\right)^{t} < \frac{1}{e} \approx 0.37,\end{aligned}$$

  di conseguenza

$$\Pr[\text{successo}] = 1 - \Pr[\text{fallimento}] \approx 0.63.$$

- Supponiamo invece $D(\chi) < \frac{t}{2}$. La probabilità di successo è ancora la probabilità che $D(\chi)$ volte la funzione hash sia diversa da t, dunque

$$\begin{aligned}\Pr[\text{successo}] &= \overbrace{\left(1-\frac{1}{t}\right)\left(1-\frac{1}{t}\right)\cdots\left(1-\frac{1}{t}\right)}^{D(\chi) \text{ volte}} \\ &= \left(1-\frac{1}{t}\right)^{D(\chi)} \geq \left(1-\frac{1}{t}\right)^{\frac{t}{2}} > \frac{1}{\sqrt{e}} \approx 0.60.\end{aligned}$$

Otteniamo quindi un algoritmo che fornisce la risposta corretta in più del 60% dei casi. □

Possiamo superare qualunque soglia di confidenza semplicemente ripetendo abbastanza volte il precedente algoritmo e scegliendo il risultato ottenuto più frequentemente. Sia dunque $H = \{h_j \mid j \in [k]\}$ una famiglia di funzioni hash ideali e indipendenti.

Algoritmo 9 Referendum di NoisyCounter

```
1: yes_votes = 0
2: for each h_j ∈ H do
3:     if NoisyCounter(h_j) = "Sì" then
4:         yes_votes++
5: if yes_votes > k/2 then
6:     return "Sì"
7: return "No"
```

Lemma 8. *$\forall\, \delta > 0$, il precedente algoritmo risolve il problema semplificato con probabilità di essere corretto $1 - \delta$.*

Dimostrazione. Sia Z_i l'indicatrice dell'evento $\{\text{NoisyCounter}(h_j) = \text{“Sì”}\}$. Il precedente lemma garantisce che $\Pr[Z_i] \geq 0.6$. Sia allora $Z = \sum_{i=1}^{k} Z_i$. Osserviamo che dalla linearità della speranza segue immediatamente che $\mathbb{E}[Z] \geq 0.6k$. Di conseguenza abbiamo:

$$\Pr\left[Z \leq \frac{k}{2}\right] \leq \Pr\left[Z \leq \frac{0.5}{0.6}\mathbb{E}[Z]\right].$$

Possiamo maggiorare il membro di destra con una delle disuguaglianze di Chernoff, cioè:

$$\Pr[Z \leq (1-\xi)\mathbb{E}[Z]] \leq \exp\left(-\frac{\mathbb{E}[Z]\cdot\xi^2}{2}\right).$$

Abbiamo dunque, ponendo $1-\xi = \frac{0.5}{0.6}$, che:

$$Pr\left[Z \leq \frac{k}{2}\right] \leq \exp\left(-\frac{0.6k\cdot\frac{0.1}{0.6}^2}{2}\right),$$

perciò possiamo rendere arbitrariamente piccola la probabilità di rispondere "No". Di conseguenza, ragionando come nella dimostrazione del precedente lemma, possiamo ottenere un algoritmo corretto con probabilità $1-\delta$ scegliendo $k = O\left(\log\frac{1}{\delta}\right)$. □

Dalla soluzione del problema semplificato ricaviamo una soluzione del problema originario scegliendo come soglie le potenze di due più piccole di n.

Fissiamo $\delta > 0$. Poiché basta soltanto un bit per memorizzare l'informazione "è stata superata l'i-esima soglia", è sufficiente inizializzare $\log n$ bit a zero corrispondenti alle potenze di due in ordine cresecente e scegliere k funzioni hash ideali e indipendenti in modo che $k = O\left(\log\frac{1}{\delta}\right)$. Consumiamo uno a uno i caratteri dello stream χ e aggiorniamo con il precedente algoritmo tutti i bit corrispondenti alle soglie; al termine dello stream ritorniamo l'indice p del bit più a sinistra impostato a 0.

L'analisi fin qui effettuata garantisce allora che il numero di elementi distinti $D(\chi)$ sia compreso fra 2^{p-1} e 2^p con probabilità maggiore di $1-\delta$.

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 10 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 9. *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:*

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

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

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

- *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:

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

□

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⇔esiste un percorso di lunghezza l tra i e i* otteniamo:

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

da cui:

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

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).*

L'algoritmo per l'inserimento è il seguente:

Algoritmo 11 Inserimento in un albero random di ricerca binario

```
1: function INSERT(x, T)
2:     n ← T.size
3:     r ← random(0, n)
4:     if r == n then
5:         insert_root(x, T)
6:     if x < T.key then
7:         insert(x, T.left)
8:     else
9:         insert(x, T.right)
```

a questo punto si pone il prolbema di inserire l'elemento nel sottoalbero che lo dovrà contenere in modo da mantenere l'integrità della struttura, per fare questo usiamo il seguente algoritmo, che separa gli elementi più piccoli di x e quelli più grandi di x, e al posto del sottoalbero di partenza mette un sottoalbero con radice x, figlio sinistro il sottoalbero contenente tutti valori più piccoli di x e figlio destro contenente tutti quelli più grandi: begin

Algoritmo 12 Inserimento nella radice di un sottoalbero

```
1: function INSERT_ROOT(x, T)
2:     rbintree S, G
3:     split(x, T, &S, &G)
4:     T ← new_node()
5:     T.left = S, T.right = G
6:     return T
7: function SPLIT(x, T, L, R)
8:     if empty(T) then
9:         *L ← empty_tree(), *R ← empty_tree()
10:        return
11:    if x < T.key then
12:        *R ← T
13:        split(x, T.left, L, &(*R.left))
14:    else
15:        *L ← T
16:        split(x, T.right, &(*L.right), R)
17:    return
```

Problema 16

Lista invertita compressa

Problema. *Prendiamo una sequenza ordinata crescente di n interi $i_1, i_2, \ldots, i_n$, come per esempio una lista invertita. La rappresentazione compressa differenziale è la sequenza S di $|S|$ bit ottenuti concatenando $\gamma(i_1), \gamma(i_2 - i_1), \ldots, \gamma(i_n - i_{n-1})$, 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 $j \in [2 \ldots n]$, 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.*

Dato un testo, costruiamo il codice di Huffman come segue

Algoritmo 13 Algoritmo per la costruzione del prefix tree del codice di Huffman

1: scorro la stringa mantenendo tante triple della forma $< p, t >$ per ogni carattere c, dove c rappresenta il carattere, p la probabilità empirica per c e t è l'albero associato al carattere (in questo caso una foglia contenente come valore c).
2: **while** ci sono triple **do**
3: prendo le due triple $< p1, t1 >$, $< p2, t2 >$ con minor valore di p
4: creo la nuova tripla $< p', t' >$
5: $p' \leftarrow p1 + p2$
6: $t'.left \leftarrow t1$
7: $t'.right \leftarrow t2$

Per ottenere i codici associati ai caratteri è sufficiente visitare l'albero, associando ad una ricorsione sul figlio sinistro l'inserimento nel codice di un bit 0 e 1 nel caso del figlio destro, o viceversa.

Si nota che le righe 6 e 7 potevano tranquillamente avere i valori invertiti, in quanto non cambierebbe né la lunghezza dei codici ottenuti, né la proprietà di prefix free, in quanto i valori codificati restano sempre sulle foglie.

Lemma 10 (Ottimalità)**.** *Sia H l'albero ottenuto con l'algoritmo precedente, e sia T un qualunque altro albero con la stessa struttura ma con le foglie permutate. Vale*

$$P(H) \leq P(T), \text{ con } P(X) = \sum_{c \in \Sigma} l_{X,c}\, p_c$$

Dimostrazione. *È sufficiente mostrare che scambiando due foglie in H otteniamo un valore più grande di $P(H)$.*

Siano l_a, p_a, l_b, p_b lunghezza e probabilità associate a due caratteri a e b, vogliamo dimostrare $l_a p_a + l_b p_b \leq l_a p_b + l_b p_a$.

$$
\begin{aligned}
&l_a p_a + l_b p_b \leq l_a p_b + l_b p_a \\
&(l_a - l_b) p_a + (l_b - l_a) p_b \leq 0 \\
&(l_a - l_b) p_a - (l_a - l_b) p_b \leq 0 \\
&(l_a - l_b)(p_a - p_b) \leq 0
\end{aligned}
$$

che è sempre vero perché $p_a \leq p_b \Rightarrow l_b \leq l_a$. □

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.*

Problema 20

Approssimazione per MIN-VC

Problema. *Il problema del MIN-VC (minimum vertex-cover) richiede, per un grafo $G = (V, E)$, di trovare un sottoinsieme $S \subseteq V$ di cardinalità minima tale che ogni arco sia incidente ad almeno un vertice di S, cioè per ogni $(u, v) \in E$ vale $u \in S$ oppure $v \in S$. Mostrare come il seguente approccio greedy fornisca una 2-approssimazione: inizializza S all'insieme vuoto e, per ogni arco $(u, v) \in E$, se u e v non sono entrambi marcati, allora marcali e aggiungili a $S := S \cup \{u, v\}$; altrimenti, scarta l'arco.*

Siano A e C gli insiemi rispettivamente di vertici e di archi selezionati dall'algoritmo; sia inoltre C^* l'insieme di vertici nella soluzione ottima.
Chiaramente C^* deve coprire gli archi di A, dunque deve contenere almeno un vertice per ogni arco in A; per di più due qualsiasi archi di A non hanno vertici in comune, pertanto ciascun vertice di C^* copre al più un arco di A, da cui

$$|C^*| \geq |A|.$$

Inoltre il nostro algoritmo seleziona un arco solo se nessuno dei due vertici sta già in C, dunque per ogni arco inserito in A aggiunge due vertici a C, ossia

$$|C| = 2|A|.$$

Unendo questi due risultati abbiamo che

$$|C| = 2|A| \leq 2|C^*|,$$

come volevasi dimostrare.

Problema 21

Approssimazione per MAX-SAT

Problema. *Per il problema MAX-SAT della soddisfacibilità di una formula booleana, si consideri il seguente algoritmo di approssimazione per massimizzare il numero di clausole soddisfatte in una data formula: Sia F la formula data, $x_1, x_2, \ldots, x_n$ le variabili booleane in essa contenute, e $c_1, c_2, \ldots c_m$ le sue clausole. Scegli i valori booleani casuali $b_1, b_2, \ldots, b_n$, ossia ciascun $b_i \in \{0, 1\}$ $(1 \leq i \leq n)$. Calcola il numero m_0 di clausole soddisfatte dall'assegnamento tale che $x_i := b_i$ $(1 \leq i \leq n)$. Calcola il numero m_1 di clausole soddisfatte dall'assegnamento tale che $x_i := \bar{b_i}$ $(1 \leq i \leq n)$, dove $\bar{b_i}$ indica la negazione di b_i. Se $m_0 > m_1$, restituisci l'assegnamento $x_i := b_i$ $(1 \leq i \leq n)$; altrimenti, restituisci l'assegnamento $x_i := \bar{b_i}$ $(1 \leq i \leq n)$. Dimostrare che il suddetto algoritmo è una r-approssimazione per MAX-SAT, indicando anche il valore di $r > 1$ (e motivando l'utilizzo di tale valore). Discutere se, in generale, la scelta di $b1, b2, \ldots, bn$ possa influenzare o meno il valore di r, motivando le argomentazioni addotte. Facoltativo: creare un'istanza di MAX-SAT in cui il suddetto algoritmo ottiene un costo che è r volte più piccolo del costo ottimo per una data scelta dei valori di $b_1, b_2 \ldots, b_n$.*

Problema 22

Approssimazione per MAX-CUT

Problema. *Il problema MAX-CUT è NP-hard ed è definito come segue per un grafo non orientato $G = (V, E)$. Una partizione di nodi $(C, V - C)$ con $C \in V$ si chiama "cut" o taglio. Un arco $e = (v, w)$ con $v \in C$ e $w \in V - C$ si chiama arco di taglio (ricordando che (v, w) e (w, v) denotano lo stesso arco in un grafo non orientato). Il numero di archi di taglio definisce la dimensione del cut $(C, V - C)$. Poiché cambiando taglio, può cambiare la sua dimensione, il problema richiede di trovare il taglio di dimensione massima e quindi gli archi di taglio corrispondenti. Dimostrare che il seguente algoritmo randomizzato è una 2-approssimazione in valore atteso, ossia che il numero medio di archi di taglio così individuati è in media almeno la metà di quelli del taglio massimo.*

1. *Per ogni nodo $v \in V$, lancia una moneta equiprobabile: se viene testa, inserisci v in C; altrimenti (viene croce), inserisci v in $V - C$.*

2. *Inizializza T all'insieme vuoto. Per ogni arco $(v, w) \in E$, tale che $v \in C$ e $w \in V - C$, aggiungi (v, w) all'insieme T. Restituisci C e T come soluzione approssimata.*

Definiamo la variabile indicatrice $X_{(u,v)}$ che vale 1 se l'arco (u, v) appartiene a T, 0 altrimenti. Allora

$$\Pr[X_{(u,v)} = 1] = \Pr[u \in C \wedge v \in V - C] + \Pr[u \in V - C \wedge v \in C] = \frac{1}{2},$$

da cui $\mathbb{E}[X_{(u,v)}] = \frac{1}{2}$. Definiamo poi $X = \sum_{(u,v)\in E} X_{(u,v)}$. Allora

$$\mathbb{E}[X] = \sum_{(u,v)\in E} \mathbb{E}[X_{(u,v)}] = \frac{1}{2}|E|.$$

Detto T^* il taglio ottimo, si ha ovviamente che $|T^*| \leq |E|$, da cui

$$\mathbb{E}[|T|] = \mathbb{E}[X] = \frac{1}{2}|E| \geq \frac{1}{2}|T^*|,$$

come volevasi dimostrare.