

## Problema 1

### *External memory mergesort*

Nel modello EMM (external memory model), mostrare come implementare il k-way merge ( $(k + 1)B \leq M$ ), ossia la fusione di  $k$  sequenze individualmente ordinate e di lunghezza totale  $N$ , con costo I/O di  $O(N/B)$  dove  $B$  è la dimensione del blocco. Minimizzare e valutare il costo di CPU. Analizzare il costo del merge sort (I/O complexity, CPU complexity) che utilizza tale k-way merge.

Dovendo ordinare una sequenza di elementi la cui dimensione compressiva è superiore a quella della memoria interna e dovendo ottimizzare la complessità di I/O dei trasferimenti con la EMM si procede come segue.

Date  $k$  sequenze individualmente ordinate, dette run, il k-way Merge Sort mantiene in memoria interna  $k$  buffer di input e uno di output, tutti di dimensione  $B$ . Quindi viene selezionato l'elemento minore tra i buffer di input limitando la ricerca al primo elemento di ciascun buffer e successivamente all'individuazione di tale elemento, esso viene spostato nel buffer di output. In caso di saturazione di quest'ultimo si procede alla copia del contenuto in memoria esterna e il buffer di output viene svuotato. Se uno dei  $k$  buffer di input viene consumato completamente, si procede all'inserimento di ulteriori  $B$  elementi copiandoli a partire dal  $k$ -esimo run. Nel caso in cui anche suddetto run fosse ultimato, il buffer di input corrispondente viene lasciato vuoto e marcato come inattivo.

Volendo valutare il costo di I/O del precedente algoritmo si evidenzia che i trasferimenti che avvengono sono pari all'ordine di  $O(N/B)$ . Ogni elemento viene trasferito una volta tra EEM e memoria principale in fase di lettura e una seconda volta quando viene scritto in output, con un costo pari a  $O(1)$ .

Abbiamo supposto che la memoria principale sia abbastanza grande da contenere  $k+1$  blocchi, ossia  $k \leq M/B - 1$ . Questa però è una semplificazione, infatti in questo caso specifico basta eseguire una sola volta il k-way Merge Sort. Diversamente bisogna iterare più volte e per ogni esecuzione il numero di run diminuisce di un fattore  $k$ , dunque viene eseguito  $\log_{M/B} N/M$ .

Quindi la complessità dell'algoritmo nel caso più generale è pari a  $O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{M}\right) = O\left(\frac{N}{B} \log_M \frac{N}{B}\right)$ .

È più delicato valutare il costo in CPU, infatti paghiamo  $O(k)$  confronti per trovare il minimo tra i buffer attivi, ottendendo quindi un costo totale di  $O(Nk)$ . Qualora mantenessimo invece i  $k$  elementi in un Heap pagheremmo un costo totale inferiore e pari a  $O(N \log_2 k)$ .

## Problema 2

### *External memory permuting*

Dati due array  $A$  e  $\pi$ , dove  $A$  contiene  $N$  elementi (non importa quali) e  $\pi$  contiene una permutazione di  $\{1, \dots, N\}$ , descrivere e analizzare nel modello EMM un algoritmo ottimo per costruire un terzo array  $C$  di  $N$  elementi tale che  $C[\pi[i]] = A[i]$  per  $1 \leq i \leq N$ .

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

Valutato il minimo tra  $N$  e  $\text{sort}(n)$ , si procede, nel primo caso, utilizzando un algoritmo banale che semplicemente esegue un'iterazione sugli elementi e scrive la permutazione.

Nel secondo, invece, i passi da seguire sono i seguenti:

1. Creare le coppie  $\langle i, \pi[i] \rangle$ , ovvero le coppie che mettono in relazione una posizione del vettore  $\pi$  con l'indice di permutazione.
2. Ordinare le coppie in base alla seconda componente (l'indice di permutazione).
3. Sostituire nelle coppie  $\pi[i]$  con  $A[\pi[i]]$ .
4. Ordinare le coppie per la prima componente.
5. Scrivere nel vettore  $C$  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  $O(N/B)$ .
- 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  $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ .
- Nel terzo passo, poiché le coppie sono ordinate per  $\pi[i]$ , si accede al vettore  $A$  in modo sequenziale, quindi il costo di I/O è  $O(N/B)$ .
- Anche per il quarto il costo di I/O è  $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ .
- Per l'ultimo passo il costo di I/O è  $O(N/B)$  in quanto sono letti e scritti  $N$  sequenzialmente.

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

## Problema 3

*Lower bound for permuting*

Estendere l'argomentazione utilizzata per il limite inferiore al problema dell'ordinamento in memoria esterna a quello della permutazione:

dati N elementi e  $e_1, e_2, \dots, e_N$  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  $e\pi[1], e\pi[2], \dots, e\pi[N]$ .

Si vuole dimostrare che il lower bound per la permutazione corrisponde a  $\Omega(\min\{N, \text{sort}(n)\})$ .

L'idea è di partire da t operazioni di input e ottenere  $N!$  permutazioni. Il minimo valore di t che ci garantisce tutte le permutazioni distinte rappresenta il lower bound.

Considerando una nuova operazione di input, possiamo scegliere al più tra N pagine. Distinguiamo quindi due casi:

- L'elemento che ho scelto viene caricato in memoria per la prima volta. Viene eseguita una operazione di I/O per caricare il blocco che lo contiene. Posso quindi permutare in  $B!$  modi i suoi elementi e disporli tra gli elementi già in memoria in al più  $\binom{M}{B}$  modi. Questo ci produce  $N \cdot B! \cdot \binom{M}{B}$
- Se invece l'elemento scelto si trova già in memoria perché ho caricato precedentemente il blocco che lo contiene, gli elementi sono già permutati e quindi non è presente il termine  $B!$ . I blocchi vengono quindi letti  $N/B$  volte.

Quindi dopo t operazioni di I/O otteniamo  $(B!)^{N/B} \cdot (N \cdot \binom{M}{B})^t$ .

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

Adoperiamo la formula di Stirling ( $\log x! = x \log x$ ) e che  $\log \binom{M}{B} = B \log \binom{M}{B}$  e otteniamo.

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

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

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

Se  $\log N \leq B \log \frac{M}{B}$  allora  $t \geq \frac{N \log \frac{N}{B}}{\log N + B \log \frac{M}{B}} = \frac{N \log \frac{N}{B}}{2B \log \frac{M}{B}} = \Omega(n \log_m n)$  con m=M/B ed n=N/B

Se invece  $\log N > B \log \frac{M}{B}$ , ovvero  $B < \sqrt{N}$  allora  $\frac{N \log \frac{N}{B}}{\log N + B \log \frac{M}{B}} = \frac{N \log \frac{N}{B}}{2 \log N} = \frac{N \log N - N \log B}{2 \log N} = \frac{1}{2} \left( N - N \frac{\log B}{\log N} \right) \geq \frac{1}{2} \left( N - \frac{1}{2} N \right) = \Omega(N)$

## Problema 4

*External memory implicit searching*

Dato un array statico A di N chiavi in memoria esterna, descrivere come organizzare le chiavi dentro A permutandole attraverso un opportuno preprocessing, in modo che sia possibile effettuare successivamente la ricerca di una chiave con  $O(\log_B N)$  trasferimenti di blocchi utilizzando soltanto  $O(1)$  blocchi di appoggio, oltre a quelli necessari a memorizzare A. (Chiaramente il tempo di CPU deve rimaner  $O(\log N)$ .) Discutere il costo di tale preprocessing in memoria esterna, che puo` utilizzare  $O(N)$  spazio aggiuntivo (al contrario della ricerca).

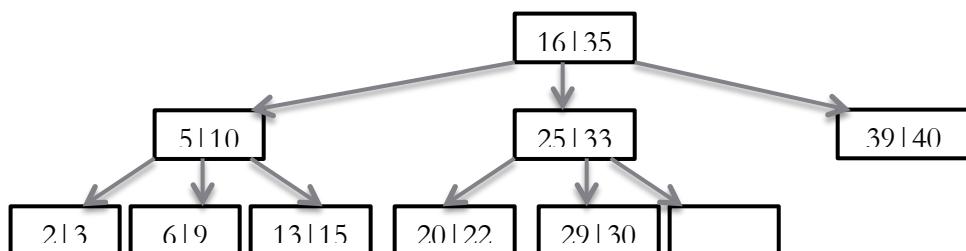
La struttura dati adatta per effettuare una ricerca in tempo  $O(\log_B N)$  è quella denominata B-ary tree. Ovvero una struttura dati simile a quella del ABR, ma i cui nodi contengono B elementi e hanno  $B+1$  figli. Tale struttura viene rappresentata implicitamente con un array per semplicità.

L'albero che andrà virtualmente costruito è di tipo statico, infatti conoscendo N e B sarà possibile sin da subito calcolarne l'altezza.

Fissato a titolo di esempio B pari a 2 ed scelto un array A con 18 elementi si può facilmente calcolare l'altezza dell'albero con la formula  $h = \log_{B+1}(N) + 1 = \log_3(18) + 1 = 3$ .

Il primo passo sarà quello di ordinare l'array in ordine crescente.

A=[2,3,5,6,9,10,13,15,16,20,22,25,29,30,33,35,39,40]



Come è evidente l'albero è completo a sinistra ed è riempito con un particolare ordine che permette, facendo B confronti per nodo attraversato a partire dalla radice, di trovare la chiave cercata in complessità massima  $O(\log_B N)$ .

Conoscendo la struttura dell'albero è possibile facilmente dedurre quanti saranno gli elementi delle foglie sottraendo a N il numero di elementi presenti nei nodi dell'albero. In questo caso si tratta di 2 elementi per la radice e 6 per i nodi centrali, ottenendo quindi 10 elementi da inserire nelle foglie.

Partendo dall'array ordinato scriveremo i primi B elementi nella posizione dell'array implicito che corrisponde alla prima foglia dell'albero. Questo offset corrisponde al valore appena calcolato, pari quindi a  $\sum_{i=0}^{h-1} (B + 1)^i B$ .

Dopo aver scritto B elementi, salveremo in un buffer temporaneo di lunghezza B il successivo elemento che andrà scritto al riempimento del buffer. A questo punto si procede nuovamente con la scrittura di B elementi successivamente a quelli appena scritti. Il successivo elemento va inserito nuovamente nel buffer. Al riempimento del buffer si potrà procedere con la scrittura di questo nell'array nella posizione "padre" rispetto alle foglie appena trattate.

Si procede ricorsivamente fino al completamento dell'array.

B	3B			9B								
	5	10				2	3	6	9	13	15	

Risulta necessario mantenere  $\log_B N$  buffer per poter mantenere i valori da scrivere in un'unica scrittura al fine di ottimizzare le operazioni di I/O.

Il costo di pre-processing per ottenere l'array permutato risulta quindi pari al costo di sorting sommato a quello di scansione lineare dell'array, quindi  $O\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right) + O(N/B) = O\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$

Per quanto riguarda invece la memoria aggiuntiva per la costruzione devo poter avere a disposizione  $\log_B N$  buffer ciascuno di dimensione B per un totale di  $\Omega(B \log_B N)$

Navigare:

Il figlio  $B*i+1+c$  con  $c=[0-B]$

Il padre  $\text{floor}\{(i-1)/k\}$

## Problema 5

*External memory distributin sort using splitters*

Utilizzando il fatto che è possibile trovare  $\sqrt{M/B}$  splitter per un insieme di N elementi con  $O(N/B)$  trasferimenti di blocchi, descrivere come creare  $\sqrt{M/B} + 1$  sottoinsiemi di chiavi, separati dagli splitter.

(Nota: con un solo splitter, questo equivale ad avere la classica partizione del quicksort dell'input in due sottoinsiemi). Utilizzando tale distribuzione, progettare un algoritmo di ordinamento (alternativo al mergesort) in memoria esterna che richieda  $O(N/B \log_{\frac{M}{B}} N/B)$  trasferimenti di blocchi.

Il risultato che si vuole ottenere è quello di avere gli N elementi divisi in gruppi, separati tra loro dagli splitter. Ogni gruppo, noto anche come run, contiene tutti gli elementi di N compresi tra gli splitter in cui lo stesso run è contenuto.

Per ottimizzare i processi di I/O al fine di ottenere quanto sopra si procede con la creazione di  $\sqrt{M/B} + 1$  buffer i quali verranno scritti solo quando pieni. Un ulteriore buffer viene creato e contiene gli elementi di N letti in sequenza. In questo modo, per tutto il buffer di elementi letti da N, confrontiamo con gli splitter trovati e assegnamo agli elementi di N il run relativo. La compressività di esecuzione di questo assegnamento è di  $O(N/B)$  I/O. Infatti, avvengono  $O(N/B)$  letture per leggere gli elementi di N e altrettante per riscriverle nel run corretto.

Al fine di ordinare l'array di N elementi, si procede ricorsivamente sui singoli run fino al raggiungimento della dimensione M (ordinabile in main memory).

Un volta ottenuti run di dimensione M e ordinati singolarmente in main memory sarà possibile procedere alla concatenazione di questi per ottenere un singolo run completamente ordinato.

La ricorsione viene applicata fino a quando i run non sono di dimensione minore o uguale a M

$$\frac{N}{(\sqrt{M/B})^i} \leq M; \frac{N}{M} \leq (\sqrt{\frac{M}{B}})^i; i \geq \log_{\sqrt{\frac{M}{B}}} N/M$$

Quindi, a livello asintotico, la ricorsione avviene  $\log_{\frac{M}{B}} N/M$  volte.

Questo porta a un costo totale pari a  $O(\frac{N}{B} \log_{\frac{M}{B}} N/M) = O(\frac{N}{B} \log_{\frac{M}{B}} N/B)$

## Problema 6

*Number of splits for (a,b)-trees*

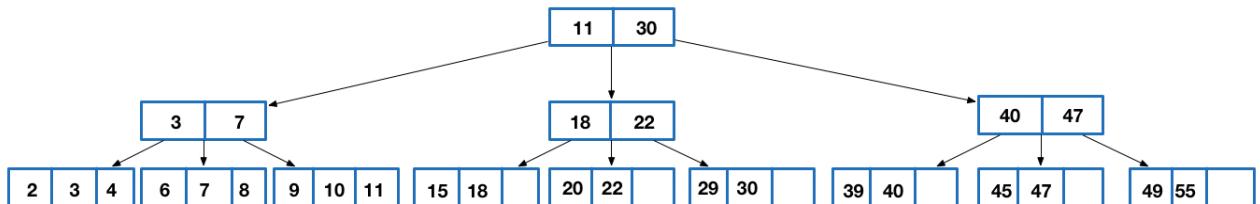
Consider the (a,b)-trees with  $a = 2$  and  $b = 3$ . Describe an example of an (a,b)-tree with  $N$  keys and choose a value of the search key  $k$  such that performing a sequence of  $m$  operations insert( $k$ ), delete( $k$ ), insert( $k$ ), delete( $k$ ), insert( $k$ ), delete( $k$ ), etc. ..., produces  $\Theta(mH)$  split and fuse operations, where  $H = O(\log_b N/b)$  is the height. Try to make the example as general as possible.

After that, consider the (a, b)-trees with  $a = 2$  and  $b = 8$ : produce some examples to check that the above situation cannot occur. Try to guess some properties from the examples using the fact that  $a = b/4$ : if they are convincing, try to prove and use them to show that the situation mentioned above cannot occur, and that the number of split and fuse operations is  $\Theta(m)$ .

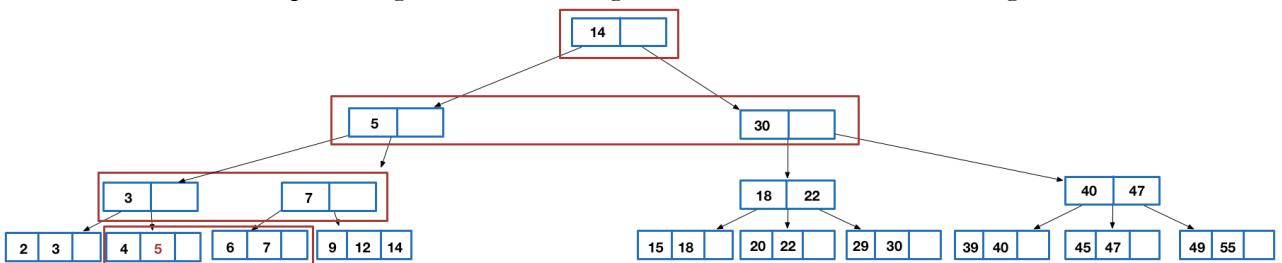
Nel caso in cui  $a$  è pari a 2 e  $b$  a 3 ci si può trovare facilmente nella situazione in cui per un inserimento si verifica uno split a partire dalle foglie fino alla radice e, successivamente, a causa della rimozione della stessa chiave ci si ritrova a dover fare fuse dei nodi relativi al ramo di quella particolare chiave. Questa situazione si verifica nel caso in cui i nodi dell'intero path dalla root e fino alla foglia dove dobbiamo effettuare l'inserimento e le foglie adiacenti alla stessa sono complete.

Questa situazione porta conseguenze, come visto, su tutto il ramo dell'albero di altezza  $h$ , pari a  $\log_B N/B$ . Ripetere quindi  $m$  inserimenti ci costerà in termini di I/O  $O(mH)$ .

L'albero della figura seguente rappresenta un (a,b)-tree per il quale scegliendo un particolare  $k$ , pari a 5 ad esempio, facendo  $m$  operazioni di inserimento e rimozione andremo a pagare  $mH$  split e merge.



Inserendo 5, infatti, il numero di elementi nella foglia diventano 4, numero maggiore di  $b$ , e quindi bisogna procedere con lo split in due foglie distinte di 2 elementi ciascuna. Questo comporta uno split a cascata fino alla root, i quanto ogni nodo interno gestisce il numero massimo di figli.



Il problema più grave si evidenzia quando andiamo a rimuovere 5. Si nota subito che la rimozione causerebbe un merge di tutti i nodi che erano stati precedentemente separati.

Per evitare questo genere di comportamento è evidente che a seguito di un inserimento e conseguente split, procedendo quindi con una rimozione dello stesso, vogliamo trovarci in una situazione "stabile", ovvero in cui il numero di elementi è maggiore di  $a$ . Volendo rappresentare quanto detto con una equazione:  $\lceil \frac{b+1}{2} \rceil - 1 \geq a$ . Volendo generalizzare per  $a$  elementi otteniamo:  $\lceil \frac{b+1}{2} \rceil - a \geq a$ .

Risolvendo ci accorgiamo che  $b \geq 4a - 1$  ci garantisce che questo genere di situazione non si verifichi.

## Problema 7

### 1-D range query

Describe how to perform a one-dimensional range queries in (a, b)- trees with  $a, b = \Theta(B)$ . Given two keys  $k_1 \leq k_2$ , the query asks to report all the keys  $k$  in the (a, b)-tree such that  $k_1 \leq k \leq k_2$ . Give an analysis of the cost of the proposed algorithm, which should be output-sensitive, namely,  $O(\log_B N + R/B)$  block transfers, where  $R$  is the number of reported keys. After that, for a given set of  $N$  keys, describe how to build an (a,b)-tree for them using  $O(\text{sort}(N))$  block transfers, where  $\text{sort}(N) = \Theta(N/B \log M/B N/B)$  is the optimal bound for sorting  $N$  keys in external memory.

Ricercare su un (a,b)-tree è molto simile a ricercare in un albero binario, eccetto che per il numero di confronti da effettuare che in questo caso sono pari al numero di nodi figli. La ricerca prende come input un puntatore al nodo padre e una chiave da ricercare all'interno dell'albero.

La ricerca della singola chiave per un (a,b)-tree ha quindi complessità pari a  $O(h) = O(\log_{B+1}(N) + 1)$ , dove  $h$  è l'altezza dell'albero.

Volendo estendere la ricerca a un range di chiavi comprese tra la chiavi  $k_1$  e  $k_2$  si procede cercando prima  $k_1$  e una volta trovata si procede linearmente con i blocchi successivi fino al raggiungimento della chiave  $k_2$ . Questo è possibile ipotizzando di avere le foglie linkate tra loro oppure di avere salvato le foglie stesse in blocchi di memoria contigui e quindi il costo di I/O aggiuntivo rispetto alla ricerca della singola chiave è pari a  $O(R/B)$ .

Questo risultato porta alla conclusione che il costo totale di I/O per la ricerca di una range di chiavi comprese tra  $k_1$  e  $k_2$  sarà pari a  $O(\log_B N + R/N)$ .

Volendo procedere alla costruzione dell'albero, come è stato già detto precedentemente, è fondamentale avere le foglie ordinate. Sarà quindi possibile procedere con una costruzione bottom-up, ovvero a partire dalle foglie fino al raggiungimento della radice. In questo modo l'unica vera complessità di I/O è dovuta all'ordinamento degli  $N$  elementi foglia e sarà quindi pari a

$$O(\text{sort}(N)) = O\left(\frac{N}{B} \log_{M/B} N/B\right).$$

## Problema 8

### *Suffix sorting in EM*

Using the DC3 algorithm seen in class, and based on a variation of mergesort, design an EM algorithm to build the suffix array for a text of N symbols. The I/O complexity should be the same as that of standard sorting, namely,  $O(N/B \log M/B N/B)$  block transfers.

L'algoritmo DC3 visto a lezione prevede la creazione di un suffix array operando in main memory, volendo estendere tale algoritmo per la memoria esterna si sostituirà il processo di ordinamento con radix sort con un k-way merge sort. Per farlo è necessario modificare opportunamente l'algoritmo. Si parte come di consueto dai trigrammi di tipo 1 e 2 ( $\text{mod } 3 = 1$  &  $\text{mod } 3 = 2$ ) e si crea un array di tuple, ciascuna formata dal trigramma e dalla sua posizione nel testo T:  $\langle 1/2\text{-gramma}, i \rangle$ .

A questo punto si procede con un sort lessicografico in memoria esterna sulla colonna del trigramma e si assegna a ciascuno di essi un “ranking”, non necessariamente univoco, ma anche duplicato come nel caso di due trigrammi uguali. Se ci si trova in questo caso, si procederà ricorsivamente con l'algoritmo sulla stringa T', così costruita:

- Ordino nuovamente le tuple per posizione i
- Genero una stringa di “ranking” del tipo  $S1\$S2$$ , dove S1 sono tutti i rank ordinati degli 1-grammi e S2 quelli dei 2-grammi.

A questo punto ripetono questa procedura fino all'ottenimento di “ranking” distinti per ciascun trigramma, con i quali, una volta completata la procedura di unione con gli 0-grammi e opportunamente ordinati, potrò ottenere il SA' relativo a T'.

Per convertire SA' in SA<sup>12</sup> userò la seguente formula:

$$SA^{12} = \begin{cases} 1 + 3 SA'[i] & \text{se } SA'[i] < n_1 \\ 2 + 3(SA'[i] - n_1) & \text{altrimenti} \end{cases}$$

Dove  $n_1$  è il numero di 1-grammi di T.

A questo punto avrò  $\langle SA^{12}, R^{12} \rangle$ , ordinando per  $SA^{12}$  e prendendo al seconda componente otterrò ISA<sup>12</sup> ovvero l'inverted suffix array.

A questo punto procedo alla costruzione di tuple, più precisamente una quintupla per gli 0-grammi e 2-grammi, una quadrupla per gli 1-grammi. Per farlo scorro il testo e, per ogni tripla, se è uno 0-grammi creo la seguente struttura:

$\langle T[i], T[i+1], i, ISA[i+1], ISA[i+2] \rangle$

dove  $T[i]$  e  $T[i+1]$  sono i primi due caratteri dello 0-gramma,  $i$  è la sua posizione,  $ISA[i+1]$  e  $ISA[i+2]$  sono rispettivamente i rank di 1-gramma e 2-gramma successivi.

Se invece incontro un 1-gramma:

$\langle T[i], i, ISA[i], ISA[i+1] \rangle$

Per il 2-gramma:

$\langle T[i], T[i+1], i, ISA[i], ISA[i+2] \rangle$

Per costruire queste tuple scorro il testo, quindi ho un costo di I/O pari a  $O(N/B)$ , inoltre dovendo inserire anche i rank relativi ai trigrammi 1-2 il costo totale viene incrementato di un ulteriore  $O(N/B)$  visto che ho precedentemente ordinato per posizione i il SA<sup>12</sup>.

Successivamente ordino gli 0-grammi sulla base della prima lettera e, nel caso di lettere uguali, in base a ISA del trigramma successivo (dato contenuto nella tupla).

Le tuple per i trigrammi 1 e 2, che si trovano nella medesima lista, vengono ordinati sulla base del loro stesso ISA.

Una volta ordinati sarà possibile procedere al merge delle due liste confrontando sempre i primi due elementi di ciascuna e prendendo il più piccolo. Si inserisce la posizione i dell'elemento in un nuovo array, alla fine della procedura quello sarà il SA.

Le combinazioni possibili sono le seguenti:

- 0-gramma vs 1-gramma
  - Hanno la prima lettera diversa, si prende il trigramma lessicograficamente più piccolo.
  - Prima lettera uguale, si procede al confronto di  $ISA^0[i+1]$  con  $ISA^2[i+1]$  e si sceglie il più piccolo
- 0-gramma vs 2-gramma
  - Hanno la prima lettera diversa, si prende il trigramma lessicograficamente più piccolo.
  - Prima lettera uguale, si procede al confronto della seconda lettera, se diversa si prende l'elemento lessicograficamente più piccolo, altrimenti si confronta  $ISA^2[i+2]$  con  $ISA^1[i+2]$

## Problema 9

### *Suffix array search*

Suppose to have run the DC3 algorithm on the text  $T$  of  $N$  symbols, so the suffix array  $SA$  of  $N$  entries is available. Design and analyze efficient algorithms to find all the occurrences of a pattern string  $P$  of  $M$  symbols in  $T$ : we say that position  $i$  in  $T$  is an occurrence of  $P$  if the substring  $T[i \dots i+M-1]$  is equal to  $P$  symbol-wise. Note that this is an **on-line query**, meaning that  $T$  and  $SA$  are fixed, while  $P$  is provided each time by the user query and makes use of  $T$  and  $SA$  to list all the occurrences of  $P$ . The complexity of the query algorithm thus proposed should be analyzed and commented in the following settings, assuming that  $P$  can be always kept in main memory: (1) both  $T$  and  $SA$  are in main memory; (2)  $T$  is main memory while  $SA$  is in external memory; (3)  $T$  is in external memory while  $SA$  is in main memory; (4) both  $T$  and  $SA$  are in external memory. Note that (2) and (3) describe a situation in which it is not possible to keep both  $T$  and  $SA$  in main memory, just only one of them.

L'algoritmo di ricerca di un pattern  $P$  di lunghezza  $M$  all'interno di una stringa  $T$  di lunghezza  $N$  è molto simile a quello di ricerca binaria. Avendo a disposizione il SA del testo  $T$ , si procede fissando due puntatori, sinistro ( $l$ ) e destro ( $r$ ), rispettivamente al primo e all'ultimo elemento.

Il passo successivo è quello di dividere in due l'array e confrontare il pattern  $P$  con gli  $M$  elementi del suffisso mediano, ovvero la sottostringa di  $T$  che va da  $i$  a  $i+M-1$  con  $i$  pari a  $SA[(l+r)/2]$ .

In base al risultato del confronto si valuta se prendere in esame la metà sinistra o quella destra del suffix array.

Si procede, quindi, ricorsivamente sulla parte di array scelta fino a quando i puntatori sinistro e destro non coincidono.

A questo punto fissiamo l'elemento trovato come il primo elemento contenente il pattern ricercato. A partire da questo elemento e fino all'ultimo del suffix array si procede nuovamente con la ricerca del pattern con lo stesso algoritmo ma “al contrario” andando alla ricerca dell'ultimo elemento contenente il pattern ricercato.

Trovati il primo e l'ultimo elemento contenenti il pattern, sappiamo che esso è contenuto anche in tutti i suffix compresi tra il risultato delle due esecuzioni.

Pseudo codice:

```
l = 0; r = n
while l < r:
    mid = (l+r) / 2
    if P > suffixAt(A[mid]):
        l = mid + 1
    else:
        r = mid
s = l; r = n
while l < r:
    mid = (l+r) / 2
    if P < suffixAt(A[mid]):
        r = mid
    else:
        l = mid + 1
return (s, r)
```

La complessità dell'algoritmo proposto risulta simile a quella della ricerca binaria. In termini di CPU infatti, ipotizzando costante il confronto del singolo carattere, ci accorgiamo che sono necessari al massimo M confronti tra il pattern P e la sottostringa del testo. Questa operazione di confronti si ripete  $\log N$  volte, per una complessità quindi di  $O(M \log N)$ . L'algoritmo si ripete due volte, per trovare il primo e l'ultimo suffisso, ma la complessità asintotica non cambia.

Analizzando invece la complessità di I/O per i quattro casi proposti, è evidente come nel primo caso, quello in cui T e SA sono in main memory, la complessità risulta costante  $O(1)$ .

Nel secondo caso, in cui solo SA è in memoria esterna, dovendo leggere l'array ci accorgiamo che la complessità di I/O risulta  $O(\log N/B)$ , poiché carichiamo blocchi di dimensione B.

Se invece è T ad essere in memoria esterna, mentre SA si trova in memoria centrale, confrontare il pattern significa confrontare al massimo i suoi M caratteri con altrettanti appartenenti alla sottostringa di T, i quali dovranno essere caricati a blocchi di dimensione B. Questa operazione costa  $M/B$  e la devo ripetere  $\log N$  volte ottenendo quindi  $O(\frac{M}{B} \log N)$ .

Nell'ultimo caso, in cui entrambi SA e T si trovino in memoria esterna, la complessità risulta pari a  $M/B * \log N + \log N/B$ , quindi  $O(M/B * \log N)$ .

## Problema 10

### *Implicit navigation in vEB layout*

Consider  $N = 2^h - 1$  keys where  $h$  is a power of 2, and the implicit cache-oblivious vEB layout of their corresponding complete binary tree, where the keys are suitably permuted and stored in an array of length  $N$  without using pointers (as it happens in the classical implicit binary heap but the rule here is different). The root is in the first position of the array. Find a rule that, given the position of the current node, it is possible to locate in the array the positions of its left and right children. Discuss how to apply this layout to obtain (a) a static binary search tree and (b) a heap data structure, discussing the cache complexity.

Per prima cosa è possibile calcolarsi  $n \% 3$ , con  $n$  la posizione  $n$ -esima nell'array, per capire se l'elemento è un nodo radice di un sottoalbero di dimensione 3. Se il modulo risulta pari a 0 in quel caso i figli sx e dx si trovano rispettivamente alla posizione  $n+1$  e  $n+2$ . In questo caso l'algoritmo termina, altrimenti si procede con la ricerca come segue.

Conoscendo la dimensione dell'array possiamo ottenere l'altezza dell'albero che sarà pari a  $h = \log(N) + 1$ . A questo punto inizializzo a 0 delle variabili che mi serviranno durante l'esecuzione dell'algoritmo: in `root` dove salverò l'indice della radice del sottoalbero dove si trova l'elemento, `offsetSx` che contiene il numero di fratelli sinistri dell'elemento, `jump` dove salvo la dimensione del sottoalbero e `rootTop` dove salvo la `root` quando il taglio mi rende un top-tree.

1. Essendo  $h$  una potenza di due è possibile dividere l'albero in sotto-alberi di altezza  $h/2$ , un top-tree e  $2^{h/2}$  bottom-tree. Questi alberi contengono  $2^{h/2} - 1$  elementi. Per capire a quale sotto-albero appartiene l'elemento dato dividiamo la sua posizione  $n$  per  $2^{h/2} - 1$  (dimensione) e ne prendiamo il floor, chiamiamo questo valore `IDAlbero`. Se il risultato sarà 0 vuol dire che l'elemento si trova nel top-tree, se maggiore di 0 invece in uno dei bottom tree. Dividendo l'array in  $2^{h/2} + 1$  blocchi, sarà possibile prendere in esame solo quella relativa al sotto-albero selezionato.

2. Identificato l'albero dove si trova l'elemento, se è un top-tree effettuo tre operazioni. Imposto a 0 la variabile `offsetSx`, aggiorno la variabile salto alla dimensione del  $2^{h/2} - 1$  e pongo `rootTop=root`. Altrimenti se è uno dei bottom-tree traslo la radice con la formula `root+=IDAlbero*dimensione`, aggiorno l'indice  $n$  rispetto alla `root` con la formula  $n:=(n \% \text{dimensione})$  e aggiungo a `offsetSx+=(IDAlbero-1)*2^{h/2-1}`

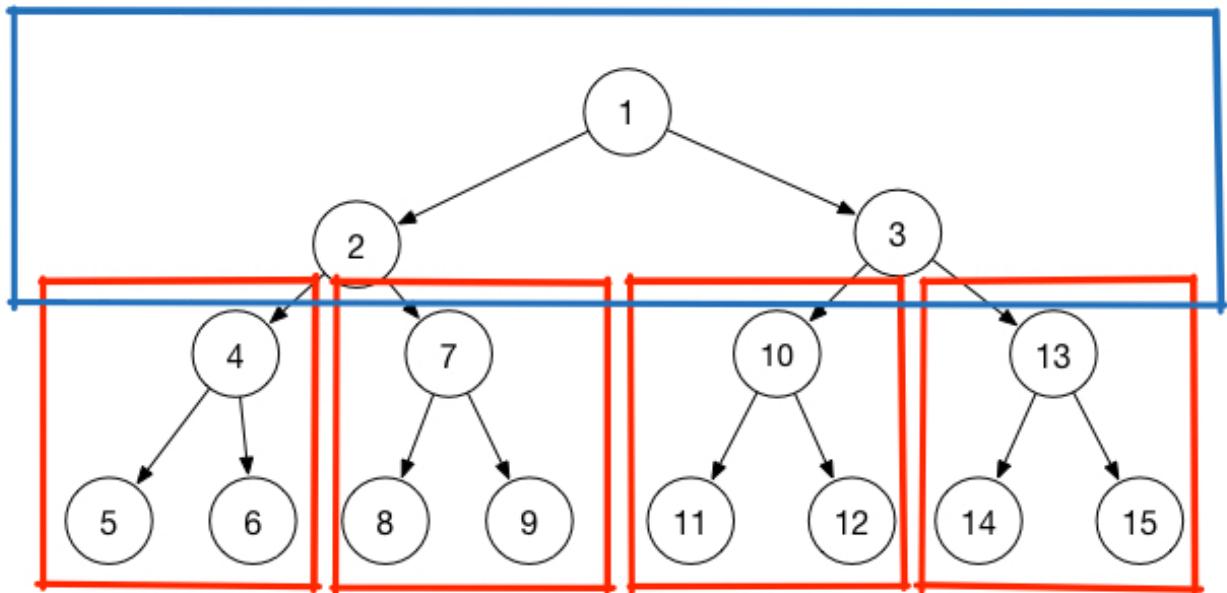
3. A questo punto ripeto ricorsivamente le operazioni 1 e 2 fino a quando non arrivo a un taglio che mi porta ad alberi di altezza  $h=2$ , ovvero sottoalberi di 3 elementi, che considero il caso base. Se il nodo è il figlio sinistro dell'albero di altezza 2 non ha fratelli a sinistra e non incremento `offsetSx`, se è il destro incremento di 1 il contatore.

4. Per concludere, è possibile trovare rispettivamente i figli sinistro e destro con le seguenti formule:  
 $SX = \text{rootTop} + \text{jump} + 2 * \text{offsetSX} * \text{jump}$   
 $DX = \text{rootTop} + \text{jump} + 2 * \text{offsetSX} * \text{jump} + \text{jump}$

È possibile adoperare un albero vEB come albero di ricerca binario. In fase di ricerca vengono attraversati  $h/k$  blocchi per raggiungere una foglia a partire dalla radice, con  $k=\log B$ .

La complessità quindi è pari a  $O(\log N / \log B) = O(\log_B N)$ .

Se invece adoperiamo l'albero vEB come heap, ad ogni rimozione dell'elemento radice sappiamo che dobbiamo sostituire la radice con l'elemento foglia più a destra. A questo punto si procede al ribilanciamento fino a che non viene ristabilita la proprietà di ordinamento, questa operazione avviene sempre e comunque su un solo path radice-foglia e quindi la complessità non cambia,  $O(\log_B N)$ .



## Problema 11

*LCP, suffix arrays, and suffix trees*

Show how to extend the DC3 construction of the suffix array SA so as to compute also the array LCP, where  $LCP[i]$  contains the length of the longest common prefix between the two suffixes indicated by  $SA[i]$  and  $SA[i + 1]$ , for  $1 \leq i \leq N - 1$ . The complexity of the original DC3 algorithm should not change. After that, show how to build the suffix tree in linear time using arrays SA and LCP for the given input text. [Hint: as seen in class, traversing the suffix tree in preorder, the suffixes represented in the leaves are those in the sequence SA and the skip values in the internal nodes are those in LCP. The idea is to build the suffix tree in preorder in which the current rightmost path is maintained, and the new leaf is assigned the next suffix in SA and is attached to the internal node (which is created if needed) in the path, with the next value in LCP as skip value.]

Per estendere l'algoritmo DC3 per la costruzione del SA al fine di calcolare contemporaneamente l'array LCP si procede normalmente con il calcolo del SA<sup>12</sup>. Se necessario, si effettua il passo ricorsivo su  $T'$  al fine di ottenere i rank distinti. A quel punto posso calcolare  $LCP'$  per la stringa  $T'$  confrontando i primi 2 caratteri di ogni trigramma con il suo precedente (non serve confrontarne 3 perché altrimenti il rank non sarebbe distinto). Finita la procedura applicata a  $T'$ , la ricorsione ritorna  $\langle SA', LCP' \rangle$ . Se la procedura ricorsiva non era necessaria si ottiene direttamente  $\langle SA^{12}, LCP^{12} \rangle$  e quindi non è necessario il passo successivo di conversione.

Per convertire  $LCP'$  in  $LCP^{12}$  si usa la seguente formula:

$$LCP^{12}[i] = 3*LCP'[i] + lcp(SA[i] + 3*LCP'[i], SA[i-1] + 3*LCP'[i])$$

Successivamente procedo normalmente con la costruzione di  $SA^0$  e al merge, come da procedura DC3. Ad ogni inserimento calcolo LCP dell'elemento inserito rispetto al precedente secondo i seguenti casi possibili:

- Inserisco 0-suffix preceduto da uno 0-suffix
  - Prima lettera diversa =>  $LCP=0$
  - Prima lettera uguale, RMQ tra gli 1-suffix che seguono entrambi
- Inserisco 0-suffix preceduto da 1-suffix o 2-suffix
  - Prima lettera diversa =>  $LCP=0$
  - Prima lettera uguale
    - 0-suffix preceduto da 1-suffix, RMQ tra 1-suffix e 2-suffix che li seguono
    - 0-suffix preceduto da 2-suffix, spostandomi ai successivi (1-suffix vs 0-suffix) mi ritrovo nella situazione iniziale e ripeto la procedura
- Inserisco 1-suffix o 2-suffix preceduto da 1-suffix o 2-suffix
  - LCP non varia rispetto a  $LCP^{12}$

Per la costruzione del suffix tree a partire dall'array LCP si parte costruendo una radice ed, essendo il primo valore di LCP sempre pari a zero, creando il primo nodo e collegandolo direttamente alla radice. Successivamente si passa al secondo suffisso, si controlla il valore di LCP con il precedente e se pari a zero si collega il nuovo nodo direttamente con la radice, diversamente si crea un nodo fittizio sull'arco che collega la radice e l'ultimo elemento inserito.

Per i successivi si procede nel seguente modo:

- se LCP è uguale a zero si collega in nuovo nodo direttamente alla root
- se LCP è maggiore di zero si risale l'albero a partire dall'ultimo nodo passando per i nodi interni contenenti i valori di LCP. Ad ogni nodo interno si confronta il valore di LCP del suffisso in

esame e se è uguale si collega il nuovo nodo a tale nodo interno. Se diverso si continua a risalire l'albero fino a che non si trovano due nodi tra loro connessi tale per cui il valore di LCP del suffisso è compreso tra quei due valori. In questo caso si procede all'inserimento di un nuovo nodo interno tra i nodi trovati e, a quest'ultimo, si collega il nuovo suffisso appena inserito.

## Problema 12

### *Deterministic data streaming*

Consider a stream of  $n$  items, where items can appear more than once in the stream. The problem is to find the most frequently appearing item in the stream (where ties broken arbitrarily if more than one item satisfies the latter). Suppose that only  $k = O(\log n)$  items can be stored, one item per memory cell, where the available storage is  $k + O(1)$  memory cells. Show that the problem cannot be solved deterministically. Hint: it is an adversarial argument based on the  $k$  items chosen by the hypothetical determinist streaming algorithm, and the fact that there can be a tie on  $> k$  items till the last minute

Avendo a disposizione  $k+1$  celle di memoria, possiamo ipotizzare di avere  $m$  bit per poter memorizzare lo stato dell'algoritmo.

Ipotizzando di inviare almeno  $2^m+1$  input diversi, dove per ogni stream di input inviato si ha un elemento con frequenza massima sempre diverso, possiamo affermare che l'algoritmo si ritroverà nella situazione in cui due stream diversi, quindi con due "most frequent" diversi, vengono mappati in memoria con la stessa configurazione.

A partire quindi dallo stesso stato l'algoritmo darà due risposte diverse, un assurdo per un algoritmo deterministico.

Nel caso invece in cui non dia due risposte distinte, allora una delle due sarà sicuramente sbagliata per ipotesi e quindi l'algoritmo non può essere deterministico.

Lo stream di input sarà formato da  $2^m+1$  caratteri distinti tutti ripetuti una sola volta ad esclusione di uno che si ripeterà due volte diventando quindi il più frequente. Il carattere che si ripete 2 volte sarà diverso per ogni stream, garantendo così sia la condizione di stream tutti diversi l'uno con l'altro che quella di "most frequent diversi".

## Problema 13

$$\{h_{a_0, a_1}\}_{h \in \mathcal{H}} \text{ where } h_{a_0, a_1} = ((a_0 x + a_1) \% p) \% b$$

$$x \in [n]; a_0 \neq 0; a_0, a_1 \in \mathbb{Z}_p; p \text{ prime} \in [b+1 \dots 2b]$$

Prove that  $\mathcal{H}$  is 2-wise independant

$$\text{pr}(h_{a_0, a_1}(x) = z \& h_{a_0, a_1}(y) = w) \cong \text{pr}(h_{a_0, a_1}(x) = z) \cdot \text{pr}(h_{a_0, a_1}(y) = w)$$

Poniamo  $h_{a_0, a_1}(x) = r_{a_0, a_1}(x) \% b$ , dove  $r_{a_0, a_1}(x) = (a_0 x + a_1) \% p$ . Sappiamo che trovare  $\text{pr}(h_{a_0, a_1}(x) = z)$  equivale a  $\text{pr}(r_{a_0, a_1}(x) = z) + \text{pr}(r_{a_0, a_1}(x) = z + b)$  con  $z < p-b$ , equivale invece a  $\text{pr}(r_{a_0, a_1}(x) = z)$  altrimenti.

Per farlo calcoliamo il rapporto tra la somma del numero di soluzioni delle equazioni  $(a_0 x + a_1) = z$  e  $(a_0 x + a_1) = z+b$  (se presente) in  $\mathbb{Z}_p$  e tutte le possibili scelte di  $a_0, a_1$ .

Ponendo  $(a_0 x - z = a_1)$  le possibili scelte di  $a_0$  che soddisfano l'equazione in  $\mathbb{Z}_p$  sono  $p-1$  e altrettante ce ne sono per  $(a_0 x - z - b = a_1)$ . Invece, tutte le possibili scelte di  $a_0$  e  $a_1$  per la generica equazione sono  $p(p-1)$ . Il rapporto produce una probabilità pari a  $2/p$  per  $z < p-b$  e  $1/p$  altrimenti.

Volendo, invece, trovare la  $\text{pr}(h_{a_0, a_1}(x) = z \& h_{a_0, a_1}(y) = w)$  i casi possibili possono essere 1, 2 o 4.

$$1. \quad \begin{cases} r_{a_0, a_1}(x) = z \\ r_{a_0, a_1}(y) = w \end{cases} \quad \text{per } z, w \geq p-b$$

$$2. \quad \begin{cases} r_{a_0, a_1}(x) = z + b \\ r_{a_0, a_1}(y) = w \end{cases} \quad \text{per } z < p-b, w \geq p-b$$

$$3. \quad \begin{cases} r_{a_0, a_1}(x) = z \\ r_{a_0, a_1}(y) = w + b \end{cases} \quad \text{per } z \geq p-b, w < p-b$$

$$4. \quad \begin{cases} r_{a_0, a_1}(x) = z + b \\ r_{a_0, a_1}(y) = w + b \end{cases} \quad \text{per } z, w < p-b$$

Per ogni sistema si ottiene 1 soluzione e la somma soluzioni può essere 1, 2 o 4 in base ai valori di  $z$  e  $w$ .

Le possibili scelte di  $a_0$  e  $a_1$  sono anche in questo caso  $p(p-1)$  e quindi si ottiene una probabilità totale di  $\frac{1}{p(p-1)}$  per  $z, w \geq p-b$ ,  $\frac{2}{p(p-1)}$  per  $z < p-b, w \geq p-b$  e per  $z \geq p-b, w < p-b$ ,  $\frac{4}{p(p-1)}$  per  $z, w < p-b$ .

Si è dimostrato, quindi, che:

$$\text{pr}(h_{a_0, a_1}(x) = z \& h_{a_0, a_1}(y) = w) \cong \text{pr}(h_{a_0, a_1}(x) = z) \cdot \text{pr}(h_{a_0, a_1}(y) = w)$$

## Problema 14

Generalize CM-sketch to  $F[i] := \pm v$ , discuss how the analysis changes w.r.t  $F[i] := \pm 1$

Per il count min sketch a valori arbitrari vale:

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

con  $\widetilde{F}[i] = \text{median}_j\{T[j, h_j(i)]\}$ .

$$\widetilde{F}[i] = F[i] + X_{j,i} = F[i] + \sum_{k=1}^n I_{j,i,k} F[k]$$

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

$$I_{jik} = \begin{cases} 1 & \text{per } k \neq i \text{ e } h_j(i) = h_j(k) \\ 0 & \text{altrimenti} \end{cases}$$

Si ottiene quindi:

$$\begin{aligned} \Pr[F[i] - 3\varepsilon\|F\| \leq F[i] + X_{j,i} \leq F[i] + 3\varepsilon\|F\|] &= \\ &= \Pr[-3\varepsilon\|F\| \leq X_{j,i} \leq 3\varepsilon\|F\|] = \\ &= \Pr[|X_{j,i}| \leq 3\varepsilon\|F\|] \geq 1 - \delta^{\frac{1}{4}} \end{aligned}$$

Invertiamo la disegualanza per poter utilizzare la disegualanza di Markov:

$$\Pr[|X_{j,i}| > 3\varepsilon\|F\|] < \delta^{\frac{1}{4}}$$

Otteniamo quindi:

$$\Pr[|X_{j,i}| > 3\varepsilon\|F\|] = \frac{E[X_{j,i}]}{3\varepsilon\|F\|}$$

$$\text{con } E[X_{j,i}] = E[\sum_{k=1}^n I_{j,i,k} F[k]] = \sum_{k=1}^n F[k] E[I_{j,i,k}]$$

$$E[I_{j,i,k}] = \Pr[i \neq k \wedge h_j(i) = h_j(k)] = \sum_{a=1}^c \Pr[i \neq k \wedge h_j(i) = a \wedge h_j(k) = a]$$

che essendo funzioni hash uniformi

$$\leq \sum_{a=1}^c \frac{1}{c} = \frac{c}{c^2} = \frac{1}{c} = \frac{\varepsilon}{e}$$

Da cui si ottiene:

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

e quindi

$$\Pr[|X_{j,i}| > 3\varepsilon\|F\|] = \frac{\frac{\varepsilon}{e} \|F\|}{3\varepsilon\|F\|} = \frac{1}{3e}$$

Notiamo che questo è valido per  $|X_{j,i}| > 3\varepsilon\|F\|$ , introduciamo quindi le variabili casuali:

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

Poniamo  $E[Y_j] = p = \Pr[|X_{j,i}| > 3\varepsilon\|F\|]$  e che  $Y = Y_1 + \dots + Y_r$  quindi  $E[Y] = rp$

Si vuole  $Y \geq r/2$ , usiamo la disegualanza di Chernoff per calcolarne la probabilità.

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

Ponendo  $(1 + \lambda)\mu = r/2$ ,  $\mu = E[Y] = rp$ ,  $1 + \lambda = \frac{1}{2p}$ , otteniamo:

$$\Pr[Y \geq r/2] < \left(\frac{e^\lambda}{(1+\lambda)^{1+\lambda}}\right)^\mu = \frac{1}{e^\mu} \left(\frac{e}{(1+\lambda)}\right)^{(1+\lambda)\mu} = \frac{1}{e^{rp}} \left(\frac{e}{1/(2p)}\right)^{r/2} = \frac{1}{e^{rp}} (2pe)^{r/2}$$

Quindi se  $\frac{1}{e^{rp}} (2pe)^{\frac{r}{2}} \leq \delta^{\frac{1}{4}} = \frac{1}{2^{r/4}}$ , visto che  $e^{rp}$  è sempre maggiore di zero ed elevando tutto a 2/r si ottiene:

$$\frac{1}{e^{rp}} (2pe)^{\frac{r}{2}} \leq \frac{1}{2^{\frac{r}{4}}} \equiv 2^{\frac{r}{4}} \leq e^{rp}/(2pe)^{\frac{r}{2}} \equiv 2^{\frac{1}{2}} \leq 1/2pe \equiv p \leq \frac{1}{2\sqrt{2}e} \cong 1/8$$

## Problema 15

Show and analyze the application of count-min sketch to range queries  $(i, j)$  for computing  $\Pr_{j \in [k]} F[j]$ . Hint: reduce the latter query to the estimate of just  $t \leq 2 \log n$  contatori  $c_1, c_2, \dots, c_t$ . Note that in order to obtain a probability at most  $\delta$  of error (i.e. that  $\Pr_{l=1}^t c_l > \Pr_{j \in [k]} F[j] + 2 \log n \|F\|$ ), it does not suffice to say that it is at most  $\delta$  the probability of error of each counter  $c_l$ : while each counter is still the actual wanted value plus the residual as before, it is better to consider the sum  $V$  of these  $t$  wanted values and the sum  $X$  of these residuals, and apply Markov's inequality to  $V$  and  $X$  rather than on the individual counters.

Per risolvere questo problema suddividiamo ogni range in modo canonico in sottorange di dimensione  $2^k$ .

$t \leq 2 \log n$  contatori

$$\Pr \left[ \sum_{l=1}^t c_l > \sum_{k=1}^j F[k] + 2\epsilon \log n \|F\| \right] < \delta$$

$$\sum_{l=1}^t c_l = \tilde{v}$$

$$\sum_{k=1}^j F[k] = v$$

$$\tilde{v} = v + X_t = v + x_1 + x_2 + x_3 \dots x_t$$

$$\Pr[\tilde{v} > v + 2\epsilon \log n \|F\|] < \delta$$

$v > 0$

$$\Pr[X > 2\epsilon \log n \|F\|]$$

Applico Markov

$$\Pr[X > 2\epsilon \log n \|F\|] \leq \frac{E[X]}{2\epsilon \log n \|F\|}$$

$$E[X] = \sum_{k=1}^n I_{jik} F[K]$$

dove  $I_{jik}$  è una variabile indicatrice per cui:

$$I_{jik} = \begin{cases} 1 & \text{per } k \neq i \text{ e } h_j(i) = h_j(k) \\ 0 & \text{altrimenti} \end{cases}$$

Per la linearità del valore atteso si può scrivere

$$E[X] = \sum_0^{2 \log n} \sum_{k=1}^n F[K] E[I_{jik}] = \|F\| \frac{\epsilon}{e} \text{ con } E[I_{jik}] = \frac{\epsilon}{e}$$

Essendo questo l'errore relativo a tutti i contatori si moltiplica per  $2 \log n$ , ottenendo quindi

$$E[X] = 2 \log n ||F|| \frac{\varepsilon}{e}$$

Andando a sostituire otteniamo:

$$\Pr[X > 2\varepsilon \log n ||F||] \leq \frac{2 \log n ||F|| \frac{\varepsilon}{e}}{2\varepsilon \log n ||F||} = \frac{1}{e}$$

Volendo che questo sia valida per tutte le  $r$  funzioni hash si ottiene  $(\frac{1}{e})^r \sim (\frac{1}{2})^r$

Da cui

$$\Pr\left[\sum_{l=1}^t C_l \leq \sum_{k=1}^j F[k] + 2\varepsilon \log n ||F||\right] \geq 1 - \delta$$

## Problema 16

*Not all equal 3-SAT*

Consider the problem NE-3-SAT, which asks if there is an assignment of the Boolean variables in a conjunctive normal form formula F, where each clause contains three literals, such that each clause of F has at least one literal true and at least one literal false. Hint: use a polynomial reduction from 3-SAT.

$$\begin{aligned}F &= (x \vee y \vee z) \\F' &= \text{NAE}(x \vee y \vee z \vee t)\end{aligned}$$

Per tutti i literals che verificano F è verificata anche F'. Infatti se la condizione  $x \vee y \vee z$  è pari a true, allora anche  $x \vee y \vee z \vee t$  sarà pari a true indipendentemente da t. Bisogna solo evidenziare la condizione Not-All-Equal, ovvero che almeno uno dei quattro literal sia diverso dagli altri, sia rispettata. È possibile quindi affermare che 3-SAT si riduce in 4-NAE-SAT.

$$F'' = \text{NAE}(x \vee y \vee w) \wedge (\bar{w} \vee z \vee t)$$

Volendo ridurre NAE-4-SAT visto in precedenza in un NAE-3-SAT è possibile, assegnando i valori che soddisfano F' a F'' e scegliendo opportunamente la variabile ausiliaria w, verificarle entrambe.

Si conclude quindi che un 3-SAT si può ridurre a un NAE-3-SAT.

## Problema 17

*Wrong greedy for MAX-CUT*

Find an example of graphs for which the greedy approach of choosing each time the vertex with the largest residual degree does not give a 2-approximation.

Nel caso in cui il grafo sia una Clique, quindi con ogni nodo connesso a tutti gli altri, qualsiasi nodo si prenda esso risulta sempre il nodo con più archi essendo tale numero uguale per tutti i nodi. Inoltre, una volta selezionato il primo, usando il greedy algorithm, si escluderanno tutti gli altri in quanto tutti connessi al nodo selezionato. Questo porta ad ottenere, con questa tipologia di grafo, un max-cut sempre pari al numero di nodi del grafo meno uno.

Il valore del max-cut per questo genere di grafo è pari al  $\lfloor n/2 \rfloor \cdot \lceil n/2 \rceil$  dove  $n$  è il numero dei nodi del grafo.

Quindi se  $\lfloor n/2 \rfloor \cdot \lceil n/2 \rceil > 2(n - 1)$  vuol dire che il greedy non verifica la 2-approximazione richiesta. Ad esempio prendendo un grafo a clique formato da 8 nodi, selezionando un nodo taglieremo 7 archi escludendo però tutti gli altri nodi del grafo. Il max cut per quel grafo risulta pari a 16 e siccome  $7 < 16/2$  si può affermare che questa è un'eccezione in cui tale algoritmo non garantisce l'approssimazione.

## Problema 18

**Dimostrazione.** Definiamo la variabile indicatrice  $X_{(u,v)}$ , che vale:

$$X_{(u,v)} = \begin{cases} 1 & \text{se } (u,v) \in T \\ 0 & \text{altrimenti.} \end{cases}$$

Allora la probabilità che tale variabile sia uguale ad uno è data da:

$$\begin{aligned} Pr[X_{(u,v)} = 1] &= \\ &= Pr[u \in V \wedge v \in V \setminus C \vee u \in V \setminus C \wedge v \in V] = \\ &= Pr[u \in V \wedge v \in V \setminus C] + Pr[u \in V \setminus C \wedge v \in V] = \\ &= \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{2}. \end{aligned}$$

Da cui sappiamo che la media di una variabile aleatoria così fatta è proprio la probabilità che valga uno:

$$E(X_{(u,v)}) = Pr[X_{(u,v)} = 1] = \frac{1}{2}.$$

Definiamo la variabile aleatoria  $X = \sum_{(u,v) \in E} X_{(u,v)}$ , somma delle variabili indicatorie su tutti gli archi in  $E$ , quindi conta il numero di archi in  $T$ . La sua media sarà:

$$E(X) = E\left(\sum_{(u,v) \in E} X_{(u,v)}\right) = \sum_{(u,v) \in E} E(X_{(u,v)}) = |E| \cdot \frac{1}{2}.$$

---

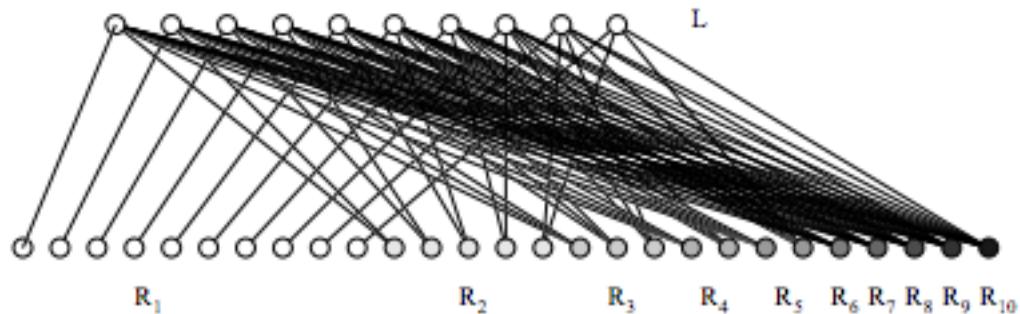
## PROBLEMA 21. APPROXIMAZIONE PER MAX-CUT

Sia  $T^*$ , la soluzione ottima. Sappiamo che sicuramente  $|T^*| \leq |E|$ , ottenendo:

$$E(|T|) = E(X) = \frac{1}{2}|E| \geq \frac{1}{2}|T^*|.$$

Abbiamo quindi dimostrato che l'algoritmo proposto è una 2-approximazione in valore atteso: ossia che il numero medio di archi di taglio della soluzione trovata è in media almeno la metà di quelli del taglio massimo.

### Problema 19



Suppose that (out of sheer bad luck) the algorithm considers an edge out of  $R_r$  first, choosing the end-point in  $L$  as the vertex to be placed in the cover. Then it picks an edge out of  $R_{r-1}$ , again choosing its end-point in  $L$  for the cover  $C$ ; and, so on. Therefore the vertex cover chosen is  $C = L$ . But  $L$  is itself a vertex cover since the graph is bipartite. It follows that the ratio achieved by this algorithm is no better than  $|R|/|L| = \Omega(\log n)$ .

## Algorithm analysis

Let us consider the behaviour of this algorithm on the graph  $B$ . It should be easy to see that GREEDY2 could also output  $R$  as a vertex cover. It could choose vertices from  $R_r$  at the very first stage. After this, it could choose vertices from  $R_{r-1}$ . In general, it would choose the highest degree vertices from  $R$  at each stage. It is very surprising that a seemingly much more intelligent heuristic does no better than the rather simple-minded heuristic GREEDY1. However this algorithm is not totally useless. It may be shown that it always achieves the ratio  $O(\log n)$  for the much more general problem of set cover and hence also for vertex cover.

### Procedimento.

Preso un grafo  $G = (V, E)$ . Bisogna cercare il più piccolo sottoinsieme di vertici tali che, preso un qualunque arco, almeno uno dei due vertici cui è collegato appartiene all'insieme. Cerchiamo una due approssimazione, cioè l'insieme deve avere massimo il doppio dei nodi dell'insieme ottimo soluzione. Chiamo  $S \subseteq V$ , il sottoinsieme dei vertici soluzione approssimata: lo inizializziamo all'insieme vuoto. Chiamo  $S^* \subseteq V$ , il sottoinsieme dei vertici soluzione ottima. Chiamo  $A \subseteq E$ , il sottoinsieme degli archi ancora da considerare nell'esecuzione dell'algoritmo: lo inizializziamo all'insieme  $E$ . Chiamiamo "marcati" i vertici che sono stati inseriti in  $S$ .

Algoritmo:

1.  $S = \emptyset; A = E;$
2. fino a quando  $A \neq \emptyset$ , ci sono ancora archi da considerare:
  - (a) estraggo un arco  $(x, y)$  da  $A$  scelto in modo casuale, ottenendo  $A = A \setminus (x, y)$ ;
  - (b) se nessuno dei due nodi appartiene all'insieme dei vertici di copertura  $[x \notin S \wedge y \notin S]$ , allora li inserisco entrambi [ $S = S \cup \{x, y\}$ ].

Vengono inseriti entrambi quando nessuno dei due appartiene all'insieme  $S$  perché potrei trovarmi nel caso in cui, prendendo ad esempio una configurazione a stella con un nodo centrale e gli altri ad esso collegati, mi troverò nel caso peggiore di aver considerato tutti i vertici intorno e non il centro, quando invece bastava prendere il nodo centrale per avere come buoni tutti quelli collegati.

Chiamiamo "matching" per un grafo, un sottoinsieme di archi a due a due non incidenti, ossia non aventi vertici in comune. Nel nostro caso è l'insieme degli archi che al momento della scelta hanno provocato l'inserimento di due nodi in  $S$ . Siamo sicuri che gli archi sono non incidenti perché quando prendo due vertici estremi di un arco, sono sicuro che tutti gli archi che escono dai vertici marcati, non

provocheranno altri inserimenti di vertici in  $S$  in quanto almeno uno dei due è marcato. Considerando la soluzione ottima  $S^*$ , possiamo notare che quando considero un arco che non ha nessun vertice marcato, può succedere che non verranno presi entrambi i suoi vertici ma uno solo potrebbe bastare.

Sappiamo sicuramente che

$$|S^*| \geq |S^* \cap S|.$$

Possiamo inoltre notare che

$$|S^*| \geq |S^* \cap S| \geq \frac{|S|}{2}$$

perché, per come abbiamo detto prima, nell'applicazione dell'algoritmo quando trovo un arco dove nessun vertice è marcato, allora li prendo entrambi; mentre può accadere che nella soluzione ottima ne basta prendere uno. Questo ci assicura che se mi basta sempre prendere un solo vertice nella soluzione ottima, allora l'intersezione è grande la metà di  $S$ , altrimenti ci sarà almeno un caso in cui li ho presi tutti e due, ottenendo una maggiorazione. Abbiamo quindi dimostrato che l'algoritmo proposto è una 2-approssimazione dell'algoritmo MIN-VC.

## Problema 20

**Dimostrazione.**

Supponiamo che la formula  $F$  sia in forma normale congiuntiva, ossia sia la congiunzione di clausole, dove le clausole sono disgiunzioni di variabili. Se non lo è, si applicano le leggi di De Morgan fino a portarla in quella forma. Denotiamo con  $x_i$  la variabile coniugata di  $x_i$ .

Dato un assegnamento  $\forall i \in [1, n] x_i := b_i$ , rappresentiamo tutte le variabili in un insieme.

Dato l'assegnamento coniugato, rappresentiamo tutte le variabili coinvolte in un altro insieme.

Supponiamo che sia questa la configurazione:

$\langle x_1, \bar{x}_2, \bar{x}_3, x_4 \rangle$  primo insieme;

$\langle \bar{x}_1, x_2, x_3, \bar{x}_4 \rangle$  secondo insieme.

Nel primo insieme ci sono tutte le variabili in forma positiva, se a loro è stato assegnato  $b = 1$ ; in forma negativa, altrimenti. Nel secondo insieme ci sono tutte le variabili in forma positiva, se a loro è stato assegnato  $b = 0$ ; in forma negativa altrimenti. Possiamo denotare con:

- $P$ : il numero di clausole formate da sole variabili del primo insieme, che sono verificate per come abbiamo fatto l'assegnamento;
- $N$ : il numero di clausole formate da sole variabili del secondo insieme, che sono verificate per come abbiamo fatto l'assegnamento opposto;
- $C$ : il numero di clausole verificate formate da elementi di entrambi gli insiemi.

Definiamo la nostra approssimazione di clausole verificate  $\max\{m_0, m_1\} = \max\{P + C, N + C\} = C + \max\{P, N\}$ .

Abbiamo  $C$  perché a seconda che si scelga l'assegnamento o il suo coniugato, quelle clausole ci sono sempre in quanto formate da variabili di entrambi gli insiemi e almeno una sarà vera, per costruzione. Abbiamo  $\max\{P, N\}$  perché in base a quale dei due massimizza il numero di clausole verificate sceglieremo quale assegnamento considerare.

Sappiamo che:

$$APPROX = C + \max\{P, N\} \leq OPT;$$

dove  $OPT$  è il valore ottimo soluzione del problema.

Sappiamo anche che possiamo maggiorare:

$$\begin{aligned} OPT &\leq C + P + N; \\ \frac{OPT}{2} &\leq \frac{C + P + N}{2}; \\ \frac{OPT}{2} &\leq \frac{C + P + N}{2} \leq C + \frac{P + N}{2}; \end{aligned}$$

dove il secondo passo è ottenuto semplicemente dividendo per due e maggiorando  $\frac{C}{2} \leq C$ .

Sappiamo che il massimo fra due numeri è maggiore della loro media per cui possiamo dire:

$$\frac{OPT}{2} \leq C + \frac{P + N}{2} \leq C + \max\{P, N\} \leq OPT.$$

Possiamo concludere che questo algoritmo è una 2-approssimazione per il problema MAX-SAT. Il valore  $r$  non dipende dall'assegnamento perché se scegliamo un assegnamento buono, quello negativo ha meno clausole verificate. Il peggior modo in cui possiamo scegliere l'assegnamento è quando metà delle clausole sono verificate, e metà della sua negazione sono verificate.

Un esempio può essere quando abbiamo:

- primo insieme:  $\langle x_1, \bar{x}_2, \bar{x}_3, x_4 \rangle$ ;
- secondo insieme:  $\langle \bar{x}_1, x_2, x_3, \bar{x}_4 \rangle$ .

Abbiamo che  $P = 2$ ,  $N = 2$ , quindi:

$$\frac{OPT}{2} \leq 0 + \max\{2, 2\} = 2 \leq OPT.$$