AE - Teoremi

Luca Corbucci

January 2019

1 Random Sampling

1.1 Stream e lunghezza conosciuta

Teorema: Conoscendo la lunghezza N dello stream, è necessario che venga rispettata la richiesta di poter selezionare "Uniformly at random" ogni elemento presente all'interno dello stream.

Dimostrazione: Per dimostrare che effettivamente ogni elemento dello stream ha probabilità $\frac{1}{N}$ di essere selezionato dobbiamo considerare i seguenti passaggi:

- Partiamo con uno stream lungo N e vogliamo selezionare m=1 elemento random dallo stream;
- Per ogni elemento dello stream abbiamo la seguente probabilità di essere selezionato. $P(1) = \frac{1}{N}$, $P(2) = \frac{1}{N-1}$, $P(3) = \frac{1}{N-2}$,..., $P(j) = \frac{1}{N-j+1}$ fino ad arrivare a $P(N) = 1$. Nel caso di P(j), $N - j + 1$ indica il numero complessivo di elementi che rimangono nella sequenza e ogni elemento ha probabilità $\frac{1}{N}$ di essere considerato;
- Consideriamo che se siamo sull'elemento J vuol dire che i precedenti J-1 non sono stati presi in considerazione. Per induzione possiamo dire che abbiamo una probabilità $\frac{1}{N}$ di selezionare random uno dei J-1 elementi precedenti. Quindi abbiamo probabilità $1 - \frac{j-1}{N}$ di non selezionarne nessuno;
- Ora consideriamo la probabilità di prendere come sample l'elemento J:

$$P(SampleJ) = P(Notsample1, ..., j-1) * P(ScegliereJ) \tag{1}$$

$$= 1 - \frac{j-1}{N} * \frac{1}{N-j+1} = \frac{1}{N} \tag{2}$$

1.2 Correttezza stream e lunghezza conosciuta

La guardia dell'if mi dice che $\frac{m-s}{n-j+1}$, se m=s allora non devo selezionare altro, se invece mi mancano da selezionare k elementi e nello stream mi rimangono k elementi allora vuol dire che li selezioniamo tutti con probabilità 1. Per dimostrare la correttezza dell'algoritmo consideriamo le combinazioni che includono $S[J]$ e lo dividiamo per quelle che includono o no $S[J]$. Quindi abbiamo:

$$\frac{Poss.\ di\ scegliere\ m-s\ elementi\ compreso\ S[J]}{Poss.\ di\ scegliere\ m-s\ elementi\ con\ S[J]\ o\ senza} \tag{3}$$

$$\frac{\binom{n-j}{m-s-1}}{\binom{n-j+1}{m-s}} = \frac{m-s}{n-j+1} \tag{4}$$

1.3 Reservoir Sampling

Teorema: ogni elemento presente all'interno del reservoir R viene selezionato con probabilità uniformly at random con probabilità $\frac{m}{n}$ **Dimostrazione:** Partiamo considerando il caso più semplice ovvero quando $m = n$ e R è formato dagli stessi elementi che troviamo in S. Creiamo R già al primo step dell'algoritmo e poi non facciamo altro. Ogni elemento viene selezionato con probabilità $\frac{m}{n} = 1$. Il passaggio induttivo da n-1 elementi a n è il seguente, un elemento è all'interno di R allo step N se c'era anche allo step $N - 1$ e se allo step N non viene eliminato ovvero:

$$\begin{aligned} P(s[J] \in R\ after\ N\ step) = P(S[J] \in R\ after\ N-1\ step) * \\ [P(S[N]\ non\ selezionato) + P(S[N]\ venga\ selezionato) * \\ P(S[J]\ non\ viene\ rimosso\ da\ R)] \end{aligned} \tag{5}$$

Questo calcolo diventa:

$$P(s[J] \in R\ after\ N\ step) = \frac{m}{n-1} * ((1 - \frac{m}{n}) + (\frac{m}{n}) * (1 - \frac{1}{m})) = \frac{m}{n} \tag{6}$$

Abbiamo dimostrato che ogni elemento viene selezionato con probabilità $\frac{m}{n}$.

2 Sorting Atomic Items

2.1 QuickSort

Teorema: Il numero di confronti non supera $2nlogn$:

Dimostrazione:

- Consideriamo $X_{u,v}$ che è la variabile che mi indica se S[u] e S[v] sono state confrontate o no e poi $p_{u,v}$ che invece mi dice la probabilità di confrontare questi due valori.
- La media dei confronti quindi dipende dalla probabilità $p_{u,v}$ che quindi va stimata.
- Per stimare $p_{u,v}$ consideriamo vari casi:
  - Se abbiamo scelto un pivot S[r] tale che S[u] e S[v] sono sempre maggiori o sempre minori allora vuol dire che non facciamo mai confronti tra S[u] e S[v].
  - Se invece scegliamo come pivot S[u] oppure S[v] allora facciamo almeno un confronto tra i due, gli altri casi sono quando il pivot è compreso tra u e v. Per il calcolo della $p_{u,v}$ ordiniamo S e otteniamo S', poi prendiamo il corrispondente di S[u] e di S[v] in S'che stanno in posizione u' e v' e calcoliamo $v' - u' + 1$. La $p_{u,v}$ quindi è 2 sulla distanza tra u' e v' quindi: $p_{u,v} = \frac{2}{v'-u'+1}$

Ora data questa probabilità possiamo dimostrare che non facciamo più di $2nLogn$ confronti:

$$E[\sum_{u,v} X_{u,v}] = \sum_{u'=1}^{n} \sum_{v'>u'}^{n} \frac{2}{v' - u' + 1} = 2 \sum_{u'=1}^{n} \sum_{k=2}^{n-u'+1} \frac{1}{k} \leq 2nLogn \tag{7}$$

L'ultimo passaggio è giustificato dalla proprietà della serie armonica.

Teorema: Il Lower Bound del sorting nel modello Ram è $\Omega(nLogn)$

Dimostrazione: Per calcolare il lower bound del sorting nel modello Ram dobbiamo utilizzare un albero di decisione, in ogni nodo dell'albero viene presa una decisione e ogni nodo poi ha due figli. Quindi l'albero è binario e se abbiamo n elementi da ordinare avremo $n!$ possibili ordinamenti che quindi corrispondono a $n!$ foglie dell'albero. Ogni path dal nodo root ad una foglia è una possibile computazione. Se l'albero ha altezza h vuol dire che all'ultimo livello dell'albero dovremo avere $n!$ foglie, quindi:

$$2^h \geq n! => h \geq logn! => h \geq nlogn \tag{8}$$

L'ultimo passaggio lo possiamo fare per l'approssimazione di Stirling. Quindi il lower bound nel ram model è $\Omega(nlogn)$

Teorema: Il lower bound del sorting nel 2 level model è $\Omega(\frac{n}{B} Log_{\frac{M}{B}} \frac{N}{M})$

Dimostrazione: Anche in questo caso dobbiamo sempre considerare l'albero di decisione in cui ogni nodo è un confronto e una operazione di I/O. Il numero delle foglie è sempre $n!$. La differenza rispetto al modello ram è il numero di figli di ogni nodo (fan-out), un solo I/O legge B elementi e ce ne sono già M-B in memoria, questo I/O mi genera un numero di possibili ordinamenti differenti dei B elementi che è pari a $\binom{M}{B}$ modi differenti, vanno però considerate anche le possibili permutazioni di B che sono $B!$ quindi il fan out dei nodi è $(\binom{M}{B})B!$. Va considerato anche un altro dettaglio però, dopo aver fatto O(n/b) I/O avremo già considerato tutti gli elementi una volta, quindi non è necessario considerare le possibili permutazioni di B quindi per alcuni livelli dell'albero di decisione il fan out dei nodi sarà $(\binom{M}{B})$.

Se consideriamo un percorso root-foglia lungo t avremo $\frac{n}{b}$ nodi che avranno un fan out $(\binom{M}{B})B!$ e $t - b$ nodi che invece avranno un fan out $(\binom{M}{B})$. Quindi il numero delle foglie dell'albero sarà:

$$((\binom{M}{B})B!)^{\frac{n}{B}} * (\binom{M}{B})^{t-\frac{n}{B}} \tag{9}$$

Questo va posto sempre maggiore o uguale a $n!$ perchè il numero di foglie deve essere quello quindi abbiamo:

$$((\binom{M}{B})B!)^{\frac{n}{B}} * (\binom{M}{B})^{t-\frac{n}{B}} \geq n! = (\binom{M}{B})^t * (B!)^{\frac{n}{B}} \geq n! \tag{10}$$

$$tLog(\binom{M}{B}) * \frac{n}{B} Log(B!) \geq Logn! \tag{11}$$

$$tLog(\binom{M}{B}) * B\frac{n}{B} Log(B) \geq nLogn \tag{12}$$

$$tLog(\binom{M}{B}) * nLog(B) \geq nLogn \tag{13}$$

$$tLog(\binom{M}{B}) \geq nLogn - nLog(B) \tag{14}$$

$$tLog(\binom{M}{B}) \geq nLog\frac{n}{B} \tag{15}$$

Con la formula del cambio di base:

$$t \geq \frac{n}{B} Log_{\frac{M}{B}} \frac{n}{B} \tag{16}$$

Quindi vuol dire che il numero di operazioni I/O da eseguire è pari a $\frac{n}{B} Log_{\frac{M}{B}} \frac{n}{B}$. Se abbiamo invece D dischi diventa $\frac{n}{DB} Log_{\frac{M}{B}} \frac{n}{DB}$.

2.2 RandSelect

Teorema: Il costo del Rand Select in termini di tempo è $O(n)$ nel Ram model e richiede $O(\frac{n}{B})$ operazioni di I/O nel 2 level model. **Dimostrazione:**

- Consideriamo una buona scelta del Pivot quella che fa in modo che le partizioni $n_<$ e $n_>$ non siano più grandi di $\frac{2n}{3}$. Per fare in modo che valga questa proprietà allora l'elemento che cerchiamo deve avere un rank compreso tra $[\frac{n}{3}, \frac{2n}{3}]$.

- La probabilità di essere in questo range è $\frac{1}{3}$ perchè equivale alla probabilità di essere nel sub array di mezzo.

- Quindi bisogna scrivere l'equazione di ricorrenza considerando la buona scelta dle pivot e la cattiva:

$$T(n) \leq O(n) + \frac{1}{3} T(\frac{2n}{3}) + \frac{2}{3} T(n) \tag{17}$$

  Dove il primo elemento è dovuto alla complessità dello spostamento degli elementi nell'array, il secondo è il costo del caso con la scelta buona del pivot (perchè ho 1/3 di probabilità che $n_<$ e $n_>$ non siano più grandi di $\frac{2n}{3}$) e poi l'ultimo è il caso in cui invece si fa la ricorsione su praticamente tutto l'array. Questa equazione si risolve sottraendo T(n) da entrambe le parti:

$$T(n) \leq O(n) + \frac{1}{3} T(\frac{2n}{3}) \tag{18}$$

  Questo equivale a O(n) nel caso del Ram model, nel caso del 2 level l'equazione diventa:

$$T(n) \leq O(\frac{n}{b}) + \frac{1}{3} T(\frac{2n}{3}) \tag{19}$$

  E quindi il numero di I/O diventa $O(\frac{n}{b})$.

2.3 Oversampling

Lemma: Dati $k \geq 2$ e a+1 = 12*ln k, un sample di dimensione $(a+1)k-1$ mi basta per dire che ogni bucket riceve meno di $\frac{4n}{k}$ elementi con probabilità almeno $\frac{1}{2}$.

Dimostrazione: Consideriamo il problema complementare e lavoriamo tramite failure sampling, diciamo che esiste 1 bucket che contiene più di $\frac{4n}{k}$ elementi con probabilità al più $\frac{1}{2}$. Prendiamo la versione ordinata di S, la chiamiamo S' è formata da $\frac{k}{2}$ blocchi di dimensione $\frac{2n}{k}$ elementi. Ora consideriamo:

- Un blocco B_i tale che la dimensione sia almeno $\frac{4n}{k}$, questo blocco si sovrappone ad un blocco t_i della sequenza e i suoi delimitatori saranno esterni a t_i e saranno ad esempio s_i e s_{i-1}.

- Ora consideriamo le probabilità:

$$\begin{aligned} P(EsisteB_i : |B_i| \geq n/k) &\leq P(Esiste\ t_j\ :\ contiene\ meno\ di\ (a+1)\ sample) \\ &\leq \frac{k}{2} P(Un\ segmento\ specifico\ contiene\ meno\ di\ (a+1)\ sample)) \end{aligned}$$

Ora consideriamo le seguenti probabilità:

- P(1 sample finisce in un certo segmento) = $\frac{2}{k}$

- Dato X il numero di sample che finiscono in un segmento vogliamo calcolare la probabilità che P(X ¡ a+1)

- Calcoliamo il numero medio di sample X:

$$E[X] = ((a+1)k - 1) + \frac{2}{k} \geq 2(a+1) - \frac{2}{k}$$
$$Sek \geq 2 allora : E[X] \geq 2(a+1) - 1 \geq \frac{3}{2}(a+1)$$
$$a + 1 \leq \frac{2}{3}E[X] = (1 - \frac{1}{3})E[X]$$

  Alla fine possiamo dire che $a + 1 = (1 - \frac{1}{3})E[X]$

  Consideriamo il Chernoff Bound:

$$P(X < (1-\delta)E[X]) \leq e^{-\frac{\delta^2}{2}E[X]} \tag{20}$$

  Se $\delta = 1/3$ e a+1 = 12lnk abbiamo che:

$$P(X < a+1) \leq P(X \leq (1 - \frac{1}{3})E[X]) \leq e^{-\frac{\delta^2}{2}E[X]}$$
$$e^{-\frac{E[X]}{18}} \leq e^{-\frac{-(q+1)}{12}} = e^{-lnk} = 1/k$$

Quindi $P(x < a + 1) \leq 1/K$.

  Sappiamo che

$$P(EsisteB_i : |B_i| \geq n/k) \leq \frac{k}{2} P(Un\ segmento\ specifico\ contiene\ meno\ di\ (a+1)\ sample)) \tag{21}$$

Sappiamo anche che

$$P(EsisteB_i : |B_i| \geq n/k) \leq 1/2\ complemento \tag{22}$$

  Conclusione: tutti i blucket hanno meno di $\frac{4n}{k}$ elementi con probabilità maggiore di $1_{\overline{2.}}$

3 Set Intersection

3.1 Costo del Mutual partitioning

Teorema: Il mutual Partitioning nel caso pessimo mi costa $O(m(1 + Log\frac{n}{m}))$

  Dimostrazione:

  Va calcolata la complessità in termini di tempo e possiamo considerare un caso pessimo e un caso ottimo:

- Al caso ottimo ogni volta che cerco il pivot all'interno dell'array, non lo trovo e risulta che il pivot è fuori dall'array, questo mi permette di andare a dimezzare ogni volta la dimensione m dell'array più breve. Quindi vuol dire che faccio per $Logm$ volte la ricerca binaria. Quindi vuol dire che nel complesso il tempo è $O(LogmLogn)$.

- Al caso pessimo invece il pivot si trova ogni volta a metà dell'array A che quindi viene ogni volta diviso in due e non scarto mai elementi di B. Quindi la relazione di ricorrenza diventa $T(n, m) = O(Logn + T(n/2, m/2))$ che ha come soluzione $O(m(1 + Log\frac{n}{m}))$. Se m è molto simile a n abbiamo un costo simile al Merge based, se m è piccolo invece abbiamo un costo simile al Binary Search.

3.2 Costo del doubling

Teorema: Il doubling mi costa $O(m(1 + Log\frac{n}{m}))$, a differenza del precedente non è ricorsivo.

  Dimostrazione:

- Supponiamo che b_j sarà compreso tra $A[i + 2^{k-1}] < b_j \leq A[i + 2^k]$

- Sappiamo che b_j è la posizione in cui si trova l'elemento che è presente in entrambi i set, questa posizione è $b_j = i'$ se consideriamo la posizione all'interno del blocco in cui faccio la ricerca allora abbiamo che la posizione è $i' - i$ Abbiamo le seguenti relazioni: $i + 2^{k-1} < i' \leq i + 2^k$ e quindi $2^{k-1} < i' - i \leq 2^k$. Quindi il primo risultato che otteniamo è avere $2^k < 2(i' - i)$.

- Consideriamo la porzione dell'array in cui facciamo la ricerca binaria, questa porzione di array la chiamiamo Δ_j. Chiamiamo i_j la posizione di b_j e quindi i_{j-1} la posizione di b_{j-1}, diciamo che i_0 vale 0. Possiamo dire che: $i_j - i_{j-1} \leq \Delta_j \leq 2^k$. Ma sapendo che $2^k < 2(i' - i)$ possiamo dire che $i_j - i_{j-1} \leq \Delta_j \leq 2^k < 2(i' - i)$.

- Consideriamo la somma dei sub array in cui facciamo la ricerca: $\sum_{i=1}^{m} \Delta_j < \sum_{i=1}^{m} 2(i_j - i_{j-1}) = 2\sum_{i=1}^{m}(i_j - i_{j-1})$ Questa è una somma telescopica quindi diventa 2n. Quindi abbiamo che $\sum_{i=1}^{m} \Delta_j < 2n$.
- L'algoritmo esegue $O(log\Delta_j)$ step quindi il costo totale della binary search è: $\sum_{i=1}^{m} log\Delta_j < \sum_{i=1}^{m} Log(2(i_j - i_{j-1})) = \sum_{i=1}^{m} 1 + Log_2(i_j - i_{j-1}) = m + \sum_{i=1}^{m} Log_2(i_j - i_{j-1})$.
- Ora utilizziamo la Jensen Inequality che mi permette di trasformare l'ultimo valore che abbiamo calcolato in: $Log_2(i_j - i_{j-1}) = m + m\sum_{i=1}^{m}(\frac{i_j - i_{j-1}}{m})$ Per la somma telescopica diventa: $m + mLog_2\frac{i_m - i_0}{m}$ Dato che i_0 vale 0 abbiamo: $m + mLog_2\frac{i_m}{m}$ Dato che $i_m < n$: $m + mLog_2\frac{n}{m}$ Il costo totale diventa: $O(m(1 + Log_2\frac{n}{m}))$.

Quindi la complessità in termine di tempo dell'algoritmo che usa il doubling è $O(m(1 + Log_2\frac{n}{m}))$ ma rispetto all'algoritmo del Mutual Partition non vengono fatte tutte le chiamate ricorsive.

3.3 Random Permuting

Teorema: La complessità è la stessa del set intersection nel 2 level model con la sola differenza che qua dobbiamo prendere in considerazione anche il costo della permutazione. Quindi abbiamo che il tempo necessario è $O(minn, mL + \frac{n}{L})$ e il numero di operazioni I/O è $O(\frac{mL}{B} + \frac{n}{BL} + m)$.

Dimostrazione: Per ottenere questa proprietà si deve fare una permutazione della posting list originale utilizzando una funzione del tipo $\Pi(x) = ax + bmodu$ dove a ed u sono coprimi. In questo modo abbiamo una distribuzione uniforme degli interi e quindi poi avremo una distribuzione uniforme delle loro distanze. È importante che si possa calcolare facilmente l'inverso della funzione che utilizziamo per fare la permutazione. Una volta eseguita la permutazione, nell'array A creiamo $\frac{n}{L}$ blocchi e assegnamo ogni elemento di A permutato ad uno di questi blocchi guardando i primi $l = log_2\frac{n}{L}$ bit degli interi permutati, la stessa cosa viene fatta anche nell'array B e se l'array B è più corto di A avremo che al più m blocchi su $\frac{n}{L}$ verranno riempiti con elementi. Poi, potremo poi utilizzare l'algoritmo usato nel 2 level model per eseguire il set intersection prendendo quindi i blocchi corrispondenti e poi andando ad eseguire il merge tra i blocchi per fare l'intersezione.

4 Ordinare le stringhe

4.1 Lower Bound

Teorema: Il lower bound per l'ordinamento delle stringhe è $\Omega(N) = \Omega(d + nLogn)$

Dimostrazione: Per ogni stringa del set S calcoliamo d_s che è il numero minimo di caratteri che dobbiamo confrontare per distinguere la stringa dalle altre. Calcoliamo d ovvero la somma dei d_s delle varie stringhe di S. Quindi, se pensiamo le stinghe di S come binarie possiamo dire che in ogni stringa abbiamo l bit che mi servono per distringuere le stringhe e poi log_2n bit che invece rappresentano il resto della stringa, ogni stringa è lunga $l + log_2n$ bit e quindi $N = n(l + log_2n)$ bit. Ora consideriamo che se abbiamo n stringhe, devo fare $\Omega(nLogn)$ confronti che mi confrontano 1 carattere per stringa e in più devo fare d confronti dove d indica la somma dei caratteri che sono diversi tra una stringa e l'altra. Quindi in totale il lower bound mi dice che la complessità in tempo è $O(d + nLogn)$ ma dato che $d = \Omega(N)$ abbiamo che il lower bound diventa $\Omega(N + nLogn)$ ovvero $\Omega(N)$. Gli algoritmi classici tipo Merge Sort e Quick Sort invece fanno un numero di confronti maggiori e la complessità in tempo diventa: $\Theta((l + logn)nLogn) = \Theta((L)nLogn) = \Theta(NLogn)$. Quindi siamo distanti di un fattore $Logn$ rispetto all'ottimo.

4.2 Correttezza LSD-First

Dimostrazione: Partiamo con due stringhe A e B che possiamo suddividere in varie parti, A = abc e B = ade. All'inizio delle due stringhe troviamo una serie di caratteri che in questo caso indichiamo con a che sono uguali tra le due stringhe, poi abbiamo b e d che rappresentano il primo carattere differente nelle due stringhe e poi abbiamo c ed e che sono le parti differenti delle due stringhe che abbiamo ordinato. Noi nell'algoritmo LSD partiamo da destra e quindi ordiniamo le stringhe basandoci sugli ultimi $|c| == |e| = P$ caratteri, dopo queste prime P fasi troviamo l'ultimo carattere differente e facciamo un altro ordinamento delle due stringhe, questo è l'ultimo perchè poi avremo un numero di ordinamenti pari ad a che però non modificherà l'ordine perchè i due insiemi di caratteri sono uguali. Quindi la sequenza finale che viene prodotta è ordinata.

4.3 Costo MSD-First

Teorema: Ogni nodo che viene creato occupa uno spazio che è $O(\sigma)$ e ne vengono creati d interni in più abbiamo n puntatori alle stringhe. Quindi in tutto portano uno spreco di memoria pari a $O(n + d * \sigma)$. La complessità in tempo necessaria per creare il trie è pari a $O(n + d * \sigma)$.

Lo spazio occupato utilizzando MSD con le tabelle hash è pari a $O(d + n)$ perchè dobbiamo allocare tutti i puntatori per i nodi interni e poi abbiamo le foglie. Il tempo invece è $O(dLog\sigma)$ che non è ottimo ma almeno abbiamo guadagnato molto in spazio.

5 Prefix Search

5.1 Interpolation Search

Teorema: La ricerca costa $O(Log\Delta)$ dove Δ è la media tra il massimo e il minimo gap tra due elementi consecutivi dell'array

Dimostrazione: Per dimostrare la complessità in tempo consideriamo che il massimo gap tra due interi presenti nell'array è maggiore della media degli elementi presenti nell'array:

$$max_{i=2,..}(x_i - x_{i-1}) \geq \frac{\sum_{i=2}^{m} x_i - x_{i-1}}{m-1} \geq \frac{x_m - x_1 + 1}{m} = b \tag{23}$$

Il numero massimo di elementi di ogni bin invece dipende dal fatto che gli elementi dell'array sono divisi da un numero di $s = min_{i=2...m}(x_i - x_{i-1})$ e quindi il numero di elementi di ogni bin diventa $\frac{b}{s}$. Quindi considerando tutte le precedenti disuguaglianze possiamo scrivere:

$$|B_i| \leq \frac{b}{s} \leq \frac{max_{i=2,..}(x_i - x_{i-1})}{min_{i=2...m}(x_i - x_{i-1})} = \Delta \tag{24}$$

Note:

- Il caso pessimo lo abbiamo se tutti gli elementi finiscono in un bin e in questo caso avremmo una complessità che è $Logm$
- L'algoritmo è cache oblivious
- L'occupazione in spazio è $O(m)$ cosa che non è necessaria in una classica binary search

Teorema: Se gli interi dell'array sono distribuiti in modo uniforme, l'algoritmo richiede un tempo $O(LogLogm)$ con alta probabilità.

Dimostrazione: Partiamo dicendo che abbiamo una distribuzione uniforme di interi e quindi ogni bucket contiene $O(1)$ intero. Assumiamo di partizionare gli interi in un numero di range pari a $r = \frac{m}{2Logm}$. Con probabilità $\frac{1}{r}$ un intero è in un range e con probabilità $(1 - \frac{1}{r})^m = (1 - \frac{m}{2Logm})^m = O(e^{-2Logm}) = O(\frac{1}{m^2})$ abbiamo un range che non contiene neanche un intero. Se diciamo che ogni range contiene almeno un elemento allora la distanza massima tra due interi vicini nell'array dovrà essere minore di 2 volte la lunghezza del range, quindi $max_i(x_i - x_{i-1}) = \frac{2U}{r} = O(\frac{ULogm}{m})$. Se prendiamo $r' = \Theta(mLogm)$ range possiamo dire che ogni coppia di range vicini tra loro contengono almeno un intero (se sta in un range poi nei due vicini potrei non avere interi). Quindi il gap minimo tra due interi della sequenza può essere limitato in questo modo: $min_i(x_i - x_{i-1}) \geq \frac{U}{r'} = \Theta(\frac{U}{mLogm})$. Ora prendiamo i due risultati e calcoliamo la divisione:

$$\Delta = \frac{ULogm}{m} * \frac{mLogm}{U} = O(Log^2 m) \tag{25}$$

Questo è il delta che volevamo calcolare che ci indica che se gli interi sono distribuiti in modo random allora il costo per l'interpolation search che di norma è $O(Log\ dimensione\ bin) = O(Log\Delta) = O(LogLogm)$.

5.2 Locality Preserving Front Coding

Teorema: Utilizzando il Locality Preserving Front Coding si consuma $O((1+\epsilon)FC(D))$ spazio e un numero di operazioni I/O pari a $O(\frac{FC(D)}{\epsilon B})$.

Dimostrazione: Consideriamo prima le stringhe che sono salvate compresse, quindi lo spazio che occupano è limitato superiormente da $FC(D)$. Le stringhe che invece sono copiate le dividiamo in due gruppi:

- Stringhe copiate Uncrowded: sono quelle stringhe che a sinistra hanno una quantità di caratteri Front coded maggiore di $\frac{C|S|}{2}$;
- Stringhe copiate Crowded: sono quelle stringhe che a sinistra hanno una quantità di caratteri Front coded minore di $\frac{C|S|}{2}$;

Da questa relazione possiamo dire che per una stringa s' copiata che viene prima di una stringa crowded, vale la seguente relazione: $|s'| \geq \frac{C|S|}{2}$.

Ora creiamo delle sequenze tali che la prima stringa è uncrowded e poi dopo c'è la massima sequenza di stringhe crowded. Possiamo dire che, prendendo una stringa crowded w_i si verifica la seguente relazione:

$$|w_{i-1}| > \frac{C|w_i|}{2}\ ovvero\ |w_i| < \frac{2|w_{i-1}|}{C} < (\frac{2}{C})^2|w_{i-2}| < (\frac{2}{C})^3|w_{i-3}| \tag{26}$$

$$< (\frac{2}{C})^{i-1}|w_1| \tag{27}$$

Facciamo la sommatoria

$$\sum_{i=0}^{x}(\frac{2}{C})^{i-1}|w_1| = \frac{c}{c-2}|w_i| \tag{28}$$

Ora consideriamo la stringa uncrowded w_1 che è preceduta da una serie di caratteri Front coded maggiori di $\frac{C|w_1|}{2}$ e questo valore è limitato da FC(D). Quindi abbiamo

$$\frac{C|w_1|}{2} < FC(D) \implies |w_1| < \frac{2FC(D)}{C} \tag{29}$$

Ora sostituiamo questo secondo valore nel primo e otteniamo:

$$\frac{c}{c-2}\frac{2FC(D)}{C} = \frac{2FC(D)}{C-2} \tag{30}$$

Per $\epsilon = \frac{2}{C-2}$ vale il teorema.

5.3 Uncompacted Trie

Teorema: L'uncompacted Trie risolve il problema del Prefix Search in $O(P + n_{occ})$ tempo dove n_{occ} è il tempo necessario per trovare le occorrenze delle stringhe e quindi contarle, il numero di operazioni I/O necessarie è $O(P + \frac{n_{occ}}{B})$. Se oltre al numero di stringhe che hanno quel prefisso vogliamo trovare anche le stringhe allora abbiamo un costo I/O aggiuntivo pari a $\frac{O(N_{occ}}{B})$ dove N_{occ} è la lunghezza delle stringhe da restituire. Il tutto supponendo che le stringhe sono salvate ordinate in memoria e sono una vicina all'altra.

5.4 Compacted Trie

Teorema: Il compacted Trie risolve il problema del prefix search in $O(P + n_{occ})$ e con $O(p + \frac{n_{occ}}{B})$ I/O. Ottenere le stringhe costa un tempo $O(N_{occ})$ e un numero di I/O pari a $\frac{N_{occ}}{B}$. Lo spazio occupato dal compacted Trie è $O(n)$ contro $O(N)$ dell'uncompacted.

5.5 Patricia Trie

Teorema: Un Patricia Trie permette di eseguire la ricerca di un pattern P[1,p] in tempo $O(p)$ con un consumo di spazio che è $O(n)$ e un numero di operazioni I/O che è pari a $O(\frac{p}{B})$ e che vengono utilizzate per comparare la stringa identificata dall'algoritmo blind search. Se $n < M$ possiamo far entrare in memoria tutto il Patricia trie, occuperò sempre $O(n)$ in memoria e poi avrò $O(N)$ per l'occupazione nella memoria esterna.

Possiamo anche utilizzare il Patricia Trie insieme al Locality Preserving Front Coding, si può sfruttare la compressione per cercare di ridurre l'occupazione di memoria delle stringhe che stanno salvate su disco. Quindi su disco vengono salvate le stringhe compresse che quindi occuperanno $O((1+\epsilon)FC(D))$, poi in memoria invece rimane sempre $O(n)$, la ricerca del pattern P la facciamo sempre in $O(p)$ con la differenza che le operazioni di I/O per cercare di capire l'lcp sono $O(\frac{p}{B} + \frac{|s|}{\epsilon B})$. Se invece vogliamo ottenere proprio le stringhe mi costa un numero di operazioni I/O pari a $\frac{(1+\epsilon)FC(D_{occ})}{B}$.

6 Prefix Search

Teorema: La ricerca costa $O(Log\Delta)$ dove Δ è la media tra il massimo e il minimo gap tra due elementi consecutivi dell'array

Dimostrazione: Per dimostrare la complessità in tempo consideriamo che il massimo gap tra due interi presenti nell'array è maggiore della media degli elementi presenti nell'array:

$$max_{i=2,..}(x_i - x_{i-1}) \geq \frac{\sum_{i=2}^{m} x_i - x_{i-1}}{m-1} \geq \frac{x_m - x_1 + 1}{m} = b \tag{31}$$

Il numero massimo di elementi di ogni bin invece dipende dal fatto che gli elementi dell'array sono divisi da un numero di $s = min_{i=2...m}(x_i - x_{i-1})$ e quindi il numero di elementi di ogni bin diventa $\frac{b}{s}$. Quindi considerando tutte le precedenti disuguaglianze possiamo scrivere:

$$|B_i| \leq \frac{b}{s} \leq \frac{max_{i=2,..}(x_i - x_{i-1})}{min_{i=2...m}(x_i - x_{i-1})} = \Delta \tag{32}$$

Teorema: Se gli interi dell'array sono distribuiti in modo uniforme, l'algoritmo richiede un tempo $O(LogLogm)$ con alta probabilità.

Dimostrazione: Partiamo dicendo che abbiamo una distribuzione uniforme di interi e quindi ogni bucket contiene $O(1)$ intero. Assumiamo di partizionare gli interi in un numero di range pari a $r = \frac{m}{2Logm}$. Con probabilità $\frac{1}{r}$ un intero è in un range e con probabilità $(1 - \frac{1}{r})^m = (1 - \frac{m}{2Logm})^m = O(e^{-2Logm}) = O(\frac{1}{m^2})$ abbiamo un range che non contiene neanche un intero. Se diciamo che ogni range contiene almeno un elemento allora la distanza massima tra due interi vicini nell'array dovrà essere minore di 2 volte la lunghezza del range, quindi $max_i(x_i - x_{i-1}) = \frac{2U}{r} = O(\frac{ULogm}{m})$. Se prendiamo $r' = \Theta(mLogm)$ range possiamo dire che ogni coppia di range vicini tra loro contengono almeno un intero (se sta in un range poi nei due vicini potrei non avere interi). Quindi il gap minimo tra due interi della sequenza può essere limitato in questo modo: $min_i(x_i - x_{i-1}) \geq \frac{U}{r'} = \Theta(\frac{U}{mLogm})$. Ora prendiamo i due risultati e calcoliamo la divisione:

$$\Delta = \frac{ULogm}{m} * \frac{mLogm}{U} = O(Log^2 m) \tag{33}$$

Questo è il delta che volevamo calcolare che ci indica che se gli interi sono distribuiti in modo random allora il costo per l'interpolation search che di norma è $O(Log\ dimensione\ bin) = O(Log\Delta) = O(LogLogm)$.

6.1 Locality Preserving Front Coding

Teorema: Utilizzando il Locality Preserving Front Coding si consuma $O((1+\epsilon)FC(D))$ spazio e un numero di operazioni I/O pari a $O(\frac{FC(D)}{\epsilon B})$.

Dimostrazione: Consideriamo prima le stringhe che sono salvate compresse, quindi lo spazio che occupano è limitato superiormente da $FC(D)$. Le stringhe che invece sono copiate le dividiamo in due gruppi:

- Stringhe copiate Uncrowded: sono quelle stringhe che a sinistra hanno una quantità di caratteri Front coded maggiore di $\frac{C|S|}{2}$;
- Stringhe copiate Crowded: sono quelle stringhe che a sinistra hanno una quantità di caratteri Front coded minore di $\frac{C|S|}{2}$;

Da questa relazione possiamo dire che per una stringa s' copiata che viene prima di una stringa crowded, vale la seguente relazione: $|s'| \geq \frac{C|S|}{2}$.

Ora creiamo delle sequenze tali che la prima stringa è uncrowded e poi dopo c'è la massima sequenza di stringhe crowded. Possiamo dire che, prendendo una stringa crowded w_i si verifica la seguente relazione:

$$|w_{i-1}| > \frac{C|w_i|}{2}\ ovvero\ |w_i| < \frac{2|w_{i-1}|}{C} < (\frac{2}{C})^2|w_{i-2}| < (\frac{2}{C})^3|w_{i-3}| \tag{34}$$

$$< (\frac{2}{C})^{i-1}|w_1| \tag{35}$$

Facciamo la sommatoria

$$\sum_{i=0}^{x}(\frac{2}{C})^{i-1}|w_1| = \frac{c}{c-2}|w_i| \tag{36}$$

Ora consideriamo la stringa uncrowded w_1 che è preceduta da una serie di caratteri Front coded maggiori di $\frac{C|w_1|}{2}$ e questo valore è limitato da FC(D). Quindi abbiamo

$$\frac{C|w_1|}{2} < FC(D)\ \ => |w_1| < \frac{2FC(D)}{C} \tag{37}$$

Ora sostituiamo questo secondo valore nel primo e otteniamo:

$$\frac{c}{c-2}\frac{2FC(D)}{C} = \frac{2FC(D)}{C-2} \tag{38}$$

Per $\epsilon = \frac{2}{C-2}$ vale il teorema.

6.2 Uncompacted Trie

Teorema: L'uncompacted Trie risolve il problema del Prefix Search in $O(P + n_{occ})$ tempo dove n_{occ} è il tempo necessario per trovare le occorrenze delle stringhe e quindi contarle, il numero di operazioni I/O necessarie è $O(P + \frac{n_{occ}}{B})$. Se oltre al numero di stringhe che hanno quel prefisso vogliamo trovare anche le stringhe allora abbiamo un costo I/O aggiuntivo pari a $\frac{O(N_{occ}}{B})$ dove N_{occ} è la lunghezza delle stringhe da restituire. Il tutto supponendo che le stringhe sono salvate ordinate in memoria e sono una vicina all'altra.

6.3 Compacted Trie

Teorema: Il compacted Trie risolve il problema del prefix search in $O(P + n_{occ})$ e con $O(p + \frac{n_{occ}}{B})$ I/O. Ottenere le stringhe costa un tempo $O(N_{occ})$ e un numero di I/O pari a $\frac{N_{occ}}{B}$. Lo spazio occupato dal compacted Trie è $O(n)$ contro $O(N)$ dell'uncompacted.

6.4 Patricia Trie

Teorema: Un Patricia Trie permette di eseguire la ricerca di un pattern P[1,p] in tempo $O(p)$ con un consumo di spazio che è $O(n)$ e un numero di operazioni I/O che è pari a $O(\frac{p}{B})$ e che vengono utilizzate per comparare la stringa identificata dall'algoritmo blind search. Se $n < M$ possiamo far entrare in memoria tutto il Patricia trie, occuperò sempre $O(n)$ in memoria e poi avrò $O(N)$ per l'occupazione nella memoria esterna.

7 Searching String by Substring

Teorema: Il costo del substring search può essere ridotto a $O(p + Logn)$

Dimostrazione:

- Abbiamo l'array lcp[1,n-1] in cui salviamo i lcp tra il suffisso i e il suffisso i+1
- Abbiamo due array di n elementi, ognuno degli elementi dell'array è legato alla tripla $< L, M, R >$ che si crea ad ogni iterazione della ricerca binaria. Dove L è il limite inferiore dell'array, M è il punto medio e R è il superiore. In particolare si va a definire gli array Llcp[] e Rlcp. $Llcp[M] = lcp(Suff_{SA[M]}, Suff_{SA[L]})$ $Rlcp[M] = lcp(Suff_{SA[M]}, Suff_{SA[R]})$
- Per ogni iterazione della ricerca binaria salviamo anche due variabili l e r: $l = lcp(P, Suff_{SA[L]})$ $r = lcp(P, Suff_{SA[R]})$
- Poi possiamo calcolare LCP(L,R)

Ci sono alcune cose che possiamo notare e che ci portano a ridurre il numero di confronti:

- Il pattern P che ricerchiamo si trova tra L e R quindi condivide con le stringhe che si trovano in questo range almeno LCP(L,R) caratteri.
- Valgolo le relazioni: $l > LCP(L,R)$ $r > LCP(L,R)$ $Lcp(Suff_{SA[M]}, P) > LCP(L,R)$

Quindi da queste due osservazioni possiamo dire che ad ogni iterazione possiamo scartare i primi $LCP(L,R)$ caratteri e possiamo partire dal successivo durante il confronto del pattern P con $Suff_{SA[M]}$. C'è un altro dettaglio da considerare, se consideriamo il caso $l \geq r$ abbiamo queste relazioni che ci permettono di escludere altri confronti:

- Se vale $l > Llcp[M]$ allora devo eseguire la ricorsione nella prima parte dell'array quindi $m = r$ ma non faccio altri confronti
- Se vale $l < Llcp[M]$ allora devo eseguire la ricorsione nella seconda parte dell'array quindi $m = l$ ma non faccio altri confronti
- Se invece vale $l = Llcp[M]$ possiamo dire che i primi l caratteri sono in comune tra il pattern P e $Suff_{SA[M]}$ quindi confrontiamo a partire da P[l+1].

In questo modo il tempo necessario per l'esecuzione dell'algoritmo passa a $O(p + Logn)$, più $O(occ)$ se vogliamo la posizione delle occorrenze delle sottostringhe. Lo spazio necessario è $O(n)$.

7.1 LCP Build

Vediamo l'implementazione dell'algoritmo, il tempo richiesto è $O(n)$ perchè ci muoviamo sempre verso destra e non facciamo dei confronti in più che non sono necessari:

```
LCP-Build(char *T, int n, char **SA):
    h = 0
    for (i=0;i<n;i++):
        q = SA^{-1}[i] # prendiamo la posizione di SA[i]
        if q > 1:
            k = SA[q-1]
            if(h > 0):
                h--
            while(T[k+h] == T[i+h]):
                h++
            lcp[q-1] = h
```

7.2 Come calcolare LCP in O(1)

Prendiamo una certa stringa T e creiamo il suo suffix tree. Consideriamo due suffissi, $X[i,x]$ e $X[j,x]$, tra questi due c'è un prefisso in comune che viene indicato dal nodo u che è antenato di entrambi. C'è quindi un legame tra il problema dell'lcp e quello dell'lca (longest common anchestor). In particolare il nodo u antenato di entrambi i suffissi avrà un $|S[u]|$ associato che rappresenta la lunghezza dell'lcp delle due sottostringhe. Lo stesso problema può essere risolto anche utilizzando il suffix array e il relativo array con i vari lcp. In particolare prese le sottostringhe $X[i,x]$ e $X[j,x]$ per cui vogliamo calcolare l'lcp, devo andare a prendere le corrispondenti posizioni nell'array lcp e poi prendere il minimo dei valori compresi tra queste due posizioni. Questo valore minimo è esattamente $|S[u]|$ che abbiamo nominato prima ed è la lunghezza del longest common prefix. Questa seconda soluzione che consiste nel trovare il minimo in un range all'interno di un array prende il nome di Range Minimum Query problem (RMQ). Il problema, dato un array A[1,n] consiste nel trovare il minimo elemento all'interno del range (i,j) in modo da restituire la posizione di questo valore minimo.

Una soluzione banale al problema consiste nel prendere tutte le possibili coppie i,j e calcolare il minimo compreso in quel range, poi si mettono tutti i risultati in una hash table in tempo $O(n^2)$ e spazio $O(n^2)$.

Un'altra soluzione è il metodo sparsification, in questo caso partiamo da un indice i e andiamo a calcolare ad ogni iterazione il minimo del range $(i, i+2^L-1)$ dove $L = Log(j-i+1)$. In questo modo l'array in cui salviamo i minimi mi occupa $O(nLogn)$. Se voglio risolvere il problema dell'RMQ di un certo range: $RMQ(i,j) = argmin(RMQ_a(i, i+2^L-1), RMQ(j-2^L+1, j))$ eseguo questa query e ottengo la risposta in $O(1)$.

Possiamo utilizzare il metodo sparsification con una variante andando a dividere in blocchi l'array A di partenza:

- Dividiamo A in n/b blocchi e ogni blocco ha $Logn$ elementi.
- Per ognuno dei blocchi calcoliamo il minimo e lo salviamo in un array A' che quindi occuperò $O(\frac{n}{Logn})$.
- Poi usiamo il metodo sparsification sull'array A' e quindi in questo modo abbiamo RMQ_A che mi occupa $(O(|A'|Log|A'|)) = O(n)$.

Ora se vogliamo risolvere le query RMQ:

- Se abbiamo la richiesta di RMQ che combacia direttamente con un blocco dell'array ottengo la risposta in O(1)
- Se abbiamo una query in cui i due limiti escono fuori da un blocco e quindi coprono un blocco per intero e due in parte dobbiamo andare a calcolare il minimo di due sottoblocchi $B_i[i,b]$ e $B_j[1,i]$. Per farlo possiamo salvarci per ogni blocco i minimi dei vari sottoblocchi con uno spreco di spazio pari a $O(b * \frac{n}{b}) = O(n)$. Anche in questo caso si risolve il problema in $O(1)$.

Il problema si presenta quando dobbiamo calcolare RMQ di un sottoblocco. Per risolverlo vengono fatte due riduzioni:

- La prima riduzione è dal problema RMQ al problema LCA
- La seconda riduzione è dal problema LCA a RMQ

Per quanto riguarda la prima riduzione, si parte con un array di interi (che sarebbe il nostro array lcp) e poi andiamo a creare un cartesian tree. Il cartesian tree lo costruisco così:

- Prendiamo il minimo dell'array e questo diventa il nodo root nel formato $< valoreminimo, posizione >$. Dove posizione = m.
- Il cartesian tree ha due figli, è binario. Il figlio sinistro sarà il minimo che troviamo nel sub array che va da 1 a m e il sub array a destra sarà il minimo del sub array che va da m a n dove n è la lunghezza complessiva dell'array.
- Ricorsivamente andiamo avanti in questo modo costruendo l'albero.

Se vogliamo calcolare il RMQ(i,j) ovvero il minimo del range (i,j), il problema può essere affrontato usando il cartesian tree andando a risolvere lca(i,j) ovvero trovando il common anchestor di i e di j all'interno del cartesian tree.

Poi va fatta una seconda riduzione da LCA e RMQ.

Per la seconda riduzione trasformiamo il problema dell'LCA da svolgere sul cartesian tree in un RMQ da svolgere su un array binario.

- Partiamo dal Cartesian tree creato al passo precedente, svolgiamo un euler tour di questo albero andando ad inserire in un array $\Delta[1, 2e]$ (dove e è il numero di archi visitati) il valore dei vari nodi che visitiamo.

- Mentre creiamo l'array dell'euler tour creiamo anche un secondo array D in cui inseriamo la profondità del nodo che stiamo visitando. La particolarità di questo array D è che c'è una differenza di 1 tra i vari elementi che sono presenti al suo interno. Ogni interò è +1 o -1 rispetto al precedente.

Ora vogliamo risolvere il problema del LCA, se prendiamo il CT, e dobbiamo trovare il LCA(i,j) tra due nodi i e j possiamo utilizzare l'array D e svolgere RMQ(pos(i), pos(j)) dove con pos indichiamo la posizione di i e j che viene trovata prendendo la i più a sinistra nell'array dell'euler tour e la j più a destra.

Il problema RMQ può essere risolto sull'array D in spazio $O(n)$ e tempo $O(1)$:

- L'array D viene diviso in blocchi di dimensione $d = \frac{1}{2}Loge$, ogni blocco D_k è formato da $\frac{2e}{d}$ elementi.

- Per ogni blocco D_k calcoliamo il minimo che viene salvato in un array A che avrà al suo interno un numero di elementi pari al numero di blocchi.

- Applichiamo il metodo sparsify sull'array appena creato creando un secondo array M che avrà dimensione $|A|\ Log\ |A|$ ovvero $\frac{e}{Loge}Log\frac{e}{Loge} = O(n)$.

Ci serve una seconda struttura dati per coprire anche il caso in cui le i,j di RMQ(i,j) sono interne ad un blocco.

- L'array D viene trasformato, ogni blocco D_k viene scritto nella forma $< D_k[1], \Delta_k >$ Dove in Δ_k ogni elemento è 1 o 0, 1 se nell'array D abbiamo un +1 rispetto all'elemento precedente e 0 se abbiamo un -1.

- Per ogni blocco salviamo anche la coppia $\Delta_k, posizioneminimo$.

Notiamo che abbiamo 2^{d-1} possibili Δ_k, li vogliamo elencare tutti quanti e per ognuno elencare la possibile posizione del minimo, in tutto sono $2^{\frac{Loge}{2}} = O(\sqrt{n})$. In più vogliamo considerare tutte le possibili coppie (i,j) prese all'interno di D_k, abbiamo $d = \frac{1}{2}Loge$ possibili valori per ognuno dei due quindi in tutto abbiamo $O(Log^2 e) = O(Log^2 n)$ possibili coppie. Vogliamo creare una tabella T in cui troviamo $< i_0, j_0, \Delta_k, posmin >$, questa tabella avrà un numero di entry pari a $O(\sqrt{n}) * O(Log^2 n)$ ovvero $O(n)$ entry e lo creiamo in $O(n)$.

Ora, se dobbiamo risolvere la query RMQ(i,j) con i,j che sono interni al blocco D_k, dobbiamo fare due passaggi:

- Prima calcoliamo i_0 e j_0 che sono le nuove posizioni di i e di j all'interno di D_k. Prendiamo la corrispondente configurazione Δ_k.

- Usiamo la tripla $< i_0, j_0, \Delta_k >$ per andare ad accedere alla tabella T in tempo $O(1)$.

Quindi per eseguire il calcolo dell'RMQ in un array A, possiamo utilizzare una struttura dati che ci permette di risolvere il problema in $O(1)$. La risoluzione di questo problema in $O(1)$ ci permette di risolvere anche il LCA in $O(1)$.

8 Statistical Coding

8.1 HuffMan Coding

Lemma: tra l'albero T e la versione ridotta c'è la seguente relazione: $L_T = L_R + P_x + P_y$.

Dimostrazione: L_T è uguale a $L_T = \sum(p_\sigma L(\sigma)) + (P_x + P_y)(L_T(z) + 1)$, invece $L_R = \sum(p_\sigma L(\sigma)) + (P_x + P_y)(L(z))$. Quindi in pratica prendendo L_T vediamo che la differenza rispetto a L_R è il +1 nella moltiplicazione e quindi è dimostrata la relazione nel lemma.

Teorema: Se prendiamo la lunghezza di tutte le codifiche dei caratteri che creiamo con l'Huffman coding abbiamo una lunghezza $L_c = \sum_{\sigma\in\Sigma} L(\sigma)P[\sigma]$. Possiamo dire che utilizzando l'Huffman coding, questa lunghezza L_c è la minore che possiamo ottenere tra tutti gli algoritmi di encoding, quindi $L_c < L'_c$ dove c è l'Huffman Coding e c' è un qualsiasi altro algoritmo.

Ora possiamo dimostrare il teorema e lo dimostriamo per induzione:

- Partiamo dicendo che vogliamo dimostrare che $L_h < L_c$ ovvero che la lunghezza che ottengo con l'encoding di Huffman è migliore di qualsiasi altro encoding.

- Partiamo con un alfabeto di 2 caratteri, ogni encoder e anche Huffman utilizzano 2 bit per codificare, 1 bit per ogni carattere, quindi qua abbiamo che Huffman è l'ottimo.
- Prendiamo adesso il caso dell'alfabeto con n caratteri e consideriamo che per n-1 caratteri Huffman è l'ottimo e quindi vale $L_h \leq L_c$.
- Nel caso n caratteri diciamo che abbiamo un encoder ottimo quindi vale $L_c \geq L_h$ abbiamo due alberi binari che sono creati dai due encoder, T_C e T_H. Per entrambi possiamo creare la versione ridotta che viene creata su $n-1$ caratteri. Quindi abbiamo R_C e R_H quindi in pratica possiamo dire che vale la relazione $L_{R_H} \leq L_{R_H}$.
- Ora consideriamo il lemma, possiamo riscrivere L_C e L_H come
  $L_C = R_{L_C} + P_x + P_y$
  $L_H = R_{L_H} + P_x + P_y$
  Quindi vale che $L_C \leq L_H$ e quindi $L_C = L_H$ perchè valeva anche l'opposto e quindi l'Huffman Code è ottimo.

Teorema: Data l'entropia H definita come $H = \sum_1^n p(\sigma) Log_2 \frac{1}{p(\sigma)}$, la lunghezza media del codeword che ottengo con Huffman è $H \leq L_H \leq H + 1$.

Questo teorema mi dice con Huffman perdiamo 1 bit per ogni simbolo che viene compresso. Se questa perdita è importante o meno dipende dal valore H dell'entropia. Se abbiamo una sequenza con un solo simbolo abbiamo una entropia pari a 0 perchè un simbolo compare con probabilità 1 gli altri con 0, in questo caso perdere 1 bit è importante. Se invece abbiamo una sequenza con tanti simboli avremo che al massimo l'entropia avrà valore $Log_2|\Sigma|$ e in questo caso perdere un bit non è importante.

8.2 Arithmetic Coding

Teorema: Il numero di bit che vengono emessi dall'Arithmetic Coding per una sequenza S di lunghezza n è al più $2 + nH_0$.
Dimostrazione: sappiamo che il numero di bit in output è pari a:

$Log_2 \frac{2}{s_n} < 2 - Log_2 sn = 2 - log_2(\prod_{i=1}^n P[S[i]]) = 2 - \sum_{i=1}^n Log_2 P[S[i]]$
Se consideriamo n_σ il numero di volte che un simbolo σ occorre in S allora possiamo indicare $P[\sigma] = \frac{n_\sigma}{n}$.
Quindi abbiamo: $2 - \sum_{i=1}^n n_\sigma Log_2 P[\sigma] = 2 - n(\sum_{\sigma \in \Sigma} P[\sigma] Log_2 P[\sigma]) = 2 + nH_0$.

9 Dictionary Problem

Teorema: Data l'hash table con chaining, la lunghezza media delle liste che si creano è pari a $1 + \frac{n}{m}$ dove m è la dimensione dell'hash table e n sono gli elementi inseriti all'interno.
Dimostrazione: Per prima cosa definiamo $I_{x,y}$:

$$I_{x,y}: \begin{cases} 1, & \text{if } h(x) = h(y) \\ 0, & \text{otherwise} \end{cases}$$

Ora consideriamo che la $P(h(x) = h(y)) < \frac{1}{m}$ e quindi abbiamo che

$$E[I_{x,y}] = 1 * P(I_{x,y} = 1) + 0 * P(I_{x,y} = 0) = P(I_{x,y} = 1) \leq \frac{1}{m} \tag{39}$$

Ora calcoliamo N_x che mi indica il numero di chiavi che collidono con x, escluso x:

$$N_x = \sum_{y \in D} E[I_{x,y}] = \sum_{y \in D} P(I_{x,y} = 1) = \frac{n-1}{m} \leq \frac{n}{m} \tag{40}$$

Se consideriamo anche x abbiamo la dimostrazione.

9.1 Universal Hash Function

Teorema: la classe H che contiene le funzioni definite come $h_a(k) = \sum_{i=0}^{r-1} a_i k_i \ mod \ m$ con m primo e a intero positivo è universale.
Dimostrazione: Prendiamo due interi x e y che sono diversi per almeno un bit, $x_0 \neq y_0$. Ora vogliamo dimostrare che il numero di collisioni è limitato da $\frac{|H|}{m}$ e quindi che abbiamo H universale.

$$h_a(x) = h_a(y) ==> \sum_{i=0}^{r-1} a_i * x_i mod m = \sum_{i=0}^{r-1} a_i * y_i mod m \tag{41}$$

$$(a_0 * x_0) \sum_{i=1}^{r-1} a_i * x_i mod m = (a_0 * y_0) \sum_{i=1}^{r-1} a_i * y_i mod m \tag{42}$$

$$a_0(x_0 - y_0) = \sum_{i=1}^{r-1} a_i * (x_i - y_i) mod m \tag{43}$$

$$a_0 = (\sum_{i=1}^{r-1} a_i * (x_i - y_i)) * (x_0 - y_0)^{-1} mod m \tag{44}$$

Ora possiamo dire che $(x_0 - y_0)^{-1}$ esiste ed è un intero compreso tra $[0, |U| - 1]$, quindi abbiamo trovato l'unico valore di a_0 per cui abbiamo una collisione. Allo stesso modo possiamo fare la stessa cosa per le altre $[a_1, ..., a_{r-1}]$, quindi in tutto abbiamo m^{r-1} collisioni e quindi

$$m^{r-1} = \frac{|U|}{m} = \frac{|H|}{m} \tag{45}$$

9.2 Hash Table Perfetta

Teorema: Data una hash table di dimensione q^2 e q chiavi da inserire all'interno dell'hash table, allora abbiamo un numero di collisioni atteso pari a $\frac{1}{2}$. Quindi la probabilità di collisioni p pari a $\frac{1}{2}$.

Dimostrazione: Dato che abbiamo q chiavi da inserire all'interno dell hash table, consideriamo le possibili coppie che possiamo creare, queste sono $\binom{q}{2}$. Se consideriamo che per definizione di hash function ogni coppia ha probabilità $\frac{1}{m}$ di avere una collisione, possiamo scrivere:

$$\binom{q}{2} * \frac{1}{m} = \binom{q}{2} * \frac{1}{q^2} = \frac{q(q-1)}{2} * \frac{1}{q^2} < \frac{q^2}{2q^2} = \frac{1}{2} \tag{46}$$

Teorema: Se inseriamo n chiavi in una tabella hash di dimensione $m = n$ allora la dimensione complessiva delle sotto tabelle che vengono create per memorizzare le n chiavi è pari a $E[\sum_{j=0}^{m-1} n_j^2] < 2n$.

Dimostrazione: Consideriamo:

$$E[\sum_{j=0}^{m-1} n_j^2] = E[\sum_{j=0}^{m-1} (n_j + 2\binom{n_j}{2})] = E[\sum_{j=0}^{m-1} (n_j) + 2\sum_{j=0}^{m-1} \binom{n_j}{2})] = n + E[2\sum_{j=0}^{m-1} \binom{n_j}{2}] \tag{47}$$

Ora consideriamo che $2\sum_{j=0}^{m-1} \binom{n_j}{2}$ indica il numero delle collisioni create da h, dato che abbiamo una universal hash function possiamo dire che questo è pari a $\binom{n}{2}\frac{1}{m}$. Quindi abbiamo

$$n + 2\binom{n}{2}\frac{1}{m} = n + 2\frac{n(n-1)}{2}\frac{1}{m} = 2n - 1 < 2n \tag{48}$$

Quindi in ognuna delle tabelle hash aggiuntive che vengono create ci stanno al più $2n$ elementi.

9.3 Cuckoo Hashing

Teorema: Per ogni coppia i,j di posizioni nella tabella, con $m \geq 2cn$ dove m è la dimensione della tabella e n sono le chiavi nel dizionario, abbiamo che la probabilità che esista un percorso da i a j di lunghezza $L \geq 1$ è al più $\frac{c^- L}{m}$.

Dimostrazione: La dimostrazione è per induzione:

- Per L=1 dobbiamo calcolare la probabilità di avere un arco che collega i a j. Abbiamo $\frac{1}{m}$ di probabilità di avere una chiave che finisce in i e $\frac{1}{m}$ che finisca in j, va moltiplicato per 2 perchè abbiamo 2 funzioni hash per ogni chiave. Ora consideriamo anche il fatto di avere n possibili chiavi e quindi possiamo calcolare la probabilità per il caso L=1. Abbiamo quindi $\sum_{n \in S} \frac{2}{m^2} = \frac{2n}{m^2}$. Dato che abbiamo $m \geq 2n$ allora $\frac{m}{c} \geq 2n$ quindi:

$$\frac{2n}{m^2} = \frac{mc^-1}{m^2} = \frac{c^-1}{m} \tag{49}$$

- Prendiamo poi il caso $L \geq 1$, sappiamo che il teorema vale fino a L-1 e consideriamo una divisione del path che va da i a j:
  - Una prima parte del percorso va da i a z ed è lungo L-1, questo ha una probabilità che è limitata da $\frac{c^{-L+1}}{m}$
  - Una seconda parte va da z a j ed abbiamo probabilità $\frac{c^{-1}}{m}$.

  Ora moltiplichiamo le due probabilità quindi otteniamo $\frac{c^{-L+1}}{m} * \frac{c^{-1}}{m}$ ovvero $\frac{c^{-L}}{m^2}$. Se consideriamo le possibili m posizioni che può prendere z allora abbiamo $\frac{c^{-L}}{m}$

Teorema: Per ogni coppia di chiavi x,y abbiamo che la probabilità che x venga hashato nella stessa posizione di y è $O(\frac{1}{m})$.

Dimostrazione: Consideriamo la probabilità che si crei un ciclo di una lunghezza L ovvero la probabilità di avere un arco che parte dalla posizione i e arriva in i. Consideriamo che stando al teorema precedente questo avviene con probabilità $\frac{c^{-L}}{m}$. Se consideriamo tutte le possibili L abbiamo:

$$\sum_{L=0}^{\infty} \frac{c^{-L}}{m} = \frac{1}{m} \sum_{L=0}^{\infty} \frac{1}{C^L} = \frac{1}{m} * \frac{1}{c-1} \tag{50}$$

Quindi è $O(m)$.

Corollario: Fissando $c \geq 3$ e prendendo $m \geq 6n(1+\epsilon)$ allora possiamo dire che la probabilità di avere un ciclo nel grafo è minore di $\frac{1}{2}$.

9.4 Bloom Filter

Teorema: Vogliamo cercare di capire il valore della probabilità di avere un falso positivo.

Dimostrazione: Per prima cosa consideriamo il caso in cui inseriamo una chiave, vogliamo vedere, dopo ogni inserimento, la probabilità che una certa posizione i rimanga a 0.

$$P(B[i] = 0 dopo uninserimento) = (\frac{m-1}{m})^r \tag{51}$$

Ora consideriamo che facciamo n inserimenti quindi questa probabilità diventa:

$$P(B[i] = 0 dopo ninserimenti) = (\frac{m-1}{m})^r n = (e^{-\frac{rn}{m}}) \tag{52}$$

Ora consideriamo la probabilità che con x non appartenente al dizionario, venga comunque data una risposta positiva:

$$P(falso\ positivo) = P(1 - e^{-\frac{rn}{m}})^r = (0.6185)^{\frac{m}{n}} \tag{53}$$

10 Treap E Skip List

10.1 Treap

Teorema: Se le priorità del treep sono random allora la profondità media è pari a $O(Log_2 n)$.

Dimostrazione: Per questa dimostrazione consideriamo un set di chiavi $x_1, ..., x_n$ in ordine crescente di valore. Definiamo un altro set:

$A_k^i = \begin{cases} 1, & \text{if } x_i \text{ è ancestor di } x_k \\ 0, & \text{otherwise} \end{cases}$

Dato questo set possiamo vedere che la profondità di un nodo k la possiamo calcolare andando a vedere quanti sono i suoi ancestor. Quindi possiamo scrivere: $depth(x_k) = \sum_{i=1}^{n} A_k^i$, se consideriamo poi la media delle varie profondità possiamo scrivere: $E[depth(x_k)] = \sum_{i=1}^{n} E[A_k^i] = \sum_{i=1}^{n} 1 * P(A_k^i = 1) + 0 * P(A_k^i = 0) = \sum_{i=1}^{n} P(A_k^i = 1)$.

Quindi ora rimane da calcolare la probabilità che un certo nodo i sia un ancestor di un altro nodo k. Per definizione di ancestor, preso un nodo x_i è ancestor di x_k se x_i è il nodo con più bassa priorità all'interno dell'intervallo (i, k). Quindi abbiamo $\frac{1}{|k-i|+1}$ probabilità di avere i come minima probabilità in quel set.

Quindi abbiamo: $E[depth(x_k)] = \sum_{i=1}^{n} P(A_k^i = 1) = \frac{1}{|k-i|+1}$ che equivale a $O(Log_2 n)$.

10.2 Skip List

Teorema: L'altezza della skip list è con alta probabilità $O(Log_2 n)$.

Dimostrazione: Abbiamo detto che per ogni elemento della lista abbiamo $\frac{1}{2}$ di probabilità di selezionarlo e portarlo al livello successivo. Quindi se consideriamo un elemento x e vogliamo la probabilità che questo arrivi al livello l ovvero che $P(L(x) \geq l)$ dobbiamo considerare la probabilità che per almeno l volte capita testa quando lancio la moneta. Quindi abbiamo $P(L(x) \geq l) = \frac{1}{2^l}$.

Ora consideriamo che abbiamo n elementi nella lista quindi abbiamo $P(L \geq l) = \sum_{i=1}^{n} \frac{1}{2^l} = \frac{n}{2^l}$. Se fissiamo $l = cLog_2 n$ allora abbiamo:

$$P(L \geq cLog_2 n) = \frac{n}{2^{cLog_2 n}} = \frac{n}{n^c} = \frac{1}{n^{c-1}} \tag{54}$$

Quindi l'altezza della skip list è $O(Log_2 n)$ con alta probabilità.