AE - Soluzioni ai problemi

Luca Corbucci

July 10, 2019

1 Sub Array di Somma massima

1.1 Soluzione lineare con proprietà

```
MaxSum = −∞
TmpSum = 0
b = 1;
for(s=1; s < n; s++) do
    TmpSum += D[s];
    if(MaxSum < TmpSum) then
        MaxSum = TmpSum
        b_0 = b
        s_0 = s
    end if
    if(TmpSum < 0) then
        TmpSum = 0
        b = s+1
    end if
end for
return MaxSum, b_0, s_o
```

Costo: Time $O(n)$, I/O: $O(\frac{n}{B})$

1.2 Soluzione con array ausiliari

- Utilizziamo un array P in cui salviamo le somme prefisse calcolate partendo dall'array D di partenza. Questo calcolo lo svolgiamo scorrendo una sola volta l'array D e inserendo in P ogni volta la somma di $P[i] = P[i-1] + D[i]$. Una volta ottenuto l'array P possiamo riscrivere il problema come:

$$max_s max_{b<s}(P[s] - P[b-1]) = max_s(P[s] - min_{b<s}P[b-1]) \tag{1}$$

- Un array M con i minimi che troviamo all'interno dell'array P. Per riempire questo array scorrendo solamente una volta D dobbiamo calcolare: $M[i] = minM[i-1], P[i]$. Quindi possiamo nuovamente scrivere la formula per il calcolo del Maximum Sub-array sum come:

$$max_s(P[s] - M[s-1]) \tag{2}$$

```
def MaximumSubArray(array):
    n = len(array)
    MaximumSum = -9999999
    minTmpSum = 0
    TmpSum = 0
    b0 = 0
    b = 0
    s0 = 0

    for i in range(0,n):
        TmpSum += array[i]

        if(MaximumSum < TmpSum - minTmpSum):
            MaximumSum = TmpSum - minTmpSum
            b0 = b
            s0 = i

        if(TmpSum < minTmpSum):
            minTmpSum = TmpSum
            b = i+1

    return MaximumSum, b0, s0

testArray = [4,-6,3,1,3,-2,3,-4,1,-9,6]
print MaximumSubArray(testArray)
```

2 Random Sampling

Caso in cui abbiamo il Disk Model e una lunghezza dell'array S da cui prendere i sample fissata:

- Soluzione con copia dei puntatori all'array S in un array S' perchè S non lo possiamo modificare. Ad ogni iterazione prendiamo un valore random h compreso tra (1,n-s) dove n è la lunghezza dell'array e s è il numero di sample che abbiamo già considerato, poi invertiamo S'[n-s] con S[h].

  Codice:

```
Inizializziamo l'array S'[1,n] = S[1,n]
for s = 0,1...m-1 do:
    p = Rand(1,n-s)
    Selezionare l'elemento puntato da S'[p]
    Swap S'[p] con S'[n-s]
```

  Costo: Time: $O(m)$ Space: $O(nLogn)$ I/O: $O(m)$

- Soluzione con l'utilizzo di un dizionario per salvare i sample che sono stati creati. Codice:

```
Inizializziamo il dizionario D
while(|D|<m) do:
    p = Rand(1,n)
    if p not in D:
        insert p in D
```

  Costo: Time: $O(m)$ Space: $O(m)$ I/O: $O(min(\frac{n}{B}, m))$ Il costo di I/O dipende dal fatto che potrei avere una m molto grande e quindi conviene prendere direttamente tutto l'array quindi avere n/b I/O.

- Soluzione con Dizionario e array X in cui viene creato il sample dei dati. L'array X deve anche essere ordinato.

  Codice:

```
Inizializziamo il dizionario D
while(|D|<m) do:
    X = array con m elementi presi random da S
    Ordinare X e vedere se ci sono duplicati
    Inseriamo gli elementi di X in D
```

Costo: Time: $O(mLogm)$ Space: $O(m)$ I/O: $O(min(\frac{n}{B}, m))$

Caso in cui abbiamo uno stream di lunghezza conosciuta:

- In questo caso la soluzione al problema consiste nell'andare a selezionare un elemento presente all'interno dello stream se il numero random compreso tra (0,1) che generiamo è maggiore di $\frac{m-s}{n-j+1}$ ovvero è maggiore del numero di sample che ancora dobbiamo generare diviso il numero di elementi dell'array che dobbiamo ancora vedere. Anche con lo streaming abbiamo probabilità $\frac{1}{n}$ di selezionare ogni elemento dello stream.

  Codice:

```
s = 0
for (j=0; j<n and s<m; j++):
    p = rand(0,1)
    if(p < (m-s)/(n-j+1)):
        select S[j]
        s++
```

Costo: Time: $O(n)$ Space: $O(m)$ I/O: $O(\frac{n}{B})$

Caso in cui abbiamo uno stream di cui però non conosciamo la lunghezza n.

- Una prima soluzione consiste nell'utilizzo di un Min Heap in cui inseriamo ad ogni passaggio un elemento dello stream se viene generato random un numero che è maggiore del minimo dello heap.

  Codice:

```
Creo il Min Heap H con m elementi dummy con valore <-∞, 0>
for each item S[j]:
    r_j = Rand(0,1)
    m = minimum Key in H
    if(r_j > m):
        extract the minimum Key
        Insert <r_j, S[j]> in H
```

Costo: Time: $O(nLogm)$ Space: $O(m)$ I/O: $O(\frac{n}{B})$

- Altrimenti si può usare il Reservoir Sampling che è stato creato da Knuth. In questo caso creiamo un array R di m elementi e ad ogni iterazione andiamo a generare random un numero h, se è minore di m assegnamo all'elemento in posizione h del reservoir sampling l'elemento che stavamo considerando nello stream.

  Codice:

```
Inizializziamo l'array R[1,m] = S[1,m]
for each next item S[j]:
    h = Rand(1,j)
    if (h<m):
        R[h] = S[j]
```

Costo: Time: $O(n)$ Space: $O(m)$ I/O: $O(\frac{n}{B})$

3 List Ranking

3.1 Pointer Jumping

La tecnica del pointer jumping consiste nell'utilizzo in parallelo di n processori che lavorano ognuno su un elemento della lista.

- Ogni processore prende il suo elemento della lista e inizializza il rank a 0 se è l'ultimo elemento della lista mentre lo inizializza a 1 se è un elemento precedente.

- Ogni processore poi esegue le seguenti operazioni: prima calcola per il nodo i il rank corrispondente ovvero $Rank[i] += Rank[Succ[i]]$ poi calcola il successore del nodo i ovvero calcola $Succ[i] = Succ[Succ[i]]$.

Il valore del rank dei vari elementi della lista non cresce in modo lineare ma viene raddoppiato ad ogni step dell'algoritmo, questo vuol dire che la crescita è esponenziale e che il costo dell'esecuzione dell'algoritmo per ognuno degli n processori è $O(logn)$ in tempo.

Costo: Time: $O(n * logn)$.

3.2 Scan e Sort nel 2 Level

- Si crea per ogni elemento della lista una tripla in cui inseriamo $< a_i, b_i, 0 >$ dove a_i è l'id del nodo, b_i è il successore e l'ultimo numero è il rank del nodo che, almeno all'inizio, viene settato a 0.
- Si fa un primo ordinamento delle triple basandoci sul secondo elemento.
- Si scorrono le triple e si creano delle nuove triple $< a_i, b_i, A[b_i] >$ dove $A[b_i]$ è il valore del rank del successore di a_i. (In questo caso l'operazione op è l'assegnamento perchè assegno il rank del successore).
- Si fa un secondo ordinamento basandoci sul primo elemento delle triple.
- Si fa la scansione delle triple e per ogni tripla si modifica il terzo valore andando a calcolare il nuovo rank (in questo caso è somma e assegnamento perchè sommiamo il rank che abbiamo con il rank del successore).

Costo: I/O: $\frac{n}{B} * logn$.

3.3 Divide E Conquer

- Divide: Creiamo un set I di elementi presi dalla lista iniziale, il set I deve essere tale che per ogni elemento presente in I, il successore dell'elemento non viene inserito nel set I. Questo vuol dire che la lunghezza del set I sarà $|I| \leq \frac{n}{2}$ e sarà $|I| > \frac{n}{c}$. Dove $c > 2$.
- Conquer: Partendo dalla lista iniziale eliminiamo gli elementi che sono presenti in I e creiamo una seconda lista L'. Per ogni elemento x presente nella lista tale che $Succ[x]$ è presente in I dobbiamo andare a calcolare: $Rank[x] += Rank[Succ[x]]$ e poi $Succ[x] = Succ[Succ[x]]$. In questo modo il $Rank[x]$ indicherà per ogni x nella nuova lista la distanza tra x e il successore attuale di x.
- Recombine: in questa fase abbiamo calcolato il rank di tutti gli elementi presenti nella lista che abbiamo creato togliendo gli elementi del set I. Per ogni elemento x che appartiene a I dobbiamo andare a calcolare $Rank[x] = Rank[x] + Rank[Succ[x]]$ dove $Rank[Succ[x]]$ indica la distanza tra il successore di x e la fine della lista ed è disponibile perchè l'abbiamo calcolato in precedenza dato che Succ[x] sta sicuramente nella lista L' che abbiamo creato. Mentre $Rank[x]$ indica la distanza tra x e il successore nella lista di partenza.

Costo: I/O: $T(n) = I(n) + O(\frac{n}{b} + T((1 - \frac{1}{c})n)$.

3.4 Coin Tossing

3.4.1 Randomizzato

L'idea del coin tossing è che per ogni elemento della lista viene lanciata una monete, se esce testa e sull'elemento successivo esce croce allora vuol dire che posso selezionare l'elemento. Abbiamo 4 possibili configurazioni di testa e croce, quindi al più in I finiscono $\frac{n}{4}$ elementi.

Costo: I/O: $O(\frac{n}{b})$.

3.4.2 Deterministico

- Per ogni elemento [1,n] della lista assegnamo un valore coin(i) pari a i-1. Quindi da 0 a n-1. Per rappresentare questo valore di coint(i) mi bastano $b = logn$ bit, per ogni elemento bit_b indica questo valore del coin.
- Ora vogliamo passare da n valori del coin a 6 valori, per farlo:
  - Per ogni i nella lista controlliamo $bit_b(i)$ e $bit_b(succ[i])$. Calcoliamo $\Pi(i)$ ovvero la posizione in cui i due valori differiscono e calcoliamo $z(i)$ ovvero il valore di $bit_b(i)$ che differisce rispetto a $bit_b(succ[i])$.
  - Ora assegnamo $coin(i) = 2 * \Pi(i) + z(i)$. Quindi ora mi bastano solamente $log(b) + 1$ bit per rappresentare il coin di i.
- Ora da 6 valori di coin vogliamo passare a 3 valori di coin. Per farlo consideriamo tutti gli elementi della lista che hanno un valore di coin compreso tra 3 e 5. Modifichiamo il loro valore di coin, $coin(i) = 1, 2, 3 - Coin(pred(i)), Coin(succ(i))$.
- Ora possiamo selezionare gli elementi di I, prendiamo tutti gli elementi tali che $coin(i) < coin(succ(i))$ e $coin(i) < coin(pred(i))$.

Costo: al caso pessimo I/O: $O(\frac{n}{b})$.

4 Sorting Atomic items

4.1 Snow Plow

```
H = Creare un Min Heap partendo dall'array U
U = vuoto
while(H non vuoto):
    Min = Estrai il minimo da H
    Scrivi min sul run in output
    prendi un elemento x da S
    if(x < min):
        Add x to U
    else:
        Add x to H
```

- Nella prima fase andiamo a prendere M elementi dall'array S da ordinare e li inseriamo nel Min Heap H.
- Ora entriamo in un while, fino a quando il Min-Heap non è vuoto, estraiamo il minimo da H e lo scriviamo nel run in output, poi prendiamo un elemento da S, se è maggiore del minimo estratto lo mettiamo nello heap, altrimenti nell'array U.
- Quando si svuota il Min-heap vuol dire che invece U sarà pieno, ci saranno M elementi, a questo punto ricominciamo dal punto precedente andando a inserire gli elementi di U nello heap e rientriamo poi nel while.

4.2 Multi Way Merge Sort

L'idea è di utilizzare un Min Heap, carichiamo k blocchi in memoria interna e poi leggiamo il primo elemento dei k blocchi e lo inseriamo all'interno del Min Heap. Per ogni elemento del Min Heap non ci salviamo solamente il suo valore ma anche il run da cui proviene, in questo modo quando estraiamo il minimo dal Min Heap per metterlo in output, potremo andare nel run corrispondente per estrarre un altro elemento da quello specifico run. Il processo di merging richiede un tempo $O(logk)$ per ogni singolo elemento e poi, presi k run la cui lunghezza è pari a z richiede un numero di operazioni di I/O pari a $O(z/B)$. In questo modo riusciamo ad aumentare il 2 della base del logaritmo che diventa M/B perchè ora non uniamo più due run alla volta ma $k = M/B - 1$. Quindi complessivamente il Multi Way Merge Sort richiede una compelessità in termini di tempo pari a $O(nlogn)$, il numero di operazioni di I/O però scende a $O(\frac{n}{b} log_{\frac{M}{B}} \frac{n}{M})$.

4.3 QS Bounded

```
BoundedQS(S,i,j):
    while(j-i > n_0):
        r = pick a random pivot
        p = Partition(S, i, j)
        if(p ≤ (i+j)/2):
            BoundedQS(S,i,p-1)
            i = p+1
        else:
            BoundedQS(S, p+1, j)
            j = p - 1
    InsertionSort(S,i,j)
```

Costo: $O(log_2 n)$ spazio aggiuntivo.

4.4 Multi Way QS

```
Crea (a-1)(k-1) sample di dati dalla sequenza in input
Ordinali in A
For i=0 to len(A):
    prendi come pivot l'elemento A[(a+1)i]
return the pivots
```

L'idea è quella di selezionare non solo k-1 pivot ma un numero superiore di pivot. Selezionando questo maggior numero di pivot andiamo ad ottenere dei pivot finali più distribuiti lungo la sequenza dei dati perchè selezioniamo il primo poi saltiamo di a+1 e selezioniamo il secondo e così via.

4.5 Dual Pivot QS

```
if(S[K]<p):
    l++
    swap S[l] con S[k]
    k++
else if(S[K]>q):
    while(non trovo un elemento < q):
        g--
    swap S[K] con S[g]
    g--
else:
    k++
```

Abbiamo due pivot, p e q, poi abbiamo 4 sezioni dell'array:

- Una prima sezione che va da [0,l] è la parte con gli elementi minori del pivot p
- Una seconda sezione va da [l,k-1] e qua troviamo gli elementi che sono maggiori di p e minori di q
- Abbiamo la sezione da [k,g] che contiene gli elementi che non abbiamo ancora considerato
- L'ultima sezione è quella da da g in poi dove ci sono gli elementi maggiori di q

5 Algoritmi su Grafi

5.1 Kruskal

Gli archi del grafo vengono ordinati in ordine crescente, ad ogni iterazione si prende l'arco di peso minimo e se non si crea un ciclo viene inserito all'interno del MST. Costo $O(ELogV)$

5.2 Prim

Creiamo una coda di priorità in cui inseriamo $< nodo, predecessore, costo >$, all'inizio pred=null e costo=0. Poi partiamo da un nodo e nella coda mettiamo gli archi che sono collegati se il costo è minore. Ad ogni step togliamo il vertice che costa di meno. A seconda di come è implementata la coda di priorità ci costa $O(E + VLogV)$, è da preferire se abbiamo un grafo con molti archi.

5.3 Sybein

Si riduce il numero dei nodi, abbiamo una lista di priorità Q ordinata in base alla prima e alla terza posizione di $< u, v, w(e), u_0, v_0$. Ad ogni iterazione togliamo il minimo e aggiungiamo $min(v, relinkTo), max(v, relinkTo), c$,

6 Set Intersection

6.1 Merge Based

Costo: Time $O(n + m)$.

6.2 Binary Search Based

Costo: Time $O(nLogm)$.

6.3 Mutual Partitioning

Il codice del Mutual Partitioning è il seguente:

```
m = |B| < n = |A|, se il contrario invertire A e B
Prendiamo l'elemento medio di B come pivot
Cerchiamo il pivot con la ricerca binaria in A
Se p = a_j
    Print p // Ho trovato una corrispondenza con il pivot
Calcolare l'intersezione A[1, j-1] ∩ B[1, p-1]
Calcolare l'intersezione A[j+1, n] ∩ B[p+1, m]
```

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.

Costo Time: $O(m(1 + Log\frac{n}{m}))$.

6.4 Doubling

- Abbiamo sempre un array A di n elementi e poi B di m elementi con n¿m. Ammesso che abbiamo cercato un certo elemento b_j in A e che questo si trova in posizione a_i, possiamo dire che l'elemento b_{j+1} si troverà nel sub array che inizia dalla posizione i+1

- Facciamo dei confronti tra l'elemento $B[b_{j+1}]$ e l'elemento $A[i + 2^k]$ per k che va da 0 a n. Quindi ogni volta raddoppiano la distanza dall'elemento che confrontiamo di una potenza di 2.

- Appena troviamo a k per cui $A[i + 2^k]$ ¿ $B[b_{j+1}]$ oppure quando andiamo oltre alla dimensione n dell'array, ci fermiamo e a questo punto facciamo la ricerca binaria di $B[b_{j+1}]$ all'interno del sub array $A[i + 1, minn, i + 2^k]$.

Costo: Time $O(m(1 + Log_2\frac{n}{m}))$.

6.5 2 Level Model

- Partizionare in modo logico l'array A di dimensione n in blocchi di dimensione L, creiamo quindi $\frac{n}{L}$ blocchi.

- Per ognuno dei blocchi creati, prendiamo il primo elemento e lo copiamo in un array ausiliario A' in cui quindi saranno presenti $\frac{n}{L}$ elementi.

- Prendiamo l'array A' e B ed eseguiamo il merge in un unico array, in questo modo gli elementi b_i di B verranno suddivisi all'interno dell'array e saranno delimitati dagli elementi a_i di A'. Quindi avremo per ogni i $a_{i-1} < b_i < a_i$. Questo step dell'algoritmo mi costa $O(\frac{n}{L} + m)$ in termini di tempo.

- Ora prendiamo ogni blocco B_i non vuoto ed eseguiamo il set intersect con il metodo del Merge con il sub array A_i corrispondente. In questo caso il costo è $O(|A_i| + |B_i|)$ quindi $O(L + |B_i|)$ ma dato che questa operazione viene eseguita al più m volte (perchè al più abbiamo m blocchi in B che non sono vuoti) e dato che $|B_i|$ vale m abbiamo che la complessità di questo step è $O(mL + m)$.

Costo: Time: $O(mL + \frac{n}{L})$. I/O: $O(\frac{mL}{B} + \frac{n}{BL} + m)$.

6.6 Random Permuting

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 $O(\frac{mL}{B} + \frac{n}{BL} + m)$. Possiamo migliorare il tempo di esecuzione dell'algoritmo considerando che dato che in B abbiamo al più m blocchi vuoti, non faremo più di m merge tra i blocchi di B e quelli di A. Quindi abbiamo che il tempo di esecuzione si riduce a $O(minn, mL + m)$ e quindi anche il numero di operazioni I/O diventa $O(\frac{mL}{B} + m + m)$.

Per quanto riguarda lo spazio: $n(Log_2\frac{u}{n} + Log_2 minL, Logn + \frac{logn}{L}$ con alta probabilità.

7 Sorting Strings

7.1 MSD

Quello che facciamo è partire con un trie vuoto, poi aggiungiamo una prima stringa e quindi creiamo un nodo unario, ad esempio se questa stringa inizia per 0 avremo un arco che esce dalla posizione 0 dell'array del nodo root e mi punta al puntatore alla stringa. Poi prendiamo la seconda stringa e la inseriamo all'interno del trie, se la seconda stringa inizia per 0 partiamo dalla posizione 0 del nodo root e aggiungiamo un branching node, poi nel branching node gli mettiamo sia il puntatore alla prima stringa inserita e sia il puntatore alla seconda stringa inserita.

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.

7.2 LSD

L'idea dell'algoritmo è di partire dall'ultima cifra delle varie stringhe che vanno ordinate e di ordinarle utilizzando il Counting Sort. Dobbiamo eseguire per L volte (lunghezza delle stringhe) il Counting sort su n elementi che hanno valore massimo σ. Ognuna delle fasi del counting sort produce un ordine differente delle stringhe che verrà sfruttato nella fase successiva, ad esempio se abbiamo 231 e 133 abbiamo che una prima fase mi mette 231 prima di 133 e poi nella seconda fase manterremo lo stesso ordinamento perchè l'algoritmo è stabile, alla fine con la terza fase mettiamo 133 prima di 231. Il fatto che l'algoritmo è stabile diventa essenziale quando abbiamo ad esempio 231 e 233, la prima fase qua diventa cruciale perchè 231 viene messo prima di 233 e poi nei passaggi successivi dato che l'algoritmo è stabile, non vengono invertiti nuovamente i due valori che sono uguali.

Il costo del Counting sort quindi è $O(L(n+\sigma))$ ovvero $O(N+L*sigma)$ mentre lo spazio occupato è $O(N)$.

7.3 Multi Key QuickSort

Il codice di Multi-Key Quicksort:

```
if |R| ≤ 1:
    return R
else:
    scegliere un pivot p in R
    R< = {s: s[i]<p[i]}
    R= = {s: s[i]=p[i]}
    R> = {s: s[i]>p[i]}
    A = Multi-KeyQuicksort(R<, i)
    B = Multi-KeyQuicksort(R=, i+1)
    C = Multi-KeyQuicksort(R>, i)

    return A+B+B
```

Confronti che effettua: $O(n(d_s+Logn)) = O(d+nLogn)$.

8 Searching strings by Prefix

- Una prima soluzione consiste nell'utilizzo di un array di puntatori. Abbiamo n puntatori che puntano ad n stringhe che sono sparse su disco. I puntatori sono ordinati mentre le stringhe su disco no. Quello che facciamo è prendere il pattern P che dobbiamo cercare come pattern e poi eseguiamo la ricerca binaria nell'array dei puntatori. La complessità in tempo è $O(pLogn)$ perchè eseguo per $O(Logn)$ volte la ricerca binaria e ad ogni iterazione vengono confrontati al più p caratteri. In termini di spazio mi costa $O(N + (1 + w)n)$. Il numero di operazioni I/O è $O(\frac{p}{B}Logn)$ se vogliamo solamente il numero delle occorrenze, se vogliamo anche ottenere le occorrenze allora dobbiamo considerare un numero aggiuntivo di I/O pari a $O(Occ)$ e un tempo aggiuntivo di $O(Occ)$.

- Con il 2 level scheme adottiamo una strategia differente, ordiniamo le stringhe anche su disco e poi dividiamo le stringhe ordinate su disco in blocchi grandi B. Sono in tutto $\frac{N}{B}$ blocchi e quindi $\frac{N}{B}$ puntatori al primo elemento dei blocchi. Per fare la ricerca del prefisso serve tempo $O(pLog\frac{N}{B})$ a cui va aggiunto O(Occ) per prendere anche le occorrenze. Lo spazio è $O(N + \frac{N}{B}Logn)$. Il numero di operazioni I/O è $\frac{p}{B}Log\frac{N}{B}$ a cui va aggiunto $O(\frac{Occ}{B})$ se vogliamo anche ottenere le stringhe.

- Usando il front coding abbiamo: Time necessario $O(p\frac{FC(D)}{B})$, I/O $O(\frac{p}{B}\frac{FC(D)}{B})$ in più se vogliamo anche le stringhe abbiamo anche un costo di I/O di $O(\frac{FC(D_{occ}}{B})$. Queste stringhe vanno anche decodificate poi, quindi ho un costo aggiuntivo.

- Compacted Trie: usando il compacted trie serve tempo $O(p+n_{occ})$ per ottenere il numero di stringhe che hanno quel prefisso, poi serve un numero di operazioni I/O pari a $O(p + n_{occ}/B)$. Se vogliamo ottenere anche le stringhe allora abbiamo un costo in tempo $O(N_{occ})$ e un numero di I/O pari a $O(\frac{N_{occ}}{B})$. Lo spazio occupato dal compacted trie è $O(n)$ perchè abbiamo n nodi interni e n foglie.

- Patricia Trie: per risolvere il problema del prefix tree con il Patricia Trie abbiamo un costo in tempo di $O(p)$ e un numero di I/O $O(\frac{p}{B})$ perchè compariamo una stringa. Possiamo anche utilizzarlo in coppia con il Locality Preserving Front Coding in modo da ridurre lo spazio usato su disco.

9 Substring Search

- Possiamo risolvere il problema del Substring Search andando ad utilizzare un suffix array.
  - Una prima soluzione in questo caso consiste nell'utilizzo della ricerca binaria all'interno dell'SA. Ho il pattern P di lunghezza p e lo cerco in SA. Il costo in tempo è $O(pLogn)$. Devo fare due ricerche la prima di P e poi di P per fare in modo di trovare la posizione del primo match e dell'ultimo con il prefisso P. Per trovare il numero di elementi che hanno quel prefisso faccio $pos_{finale} - pos_{iniziale} + 1$. Lo spazio è $O(n(Logn + Log\sigma))$ perchè abbiamo da memorizzare sia i puntatori alle stringhe sia le stringhe.
  - Alternativa: possiamo usare delle ottimizzazioni utilizzando array L_{lcp}, R_{lcp} e le variabili l e r. In questo modo abbassiamo il costo in tempo per fare la ricerca a $O(p + Logn)$. Se devo prendere le stringhe abbiamo un costo aggiuntivo di $O(occ)$.

- Possiamo risolvere il problema utilizzando un suffix tree, in questo caso il problema viene risolto in tempo $O(p + occ)$ e in spazio $O(n)$.

10 Integer Encoding

- Fixed Length: usiamo per ogni intero della sequenza $Logm$ bit dove m è l'intero massimo della sequenza. Ottimo per $Pr[x] = \frac{1}{m}$.

- Unary Encoding: dato x, rappresentiamo con x-1 0 e un 1 finale. Lunghezza totale x. Ottimo per $Pr[x] = \frac{1}{2^{-x}}$.

- Gamma Code: dato x, calcoliamo B(X) e poi $|B(X)|$. L'encoding è formato da $|B(X)|$ scritto in unary più $B(X)$. Ottimo per $Pr[x] = \frac{1}{2x^2}$.

- Delta Code: dato x, calcoliamo B(X) e poi $|B(X)|$. L'encoding è formato da $|B(X)|$ che viene codificato con Gamma code e da $B(X)$. Ottimo per $Pr[x] = \frac{1}{2x(Logx)^2}$.

- Rice Code: partiamo con un intero da codificare e un parametro k. Calcoliamo $q = \frac{x-1}{2^k}$ e poi $r = x - 2^k * q - 1$. q viene rappresentato con unary encoding e r viene rappresentato con binary encoding. Quindi in tutto spazio $k + q + 1$. Ottimo per $Pr[x] = (1 - p)^{x-1}p$.

- PForDelta Code: abbiamo un set di interi che sono compresi tra $[base, base + 2^b - 1]$ e li mappiamo in $[0, 2^b - 1]$. Usiamo anche il gap encoding in questo caso dopo averli mappati. Ora gli interi che possono essere scritti con b bit li scrivo in binario, poi riservo l'ultima configurazione per il carattere escape e salvo in un altro array gli interi che non possono essere scritti in binario. In tutto ho b bit per ogni intero oppure b+w bit se li salvo nell'array delle eccezioni.
- Variable Byte Code: scrivo in binario e divido in blocchi di 7 bit. Ai vari blocchi metto davanti 1 se non è l'ultimo e 0 se è l'ultimo. Nella decodifica mi fermo quando arrivo ad un blocco che ha valore minore di 128.
- Interpolative Code: procedura ricorsiva, viene calcolato l= estremo sinistro dell'array, r=estremo destro, low=valore minimo, hi=valore massimo. M punto medio dell'array e s_m valore nel mezzo dell'array. Vale la relazione $low + m - l < s_m < hi + m - r$. Quindi devo codificare $s_m - low - m + l$ in $hi - low + l - r + 1$ bit.
- Elias Fano Code: non si parte con il gap encoding. Abbiamo n elementi, U è il maggiore (potenza di 2 maggiore). Tutti gli elementi sono codificati in binario poi si divide ogni elemento codificato in due parti, una di dimensione $l = Log_2 \frac{u}{n}$ e una $w = b - l$. La parte low la scrivo esplicitamente in binario in un array, la parte high la scrivo in un array andando a contare le occorrenze di ogni gruppo di bit e poi scrivo in unario questo numero di occorrenze. Spazio complessivo: $nLog\frac{u}{n} + 2n$.

11 Statistical Coding

11.1 Huffman Coding

L'Huffman Coding è un algoritmo greedy che però è ottimo. L'idea è quella di costruire un albero binario in cui abbiamo come foglie i vari caratteri del nostro alfabero Σ con associata la probabilità $P[\sigma]$. L'algoritmo funziona in questo modo:

- Partiamo con una lista dei vari caratteri dell'alfabeto, ordinati in base alla probabilità $P[\sigma]$ crescente.
- Ad ogni passaggio prendiamo i due caratteri che hanno la probabilità più bassa e li uniamo creando un nodo padre.
- Rimuoviamo dalla lista le probabilità dei nodi che abbiamo unito e aggiungiamo la probabilità del nodo padre che corrisponde alla somma delle probabilità dei nodi figli.

In alcuni casi possiamo avere dei caratteri che vanno uniti e che hanno una probabilità uguale, quindi potremmo fare scelte differenti che ci porterebbero ad avere degli alberi differenti. Il problema viene risolto perchè ogni volta, in caso di caratteri con la stessa probabilità, si scelgono quelli che sono più vecchi ovvero che sono stati generati prima. Questa cosa si implementa con una seconda coda di priorità in cui però ordiniamo in base al momento della creazione del nodo e allo stesso tempo anche in base al valore della probabilità.

11.2 Canonical Huffman

Il canonical Huffman Coding inizialmente crea l'albero come nella classica codifica di Huffman e per ogni simbolo σ si calcola la corrispondente lunghezza $L(\sigma)$ della rappresentazione codificata. Canonical Huffman Coding utilizza le seguenti strutture dati:

- Un array *num* in cui $num[l]$ indica la quantità di simboli che hanno una codifica di lunghezza l
- Un array *symb* in cui $symb[l]$ è una lista in cui inseriamo i simboli che vengono codificati con una codifica di lunghezza l
- Un array firstCode in cui in $fc[l]$ memorizziamo la codifica del primo simbolo codificato con l bit.

11.3 Arithmetic Coding

L'algoritmo di compressione dell'arithmetic coding è il seguente:

```
s_0 = 1
l_0 = 1
i = 1
while i <= n:
    s_i = s_{i-1}*P[S[i]]
    l_i = l_{i-1} + s_{i-1}*f[S[i]]

output = < x ∈ [ln, ln + sn), n >
```

La procedura è iterativa e vengono svolte le seguenti operazioni:

- Partiamo con una stringa da codifica che è formata da caratteri che sono in un alfabeto e per ogni carattere abbiamo una probabilità che compaia. Poi Calcoliamo anche la probabilità cumulativa, $f[a] = 0$, $f[b] = P[a]$, $f[c] = P[a] + P[b]$ ecc.
- Ora ad ogni iterazione si prende un carattere dalla stringa da codificare e si calcola s_i e l_i.
- Arrivati alla fine avremo in output $< x \in [ln, ln + sn), n >$, quindi non avremo una coppia ma un intero x che è all'interno del range e la lunghezza della sequenza codificata.

Teorema: Il numero di bit che vengono emessi dall'Arithmetic Coding per una sequenza S di lunghezza n è al più $2 + nH_0$.

12 Dictionary Based Compressor

12.1 LZ77

L'algoritmo di compressione LZ77 si basa su una "sliding windows" di dimensione W che contiene parte della stringa di input. L'algoritmo funziona in modo induttivo, quando arrivo alla posizione i della stringa S, suppongo di aver già compresso i precedenti i-1 caratteri. Il funzionamento della fase di parsing è il seguente:
Ci troviamo in posizione i nella stringa S, cerchiamo la sottostringa α che parte da i, a sinistra di i. Quindi mi sposto verso sinistra di d caratteri e quando trovo una sottostringa in W che match α emetto una tripla $< d, |\alpha|, c$ dove c è il primo carattere della stringa che ha un mismatch. La sottostringa α ha una dimensione variabile che dipende dalla dimensione della sottostringa che matcha e che trovo nella sliding window W. È importante notare che d può essere minore di $|\alpha|$ e quindi potremmo avere un overlap tra la stringa che vogliamo codificare e la stringa che abbiamo in W.

12.2 LZ78

L'idea è di creare un dizionario in modo incrementale quando troviamo una nuova sottostringa da codificare. L'algoritmo funziona in questo modo:

- Viene creato un dizionario che inizialmente è vuoto
- Ammesso di essere arrivati alla posizione i della stringa e quindi di aver codificato fino a quel punto, cerchiamo la sottostringa di S che parte dalla posizione i e che è presente nel dizionario. Quando troviamo la sottostringa di lunghezza massima che è presente nel dizionario, emettiamo in output una coppia che è formata dal'ID della stringa trovata nel dizionario e dal primo carattere che segue la stringa $< ID, next_char >$.
- Poi inseriamo nel dizionario una nuova entry che è uguale a quella che abbiamo trovato con l'aggiunta, in fondo, del $next_char$. Anche questo nuovo elemento prende un nuovo ID.

La struttura dati che viene utilizzata per mantenere questi dati è un uncompacted trie in cui inseriamo in ogni nodo la coppia $< ID, next_char >$. Tramite la sequenza codificata possiamo ricostruire la stringa S di partenza e anche l'uncompacted trie.

13 Dictionary Problem

13.1 Universal Hashing

Universal Hashing: Data H collezione finita di hash function che mappa un universo U in interi $0, 1...m - 1$ allora H è universale se e solo se

$$|h \in H : h(x) = h(y)| \leq \frac{|H|}{m} \tag{3}$$

Questo mi dice anche che la probabilità di avere $P(h(x) = h(y))$ sarà uguale a $\frac{\frac{|H|}{m}}{|H|}$ ovvero $\frac{1}{m}$.

13.2 Perfect, Minimal e Order Preserving Hashing

Definizione: Una funzione hash $h: U -> 0, 1, ..., m-1$ si dice perfetta se per ogni coppia x, y abbiamo che $h(x) \neq h(y)$.

Definizione: Una funzione hash si dice perfetta se data la dimensione m della tabella e la dimensione n del dizionario di elementi da inserire nella tabella, abbiamo che $m = n$. Ovvero se abbiamo un elemento per ciascun bucket.

Definizione: Una funzione hash si dice preserving order se prese due chiavi k_0 e k_1 con $k_0 < k_1$ abbiamo che $h(k_0) < h(k_1)$.

13.3 Cuckoo Hashing

Come funziona il Cuckoo Hashing:

- Scegliamo due funzioni hash, $h_1(x)$ e $h_2(x)$
- Per ogni chiave che viene inserita calcoliamo $h_1(x)$ e $h_2(x)$ e abbiamo varie possibili situazioni:
  - La posizione $h_1(x)$ o $h_2(x)$ è libera, in questo caso inseriamo senza problemi e creiamo un arco tra $h_1(x)$ e $h_2(x)$.
  - Entrambe le posizioni sono occupate, in questo caso dobbiamo spostare uno degli elementi e lo spostiamo nella posizione a cui è collegato e poi facciamo l'inserimento del nuovo valore.

13.4 Bloom Filter

Il funzionamento del bloom filter è abbastanza semplice:

- Abbiamo r funzioni hash $h_0, ...h_{r-1}$
- Abbiamo un array di bit di dimensione m
- Abbiamo n chiavi da inserire

Per ogni chiave che deve essere inserita calcoliamo r volte l'hash utilizzando le r funzioni e settiamo a 1 il bit corrispondente. Una volta che dobbiamo fare una ricerca ci sono due situazioni:

- La ricerca ci porta a trovare un $B[i] = 0$, in questo caso possiamo dire con sicurezza che quella chiave non è presente all'interno dell'array di bit
- Se invece abbiamo per tutte le funzioni hash $B[i] = 1$ allora possiamo dire che:
  - La chiave è effettivamente presente all'interno del bloom filter
  - La chiave non è presente ma comunque viene restituito si e quindi abbiamo un falso positivo.

14 Treaps e Skip List

14.1 Treaps

14.2 Treaps

Un treap è una struttura ad albero in cui ogni nodo contiene due informazioni, una chiave per la ricerca e un valore che indica la priorità. I nodi sono ordinati come se fosse un albero binario di ricerca in base alla chiave, allo stesso tempo però sono ordinati in base alla priorità come se fosse un Min Heap. Quindi, dato un nodo dobbiamo rispettare alcune regole:

- Gli elementi a sinistra sono tutti minori dal punto di vista del valore e quelli a destra sono tutti maggiori
- Andando verso il basso diminuisce il valore della priorità dei vari nodi.

Ogni nodo poi ricorsivamente è un treap e quindi valgono le due proprietà.

14.3 Skip List

Si tratta di un'altra struttura dati che ha una struttura differente rispetto agli alberi binari e ai treep ma che ha le stesse proprietà interessanti. Abbiamo una lista di n elementi, in questo caso la ricerca di un certo elemento andrebbe a costare al caso pessimo $O(n)$ perchè dovremmo scorrere tutta la lista. L'idea è quella di creare un secondo livello della lista in cui andiamo ad inserire parte degli interi che sono nella prima lista. Per ognuno degli interi lanciamo una moneta quindi abbiamo probabilità $\frac{1}{2}$ di finire nel livello successivo e $\frac{1}{2}$ di non finirci. In questo modo la ricerca viene effettuata prima nel livello con meno elementi e poi quando troviamo un elemento che è maggiore di quello che cerchiamo allora andiamo nel primo livello e facciamo una ricerca in una parte della lista completa. Dato che ogni nodo ha probabilità $\frac{1}{2}$ di capitare nella seconda lista allora al più avremo da visitare $\frac{n}{2}$ nodi.