ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea in Matematica

ALGORITMI DI COMPRESSIONE SECONDO LEMPEL ZIV

Tesi di Laurea in Informatica

Relatore:
Chiar.mo Prof.
SIMONE MARTINI

Presentata da:
ANDREA RINALDI

Sessione II
Anno Accademico 2014/2015

A chi mi ha dato la possibilità di conoscere.
A chi continua a volermi bene nonostante le mie assenze.
Alla mia compagna di viaggio.

Indice

Introduzione

1 Lempel Ziv: Fattorizzazione LZ77 **5**

1.1 Metodi di Compressione Lossless 5

1.2 Fattorizzazione LZ77 6

1.2.1 Esempi 8

1.3 Decodifica 11

1.4 Breve Sintesi dei Problemi di LZ77 14

2 Strutture Dati **15**

2.1 Suffix Tree 15

2.1.1 Descrizione del Suffix Tree 15

2.1.2 Il Problema del Matching Esatto 17

2.2 Suffix Array 18

2.3 Longest Common Prefix Array 19

2.4 Longest Previous Factor Array 20

2.5 Quasi Suffix Array 21

2.6 Burrows-Wheeler Transform Array 22

3 Gli Algoritmi LZ **25**

3.1 The CPS1 Algorithms 25

3.2 CPS2 33

4 Risultati Sperimentali **35**

4.1 Confronto Teorico 35

4.2 Implementazione 36
4.3 Discussione dei Risultati del Test 38

5 Conclusioni 41

Introduzione

Supponiamo di volere andare in campeggio. L'oggetto indispensabile sarà sicuramente la tenda dove dormire. Trasportare la tenda già montata su una macchina è difficile se non impossibile. Per questo si preferisce trasportarla da smontata e costruirla una volta arrivati a destinazione.
Il concetto appena illustrato è quello della compressione dei dati, dove la tenda sta ai programmi o file, come la macchina agli strumenti internet o dispositivi di memorizzazione.
La compressione dei dati è, e sarà, molto importante nello sviluppo delle moderne tecnologie. Lo studio di questo problema è spinto dalla necessità di memorizzare informazioni occupando uno spazio sempre minore e dal bisogno di trasmettere dati occupando la minore banda possibile.

Fra i metodi di compressione lossless che permettono la ricostruzione del dato originale nella sua completezza, notevole importanza ha la famiglia degli algoritmi LZ di cui vogliamo trattare. Molti algoritmi lossless si basano sulle idee che nel 1977 i ricercatori israeliani Abraham Lempel e Jacob Ziv hanno presentato nell'articolo *"A universal Algorithm for sequential Data compression"*.
In questo elaborato vogliamo quindi presentare nel Capitolo 1 il metodo di compressione di Lempel Ziv LZ77, mostrarne le strutture dati fondamentali nel Capitolo 2, trattare qualche sua realizzazione concreta (gli algoritmi CPS1 e CPS2) nel Capitolo 3, ed infine, nel Capitolo 4, sfruttando i dati forniti dal test descritto da Al-Hafeedh et al.[1], confrontare l'efficienza degli algoritmi in termini di spazio e tempo utilizzati.

Capitolo 1

Lempel Ziv: Fattorizzazione LZ77

1.1 Metodi di Compressione Lossless

Come anticipato nell'introduzione i metodi di compressione si dividono in due categorie:

- Lossy Data Compression
- Lossless Data Compression

Appartengono alla prima categoria quei metodi che, decodificando un oggetto codificato, ottengono una approssimazione dell'oggetto iniziale. Questo tipo di compressione viene usata spesso per file immagini, dove la perdita di qualche pixel non diminuisce di molto la qualità dell'immagine; lo stesso possiamo dire per file audio e video.
Decodificando un file codificato attraverso un metodo del secondo tipo, si ottiene invece esattamente il file di partenza. Per questo motivo i Lossless Data Compression vengono molto utilizzati per comprimere file di testo, programmi, documenti, database. Un esempio sono i formati ZIP e Rar.

È proprio dei procedimenti lossless che esistano dei dati che tali metodi di compressione non riescono a comprimere in modo adeguato: ci saranno quindi alcuni dati in input che non si comprimeranno una volta elaborati dall'algoritmo.

Fra il 1976 e il 1978 i due scienziati Lempel e Ziv pubblicarono tre ricerche che diedero vita alle fattorizzazioni LZ77 e LZ78 che non ammettono perdita di informazioni:

- Lempel, A. and Ziv, J. 1976 *"On The Complexity of finite sequences"*. IEEE Trans. Inf. Theory 22, 75-81;
- Lempel, A. and Ziv, J. 1977 *"A universal algorithm for sequential data compression"*. IEEE Trans. Inf. Theory 23, 3, 337-343;
- Lempel, A. and Ziv, J. 1978 *"Compression of individual sequences via variable-rate coding". IEEE Trans. Inf Theory 24, 5, 530-536.*

1.2 Fattorizzazione LZ77

Sia χ una stringa cioè una sequenza di caratteri appartenenti ad un alfabeto Σ di grandezza σ. Sia $n =\mid \chi \mid$ la lunghezza della stringa. Per $1 \leq i \leq j \leq n$ denotiamo con $\chi[i, \ldots, j]$ una sottostringa di lunghezza $j - i + 1$ di χ.
Quando $j = i$ scriviamo $\chi[i] = \chi[i, \ldots, j]$.

Definizione 1.1. *La fattorizzazione LZ di χ è la decomposizione di $\chi = w_1, w_2, \ldots, w_k$ dove per ogni w_j, con $j \in 1, \ldots, k$ si ha:*

- ▷ *w_j è un carattere che non si presenta in $w_1, w_2, \ldots, w_{j-1}$;*
- ▷ *oppure, w_j è la più lunga sottostringa che si trova in almeno due occorrenze fra $w_1, w_2, \ldots, w_j$.*

Gli algoritmi dunque si basano sull'idea che in una stringa data in input ci siano più sottostringhe che vengono ripetute. Ogni occorrenza di tale stringa ripetuta viene quindi sostituita da un puntatore che indica la posizione in cui tale sottostringa si è presentata per la prima volta nel testo.
In generale gli algoritmi LZ non lavorano sull'intera stringa data in input, ma solo su una parte individuata ad ogni passo da una sliding window (finestra di testo) di piccole dimensioni.
Tale finestra è composta da due parti:

1. La prima parte, solitamente di qualche migliaio di caratteri, è rappresentata dal **corpo principale**, o **Search Buffer**, la cui dimensione dipende dalla grandezza della finestra scorrevole, ed è costituita da un insieme di elementi già codificati.
2. La seconda parte è di dimensioni molto minori (solitamente 18 caratteri) e viene chiamata **Look-Ahead Buffer** ed è una finestra contenente i primi caratteri che devono ancora essere esaminati e che sono immediatamente successivi al Search Buffer.

Per codificare la sottostringa contenuta nel Look-Ahead Buffer quindi è necessario guardare indietro nel Search Buffer della sliding window.
La divisione fra le due parti della finestra avviene attraverso un puntatore i che identifica l'inizio del Look-Ahead Buffer. Un secondo puntatore j invece cerca nel Search Buffer la più lunga sequenza che sia un prefisso del Look-Ahead Buffer. Una volta individuata tale sequenza, la parte di cui è stata trovata una corrispondenza nel testo già processato, viene sostituita da una tripla (POS,LEN,λ) così composta:

- POS indica la posizione nel Search Buffer del primo carattere della più lunga sottostringa comune e nel caso non esistesse, la posizione del primo carattere del Look-Ahead Buffer.
- LEN è la lunghezza della sottostringa comune. Se LEN=0 vuol dire che la prima lettera del Look-Ahead Buffer non era mai stata processata e viene aggiunta per la prima volta nel dizionario che è il Search Buffer.
- λ è invece il carattere successivo al prefisso nel Look-Ahead Buffer.

Una volta sostituito il testo con la tripla (POS,LEN, λ) accade che:

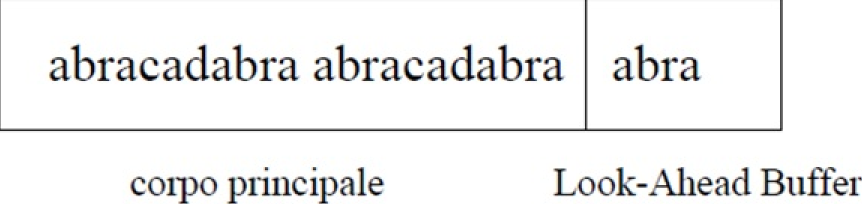
Se LEN$\neq$0 la sliding window si sposta verso destra di tante posizioni quanto è la lunghezza della stringa codificata (LEN). A questo punto viene esaminata una nuova sottostringa, che parte da λ, rispetto ad un nuovo dizionario, cioè il Search Buffer traslato.

Se LEN=0 la sliding window trasla verso destra di una posizione. λ viene inserito nel dizionario e viene esaminata una nuova sottostringa che parte dal carattere successivo a λ.

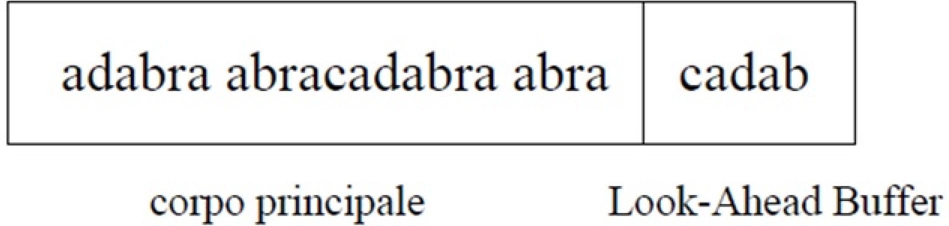
In generale, una volta decomposta la stringa χ , al suo posto compariranno le triple (POS,LEN, λ) mediante le quali potremo sempre recuperare la stringa originale χ.

1.2.1 Esempi

Esempio 1.2. Supponiamo di considerare χ=*abracadabra abracadabra abracadabra* e che ad un certo istante t la sliding window sia questa:



Notiamo che la stringa " *abra*" (con lo spazio incluso), è già contenuta nel corpo principale e quindi viene identificata con la tripla (12,5,$).
Dove 12 è la posizione nel corpo principale del primo carattere della prima occorrenza di " *abra*", mentre 5 è la lunghezza della stringa per la quale abbiamo una occorrenza. $ è un simbolo arbitrario. Il terzo elemento della tripla dovrebbe contenere il carattere successivo nel Look-Ahead Buffer al prefisso trovato, ma in questo esempio la seconda parte della finestra contiene solamente 5 caratteri ed è per questo motivo che scriviamo $ invece di "c".
Al passo successivo dopo aver spostato la sliding window della lunghezza del match trovato al passo precedente (5), la finestra sarà composta in tal modo:



Esempio 1.3. Una stringa del tipo "*abaabaab*" quando viene analizzata dall'algoritmo, viene divisa in a.b.a.abaab.

Ogni fattore viene sintetizzato con una tripla indicante la posizione dove è apparso precedentemente, la sua lunghezza e la lettera successiva, così da ottenere queste triple: (1,0,a),(2,0,b),(1,1,a),(1,5,$) dove $ è un simbolo arbitrario.
Al primo passaggio il Search Buffer è vuoto. Dunque "*a*" non appartiene al dizionario e viene prodotta la tripla (1,0,a). "*a*" entra a far parte del Search Buffer e la finestra viene traslata di una posizione verso destra.
Al secondo passaggio si trova di nuovo un carattere ("*b*") che non è mai stato codificato prima e che quindi non fa parte del dizionario. Viene prodotta la tripla (2,0,b) e "*b*" entra nel Search Buffer.
Al terzo passaggio finalmente troviamo un prefisso. Infatti "*a*" fa parte del Search Buffer. Viene prodotta la tripla (1,1,a), la lettera del mismatch è la successiva "*a*" e la finestra si sposta verso destra di una posizione.
Al quarto passaggio "*aba*" costituisce il Search Buffer mentre "*abaab*" il Look Ahead Buffer. Notiamo che il prefisso "*abaab*" è già presente nella stringa, inizia alla posizione 1 e finisce alla posizione 5 della stringa iniziale, cioè comprende tutto il Search Buffer e i primi caratteri del Look Ahead Buffer. Il metodo LZ permette di poter codificare anche sottostringhe, che si ripetono, di una lunghezza più lunga della dimensione del Search Buffer comprendendo anche caratteri che si trovano all'inizio del Look Ahead Buffer. Viene quindi prodotta la tripla (1,5,$).

Esempio 1.4. χ=*abaababaabbaabbbbbbbb*
Dimensione del search buffer = 5
Dimensione del Look-Ahead Buffer = 6
Per descrivere i passaggi sfrutteremo la Legenda 1.1:

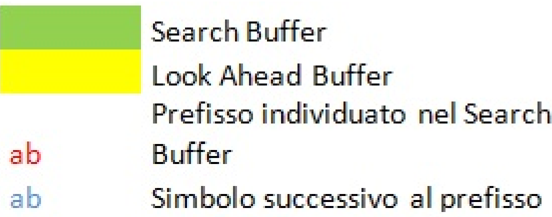


Figura 1.1: Legenda

Osserviamo che per esporre in modo più chiaro il funzionamento di LZ, è stato supposto che la dimensione del Look-Ahead Buffer sia maggiore di quella del Search Buffer; ciò nelle realizzazioni di LZ non capita mai.

Nei primi due passi dell'algoritmo, siccome il dizionario all'inizio è vuoto, non si trovano prefissi nel Search Buffer.

a baaba baabbaabbbbbbbb

Viene quindi prodotta la tripla (1,0,a).

Successivamente:

a b aabab aabbaabbbbbbbb

E si ottiene la tripla (2,0,b).

Nei passaggi successivi invece si trova un prefisso nel Search Buffer.

Terzo Passaggio:

a b a a baba abbaabbbbbbbb

Il prefisso viene codificato in (1,1,a).

Quarto Passaggio:

aba aba b aa bbaabbbbbbbb

Il prefisso viene codificato in (1,3,b).

Osserviamo che da questo passaggio il Search Buffer ha raggiunto la sua dimensione massima e dunque ad ogni codifica tutta la sliding window verrà traslata verso destra. Ci saranno quindi caratteri rimasti a sinistra della finestra che non saranno più utilizzabili come dizionario per le ripetizioni. POS indicherà la posizione dei caratteri sempre relativi alla finestra e non alla stringa originale.

Quinto Passaggio:

a baab a baab b a abbbbbbbb

Viene codificata la tripla (1,4,b).

Come nell'Esempio 1.3, nel passaggio seguente vedremo che il prefisso individuato per il match nel Search Buffer può comprendere anche caratteri del Look-Ahead buffer in modo da poter codificare prefissi più lunghi.

Sesto Passaggio:

abaab a baab baabb b bbbbbb

Il prefisso viene codificato in (2,5,b).

Settimo Passaggio:

abaababaab baa bb bbbbbb b

Viene codificata la tripla (4,6,$).

Come nell'Esempio 1.2, siccome viene codificata tutta la sottostringa presente nel Look-Ahead Buffer, il carattere successivo al prefisso non è incluso nel Look-Ahead Buffer e pertanto viene usato un carattere arbitrario $ e non il successivo carattere "b" della stringa.

Ultimo passaggio:

abaababaabbaabbb b bbbb b

Il prefisso viene codificato in (1,1,$).

Le triple dunque sono: (1,0,a), (2,0,b), (1,1,a), (1,3,b), (1,4,b), (2,5,b), (4,6,$), (1,1,$).

1.3 Decodifica

Una volta che la stringa χ è stata compressa, siamo in possesso di una lista di triple. In un successivo momento potrebbe tornare utile ricostruire la stringa originale. Vi è quindi un **metodo di decodifica** che ricostruisce la χ ricevendo in ingresso la sequenza di triple.

Per esporre in modo più esauriente faremo riferimento all'Esempio 1.4, dove le triple

ottenute sono state (1,0,a), (2,0,b), (1,1,a), (1,3,b), (1,4,b), (2,5,b), (4,6,$), (1,1,$).
L'algoritmo di decodifica parte dalla testa della lista di triple e ricostruisce la stringa originale iniziando dal carattere posto nella prima posizione fino ad arrivare all'ultimo.
Quando incontra una tripla con **LEN=0** aggiunge alla stringa ψ, inizialmente vuota che alla fine dovrà coincidere con χ, la lettera posta come terzo elemento della tripla.
Nel nostro caso ciò accade nei primi due passaggi (useremo ‖ per dividere i caratteri che sono già stati decifrati dalle triple che ancora devono essere analizzate):

1. ‖(1,0,a), (2,0,b), (1,1,a), (1,3,b), (1,4,b), (2,5,b), (4,6,$), (1,1,$);
2. a ‖ (2,0,b), (1,1,a), (1,3,b), (1,4,b), (2,5,b), (4,6,$), (1,1,$);
3. ab ‖ (1,1,a), (1,3,b), (1,4,b), (2,5,b), (4,6,$), (1,1,$);

I caratteri decifrati entrano nel Search Buffer e verranno sfruttati come dizionario per la decifrazione di quelli successivi. Se il Search Buffer ha già raggiunto la sua massima capienza (nel nostro esempio 5 caratteri), i caratteri appena decifrati entrano nel Search Buffer e a far posto loro sono i caratteri che da più tempo fanno parte del Corpo Principale, quindi quelli delle prime posizioni.
Il puntatore **POS**, primo elemento della tripla, è un indice che indica la posizione del carattere all'interno del Search Buffer che non per forza coincide con la posizione del carattere nella stringa come mostrato nell'Esempio 1.4 fra il quarto e il quinto passaggio.
Dunque, se **LEN≠0** l'algoritmo cerca quale carattere si trova nella posizione del Search Buffer indicata dall'elemento POS della tripla; a questo punto copia solo tale carattere se LEN=1, mentre copia anche i successivi n caratteri se LEN=n+1.
Indichiamo in grassetto **abcdefghi** i caratteri già decodificati, presenti nel Search Buffer, usati come dizionario per la decodifica dei caratteri indicati in corsivo *abcdefghi*.

4. **a**b*a*‖(1,3,b), (1,4,b), (2,5,b), (4,6,$), (1,1,$);
5. **aba***aba*‖(1,4,b), (2,5,b), (4,6,$), (1,1,$);

Indichiamo con | l'inizio del Search Buffer in quanto col passaggio precedente si è raggiunta la sua dimensione massima: a|baaba‖(1,4,b), (2,5,b), (4,6,$), (1,1,$).

A questo punto la tripla (1,4,b) indica di prendere il primo carattere del Search Buffer ("b") ed anche i successivi 3 caratteri ("aab").

6. a**baab**a*baab*‖(2,5,b), (4,6,$), (1,1,$).

La stringa ora si presenta così suddivisa: abaab|abaab‖(2,5,b), (4,6,$), (1,1,$).
In questo caso, la tripla (2,5,b) fa partire dalla posizione 2 del Search Buffer e fa copiare i successivi 5 caratteri. Così però si copia anche un carattere che è posto al di fuori del Search Buffer. Non si riscontrano comunque problemi in quanto questo passaggio, come tutti gli altri, non decodifica insieme l'intero blocco, ma solo **carattere per carattere**. Dunque quando si dovrà decifrare il quinto carattere di questo blocco saremo già a conoscenza dell'elemento posto una posizione dopo la fine del Search Buffer. Indichiamo con (***abcdefghi***) i caratteri che in questo passaggio fungeranno da dizionario per la decifrazione di successivi caratteri, ma che inizialmente non possiamo vedere come tali in quanto a loro volta non sono stati ancora decifrati.

7. abaaba**baab**‖(***b***)(4,6,$), (1,1,$).

Suddividendo questo settimo passaggio in cinque fasi (LEN=5) ognuna per un carattere da decifrare, notiamo che (***b***) è il primo carattere del blocco di cinque che sta per essere decifrato. Nella fase 1 non fa parte del corpo principale, ma durante la decifrazione di questa fase l'algoritmo copia il carattere posto in posizione 2 del Search Buffer ("b") e questo sarà il primo carattere del blocco di lunghezza 5 che si sta decifrando. Questo carattere è anche l'ultimo da usare come dizionario per decifrare l'ultimo carattere del blocco. Dunque quando si arriva alla fase 5, l'algoritmo di decifrazione sa quale carattere copiare, nonostante non faccia parte del Search Buffer, in quanto è già stato decifrato.
Ricominciando da questa suddivisione abaababaab|baabb‖(4,6,$), (1,1,$), i passi successivi consistono in:

8. abaababaab|baa**bb**‖ (***bbbb***)(1,1,$);

9. abaababaabbaab|**b**bbbb‖b.

Abbiamo così ricostruito la stringa χ=*abaababaabbaabbbbbbbb*.

1.4 Breve Sintesi dei Problemi di LZ77

Il metodo di compressione LZ77 presenta alcuni problemi che sono facilmente intuibili.
Essendo la sliding window di **dimensioni limitate**, se in un testo si presentano molte ripetizioni ad una distanza maggiore dell'ampiezza della finestra scorrevole, l'algoritmo LZ77 non riesce a sfruttare queste proprietà periodiche.
Un secondo problema, correlato al precedente, deriva dalla **limitata ampiezza** del Look-Ahead Buffer che impedisce di codificare con un unico puntatore occorrenze molto lunghe. Nell'Esempio 1.2 della sezione precedente infatti, avremmo potuto codificare l'intera parola "*abracadabra*" se fosse stata tutta contenuta nel Look-Ahead Buffer.

Perché allora non aumentare le dimensioni del dizionario (il corpo principale della finestra scorrevole) e del Look-Ahead Buffer? In questo modo potremmo rilevare ripetizioni più lunghe, e dunque ottenere compattazione migliore. D'altra parte, però, la ricerca di ripetizioni più lunghe potrebbe essere computazionalmente più costosa, e potrebbe dunque ridurre l'efficienza (cioè il tempo di calcolo) dell'algoritmo. A ben vedere, poi, anche la compattazione potrebbe non essere necessariamente migliore. Infatti, aumentando la dimensione del dizionario avremmo bisogno di un numero maggiore di bit per indicare la posizione nel dizionario. Analogamente, nel secondo caso una maggiore ampiezza del buffer implicherebbe l'aumento di bit per codificare la lunghezza di una frase. In conclusione, avremmo meno triple, ma per codificare *ognuna* di esse avremmo bisogno di un numero maggiore di bit. In conclusione è solo bilanciando tutti i valori in gioco che si ottiene un buon algoritmo.

Capitolo 2

Strutture Dati

Per implementare il metodo di compressione LZ77 possiamo sfruttare alcune strutture dati. In questo capitolo esponiamo le principali.

2.1 Suffix Tree

Un Suffix Tree (ST) è una struttura dati che mostra la composizione interna di una stringa. L'applicazione classica di ST è il problema della sottostringa.

2.1.1 Descrizione del Suffix Tree

Un ST è così definito:

Definizione 2.1. *Un Suffix Tree* A *per una stringa* S *con* m *caratteri è un albero radicato tale che:*

- *ha esattamente* m *foglie numerate da 1 ad* m*;*
- *Ogni nodo interno, al di fuori della radice, ha almeno 2 figli;*
- *Ogni arco è etichettato con una sottostringa di* S*;*

- *Nessuna coppia di archi, uscente dallo stesso nodo, può avere delle etichette che iniziano con lo stesso carattere;*

- *Per ogni foglia i, la concatenazione delle etichette lungo il cammino dalla radice alla foglia i è il suffisso di* S *che parte dalla posizione i, cioè coincide con* S*[i,...,m].*

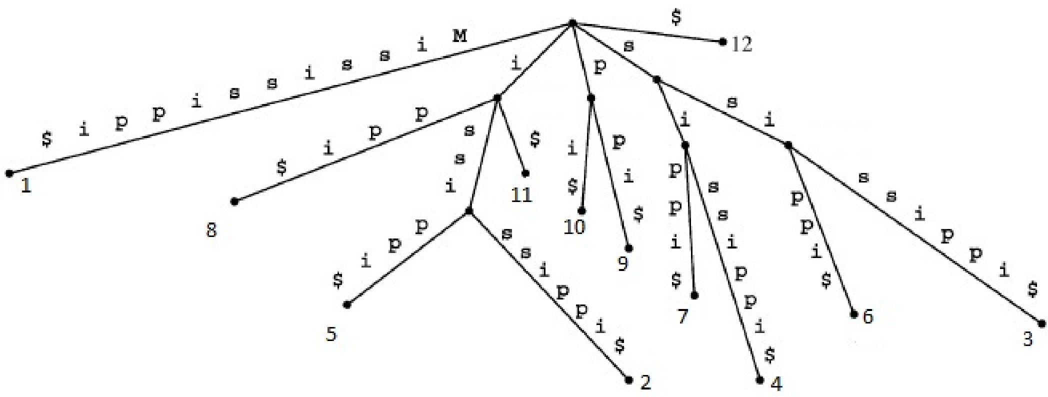


Figura 2.1: Suffix Tree della stringa Mississippi

Come possiamo notare nella Figura 2.1 compare il carattere \$ al termine della stringa S=Mississippi. Questo perché la definizione 2.1 non garantisce l'esistenza di un albero per qualsiasi stringa S data in input. In particolare, se qualche suffisso è prefisso di un altro suffisso di S, il cammino di tale suffisso non terminerebbe in una foglia, quindi si otterrebbe un ST che non soddisfa le condizioni della definizione 2.1. Se per esempio togliessimo il carattere sentinella \$, il suffisso "*i*" sarebbe prefisso dei suffissi "*issisissippi*", "*issippi*", "*ippi*" e quindi il cammino indicante "*i*" non finirebbe in una foglia.
Per risolvere questo problema, si assume che l'ultimo carattere della stringa S sia un carattere che non si trova da nessun'altra parte nella stringa. In questo modo il suffisso di S non sarà mai il prefisso di una sottostringa all'interno di S.
Come da definizione, osservando la Figura 2.1, il cammino dalla radice alla foglia 1 ricostruisce l'intera stringa "*Mississippi\$*", mentre il cammino dalla radice alla foglia 2 individua la sottostringa $S[2, \ldots, 12]$=*ississippi\$* e così via gli altri cammini.

Definizione 2.2.
- *L'etichetta di un cammino dalla radice ad un nodo è la concatenazione ordinata delle sottostringhe che etichettano gli archi del cammino.*
- *L'etichetta di un nodo ν è l'etichetta del cammino dalla radice a ν.*
- *L'etichetta della radice è la stringa vuota, mentre l'etichetta della foglia è il suffisso associato a tale foglia.*

Definizione 2.3.

Per ogni nodo ν di un Suffix Tree, la profondità di ν è il numero di caratteri presenti nella etichetta del nodo.

Dato un testo T di lunghezza n, la costruzione dell'albero richiede un tempo $O(n\log \sigma)$, dove $\sigma \in O(n)$. Data una stringa S di lunghezza m, in un tempo $O(m)$ si riesce a trovare una occorrenza di S in T o determinare che S non è contenuta in T. Quindi la ricerca dell'occorrenza all'interno del testo T avviene in un tempo che non dipende dalla lunghezza di T, ma è proporzionale a quella di S. Esiste anche un algoritmo che costruisce l'albero in un tempo $\Theta(n)$, ma che non è conveniente per stringhe molto lunghe.
Vediamo quindi come questo albero sia utile nell'affrontare il problema del Matching esatto, cioè il problema che data una stringa P, chiede di scoprire se questa è contenuta, e se sì quante volte, nel testo T.

2.1.2 Il Problema del Matching Esatto

Sia data una stringa campione P di lunghezza n e un testo T di lunghezza m. Descriviamo dunque l'approccio al problema del Matching esatto attraverso il Suffix Tree. Si parte dalla costruzione di A, il ST associato al testo T. Successivamente si confrontano i caratteri di P con l'unico cammino individuato in A. Il cammino individuato è unico perché due cammini che escono da un nodo comune hanno l'etichetta che inizia con caratteri diversi come da definizione 2.1.
Il confronto va avanti fino a quando o finiscono i caratteri che costituiscono P, o non si trovano più corrispondenze. Nell'ultimo caso, P non compare da nessuna parte in T. Nel primo caso invece, ogni foglia del sottoalbero che si trova sotto il punto dell'ultima corrispondenza viene numerata con la posizione iniziale di P in T, ed ogni posizione di

P in T numera in questo modo una foglia. Quando tutto P trova una corrispondenza in T, la corrispondenza parte da una posizione j se e solo se P è un prefisso di $T[j, \ldots, m]$. Ciò capita se e soltanto se P etichetta una parte iniziale del cammino dalla radice alla foglia j.

La Figura 2.2 mostra una parte del ST della stringa T=*awyawxawxz* dove vengono

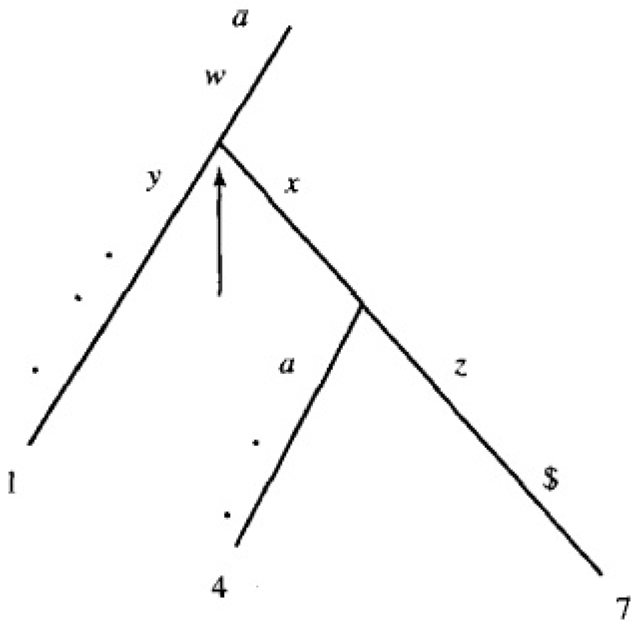


Figura 2.2: Tre corrispondenze di *aw* in *awyawxawxz*

messe in risalto le tre corrispondenze con P=*aw* che si trovano nelle posizioni 1, 4, 7. P corrisponde al cammino fino al punto indicato con la freccia. Tutte le foglie che si trovano nell'albero sotto tale punto sono numerate con la posizione del primo carattere di ciascuna corrispondenza, cioè 1, 4, 7.
In generale se nel testo T ci sono k occorrenze di P, il sottoalbero preso partendo dall'ultima corrispondenza e scendendo, ha esattamente k foglie.

2.2 Suffix Array

Un Suffix Array (SA) è una lista ordinata contenente tutti i suffissi di una stringa in ordine lessicografico. Il SA di χ è un array [1,...,n] nella quale SA[j]=i se e solo se il

suffisso i è il j-esimo in ordine lessicografico fra tutti i suffissi di χ.

Nella Figura 2.3 troviamo nella prima colonna un indice della posizione nell'array

i	**SA**[i]	x[**SA**[i]..n]
1	6	*aab*
2	3	*aabaab*
3	7	*ab*
4	4	*abaab*
5	1	*abaabaab*
6	8	*b*
7	5	*baab*
8	2	*baabaab*

Figura 2.3: Suffix Array di χ=*abaabaab*

ordinato. Nella seconda colonna troviamo SA[i], cioè la posizione del primo carattere del suffisso corrispondente alla posizione i dell'array. Nella terza invece, troviamo il suffisso corrispondente.

I Suffix Array possono essere utilizzati per risolvere gli stessi problemi per la quale si usano i Suffix Tree. Nonostante il SA necessiti di mantenere in memoria il testo originale, dovendo memorizzare solamente dei puntatori, un Suffix Array occupa meno spazio del corrispondente Suffix Tree.

La sua implementazione richiede un tempo $\Theta(n)$ nel peggiore dei casi.

2.3 Longest Common Prefix Array

L'array Longest Common Prefix (LCP) è una struttura dati che viene usata spesso insieme al Suffix Array.

Denotiamo con lcp(i,j) la lunghezza del più grande prefisso comune fra i suffissi i e j. L'array LCP contiene le lunghezze dei più grandi prefissi comuni fra suffissi successivi in SA. Quindi:

$$\text{LCP}[i]=\text{lcp}(\text{SA}[i\text{-}1], \text{SA}[i]), \text{ per } 1<i\leq \text{n}$$

Tornando all'esempio in cui χ=abaabaab, aggiungendo una quarta colonna alla tabella della Figura 2.3, con i risultati di LCP[i], otteniamo la tabella della Figura 2.4.

Dato una stringa χ e il corrispondente Suffix Array, il vettore LCP può essere costruito

i	SA[i]	x[SA[i]..n]	LCP[i]
1	6	*aab*	0
2	3	*aabaab*	3
3	7	*ab*	1
4	4	*abaab*	2
5	1	*abaabaab*	5
6	8	*b*	0
7	5	*baab*	1
8	2	*baabaab*	4

Figura 2.4: SA e LCP di χ=*abaabaab*

in un tempo $\Theta(n)$. Sono stati ideati molti algoritmi per costruire l'array LCP, alcuni dei quali veloci ma richiedenti parecchio spazio, mentre altri meno veloci ma che consumano meno memoria e hanno bisogno di meno spazio per la loro esecuzione.

2.4 Longest Previous Factor Array

Data una stringa χ, introduciamo il Longest Previous Factor array (LPF) definito come segue. Per ogni posizione i in χ, LPF[i] fornisce la lunghezza del più lungo fattore di χ, partendo dalla posizione i, che ha precedentemente una corrispondenza in χ. Quindi LPF[i] è:

$$\text{LPF}[i]=\max\ (\ \{\ \ell \mid \chi[i \ldots i+\ell\text{-}1] \text{ è un fattore di } \chi[1 \ldots i+\ell\text{-}2]\ \} \bigcup \{0\}\ )$$

i	SA[i]	x[SA[i]..n]	LCP[i]	LPF[i]
1	6	*aab*	0	0
2	3	*aabaab*	3	0
3	7	*ab*	1	1
4	4	*abaab*	2	5
5	1	*abaabaab*	5	4
6	8	*b*	0	3
7	5	*baab*	1	2
8	2	*baabaab*	4	1

Figura 2.5: SA, LCP e LPF di χ=*abaabaab*

Riprendendo l'esempio della tabella di Figura 2.4, aggiungendo una quinta colonna a destra, con i valori di LPF[i], otteniamo la tabella di Figura 2.5. Infatti:

Prendendo i=1 otteniamo il suffisso *abaabaab* che non ha alcun fattore precedente in quanto, essendo l'intera stringa χ, non c'è niente che stia prima.

Prendendo i=2 otteniamo il suffisso *baabaab* che non ha alcun fattore precedente in quanto inizia per "b" mentre nella parte precedente della stringa vi è solamente il carattere "a".

Prendendo i=3 otteniamo il suffisso *aabaab* che ha un carattere (χ[1]="a") come fattore precedente.

Prendendo i=4 otteniamo il suffisso *abaab* che ha la sottostringa $\chi[1,\ldots,5]$ come fattore precedente.

e così via.

Gli algoritmi che calcolano LPF sono tutti basati sulla costruzione precedente del Suffix Array associato e richiedono solamente l'aggiunta di una costante allo spazio in memoria usato per la realizzazione di SA e LCP.

2.5 Quasi Suffix Array

Ogni tanto associato all'array LPF si trova il Quasi Suffix Array (QSA). Il QSA si comporta in modo simile ad LPF, ma, invece di fornire la lunghezza del più grande fattore comune, fornisce la posizione nella stringa χ del primo carattere del fattore comune trovato.
Quindi $\forall\ i \in 1,\ldots,n$:

QSA[i]=0 $\Longleftrightarrow$ LPF[i]=0

QSA[i]=j per qualche $j \in 1,\ldots,j$-1 tale che: $\chi[j,\ldots,j+\text{LPF}[i]\text{-}1]=\chi[i,\ldots,i+\text{LPF}[i]\text{-}1]$

Riprendendo l'esempio con χ=*abaabaab* e la tabella di Figura 2.5, aggiungendo una ulteriore colonna per i risultati di QSA[i] otteniamo la Figura 2.6. Notiamo che le due

i	**SA**[i]	x[**SA**[i]..n]	**LCP**[i]	**LPF**[i]	**QSA**[i]
1	6	*aab*	0	0	0
2	3	*aabaab*	3	0	0
3	7	*ab*	1	1	1
4	4	*abaab*	2	5	1
5	1	*abaabaab*	5	4	2
6	8	*b*	0	3	3
7	5	*baab*	1	2	4
8	2	*baabaab*	4	1	2

Figura 2.6: SA, LCP, LPF e QSA di χ=*abaabaab*

strutture dati LPF e QSA insieme forniscono esattamente le informazioni necessarie per poter scrivere i primi due argomenti, POS e LEN, della tripla di LZ che fattorizza una stringa.

2.6 Burrows-Wheeler Transform Array

L'array della Trasformata di Burrows-Wheeler (BWT) è un'altra importante struttura dati. L'idea della trasformata si basa sull'applicazione di una trasformazione reversibile ad un blocco di testo in modo da formare un nuovo blocco contenente gli stessi caratteri, ma che sia più facile da comprimere.
La BWT prende in input una stringa χ di n caratteri. Successivamente crea tutte le n rotazioni del testo χ e le ordina in ordine lessicografico. Infine estrae l'ultimo carattere di ogni rotazione del testo formando una nuova stringa ψ, dove l'i-esimo carattere di ψ è l'ultimo carattere della i-esima rotazione ordinata.
Se la stringa iniziale contiene sottostringhe che vengono spesso ripetute, allora nella BWT di tale stringa troveremo diversi punti in cui lo stesso carattere si ripete più volte. Comprimere lunghe sequenze di caratteri uguali è molto più facile e dunque la stringa trasformata sarà più facilmente comprimibile.
Vogliamo ora trovare BWT di χ=*abaabaab*. A χ viene aggiunto il carattere sentinella \$ che separa il carattere posto all'inizio da quello alla fine di χ. Successivamente vengono elencate tutte le possibili rotazioni del testo, senza considerare \$ come un elemento che occupi uno spazio nella stringa e che quindi richieda una rotazione in più del testo. Le

rotazioni vengono infine ordinate (Figura 2.7).

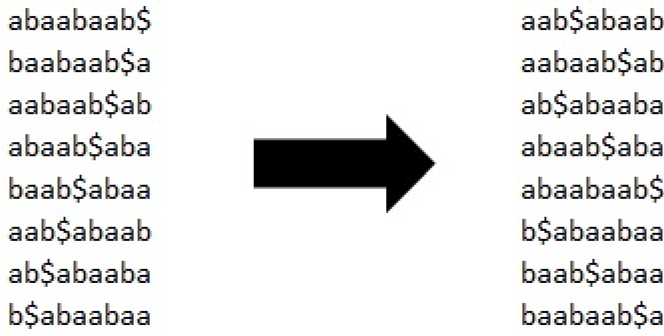


Figura 2.7: Tutte le rotazioni del testo χ=*abaabaab* e il loro ordinamento

A questo punto viene preso l'ultimo carattere di ciascuna rotazione. La n-upla ψ=[b, b, a, a, $, a, a, a] è l'array risultante della BWT applicata alla stringa χ. Aggiungendo tale stringa come settima colonna della tabella di Figura 2.6, otteniamo la Figura 2.8.

i	**SA**[i]	x[**SA**[i]..n]	**LCP**[i]	**LPF**[i]	**QSA**[i]	**BWT**[i]
1	6	*aab*	0	0	0	*b*
2	3	*aabaab*	3	0	0	*b*
3	7	*ab*	1	1	1	*a*
4	4	*abaab*	2	5	1	*a*
5	1	*abaabaab*	5	4	2	$
6	8	*b*	0	3	3	*a*
7	5	*baab*	1	2	4	*a*
8	2	*baabaab*	4	1	2	*a*

Figura 2.8: SA, LCP, LPF, QSA e l'array BWT del testo χ=*abaabaab*

La cosa più interessante di questa trasformazione non è il fatto che fornisce la possibilità di comprimere in modo più semplice, in quanto per ottenere ciò si potrebbero riordinare i caratteri in ordine alfabetico. Ciò che è interessante è la sua reversibilità.
Possiamo infine definire in modo essenziale l'array BWT di una stringa χ supponendo di conoscere il suo SA, infatti:

Se SA[i]>1, allora BWT[i]=χ[SA[i]-1]

Se SA[i]=1, allora BWT[i]=$

Capitolo 3

Gli Algoritmi LZ

In questo capitolo vogliamo descrivere qualche algoritmo che realizza LZ77 sfruttando alcune strutture dati descritte nel capitolo precedente.
L'obiettivo di questi algoritmi è di scomporre una generica stringa χ come da definizione 1.1, o, equivalentemente, di fornire una serie di coppie di interi (POS, LEN) come spiegato nel Capitolo 1. Il terzo elemento, di quella che era stata definita come una tripla (POS, LEN, λ), può essere omesso in quanto attraverso la serie (POS, LEN) si riesce ad ottenere la fattorizzazione di una generica stringa χ nella forma descritta dalla definizione 1.1.
Parleremo dunque della famiglia di algoritmi CPS1 e dell'algoritmo CPS2, entrambi descritti da Chen, Puglisi e Smyth [14][15].

3.1 The CPS1 Algorithms

La dicitura CPS1 racchiude una famiglia di algoritmi, per la realizzazione della fattorizzazione LZ, basati sul calcolo di Suffix Array (SA) e di Longest Common Prefix Array (LCP).
Questa famiglia di algoritmi calcolano due array, uno per POS, uno per LEN. Insieme questi array forniscono la fattorizzazione LZ per ogni posizione della stringa, anche per quei caratteri che, facendo parte di una sottostringa più grande con una corrispondenza nella parte precedente della stringa, sono già stati fattorizzati quando è stato analizzato il primo elemento della sottostringa ripetuta a cui appartengono. Un esempio di ciò si

può vedere nella Figura 3.1, dove il testo x si fattorizza in $a.b.a.aba.ba$, ma nonostante ciò anche le posizioni 5, 6 e 8 possiedono il corrispondente valore negli array POS e LEN.

	1	2	3	4	5	6	7	8
x =	a	b	a	a	b	a	b	a
POS =	1	2	1	1	2	1	2	1
LEN =	0	0	1	3	2	3	2	1

Figura 3.1: I due array POS e LEN della stringa x=*abaababa*

Prendiamo quindi una stringa χ=$\chi[1,\ldots,n]$ in un alfabeto Σ di dimensione α. Decidiamo inoltre che per "suffisso i" si intenda $\chi[i,\ldots,n]$ e che con lcp(i_1,i_2) si indichi il più lungo prefisso comune fra i suffissi i_1 e i_2. Ricordiamo che:

- ◇ SA[j]=i $\Longleftrightarrow$ il suffisso i è il j-esimo in ordine lessicografico fra tutti i suffissi di χ.
- ◇ LPF[i]=max ($\{ \ell \mid \chi[i \ldots i+\ell\text{-}1]$ è un fattore di $\chi[1 \ldots i+\ell\text{-}2] \} \bigcup \{0\}$)

Esaminando da sinistra verso destra gli array POS e LEN di varie fattorizzazioni, sono state fatte due osservazioni che scopriremo essere sfruttate da CPS1.

Osservazione 3.1.

1. Se LCP[i_1]$<$LCP[i_1+1], allora i suffissi j=SA[i_1] e j'=SA[i_1+1] hanno il più lungo prefisso comune di lunghezza LCP[i_1+1] che non è condiviso con SA[i_1-1] né con qualsiasi altro suffisso che si trovi in posizioni precedenti di SA.

2. Se LCP[i_2]$>$LCP[i_2+1], allora i suffissi j=SA[i_2-1] e j'=SA[i_2] hanno il più lungo prefisso comune di lunghezza LCP[i_2] che non è condiviso con SA[i_2+1] né con qualsiasi altro suffisso che si trovi in posizioni successive di SA.

Data l'osservazione 3.1 (1), CPS1 immette in una pila S (stack, struttura dati del tipo LIFO, last in first out) ogni posizione i_1 che indica l'inizio in SA di una sequenza di suffissi tutti con lcp almeno LCP[i_1+1]. Durante l'analisi da sinistra verso destra di SA, se il valore di LCP corrispondente diminuisce, allora la posizione i_2 della discesa indica, data la seconda parte dell'osservazione 3.1, la fine di almeno una sequenza di due o più

suffissi che condividono lo stesso lcp.
Più è grande la differenza di valori di lcp lungo la discesa, più sono le sequenze di suffissi con gli stessi prefissi in comune, che termineranno giunta la discesa nell'array LCP. Riprendendo l'esempio con χ=*abaababa*, riordinando tutti i suffissi e dopo aver calcolato l'array LCP otteniamo la Figura 3.2.

Da questa figura possiamo comprendere meglio ciò che stavamo dicendo. Dalla posizione SA[1] inizia la sequenza di tutti i suffissi che condividono il carattere "*a*" come prefisso. Mentre dalla posizione SA[3] inizia la sequenza di tutti i suffissi che condividono la sottostringa "*aba*". Quando in LCP avviene la discesa da 3 a 0, non solo termina la sequenza "*aba*", ma, essendo molto grande il dislivello, termina anche la sequenza più lunga "*a*".

SA		LCP
8	a	0
3	aababa	1
6	aba	1
1	abaababa	3
4	ababa	3
7	ba	0
2	baababa	2
5	baba	2

Figura 3.2: SA e LCP di χ=*abaababa*

La pila dunque viene svuotata fino a quando viene trovata una posizione i per la quale LCP[i]$\leq$LCP[i_2+1]. A questo punto ogni sequenza i_1,$i_1+1 \ldots i_2$, che rinominiamo s_1,$s_2 \ldots s_k$ specifica le corrispondenti posizioni in χ dei suffissi ($p_1 = SA[s_1]$, $p_2 = SA[s_2] \ldots p_k = SA[s_k]$). Per poter calcolare i valori dell'array POS, è sufficiente analizzare queste posizioni in coppia p_{h-1},p_h, con h che decresce da k fino a 2, assegnando a POS[p] il valore di q, dove p è il più grande fra p_{h-1} e p_h e q il più piccolo. Per ogni posizione p assegnata in POS, il corrispondente valore LEN[p] sarà LCP[i_2]. Ad ogni passaggio, per garantire che una posizione più a sinistra in χ sia sempre accessibile, dobbiamo porre SA[s_{h-1}]=q.
Dopo che tutte le sottostringhe corrispondenti all'lcp corrente sono state analizzate, i

```
— Using SA_x and LCP_x, compute POS[1..n] and LEN[1..n].
i1 ← 1; i2 ← 2; i3 ← 3
while i3 ≤ n+1 do
 — Identify the next position i2 < i3 with LCP[i2] > LCP[i3].
    while LCP[i2] ≤ LCP[i3] do
        push(S, i1); i1 ← i2; i2 ← i3; i3 ← i3+1
 — Backtrack using the stack S to locate the first i1 < i2 such that
 — LCP[i1] < LCP[i2], at each step setting the larger position in POS
 — corresponding to equal LCP to point leftwards to the smaller one,
 — if it exists; if not, then POS[i] ← i.
    q ← SA[i2]; ℓ2 ← LCP[i2]
    assign(POS, LEN, p, q)
    while LCP[i1] = ℓ2 do
        i1 ← pop(S)
        assign(POS, LEN, p, q)
    SA[i1] ← q
 — Reset pointers for the next stage.
    if i1 > 1 then
        i2 ← i1; i1 ← pop(S)
    else
        i2 ← i3; i3 ← i3+1

procedure assign(POS, LEN, p, q)
p ← SA[i1]
if p < q then
    POS[q] ← p; LEN[q] ← ℓ2; q ← p
else
    POS[p] ← q; LEN[p] ← ℓ2
```

Figura 3.3: Algoritmo CPS1-1

valori dei puntatori vengono resettati per vedere se altre coppie (POS,LEN) possono essere calcolate in quanto ci sono altre sequenze che terminano in i_2. In caso contrario i_2 diventa i_3 ed i_3 diventa $i_3 + 1$ e si ricomincia la ricerca.

L'algoritmo, di cui è riportato uno pseudocodice in Figura 3.3, fa uso di tre puntatori i_1,i_2,i_3, per realizzare l'idea appena descritta. Questi puntatori indicano le posizioni in SA e mantengono sempre invariata la condizione: $1 \leq i_1 < i_2 < i_3 \leq n{+}1$.

i_1 indica la posizione più a sinistra in SA di una sequenza di suffissi con lcp di lunghezza almeno LCP[i_1+1];

i_2 indica la fine di almeno una sequenza di suffissi che condividono lo stesso lcp;

i_3 indica la posizione che proseguendo da sinistra verso destra è successiva a i_2 e che non è già stata analizzata.

Prendiamo come esempio la stringa χ=*abaababa*. Il nostro obiettivo è quello di ricostruire la tabella di Figura 3.1, avendo a disposizione solamente SA e LCP mostrati in Figura 3.2.
All'inizio vengono posti i_1,i_2,i_3 nei primi tre posti. Poniamo poi che LCP[n+1], dove n è la lunghezza della stringa, assuma il valore 0 e che il suo corrispondente valore nell'array SA sia "_ ".

	1	2	3	4	5	6	7	8	9
SA	8	3	6	1	4	7	2	5	-
LCP	0	1	1	3	3	0	2	2	0
	i_1	i_2	i_3						

A questo punto gli indici scorrono in avanti e i_1 aggiunge nella pila le posizioni 1 e 3 che indicano l'inizio di una sequenza. Quando i_2 giunge alla posizione 5, fra LCP[i_2] e LCP[i_3] si trova una notevole diminuzione dei valori. Questo indica che nella posizione i_2 termina almeno una sequenza. A questo punto i_1 torna sulle posizioni, che precedentemente erano state aggiunte nella pila. La prima posizione è i=3.

SA	8	3	6	1	4	7	2	5	-
LCP	0	1	1	3	3	0	2	2	0
				i_1	i_2	i_3			

SA	8	3	6	1	4	7	2	5	-
LCP	0	1	1	3	3	0	2	2	0
			i_1		i_2	i_3			

Fra i_1 e i_2 si trovano i seguenti indici: (3,4,5). Questi vengono valutati a coppie (p_h,p_{h-1}) in ordine decrescente. Si pone quindi che p_k=SA[k], ottenendo p_3=6, p_4=1,

p_5=4. Guardando la prima coppia (p_5,p_4)=(4,1) poniamo p uguale al valore massimo (4) e q uguale a quello minimo (1). Infine POS[p]=POS[4]=1, di conseguenza LEN[p]=LEN[4]=LCP[i_2]=3 e SA[p]=q. I due array POS e LEN diventano quindi:

POS[_ ,_ ,_ ,1,_ ,_ ,_ ,_]
LEN[_ ,_ ,_ ,3,_ ,_ ,_ ,_]

Dalla seconda coppia (p_4,p_3)=(1,6) otteniamo p=6, q=1 e di conseguenza POS[6]=1, LEN[6]=3 e SA[3]=1.

POS[_ ,_ ,_ ,1,_ ,1 ,_ ,_]
LEN[_ ,_ ,_ ,3,_ ,3 ,_ ,_]

Ora, dopo aver estratto dalla pila il valore 3, viene estratto il valore 1. Anche dalla posizione 1 inizia una sequenza che termina nella posizione 5. i_1 dunque si posiziona nel posto 1, lì dove inizia la sequenza analizzata, mentre i_2 si pone nell'ultimo posto della sequenza avente lcp condiviso con quello dei posti che lo separano da i_1.

SA	8	3	1	1	4	7	2	5	-
LCP	**0**	1	1	3	3	0	2	2	**0**
		i_1	i_2			i_3			

SA	**8**	3	1	1	4	7	2	5	-
LCP	**0**	1	1	3	3	0	2	2	**0**
	i_1		i_2			i_3			

Vengono quindi calcolati, con il metodo spiegato precedentemente, i seguenti valori: POS[3]=1, LEN[3]=1, SA[2]=1.

POS[_ ,_ ,1 ,1,_ ,1 ,_ ,_]
LEN[_ ,_ ,1 ,3,_ ,3 ,_ ,_]

Successivamente: POS[8]=1, LEN[8]=1], SA[1]=1.

POS[_ ,_ ,1 ,1,_ ,1 ,_ ,1]
LEN[_ ,_ ,1 ,3,_ ,3 ,_ ,1]

Proseguendo in questo modo (Figura 3.4) alla fine si ottengono gli array POS, LEN di Figura 3.1.

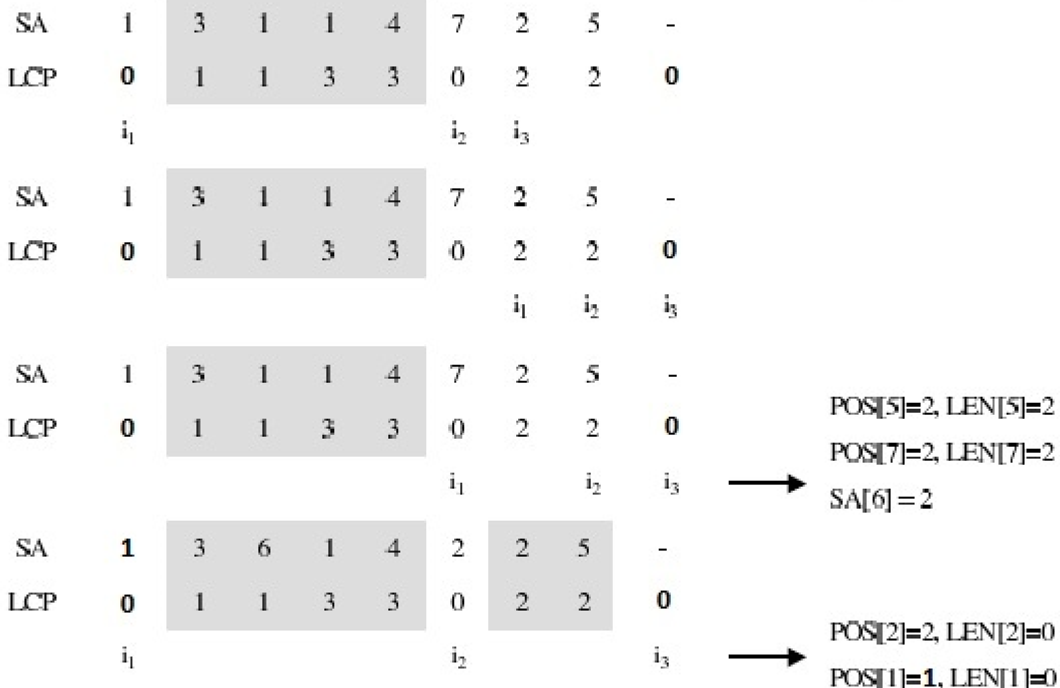


Figura 3.4: Algoritmo CPS1-1 applicato alla stringa χ=*abaababa*

L'algoritmo appena descritto si chiama CPS1-1.
Un secondo algoritmo, sempre della famiglia CPS1, sfrutta il fatto che i puntatori i_1,i_2,i_3 non tornano mai su posizioni per le quali è stato già calcolato POS[SA[i]]. Questo si può notare osservando l'esempio nella Figura 3.4 dove la parte ombreggiata non viene più attraversata dai puntatori. Segue che lo spazio in memoria utilizzato per SA e LCP, può essere dinamicamente riutilizzato per salvare le informazioni calcolate riguardanti POS (posizione e contenuto). Chiamiamo questa variante CPS1-2.
Osserviamo però che può essere risparmiato ancora più spazio rimuovendo dall'algoritmo CPS1-2 ogni riferimento a LEN e calcolando solamente l'array POS. In questo modo non viene assegnato alcuno spazio all'array LEN. Dopo aver calcolato l'array POS, lo spazio precedentemente usato per LCP rimane libero e può essere riutilizzato per LEN. Il calcolo di LEN può essere effettuato in un tempo lineare tramite il semplice algoritmo mostrato nella Figura 3.5 assegnando a POS[n+1] un valore sentinella \$.
Chiamiamo questa terza versione CPS1-3.

```
function LEN(x, POS, i)
j ← POS[i]
if j = i then
    LEN ← 0
else
    ℓ ← 1
    while x[i+ℓ] = x[j+ℓ] do
        ℓ ← ℓ+1
    LEN ← ℓ
```

Figura 3.5: Calcolo di LEN corrispondente a POS[i]

Poiché ad ogni giro del ciclo *while* principale viene calcolata almeno una posizione di POS, segue che il tempo di esecuzione degli algoritmi CPS è lineare in n (lunghezza della stringa che si vuole fattorizzare).
Lo spazio richiesto da CPS1-1 è di $4.25n$ parole di memoria (per χ, SA, LCP, POS e LEN) più lo spazio dedicato alla pila di grandezza massima s parole, dove s al più vale come la massima profondità dell'albero ST associato.
CPS1-2 richiede invece uno spazio di 3.25 parole più quello dedicato alla pila, in quanto memorizza POS sovrascrivendo gli array SA e LCP.
CPS1-3 invece è implementato in due varianti a seconda del modo usato per calcolare l'array LCP:

$CPS1-3a$ è leggermente il più veloce dei due sebbene usi esattamente $3.25n$ parole inclusa la pila;

$CPS1-3b$ che calcola LEN[i] solamente su richiesta, senza la necessità di dedicare uno spazio in memoria all'array LEN e che pertanto usa solamente $2.25n$ parole più la pila.

Tutte le varianti di CPS1 richiedono un tempo $\Theta(n)$ nel caso peggiore e tutte producono un output formato da una coppia di array (POS,LEN).

3.2 CPS2

Il punto debole degli algoritmi CPS1 è che calcolano POS, e alcuni anche LEN, per ogni posizione della stringa, anche per quelle non necessarie. Il prossimo algoritmo è stato pensato per evitare questo spreco.
L'algoritmo CPS2 è un algoritmo che occupa molto meno spazio rispetto agli algoritmi della famiglia CPS1. CPS2 fa uso solamente della stringa χ e di SA insieme ad una struttura dati *RMQ* (Range Minimum Query). RMQ risolve il problema di trovare la posizione dove si trova il minimo valore che viene assunto in una porzione (sub-array) di un array.

Definizione 3.2. *Dato un array A[1,...,n] munito di ordine totale, $RMQ_A(lr, rb)$ è l'indice del più piccolo elemento in A[lr,...,rb] cioè $RMQ_A(lr, rb)$=Ind$(\min_{k \in \{lr,...,rb\}} A[k])$, dove $lr, rb \in \{1, ..., n\}$.*

$RMQ_{SA}(i, j)$ quindi fornisce l'indice del valore minimo fra SA[i], SA[i+1],..., SA[j] e può essere calcolato in un tempo costante e richiede uno spazio $O(n)$.
CPS2 fa uso di una funzione `lzfactor` (Figura 3.6) per calcolare la lunghezza e la posizione dei fattori LZ che iniziano nella posizione i di χ.
CPS2 mantiene invariante il fatto che SA[$lb, ..., rb$] contiene tutti i suffissi che hanno come prefisso $\chi[i, ..., j-1]$ e che almeno uno dei suffissi di questo range inizia in una qualche posizione $p < i$ in χ. Questa condizione è garantita dalla funzione `refine` e da RMQ_{SA}. $SA[RMQ_{SA}(lb, rb)]$ usa rmq per calcolare il minimo di $SA[lb, ..., rb]$. Il minimo è ripetutamente calcolato per range $(lb, ..., rb)$ sempre più ristretti fino a quando viene identificata la più lunga sottostringa che inizia in i e che ha una corrispondenza in una posizione precedente di χ.
Ogni chiamata di `refine` ha un tempo di esecuzione $O(\log(n))$. Notiamo che `refine` non produrrà mai un intervallo vuoto perché stiamo effettuando la ricerca usando un suffisso della stringa stessa come modello, quindi un prefisso con $\chi[i, ..., j]$ si troverà sicuramente. Un metodo efficiente per calcolare i limiti superiori ed inferiori (*lb*,*rb*) del range è effettuare una ricerca binaria che sfrutti il fatto che ogni suffisso in SA[*lb*,*rb*] è in ordine lessicografico [15].

```
— Using SAx and RMQSA compute the position
— and length of the LZ factor beginning at i in x.
function lzfactor(x, SA, i)
match ← i
lb ← 1; rb ← n; j ← i
repeat
    (lb, rb) ← refine(lb, rb, j−i, x[j])
    min ← SA[RMQSA(lb, rb)]
    if min < i then
        match ← min; j ← j+1
until min ≥ i or j > n
return (match, j−i)
```

Figura 3.6: Algoritmo `lzfactor` per cercare la lunghezza e la precedente ripetizione di un fattore LZ ad una data posizione i nella stringa χ

```
output (1, 1)
i ← 2
while i ≤ n do
    (POS, LEN) ← lzfactor(x, SA, i)
    output (POS, LEN)
    i ← i+LEN
```

Figura 3.7: Algoritmo CPS2

Per calcolare la fattorizzazione LZ finale si applica il semplice algoritmo di Figura 3.7. Ogni linea della funzione `lzfactor` esegue in tempo costante ad eccezione di `refine`. Inoltre per produrre l'intera fattorizzazione LZ facciamo al più $n-1$ chiamate di `refine`. Questo vuol dire che il tempo di esercizio totale è $O(n \log(n))$.
Lo spazio richiesto invece è di 1.5 parole, molto meno di quello che viene dedicato agli algoritmi CPS1.
Osserviamo infine che, siccome RMQ fornisce il valore minore ad ogni passaggio, CPS2 associa ad ogni fattore la posizione iniziale della corrispondenza posta più a sinistra. Ciò non era invece scontato in CPS1.

Capitolo 4

Risultati Sperimentali

Questo capitolo presenterà un confronto fra gli algoritmi descritti nel Capitolo 3, sia da un punto di vista teorico, sia da quello sperimentale. Descriveremo quindi i test effettuati da Al-Hafeedh et al. [1] e i loro risultati.

4.1 Confronto Teorico

Algoritmo	Stima Asintotica del Tempo di Esecuzione nel Caso Peggiore	Spazio (parole di memoria)
CPS1-1	$\Theta(n)$	$4.25n+$
CPS1-2	$\Theta(n)$	$3.25n+$
CPS1-3a	$\Theta(n)$	$3.25n$
CPS1-3b	$\Theta(n)$	$2.25n+$
CPS2	$\Theta(n \log n)$	$1.5n$

Figura 4.1: Confronto Teorico degli Algoritmi LZ

La tabella mostrata nella Figura 4.1 riassume le informazioni date precedentemente riguardanti gli algoritmi trattati nel Capitolo 3. La famiglia CPS1 esegue in tempo lineare, mentre CPS2 è eseguito in tempo linearitmico, ma richiede meno spazio. Il "+" indica che, oltre alle parole di memoria scritte in tabella, viene richiesto un ulteriore spazio per pila.

4.2 Implementazione

Gli algoritmi descritti sono molto diversi fra loro. La famiglia CPS1 sfrutta strutture dati (SA e LCP) per eseguire il più velocemente possibile. CPS2 invece sfrutta strutture dati limitate (solo SA), le più semplici possibili, in modo da ridurre lo spazio di memoria necessario alla fattorizzazione. Il compromesso richiede però che il tempo di esecuzione sia maggiore.
Per la costruzione di SA è stata usata la funzione `libdivsufsort` descritta da Mori [17] mentre per la costruzione di LCP l'algoritmo descritto da Puglisi e Turpin [18]. Infine per $RMQ = RMQ_{SA}$ viene impiegato il procedimento proposto da Fischer e Heun [19]. Siccome la pre-elaborazione solitamente è una componente importante del tempo di esecuzione di questi algoritmi, mostriamo tali tempi nella Figura 4.2. Siccome le stringhe in input hanno lunghezze molto diverse fra loro, i tempi sono stati divisi per il numero di caratteri dati in input, ottenendo quindi una potenza.

Stringa	SA	LCP	RMQ
fibo36	0.36	1.10	0.02
fss10	0.35	1.06	0.02
rand2	0.21	0.19	0.02
rand21	0.28	0.14	0.02
chr22	0.23	0.20	0.02
chr19	0.25	0.21	0.02
prot-a	0.29	0.19	0.02
bible	0.17	0.15	0.02
howto	0.21	0.24	0.02
mozilla	0.17	0.19	0.02

Figura 4.2: Tempi di Esecuzione in Microsecondi per Simbolo in Input, per il calcolo di SA, LCP e RMQ

Descriviamo quindi con quali strumenti sono stati effettuati i test esposti in *A comparison of index-based Lempel Ziv LZ77 factorization algorithms* da Al-Hafeedh et al. [1].

Hardware. Tutti i test sono stati condotti su un Server SUN X4600 M2 con quattro processori 2.6 GHz Dual-Core AMD Opteron(tm) 8218 e 32GB di RAM (avente 64-bit come lunghezza parole di memoria).

Software. Il sistema operativo è un Redhat Linux 5.3 che gira su un kernel 2.6.18. Tutte le implementazioni sono state scritte in `C++`, compilate usando GNU `g++` (gcc versione 4.1.2) con l'opzione -O3 e attentamente testate.

Misurazioni delle Prestazioni. Per misurare le prestazioni degli algoritmi è stata utilizzata la libreria di funzioni standard `clock()` di `C++`. Per ogni stringa come risultato finale è stato preso in considerazione il minimo fra 10 cicli, anziché la media, basandosi sul fatto che il tempo minimo sia più veritiero in quanto viene meno condizionato da interferenze casuali del sistema operativo e degli altri utenti. Per misurare il massimo spazio in memoria utilizzato dagli algoritmi è stata utilizzata la libreria glibc.

Dati dei Test. Gli esperimenti sono stati condotti su un campione di file scelti fra la collezione in `http://www.cas.mcmaster.ca/~bill/strings/`.
Le stringhe scelte per i test sono mostrate nella Figura 4.3. Si possono dividere in cinque gruppi:

· Stringhe altamente periodiche. Sono stringhe che nella pratica non si incontrano spesso e che contengono molte ripetizioni (le stringhe di Fibonacci, le stringhe binarie costruite da Franek et al. [20]);

· Stringhe con veramente poche ripetizioni (stringhe casuali con alfabeti piccoli e grandi). Anche questo tipo di stringhe è raro nella pratica;

· Sequenze di DNA con alfabeto {a,c,g,t};

· Sequenze di proteine con alfabeto formato da 20 caratteri;

· Stringhe di un alfabeto molto grande (Caratteri ASCII o Letteratura).

Per ogni stringa forniamo la lunghezza in caratteri (bytes), la dimensione dell'alfabeto σ, il numero di fattori della fattorizzazione LZ e la lunghezza del fattore più grande.

String	Length	σ	No. Factors	Max Factor	Description
fibo36	14930352	2	35	5702887	36th Fibonacci string
fss10	12078908	2	44	5158310	10th run rich string [Franek et al. [20]]
random2	8388608	2	385232	42	Random string, small alphabet
random21	8388608	21	1835235	9	Random string, larger alphabet
chr22	34553758	4	2554184	1768	Human Chromosome 22
chr19	63811651	4	4411679	3397	Human Chromosome 19
prot-a	16777216	23	2751022	6699	Small Protein dataset
bible	4047392	62	337558	549	King James Bible
howto	39422105	197	3063929	70718	Linux Howto files
mozilla	51220480	256	3823511	41323	Mozilla binaries

Figura 4.3: Descrizione delle Stringhe Usate negli Esperimenti

4.3 Discussione dei Risultati del Test

Stringa	CPS1-2	CPS1-3b	CPS2
fibo36	1.51	1.56	0.52
fss10	1.48	1.58	0.52
rand2	0.48	0.56	1.88
rand21	0.51	0.59	2.83
chr22	0.54	0.65	2.96
chr19	0.59	0.72	3.30
prot-a	0.57	0.66	2.73
bible	0.40	0.48	1.33
howto	0.55	0.68	1.88
mozilla	0.45	0.60	1.72

Figura 4.4: Tempi Totali di Esecuzione, in Microsecondi per Simbolo dato in Input, di ogni Algoritmo per la Fattorizzazione LZ

Fra gli algoritmi CPS1 descritti sono stati testati CPS1-2 e CPS1-3b, cioè il più veloce e il più lento fra quelli proposti.
Nelle Figure 4.4 e 4.5 sono forniti, per ogni algoritmo LZ testato, il tempo totale di esecuzione (calcolato in microsecondi per simbolo in input) e il picco di memoria utilizzata (calcolata in parole per simbolo in input). Entrambe le tabelle delle figure tengono conto

Stringa	CPS1-2	CPS1-3b	CPS2
fibo36	4.05	3.05	1.44
fss10	3.94	2.93	1.44
rand2	3.52	2.52	1.44
rand21	3.86	2.86	1.44
chr22	3.65	2.65	1.44
chr19	3.65	2.65	1.44
prot-a	3.82	2.82	1.44
bible	3.72	2.27	1.44
howto	3.73	2.25	1.44
mozilla	3.95	2.95	1.44

Figura 4.5: Memoria Massima Utilizzata, in Parole di Memoria per Simbolo dato in Input, per gli Algoritmi LZ

anche del contributo dato dalla pre-elaborazione.
Possiamo notare che:

Fra i due algoritmi CPS1, CPS1-3b ha un piccolo vantaggio in termini di spazio compensato da uno svantaggio nel tempo di esecuzione;

Quando lo spazio richiesto è un problema, e capita spesso, l'algoritmo CPS2 pare essere la strategia migliore;

Per stringhe altamente periodiche, quindi con pochi fattori ma molto lunghi, l'algoritmo CPS2 è migliore rispetto a quelli CPS1, non solo dal punto di vista dello spazio, ma anche dal punto di vista del tempo di esecuzione.

Capitolo 5

Conclusioni

In questo elaborato abbiamo descritto il metodo di fattorizzazione LZ77 ideato da Lempel e Ziv. Abbiamo mostrato non solo come avviene la fattorizzazione, ma anche la decodifica. Sono state illustrate le strutture dati fondamentali per la costruzione degli algoritmi che realizzano LZ77. Fra le realizzazioni di LZ che tanti autori hanno proposto dal '77 ad oggi, abbiamo scelto di approfondire la famiglia di algoritmi CPS1 e l'algoritmo CPS2.
Abbiamo visto che CPS2 è un algoritmo che solitamente usa poco più di una parola per simbolo di testo dato in input e che quando testato su alfabeti binari si è rivelato essere il metodo migliore. Ad ogni modo, per alfabeti di dimensioni maggiori, CPS2 è un algoritmo nettamente più lento rispetto a CPS1.
Siccome CPS1-3b calcola LEN[i] solamente su richiesta, risparmia spazio rispetto gli altri algoritmi CPS1. Per stringhe molto grandi, per la quale è necessario risparmiare spazio, CPS1-3b potrebbe quindi facilmente essere l'algoritmo scelto anche se più lento di CPS1-2.
Vista l'importanza di minimizzare sempre di più lo spazio utilizzato, la sfida per il futuro sarà quella di progettare algoritmi che sfruttino poche e semplici strutture dati, come CPS2, che siano però competitivi nei tempi di esecuzione con gli algoritmi che sfruttano a pieno le strutture dati (CPS1). In alternativa, visto che buona parte del tempo totale di esecuzione viene impiegato per la pre-elaborazione dei dati, si potrebbero cercare metodi che evitino questi calcoli.

Bibliografia

[1] Al-Hafeedh, A., Crochemore, M., Ilie, L., Kopylova, E., Smyth, W. F., Tischler, G., and Yusufu, M. 2012. A comparison of index-based Lempel Ziv LZ77 factorization algorithms. ACM Comput. Surv. 45, 1, Article 5 (November 2012), 17 pages.

[2] Wikipedia, Compressione dei dati — Wikipedia, L'enciclopedia libera, 2015, `//it.wikipedia.org/w/index.php?title=Compressione_dei_dati&oldid=71945729`, [Online; in data 20-Luglio-2015].

[3] Wikipedia, Compressione dati senza perdita — Wikipedia, L'enciclopedia libera, 2015, `//it.wikipedia.org/w/index.php?title=Compressione_dati_senza_perdita&oldid=70708835`, [Online; in data 20-Luglio-2015].

[4] Lempel, A. and Ziv, J. 1976 *"On The Complexity of finite sequences"*. IEEE Trans. Inf. Theory 22, 75-81;

[5] Lempel, A. and Ziv, J. 1977 *"A universal algorithm for sequential data compression"*. IEEE Trans. Inf. Theory 23, 3, 337-343;

[6] Lempel, A. and Ziv, J. 1978 *"Compression of individual sequences via variable-rate coding". IEEE Trans. Inf Theory 24, 5, 530-536.*

[7] Epifanio, C., *LA COMPRESSIONE, dispense per il corso di tecnologie informatiche*, `http://math.unipa.it/~epifanio/tecnologie/compr07_08.pdf`

[8] Gusfield, D. 1997. *Algorithms on Strings, Trees and Sequences.* Cambridge University Press.

[9] Stoye, J. and Gusfield, D. 2002. Simple and flexible detection of contiguous repeats using a suffix tree. *Theoret. Comput. Sci. 270*, 1-2, 843-856.

[10] Foschini, L., Tesi di Laurea *Studio di un metodo efficiente nella compressione dati*, Università degli studi di Pisa, Facoltà di Ingegneria Corso di laurea in Ingegneria Informatica, `http://lucafoschini.com/papers/Bachelor.pdf`.

[11] Crochemore, M., and Ilie, L. 2008. Computing longest previous factor in linear time and applications. *Inf. Process. Lett.* 106, 2, 75-80.

[12] Burrows, M., Wheeler, D.J., 1994. A block-sorting lossless data compression algorithm. Tech. rep. 124, Digital Equipment Corporation.

[13] Trasformata di Burrows-Wheeler. (15 agosto 2015). Wikipedia, L'enciclopedia libera. Tratto il 11 settembre 2015, 15:05 da `//it.wikipedia.org/w/index.php?title=Trasformata_di_Burrows-Wheeler&oldid=74612753`.

[14] Chen, G., Puglisi, S., and Smyth, W. F. 2007. Fast and practical algorithms for computing all the runs in a string. In *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM'07).* B. Ma and K. Zhang, Eds. Lecture Notes in Computer Science Series, vol. 4580, Springer, Berlin, 307-315.

[15] Chen, G., Puglisi, S., and Smyth, W. F. 2008. Lempel-Ziv factorization using less time and space. *Math. Computer Sci. 1*, 4, 605-623.

[16] Lewenstein, M. and Valiente, G., 2006. Combinatorial Pattern Matching. In *17th Annual Symposium, CPM 2006, Barcelona, Spain, July 5-7, 2006, Proceedings.* Springer Science & Business Media. 414p. .

[17] Mori, Y. 2005. DivSurfSort. `http://code.google.com/p/libdivsufsort/`.

[18] Puglisi, S. and Turpin, A. 2008. Space-time tradeoffs for longest-common-prefix array computation. In *Proceedings of the 19th International Symposium on Algorithms and Computation.* S.-H. Hong, H. Nagamochi, and T. Fukunaga, Eds. Lecture Notes in Computer Science Series, vol. 5369. Springer, Berlin/Heidelberg, 124-135.

[19] Fischer, J. and Heun, V. 2007. A new succint representation of RMQ-information and improvements in the enhanced suffix array. In *Proceedings of the* 1st *International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE 2007)*, B. Chen, M. Paterson, and G. Zhang, Eds. Lecture Notes in Computer Science Series, vol. 4614. Springer, Berlin/Heidelberg, 459-470.

[20] Franek, F., Simpson, R. J., and Smyth, W. F. 2003b. The maximum number of runs in a string. In *Proceedings of the 14th Australasian Workshop on Combinatorial Algorithms*. M. Miller and K. Park, Eds. SNU Press, Korea, 26-35.

Ringraziamenti

Si dice che io abbia raggiunto questo traguardo, sicuramente le mie forze saranno orientate a far sì che sia soltanto un altro inizio.
Ringrazio tutte le persone che si sentono incluse nella dedica, ma in particolar modo ringrazio i miei compagni di facoltà Geni, Debora e Caterina che spesso mi hanno offerto con grande generosità il loro tempo e il loro sostegno.
Infine, vorrei ringraziare il Professor Martini per la sua estrema disponibilità.