Corso: Gestione ed elaborazione grandi moli di dati
Lezione del: 6 giugno 2006
Argomento: Tecniche di compressione con dizionario
Scribes: Fabrizio Lana, Daniele Masato e Luca Polin

1 L'algoritmo LZ77

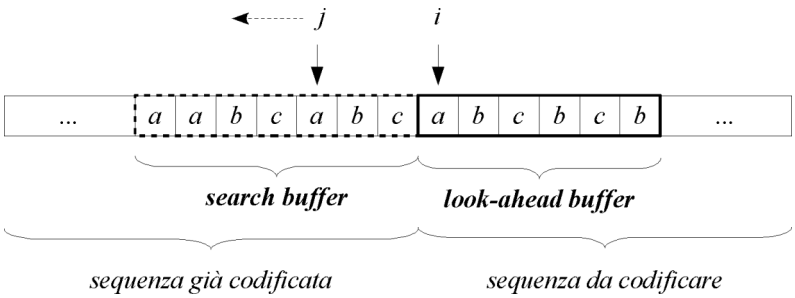
Questo algoritmo di compressione si basa su un approccio adattivo, ovvero mediante il quale viene ricavato un modello per la sorgente direttamente durante il processo di codifica, facendolo successivamente evolvere attraverso l'analisi della sequenza da comprimere.

La maggior parte delle tecniche di compressione adattive basate su dizionario si sono sviluppate a partire dagli articoli pubblicati dai ricercatori israeliani Jacob Ziv e Abraham Lempel nel 1977 e, successivamente, nel 1978 (da cui gli acronimi LZ77 e LZ78).

1.1 Descrizione generale

L'idea di base che contraddistingue tale approccio è la seguente: data una sorgente S, che genera una sequenza X di simboli appartenenti ad un alfabeto A, il dizionario D è contenuto nella porzione di tale sequenza precedentemente codificata; inoltre, inizialmente non è noto alcun modello per S, ma solamente A ed una codifica dei suoi simboli. Operando in questo modo, ci si svincola dal concetto di sorgente e si sfruttano prevalentemente le caratteristiche di X che vengono individuate durante il processo di compressione.

La codifica della sequenza X avviene utilizzando una finestra scorrevole composta di due parti: un *search buffer*, che contiene la porzione di X appena codificata, e un *look-ahead buffer* contenente il successivo segmento della sequenza da codificare. Il codificatore mantiene un puntatore i che identifica l'inizio del look-ahead buffer e, a partire da esso, esamina all'indietro il search buffer al fine di individuare (se esiste) il più lungo prefisso del look-ahead buffer che è contenuto anche nel search buffer. Un puntatore j rappresenterá l'inizio di tale prefisso nel search buffer.



Possono quindi presentarsi due casi:

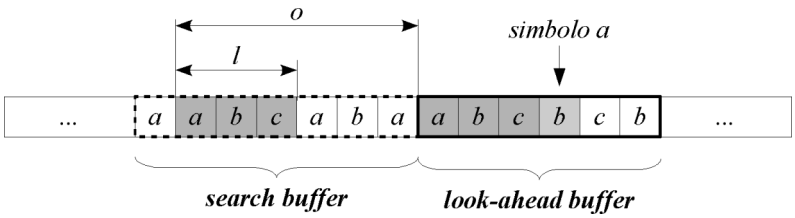
1. la ricerca del prefisso va a buon fine ed il codificatore produce una tripla (o, l, a) dove

   o è la distanza tra i puntatori i e j, chiamata *offset*

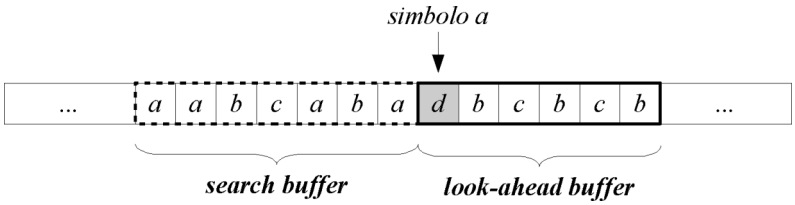
   l è la lunghezza del prefisso individuato

   a è il simbolo che compare nel look-ahead buffer dopo il prefisso

   Si noti che il prefisso non deve necessariamente essere interamente contenuto nel search buffer, ma può proseguire nel look-ahead buffer, perciò l può assumerne valori maggiori di o (vedi esempio a seguire).

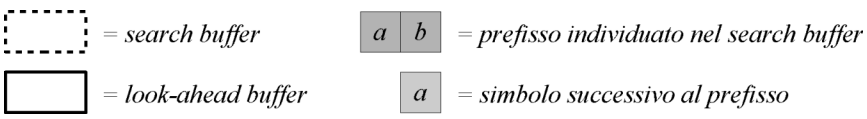


2. la ricerca ha esito negativo, ovvero non esiste alcun prefisso nel search buffer che inizia con il primo simbolo nel look-ahead buffer; in questo caso il codificatore produce una tripla $(0, 0, a)$ dove a rappresenta proprio tale simbolo.



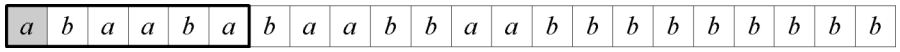
Si può osservare che, sebbene nel primo caso non sia necessario memorizzare il terzo elemento della tripla, questa scelta diventa obbligata quando si verifica la seconda situazione, per consentire al codificatore di proseguire nell'analisi di X ed al tempo stesso mantenere l'informazione associata ad a. La codifica di X viene effettuata mediante la concatenazione di triple dei due tipi viste sopra. La compressione si ottiene ovviamente se i prefissi individuati sono sufficientemente lunghi: in questo caso infatti le triple verranno sicuramente rappresentate con un numero di bit inferiore rispetto alla codifica banale dei prefissi stessi.

Esempio:
sequenza da codificare $X = abaababaabbaabbbbbbbbb$
taglia del search buffer = 5, taglia del look-ahead buffer = 6

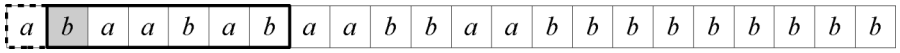


Fase di codifica

Durante i primi due passi dell'algoritmo non viene individuato alcun prefisso nel search buffer (in quanto non è stata codificata una porzione di X sufficientemente lunga), quindi i primi due elementi delle triple prodotte sono pari a 0.

1. 

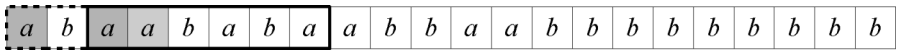
   $\longrightarrow$ viene prodotta la tripla $(0, 0, a)$

2. 

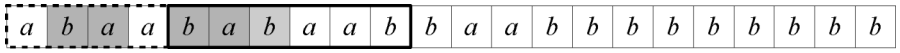
   $\longrightarrow$ viene prodotta la tripla $(0, 0, b)$

Nei successivi quattro passi, invece, il prefisso è presente nel search buffer, quindi:

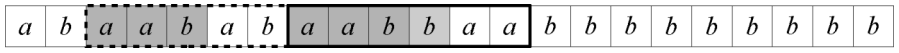
- si calcolano i valori di o e l
- la finestra viene fatta scorrere in avanti di $l + 1$ posizioni (a causa del simbolo a memorizzato)

3. 

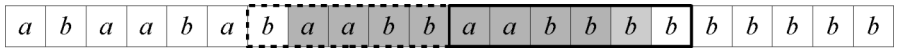
   $\longrightarrow$ viene prodotta la tripla $(2, 1, a)$

4. 

   $\longrightarrow$ viene prodotta la la tripla $(3, 2, b)$

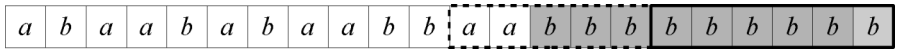
5. 

   $\longrightarrow$ viene prodotta la tripla $(5, 3, b)$

6. 

   $\longrightarrow$ viene prodotta la tripla $(4, 4, b)$

Il passo che segue merita un'analisi più attenta:

7. 

   $\longrightarrow$ viene prodotta la tripla $(3, 5, b)$

in questo caso infatti si potrebbe utilizzare il prefisso bbb individuato nel search buffer per codificare tre simboli nel look-ahead buffer; estendendo però tale prefisso anche nel look-ahead buffer si nota che è possibile codificarne fino a 5 simboli (questo passo diventerà più chiaro esaminando il comportamento dell'algoritmo di decodifica descritto in seguito). L'ultimo simbolo del look-ahead buffer viene infine inserito come terzo elemento della tripla.

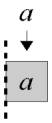
In conclusione si ottiene la sequenza di triple: $(0, 0, a)$; $(0, 0, b)$; $(2, 1, a)$; $(3, 2, b)$; $(5, 3, b)$; $(4, 4, b)$; $(3, 5, b)$.

Fase di decodifica

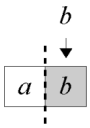
L'algoritmo riceve in ingresso la sequenza di triple e ricostruisce X originale. La linea tratteggiata indica il limite della porzione di X già decodificata.

Le prime due triple sono semplici da decodificare poiché non c'è alcun prefisso già decodificato utilizzabile, quindi i successivi simboli di X saranno rispettivamente a e b.

1. $(0, 0, a) \longrightarrow$

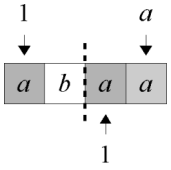


2. $(0, 0, b) \longrightarrow$

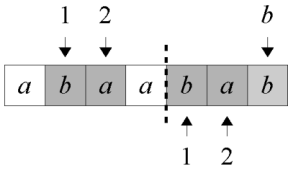


Le quattro triple successive, invece, fanno uso di un prefisso presente nel segmento decodificato: le cifre indicano l'ordine con il quale vengono effettuate le operazioni di copia dei simboli; alla fine di X viene inoltre accodato il terzo elemento della tripla.

3. $(2, 1, a) \longrightarrow$



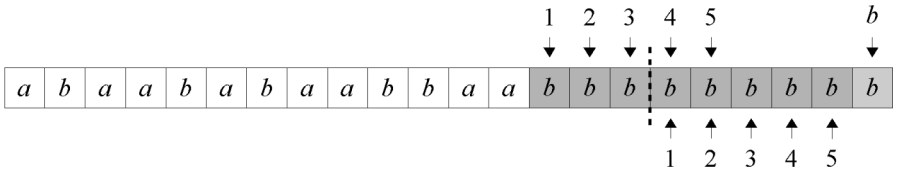
4. $(3, 2, b) \longrightarrow$



I passi 5 e 6 si possono ricavare facilmente operando allo stesso modo.

Ecco invece l'ultimo passo in dettaglio: si può notare che anche la settima tripla viene decodificata correttamente in quanto la copia del prefisso non viene effettuata in blocco, ma simbolo per simbolo.

7. $(3, 5, b) \longrightarrow$



1.2 Implementazione dell'algoritmo

Data la sequenza X, la taglia del search buffer S e la taglia del look-ahead buffer w, l'algoritmo che segue effettua la codifica e restituisce la sequenza Y di triple. L'algoritmo successivo, invece, ricostruisce la sequenza X a partire da Y.

LZ77Code (X, s, w)

$i \leftarrow 1$ /* inizio del look-ahead buffer */
$j \leftarrow 1$ /* indice della prossima tripla da produrre */
$Y \leftarrow$ sequenza vuota

while $i \leq N$ **do**
Trova la coppia (o, l) tale che o e l soddisfino le seguenti proprietà:

- $l \leq w - 1$
- $(i + l) \leq N$
- $o \in [0, s]$
- $X[i - o \ ... \ i - o + l - 1] = X[i \ ... \ i + l - 1]$

$Y[j] \leftarrow (o, l, X[i + l])$
$j \leftarrow j + 1$
$i \leftarrow i + l + 1$
end while

return Y

LZ77Decode (Y)

$i \leftarrow 1$ /* prossimo simbolo da decodificare */
$t \leftarrow |Y|$ /* numero di triple in Y */
$X \leftarrow$ sequenza vuota

for $j \leftarrow 1$ to t **do**
$(o, l, a) \leftarrow Y[j]$

for $k \leftarrow 0$ to $l - 1$ **do**
$X[i + k] \leftarrow X[i - o + k]$
end for

$X[i + l] \leftarrow$ a
$i \leftarrow i + l + 1$
end for

return X

Il decodificatore si comporta più semplicemente rispetto al codificatore, in quanto deve limitarsi a mantenere in memoria la porzione di sequenza già decodificata dalla quale attingere i nuovi simboli, come indicato dalle triple. In *LZ77Code* non è stato riportato lo pseudocodice per la ricerca del prefisso più lungo in quanto può essere implementato in modi diversi, a seconda delle prestazioni (complessità e rate) che si vogliono ottenere.

L'algoritmo LZ77 presenta diverse varianti, sia per l'implementazione sia per la scelta della taglia dei buffer, ed è alla base di molti pacchetti software di compressione quali ZIP, PKZIP, GZIP [GZIP] e del formato grafico PNG; in particolare in tali casi viene utilizzata la variante dell'algoritmo denominata *Deflate*.

2 Una variante di LZ77: l'algoritmo Deflate

Deflate [R1951] (letteralmente "sgonfiare") utilizza un search buffer di taglia 32 KB ed un look-ahead buffer di taglia 258 B: il primo è molto più grande rispetto al secondo perché più è lunga la sequenza all'interno della quale cercare il prefisso, più è probabile che questo sia lungo, quindi permetta di ottenere una codifica più compatta. Questi parametri influiscono inoltre sulla velocità d'esecuzione, dato che più grandi sono i buffer, più lungo è il tempo richiesto per ricercare possibili prefissi.

È comunque bene specificare che l'aumento della taglia del search buffer oltre una certa soglia non comporta necessariamente un sostanziale miglioramento del rate di compressione, in quanto saranno necessari più bit per rappresentare gli offset e le lunghezze dei prefissi nelle triple.

2.1 Ricerca del prefisso di lunghezza massima

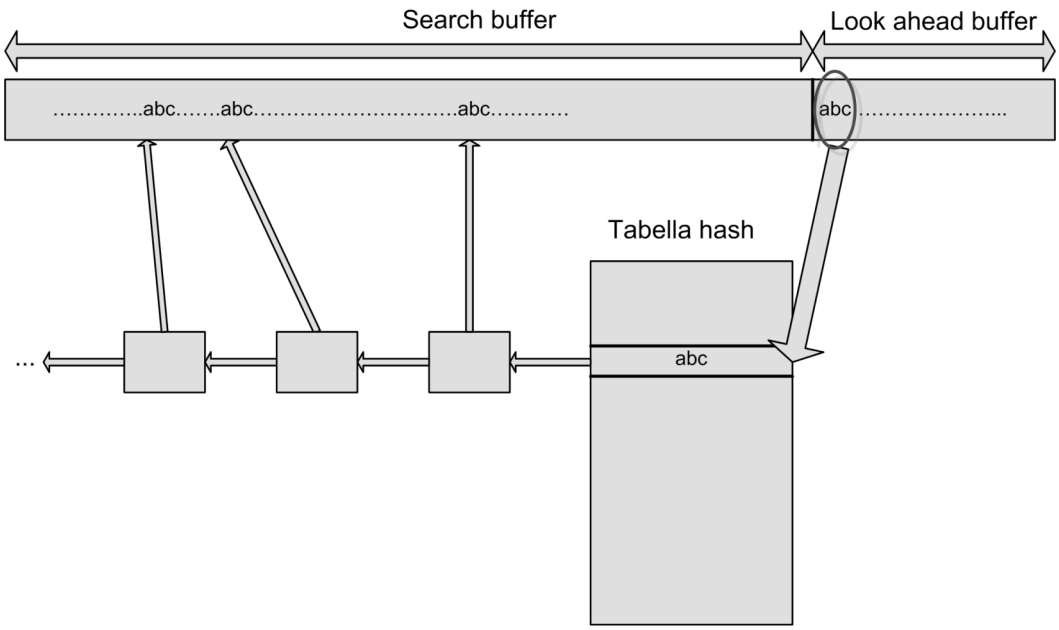
Dato che la ricerca del prefisso di lunghezza massima è un problema piuttosto complesso, spesso le varie implementazioni si accontentano di trovare in modo efficiente un prefisso abbastanza lungo, anche se non ottimo.

L'algoritmo *Deflate* risolve questo problema mediante l'utilizzo di una *tabella hash* con le seguenti caratteristiche:

- la chiave è formata da terne di simboli;
- l'accesso alla tabella viene effettuato mediante i primi tre caratteri del look-ahead buffer.

I bucket associati a ciascuna chiave contengono un puntatore ad una lista che mantiene le posizioni delle occorrenze della chiave nel search buffer, ovvero puntatori alle occorrenze del primo simbolo della terna nel search buffer.

Gli elementi della lista sono inoltre ordinati in modo tale che il primo puntatore indichi l'occorrenza più vicina all'inizio del look-ahead buffer. La tabella hash viene continuamente popolata con tutti i prefissi di lunghezza 3 incontrati, senza alcuna cancellazione. Si possono anche verificare collisioni; a causa di questa eventualità, nella stessa lista si potrebbero trovare parecchie occorrenze di terne (della lista) diverse.É utile stabilire una regola per fermarne l'esplorazione nel caso risultasse troppo onerosa in termini di tempo. Nel caso di *Deflate*, le occorrenze troppo vecchie non vengono considerate e le liste vengono troncate quando superano una determinata taglia.



Una volta individuata la posizione dalla quale inizia un possibile prefisso, si confrontano i simboli successivi fintantoché essi coincidono, quindi si salvano in memoria posizione e lunghezza attuali e si controlla l'occorrenza successiva puntata dalla lista. Alla fine si prenderá il prefisso piú lungo tra quelli controllati.

2.2 Fase di compressione

Deflate comprime le triple prodotte tramite una codifica di Huffman, con approccio semi-statico, che prevede i seguenti passi:

- vengono letti blocchi di 64 KB dalla sequenza da comprimere;
- per ogni blocco vengono generate due diverse codifiche, C per l'offset e C' per le lunghezze e i simboli;
- si trasmettono al ricevitore sia C e C', sia la codifica del blocco.

La scelta di utilizzare la stessa codifica per lunghezze e caratteri garantisce un'ulteriore aumento del rate di compressione se si opera in questo modo:

- la trasmissione di una tripla $(0, 0, a)$, avviene inviando solo $C'(a)$; il ricevitore decodificherà un simbolo, lo accoderà ad X e si aspetterà una ulteriore tripla;
- la trasmissione di una tripla (o, l, a), invece, avviene inviando $C'(l)$, seguita da $C(o)$ e $C'(a)$; in questo caso il ricevitore decodificherà una lunghezza, quindi procederà con la decodifica dei successivi elementi della tripla corrente e con l'accodamento ad X del prefisso indicato.

2.3 Fase di decompressione

La principale difficoltà della decompressione (*Inflate*, "gonfiare"), consiste nell'individuare un algoritmo veloce per l'esplorazione degli alberi di Huffman.

L'idea di base consiste nel decodificare velocemente le codeword frequenti e più lentamente le codeword più lunghe (ma allo stesso tempo più rare). Per far ciò l'algoritmo si avvale, in sostituzione dell'usuale albero di Huffman, di una tabella di look-up nella quale sono collocate le codeword e i simboli associati. La tabella contiene 2^n elementi che rappresentano tutte le possibili codeword di lunghezza n, di conseguenza un simbolo viene duplicato in tutte le entry della tabella la cui rappresentazione binaria inizia come la codeword abbinata a tale simbolo. Ad esempio, data una tabella con entry di taglia 9 bit ed una codeword di lunghezza 4 bit, il simbolo ad essa associato verrà duplicato 32 volte, nelle posizioni in cui la rappresentazione binaria della entry differisce per gli ultimi 5 bit dalla codeword; un simbolo associato ad una codeword di 9 bit apparirà invece una sola volta. Per le codeword di lunghezza superiore a n la tabella conterrà solo i primi n bit, che fungeranno da puntatore ad un'ulteriore tabella di secondo livello. A seconda del valore n, il procedimento può essere ripetuto k volte, ottenendo così $k+1$ tabelle. Infine, accanto ad ogni simbolo, viene memorizzata nella tabella anche la lunghezza effettiva della codeword ad esso associata.

L'algoritmo legge quindi il flusso di dati da decodificare a gruppi di n bit, che utilizza come chiave per accedere alla relativa entry della tabella e ricavarne il simbolo associato. Se la lunghezza effettiva della codeword è inferiore ad n, i bit rimanenti verranno preposti al successivo gruppo per poter proseguire la decodifica. È necessario precisare che tale procedimento funziona perché si sta utilizzando un codice prefisso.

Lo scopo dell'algoritmo è dunque quello di determinare un valore di n che ottimizzi le prestazioni, ovvero di individuare il giusto compromesso tra la lunghezza della tabella (che richiede tempo per essere compilata, soprattutto per i duplicati) e il tempo di decodifica. *Inflate* tende a mantenere il valore di n vicino al numero di bit necessari per rappresentare tutti i simboli dell'albero, più un bit.

Esempio: si consideri la seguente codifica di Huffman su un alfabeto A di 10 simboli, con lunghezza massima di 6 bit.

Simbolo	Codeword	Probabilità	Simbolo	Codeword	Probabilità
a	0	1/2	b	10	1/4
c	1100	1/16	d	11010	1/32
e	11011	1/32	f	11100	1/32
g	11101	1/32	h	11110	1/32
i	111110	1/64	j	111111	1/64

Siano $n = 3$ bit e $k = 1$, le tabelle di look-up che ne risultano sono riportate nella pagina successiva.

Tabella T

Elemento	Simbolo	Lunghezza codeword
000	a	1
001	a	1
010	a	1
011	a	1
100	b	2
101	b	2
110	puntatore alla tabella U	
111	puntatore alla tabella V	

Tabella U

Elemento	Simbolo	Lunghezza codeword
00	c	1
01	c	1
10	d	2
11	e	2

Tabella V

Elemento	Simbolo	Lunghezza codeword
000	f	2
001	f	2
010	g	2
011	g	2
100	h	2
101	h	2
110	i	3
111	j	3

Si nota che i simboli con codeword più corte di 3 bit vengono duplicati in tutte le entry la cui rappresentazione inizia con tale codeword. La tabella U ha elementi di taglia 2 perché la più lunga codeword che inizia con 110 ha taglia 5 bit, mentre la tabella V ha elementi di taglia 3 perché la più lunga codeword che inizia con 111 ha taglia 6 bit.

Utilizzando questa struttura che contiene in totale 20 entry *Inflate* effettuerà, oltre ad un accesso alla tabella T, al più un secondo accesso nella tabella U o nella tabella V (esse sono infatti allo stesso livello). Il numero medio di accessi per simbolo è dato da:

$$P(a) + P(b) + 2\,(1 - P(a) - P(b)) = \tfrac{1}{2} + \tfrac{1}{4} + 2\left(1 - \tfrac{1}{2} - \tfrac{1}{4}\right) = 1.25$$

perché i simboli a e b vengono decodificati mediante un singolo accesso, mentre i rimanenti (che si verificano con probabilità 0.25) richiedono 2 accessi.

Se fosse stata utilizzata una singola tabella di look-up, essa sarebbe stata composta da ben 64 entry, quindi, seppur garantendo in ogni caso un solo accesso per simbolo, il tempo richiesto per generarla sarebbe stato elevato. Costruendo invece l'albero di Huffman per l'alfabeto dato, per decodificare un simbolo sarebbe stato necessario effettuare tanti accessi

ai nodi della struttura quanti sono i bit che compongono la codeword associata al simbolo; il numero medio di accessi sarebbe dunque risultato:

$$\sum_{a \in A} P(a)|C(a)| = 2.22$$

3 Limiti dell'algoritmo LZ77

L'algoritmo LZ77 fa una assunzione implicita: i pattern ripetuti sono sufficientemente vicini, in particolare le loro occorrenze cadono all'interno del search buffer. Questa assunzione fa sì che, nel caso ci siano pattern frequenti ma a distanza maggiore della taglia del search buffer, essi vengano codificati senza sfruttare le loro occorrenze ripetute. Si pensi ad esempio ad una sequenza periodica di periodicità più lunga della finestra: essa ha sicuramente bassa entropia, ma l'algoritmo non riesce a sfruttarne la periodicità e produce una codifica con rate piuttosto scarso. L'algoritmo LZ78 si differenzia proprio per il fatto che questa assunzione non compare.

L'idea alla base dell'algoritmo di Ziv e Lempel del 1978 è proprio quella di individuare la presenza di prefissi anche a distanza maggiore della massima consentita dal search buffer: per far questo viene costruito dinamicamente un dizionario contenente le sottosequenze incontrate più frequentemente.

Bibliografia

[KS06] K. Sayood. *Introduction to Data Compression* 3^{rd} *Ed.*, pagine 1-54 e 117-140 Morgan Kaufmann, 2006.

[GZIP] Home Page del software di compressione GZIP.
Sito web `http://www.gzip.org/algorithm.txt`.

[R1951] Request For Comments 1951
Sito web `http://tools.ietf.org/html/1951`.