A-DECODE(n, s_n, f)
1 $s \leftarrow s_n$
2 **for** $i \leftarrow 1$ **to** n
3 **do** Trova x tale che $s \in [s_x, d_x)$
4 Scrivi x sul file f di output
5 $s \leftarrow (s - s_x)/(d_x - s_x)$
6 Chiudi f

7.3 Codifiche Lempel-Ziv

Esistono due famiglie di compressori note con le sigle LZx (dove L e Z sono le iniziali dei due studiosi A. Lempel e J. Ziv e x identifica la versione e/o l'implementazione), una basata sull'utilizzo di un dizionario e l'altra basata sulla nozione di *sliding window*, cioè *finestra scorrevole*. La versione che analizzeremo in questi appunti è nota come algoritmo di compressione *Lempel-Ziv-Welch* (LZW); si tratta essenzialmente dell'implementazione data da T. Welch dell'algoritmo *LZ78*, pubblicato da Lempel e Ziv in un articolo scientifico del 1978. LZW è implementato nel comando *compress* di Unix/Linux.

LZW è basato sull'uso di una struttura dati *dizionario* che associa codici numerici a segmenti di testo che sono già stati precedentemente osservati nell'input. L'algoritmo opera in modo tale che l'insieme dei segmenti memorizzati nel dizionario, e ai quali è dunque associato un codice, soddisfa una proprietà di *chiusura* rispetto ai prefissi: se α è nel dizionario, allora ogni prefisso di α è nel dizionario. Proprio in virtù di questa proprietà, almeno da un punto di vista logico il dizionario può essere memorizzato usando un trie in cui ogni nodo (e non solo le foglie) contenga informazione. Da un punto di vista implementativo, come vedremo, si ricorre tuttavia a strutture dati più efficienti.

Anche l'algoritmo di Huffman utilizza un trie, ma presenta significative differenze con LZW. Innanzitutto Huffman utilizza codeword a

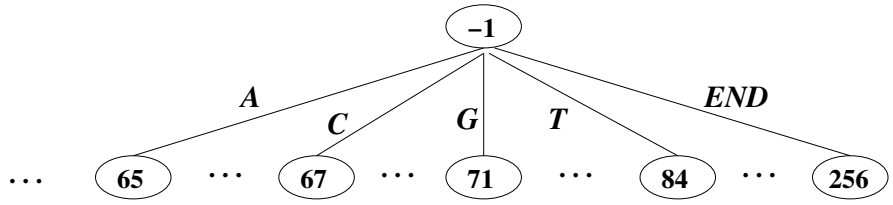

Figura 7.7: Trie iniziale per un alfabeto di 256 caratteri più il "carattere" *END*. La figura illustra solo alcuni simboli, che compariranno negli esempi di questo paragrafo.

lunghezza variabile (prefix-free) mentre LZW usa codeword di lunghezza fissa per rappresentare sequenze di testo di lunghezza variabile[3]. In secondo luogo, in Huffman il trie deve essere incluso nel file compresso (almeno nella versione statica che abbiamo studiato) mentre in LZW esso viene ricostruito dall'algoritmo di decompressione.

Consideriamo come esempio guida il caso di un testo sull'alfabeto ASCII esteso (8 bit). In queste circostanze il trie viene inizializzato con i codici dei 256 caratteri, più il codice di un carattere speciale che servirà per segnalare la fine della stringa di bit che rappresenterà il testo compresso. Si veda la figura 7.7.

Oltre al dizionario/trie, l'algoritmo usa due variabili fondamentali:

1. un contatore c per il numero di segmenti di caratteri distinti ai quali l'algoritmo assegna un codice (compresi quelli formati da un solo carattere, con i quali viene inizializzato il trie); il codice assegnato ad un segmento è proprio il valore che il contatore c ha nel momento di assegnazione;

2. una stringa s che rappresenta il segmento attuale[4].

[3]Più precisamente, come vedremo, LZW usa sequenze di lunghezza che è fissata per un gruppo di sequenze ma progressivamente crescente da gruppo a gruppo.

[4]Con abuso di notazione, ove non possano sorgere ambiguità, nella discussione che segue utilizzeremo il simbolo s, eventualmente con un indice, per denotare tanto un segmento di testo quanto il codice numerico ad esso associato.

Il ciclo fondamentale dell'algoritmo mantiene l'invariante che s è un segmento al quale l'algoritmo ha assegnato un codice. La generica, i-esima iterazione consiste dapprima in un'analisi della stringa sx_i, ottenuta concatenando ad s l'i-esimo carattere nel file di input. Le operazioni successive dipendono dal fatto che esista o meno nel trie un nodo p con path label[5] sx_i.

- Se esiste un tale nodo p evidentemente un segmento sx_i è già stato incontrato nell'input e ad esso è stato associato un codice. In questo caso l'algoritmo semplicemente pone $s \leftarrow sx_i$ perché deve verificare (leggendo il prossimo simbolo x_{i+1}) se sx_i non sia parte di un segmento già codificato ancora più lungo. Si noti che l'invariante qui è mantenuto per l'ipotesi fatta sul segmento sx_i.

- Altrimenti sx_i non ha ancora un codice mentre, in base all'invariante, il segmento s ce l'ha. In questo caso, l'algoritmo scrive s (inteso come codice) sul file di output, attribuisce un codice al nuovo segmento sx_i inserendolo nel dizionario/trie (ed aggiornando la variabile contatore c) ed infine re-inizializza la variabile s al valore x_i. Si noti che l'invariante qui è mantenuto perché tutti i singoli caratteri hanno inizialmente un codice.

Affinché l'invariante valga all'inizio della prima iterazione è sufficiente immaginare che l'informazione memorizzata nella radice del trie sia il codice associato alla stringa vuota, che in effetti è proprio la path label della radice. Tale codice non verrà comunque mai utilizzato.

L'algoritmo completo, in pseudo-codice, è fornito di seguito. Esso utilizza una funzione SEARCH(α, T) che restituisce il nodo del trie T la cui path label è la stringa α, se un tale nodo esiste, e NIL altrimenti, e una funzione CHAR(c) che restituisce il carattere con codice c. La funzione `lg` è definita nel modo seguente:

[5]Ricordiamo che la path label di un nodo v in un trie T è semplicemente la stringa ottenuta concatenando i caratteri che etichettano gli archi dell'unico cammino che unisce la radice di T a v.

$$lg(x) = \begin{cases} 1 & \text{se } x = 0 \\ \lfloor \log x \rfloor + 1 & \text{altrimenti} \end{cases}$$

LZW-ENCODE(f, g)

```
 1  T ← new node()
 2  c ← 0
 3  while c < |Σ|
 4        do p ← new node(c)
 5           Crea l'arco (T, p) con etichetta CHAR(c)
 6           c ← c + 1
    ▷ Assegna al carattere speciale END il codice c = |Σ|
 7  p ← new node(c)
 8  Crea l'arco (T, p) con etichetta END
 9  c ← c + 1
10  s ← ϵ
11  while not end of file f
12        do leggi carattere x da f
13           if SEARCH(sx, T) ≠ NIL
14              then s ← sx
15              else p ← SEARCH(s, T)
16                   Scrivi su g il codice memorizzato
                        nel nodo p usando lg(c) bit
17                   c ← c + 1
18                   q ← new node(c)
19                   Crea l'arco (p, q) con etichetta x
20                   s ← nodo associato a SEARCH(x, T)
21  Scrivi il codice in SEARCH(s, T) su g usando lg(c) bit
22  Scrivi il codice |Σ| su g usando lg(c) bit
23  Chiudi f
24  Chiudi g
```

Si noti che i codici vengono prodotti in output (righe 16, 21 e 22 dell'algoritmo LZW-ENCODE) usando $\lg(c)$ bit; questo perché c indica anche il numero di segmenti codificati "nel momento in cui viene effettuata la scrittura". $\lg(c)$ è dunque il numero di bit necessario e sufficiente per poterli rappresentare tutti. Come vedremo, in fase di decodifica la variabile c segue un'identica evoluzione e dunque consente di leggere sempre il numero corretto di bit dal file compresso.

Esempio 25 La tabella 7.1 e le figure 7.8, 7.9 e 7.10 illustrano il comportamento dell'algoritmo su input `ACAGACGATACA`. ◇

Iterazione	Input	s	c	Output	Nuovo segmento	Codice
1	A	A	257			
2	C	C	258	65	AC	257
3	A	A	259	67	CA	258
4	G	G	260	65	AG	259
5	A	A	261	71	GA	260
6	C	AC	261			
7	G	G	262	257	ACG	261
8	A	GA	262			
9	T	T	263	260	GAT	262
10	A	A	264	84	TA	263
11	C	AC	264			
12	A	A	265	257	ACA	264
13	*eof*	A	265	65,256	AA	265
	eof	A	265	65,256		

Tabella 7.1: Variabili e azioni rilevanti compiute nel ciclo fondamentale (righe 11-20) dell'algoritmo LZW-ENCODE su input il file ASCII `ACAGACGATACA`. Si noti che, all'inizio del ciclo vale $s = \epsilon$ e $c = 257$. Si noti infine che i due ultimi codici vengono prodotti in output al di fuori del ciclo principale (ultima riga della tabella).

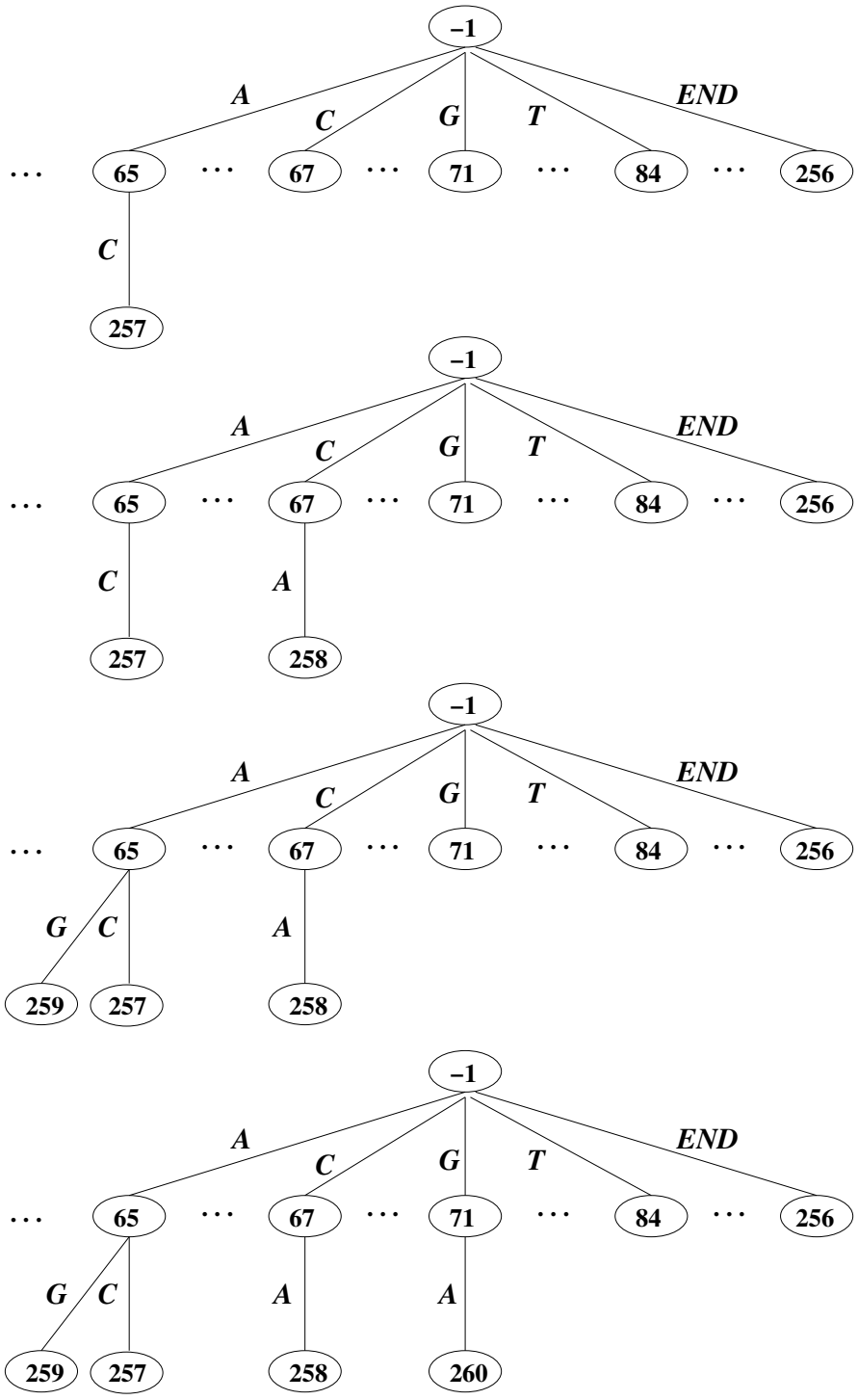

Figura 7.8: Evoluzione del dizionario/trie dell'esempio 25 (parte I).

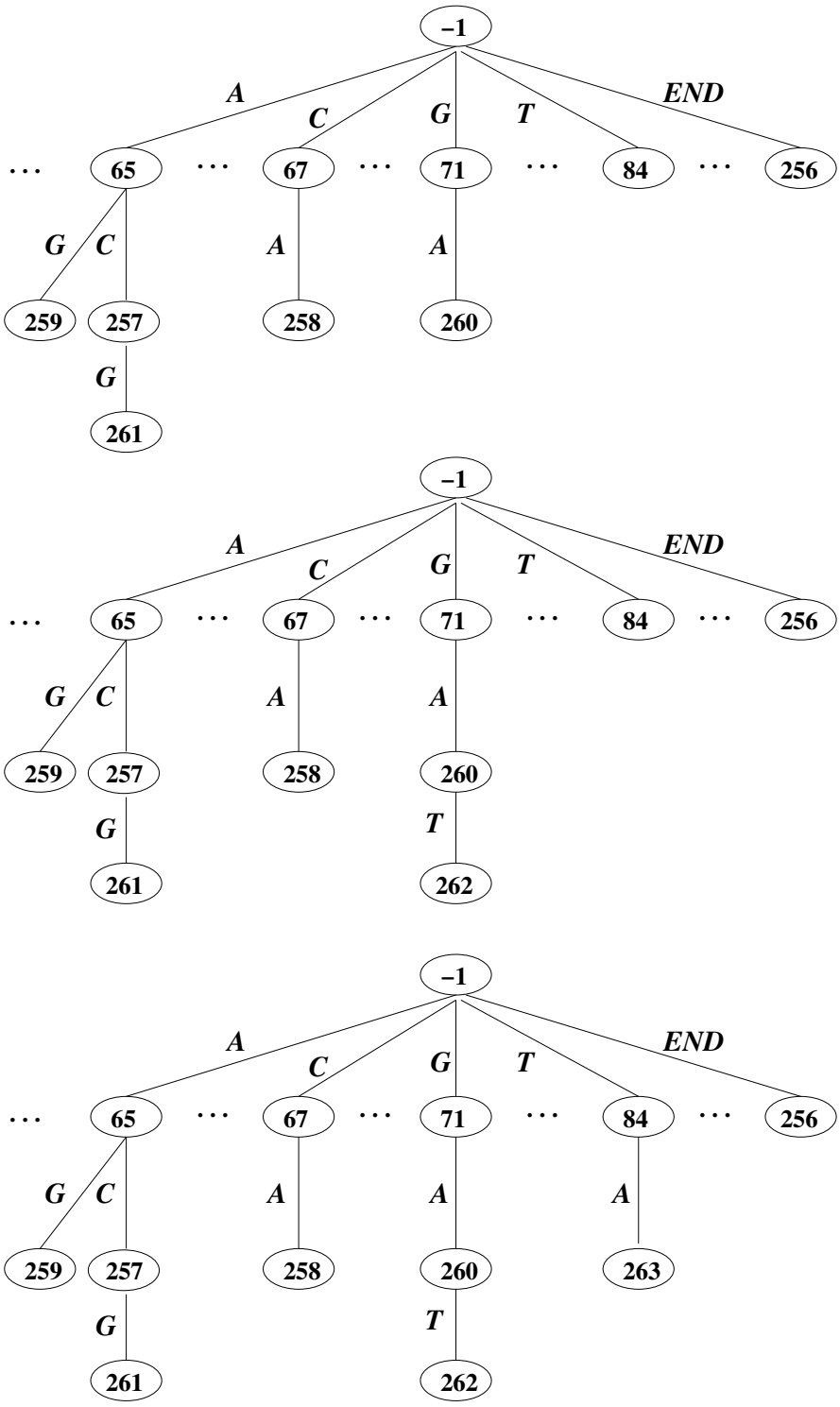

Figura 7.9: Evoluzione del dizionario/trie dell'esempio 25 (parte II).

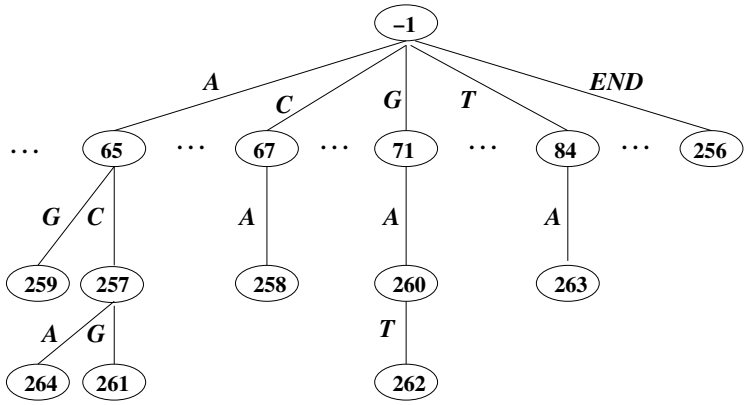

Figura 7.10: Evoluzione del dizionario/trie dell'esempio 25 (ultima inserzione).

L'algoritmo di decompressione utilizza i codici ricevuti, oltre che per produrre in output il file di testo orginale, anche per procedere ad una "ricostruzione" incrementale del trie così come lo ha generato il codificatore. Anzi, è proprio questa ricostruzione che consente la decodifica dei codici che non si riferiscano a semplici caratteri. Vediamo ora di spiegare la logica seguita dall'algoritmo di decompressione.

Siano $s_1, s_2, \ldots, s_m$ i codici prodotti in output dall'algoritmo LZW-ENCODE, per un qualche m che in generale sarà sensibilmente minore del numero n di caratteri nel file di input. Riconsideriamo con attenzione le azioni svolte dall'algoritmo LZW-ENCODE dopo la scrittura del generico codice s_k (righe da 17 a 20). Dopo aver incrementato il valore del contatore c, l'algoritmo crea un nuovo nodo nel trie associato al segmento $s'_k = s_k x$ (cioè c diventa il codice del segmento $s_k x$) e inizializza la variabile s al valore x, dove x è il carattere di input letto alla riga 12. In questo modo x diviene il primo carattere del segmento codificato dal prossimo codice prodotto in output, e cioè s_{k+1}. Si noti che il codice corrispondente al segmento s'_k potrà eventualmente comparire più avanti nel file di output. In particolare, può succedere

che $s'_k = s_{k+1}$, cioè che s'_k coincide con la prossima codeword emessa in output dal codificatore. Questo accade se la sequenza codificata da s_k si ripete immediatamete nel file di input seguita subito dopo dal carattere x. In particolare, per i nostri scopi è importante osservare come, in questo caso, x sia anche il primo carattere codificato da s_k.

Esempio 26 Si consideri la stringa $\alpha =$ `ATATATA`... sull'alfabeto ASCII esteso. La sequenza di codici prodotti in output dall'algoritmo LZW-ENCODE è 65 84 257 259 ... Come si può notare, vengono prodotti in sequenza i due codici 257 e 259 che rappresentano le sequenze `AT` e `ATA`. In questo caso $s_3 =$ `AT` e $s_4 = s'_3 = s_3$`A` = `ATA`. ◇

La condizione $s'_k = s_{k+1}$, esemplificata nell'esempio 26, è di potenziale difficoltà per il decodificatore. Ad essa ci riferiremo usando la locuzione *condizione di prolungamento* in quanto, con riferimento al trie, il codice per s_{k+1} si trova seguendo un cammino che "prolunga" quello sul quale si trova il codice per s_k.

Consideriamo ora in maniera dettagliata le operazioni che deve svolgere l'algoritmo di decompressione dopo la lettura del generico codice s_k. Supponiamo che la ricostruzione del trie già effettuata consenta all'algoritmo di decodificare s_k. Per poter eseguire sul trie le stesse operazioni svolte dal codificatore alle righe 17-20 e, in particolare, per definire il codice corrispondente al segmento s'_k, il decodificatore deve conoscere il carattere x. Questo tuttavia è il primo carattere del segmento codificato da s_{k+1} e dunque l'aggiornamento del trie potrà avvenire solo dopo aver decodificato anche s_{k+1}. Questo suggerisce come le computazioni del decodificatore debbano procedere in modo "sfalsato" rispetto a quelle del codificatore e precisamente:

....
decodifica s_k e aggiorna il trie con il codice per la sequenza s'_{k-1};
decodifica s_{k+1} e aggiorna il trie con il codice per la sequenza s'_k;
decodifica s_{k+2} e aggiorna il trie con il codice per la sequenza s'_{k+1};

...

Consideriamo ora le "condizioni iniziali", che sono necessarie affinché questo processo possa correttamente partire. Quando $k = 1$ il codice è già presente nel dizionario per il semplice fatto che, fissato l'alfabeto Σ, le inizializzazioni del trie possono procedere allo stesso modo nel codificatore e nel decodificatore. Inoltre, nessun aggiornamento del dizionario deve essere effettuato perché il codice s'_0 non è definito. Per $k = 2$ il valore da decodificare è ancora relativo ad un segmento di un solo carattere x (la verifica è immediata) e dunque anch'esso si trova già nel dizionario. Inoltre, l'algoritmo è in grado di definire il codice s'_1 che sarà relativo al segmento[6] s_1 concatenato con il carattere x appena decodificato.

Il processo di decodifica parte quindi correttamente e, in generale, ogni nuovo codice s'_k che viene definito potrà eventualmente servire in seguito per decodificare un codice s_{k+j} in input. Ora, se $j > 1$ non c'è nessun problema. Tuttavia abbiamo già visto (esempio 26) come possa occorrere anche la condizione di prolungamento, cioè la situazione $s'_k = s_{k+1}$. In tal caso, un passo della computazione avrebbe il seguente aspetto "circolare"

...

decodifica s_{k+1} e aggiorna il trie con il codice per $s'_k = s_{k+1}$;

...

Tuttavia, abbiamo già chiarito come, in questo caso, s_{k+1} abbia la seguente forma: $s_{k+1} = s_k x$, dove x coincide proprio con il primo carattere di s_k. Ne consegue, quindi, che la decodifica di s_{k+1} è possibile usando esclusivamente il segmento s_k.

Siamo a questo punto in grado di fornire lo pseudocodice dell'algoritmo di decodifica. L'algoritmo utilizza una funzione FIND(s, T) (speculare alla funzione SEARCH dell'algoritmo di codifica) che trova il nodo del trie T nel quale è memorizzato il codice s.

[6] Si ricordi l'abuso di notazione in base al quale s_i indica sia un segmento sia il suo codice.

```
LZW-DECODE(f, g)
    ▷ Inizializzazione come in LZW-ENCODE
 1  T ← new node()
 2  c ← 0
 3  while c < |Σ|
 4        do p ← new node(c)
 5           Crea l'arco (T, p) con etichetta CHAR(c)
 6           c ← c + 1
    ▷ Assegna al carattere speciale END il codice c = |Σ|
 7  p ← new node(c)
 8  Crea l'arco (T, p) con etichetta END
 9  c ← c + 1
10  Leggi un codice s (su lg(c) bit) da f
11  p ← FIND(s, T)
12  Scrivi la path label ℓ_p di p su g
13  Leggi un codice t (su lg(c) bit) da f
14  while t ≠ |Σ|
15        do if ⟨ Condizione di prolungamento ⟩
16              then Scrivi la path label ℓ_p di p su g
17                   x ← ℓ_p[0]
18                   Scrivi il carattere x su g
19              else q ← FIND(t, T)
20                   Scrivi la path label ℓ_q di q su g
21                   x ← ℓ_q[0]
22           c ← c + 1
23           r ← new node(c)
24           Crea l'arco (p, r) con etichetta x
25           s ← t
26           if ⟨ Condizione di prolungamento ⟩
27              then p ← r
28              else p ← q
29           Leggi un codice t (su lg(c) bit) da f
30  Chiudi f e g
```

Esempio 27 Consideriamo dapprima il comportamento dell'algoritmo su un esempio nel quale non si verifica mai la condizione di prolungamento. Un tale esempio ci è fornito dal file dell'esempio 25. Il file compresso (con LZW-ENCODE) contiene la sequenza di codici numerici

65 67 65 71 257 260 84 257 65 256

rappresentati su 9 bit. Analogamente a quanto visto per la compressione, la tabella 7.2 (con le figure 7.11-7.14) illustra l'evoluzione delle variabili s e t nonché l'output prodotto dalle diverse iterazioni dell'algoritmo. Ogni riga della tabella 7.2 fotografa la situazione dopo che è stata eseguita l'istruzione di riga 19. La riga relativa all'iterazione 0 fotografa dunque la situazione al primo ingresso nel ciclo. ◇

Iterazione	c	s	t	p, q	Output già prodotto
0	256	65	67	figura 7.11	`A`
1	257	67	65	figura 7.12a	`AC`
2	258	65	71	figura 7.12b	`ACA`
3	259	71	257	figura 7.12c	`ACAG`
4	260	257	260	figura 7.12d	`ACAGAC`
5	261	260	84	figura 7.13a	`ACAGACGA`
6	262	84	257	figura 7.13b	`ACAGACGAT`
7	263	257	65	figura 7.13c	`ACAGACGATAC`
8	264	65	256	figura 7.14	`ACAGACGATACA`

Tabella 7.2: Evoluzione delle variabili e azioni rilevanti compiute nel ciclo fondamentale dell'algoritmo LZW-DECODE su input costituito dal file compresso 65 67 65 71 257 260 84 257 65 256.

Quel che rimane da stabilire è come sia possibile rilevare l'occorrenza della condizione di prolungamento e illustrare il relativo comportamento dell'algoritmo LZW-DECODE con un esempio.

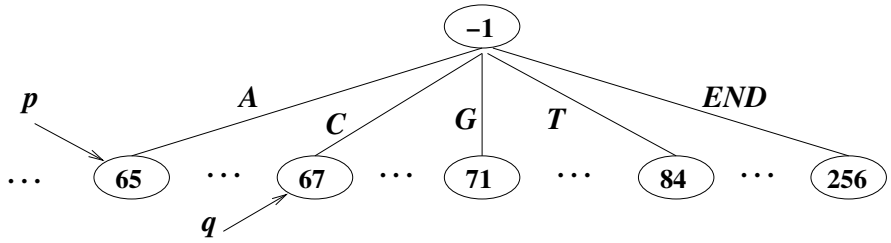

Figura 7.11: Trie iniziale costruito dall'algoritmo di decompressione.

Naturalmente, per quanto riguarda il primo punto la soluzione più semplice consiste nel verificare l'assenza nel dizionario del codice s_k appena letto. Possiamo cioè implementare la funzione FIND(s, T) in modo tale che restituisca il valore NIL nel caso in cui s non sia presente. Vogliamo tuttavia studiare più attentamente il funzionamento del decodificatore; in questo modo troveremo una semplice condizione di verifica che non richiede l'accesso al dizionario. Al riguardo, osserviamo che, a partire dal momento in cui è stato letto il secondo codice nel file compresso (cioè s_2), l'algoritmo LZW-DECODE genera, ad ogni passo, esattamente un nuovo codice (cioè un codice diverso da quelli relativi ai segmenti formati da un singolo carattere). Poiché il primo nuovo codice ha valore numerico $|\Sigma| + 1$, possiamo affermare che, al momento della lettura di s_k, l'algoritmo LZW-DECODE dispone di tutti i codici con valore numerico non superiore a $|\Sigma| + k - 2$, e soltanto quelli. Ad esempio, nel caso di alfabeto ASCII, quando legge s_3 l'algoritmo ha a disposizione i codici fino a 257=256+1; quando legge s_4 ha a disposizione i codici fino a 258=256+2 e così via. Ora, se si osserva l'esempio 26, si vede proprio che $s_4 = 259 = 256 + 3$, un codice che a quel punto non è disponibile. Per rilevare la condizione di prolungamento è dunque sufficiente verificare se $s_k > c - 1$, in quanto, all'inizio dell'iterazione, vale $c = |\Sigma| - 1$.

Consideriamo da ultimo il funzionamento di LZW-DECODE sull'input dell'esempio 26 che, come sappiamo, viene codificato come sequen-

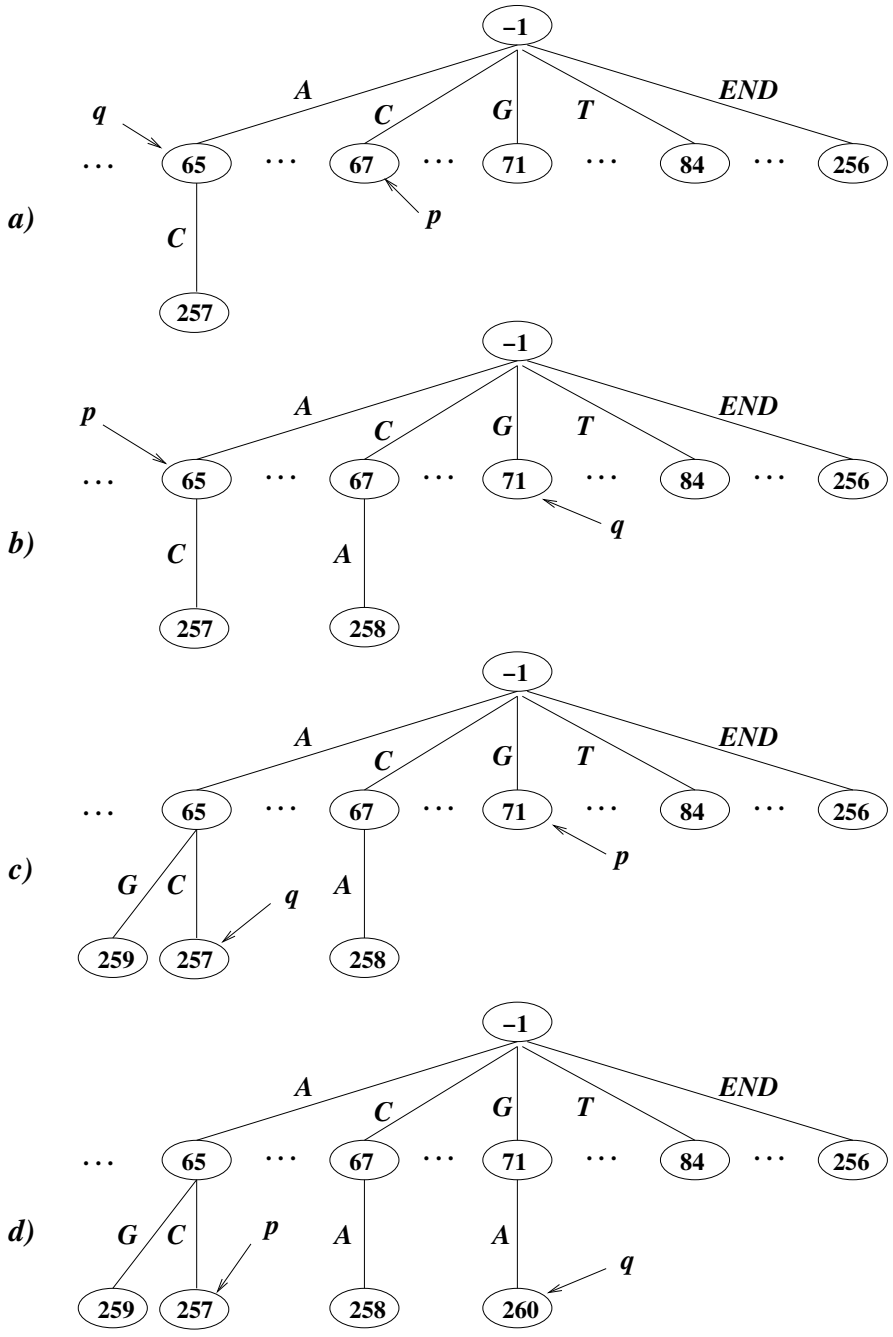

Figura 7.12: Evoluzione del trie prodotta dall'algoritmo di decompressione (iterazioni 1-5).

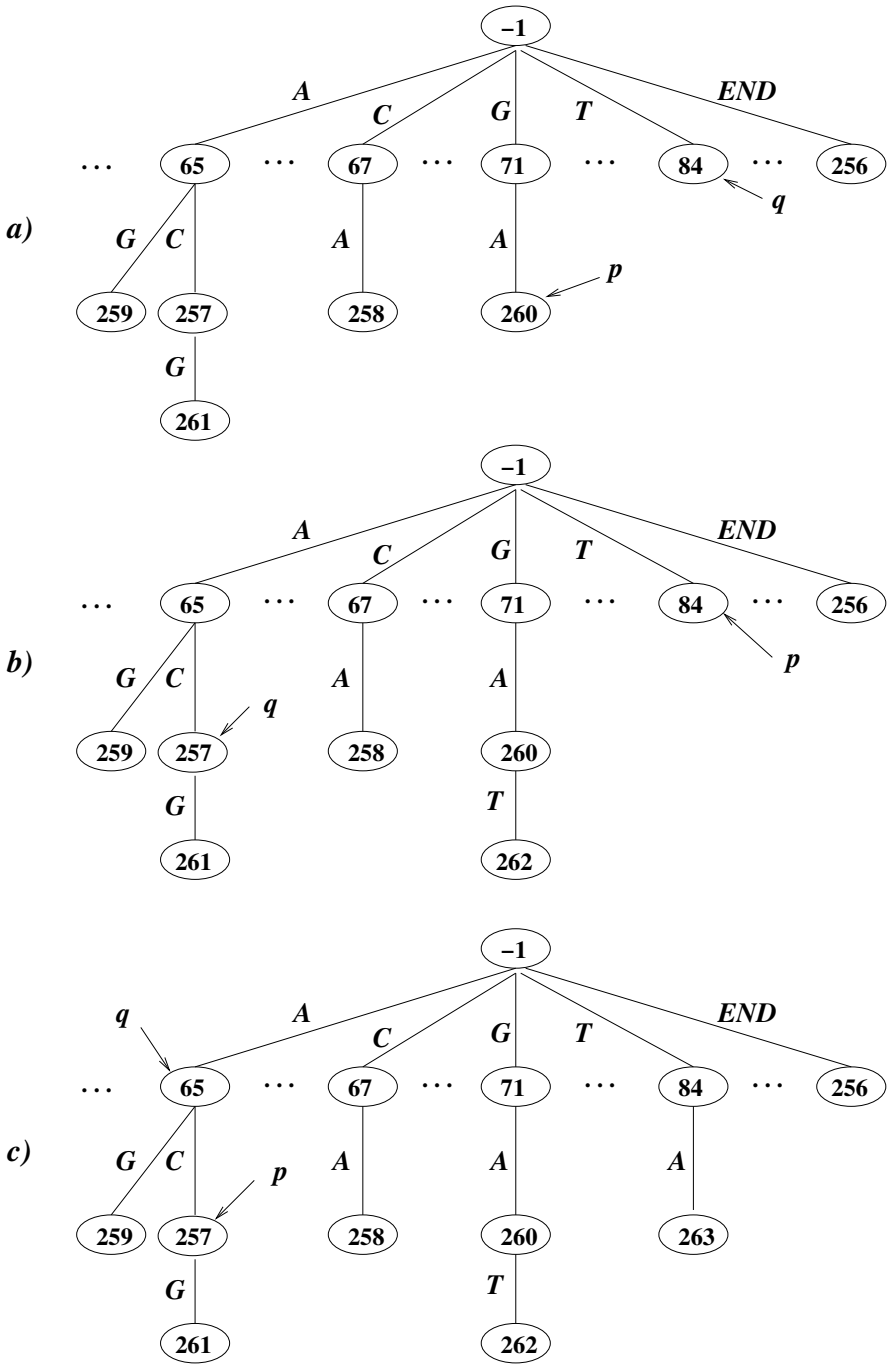

Figura 7.13: Evoluzione del trie prodotta dall'algoritmo di decompressione (iterazioni 6-8).

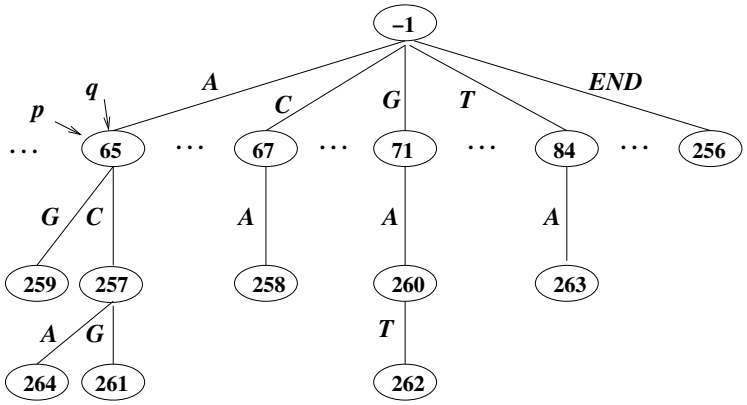

Figura 7.14: Fotografia finale del trie prodotto dall'algoritmo di decompressione.

za 65 84 257 259 .... Dopo la lettura del codice 84, l'algoritmo produce il codice 257 per il segmento `AT`; questo "lavoro" fornisce subito i suoi frutti, in quanto permette di decodificare il successivo valore numerico nel file, che è proprio 257. Usando il primo carattere del segmento `AT` appena decodificato, l'algoritmo può inoltre definire 258 come codice per il segmento `TA`. Il successivo codice nel file è ora il valore "incriminato" 259. Poiché in questo caso sappiamo che il segmento codificato coincide (a meno dell'ultimo carattere) con quello decodificato al passo precedente, l'algoritmo stampa (riga 16) la path label del nodo p relativo proprio all'ultimo valore decodificato. Sappiamo inoltre che l'ultimo carattere del segmento dovrà essere uguale al primo (e dunque uguale al primo del segmento precedente), e questo spiega le istruzioni delle righe 17 e 18. Inoltre, se si verifica la condizione di prolungamento, il puntatore p (da utilizzare nell'iterazione successiva) viene semplicemente spostato sul nodo figlio r, altrimenti (riga 28) a p viene assegnato il valore q che "che punta" ad un nodo su altro ramo del trie.

Aspetti implementativi

La più importante decisione implementativa consiste nella scelta della struttura dati che dovrà rappresentare il dizionario. Osserviamo tuttavia preliminarmente come l'implementazione non debba di necessità essere la stessa nelle due fasi di compressione e decompressione. Cominciamo dunque con l'analizzare le richieste che il codice di compressione pone alla struttura dati. È immediato rendersi conto del fatto che l'algoritmo LZW-ENCODE accede ai vari nodi del trie sempre seguendo un cammino dalla radice verso le foglie. Ciò di cui abbiamo bisogno è dunque la disponibilità di due funzioni che "simulino" la discesa nel trie e la creazione di nuovi nodi (quando la discesa si interrompe). Più precisamente, serve:

(i) un'operazione di ricerca che, dato un nodo p e un carattere x, restituisce, se esiste, il nodo q figlio di p tale che l'arco (p, q) ha etichetta x, o il valore *NIL* altrimenti;

(ii) un'operazione di inserimento che, dato un nodo p, un carattere x e un codice c, inserisce un nuovo nodo q nel trie come figlio di p, associa a q il codice c e pone x come etichetta sull'arco (p, q).

Nel caso della compressione, l'unica informazione esplicita che serve per implementare le precedenti operazioni è, per ogni dato nodo, la codeword ad esso corrispondente. Le informazioni che "simulano" la struttura ad albero sono infatti garantite dall'implementazione delle funzioni stesse. Per l'algoritmo di compressione possiamo dunque sostituire al trie un dizionario generico che memorizza oggetti; ad ogni oggetto p è associata una proprietà, code(p), il cui valore è la path label del nodo del trie rappresentato da p. Come vedremo in seguito, per la decompressione dovremo considerare altre proprietà.

Un'implementazione efficiente del dizionario e delle funzioni di accesso e inserimento utilizza una tabella hash. Poiché entrambe le ope-

razioni hanno come parametri un nodo del trie[7] e un carattere, anche le chiavi d'accesso alla tabella saranno composte da coppie $\langle p, x\rangle$, dove p è un oggetto e x un carattere. La più semplice rappresentazione dei nodi è probabilmente quella che utilizza numeri interi; più precisamente, se un nodo p è memorizzato in una certa posione $I(p)$ della tabella, allora l'indice $I(p)$ è il numero intero che rappresenta p nelle operazioni di accesso e inserimento. Per semplicità notazionale, tuttavia, useremo p per denotare tanto il nodo quanto l'indice concreto che lo rappresenta. Si noti che, secondo quanto anticipato, l'unica informazione contenuta in una posizione p della tabella è la codeword associata al nodo p.

Sia dunque H la tabella hash. Indicheremo con $H.\textsc{search}(\langle p, x\rangle)$ la funzione di ricerca che restituisce (un intero che rappresenta) il nodo figlio di p raggiungibile dall'arco etichettato con x, se questo esiste, o altrimenti il valore speciale *NIL*. Indicheremo invece con $H.\textsc{insert}(\langle p, x\rangle, c)$ la funzione di inserimento di un nuovo nodo come figlio di p. Come convenzione, possiamo infine rappresentare la radice del trie mediante l'intero -1.

Con le posizioni fatte nel precedente paragrafo, potremmo ad esempio accedere al nodo associato alla stringa `AGA` mediante la seguente sequenza di applicazioni della funzione $H.\textsc{search}$:

$$\begin{aligned} p &\leftarrow H.\textsc{search}(\langle -1, \texttt{A}\rangle) \\ q &\leftarrow H.\textsc{search}(\langle p, \texttt{G}\rangle) \\ r &\leftarrow H.\textsc{search}(\langle q, \texttt{A}\rangle) \end{aligned}$$

Ad un'analisi superficiale, la precedente sequenza non comporta grandi benefici rispetto all'uso di un trie: per accedere ad un nodo a profondità tre si eseguono tre chiamate della funzione hash. Generalizzando, il costo di accesso risulta proporzionale alla profondità del nodo (in quanto la valutazione della funzione hash richiede tem-

[7] Più precisamente, *un oggetto che rappresenta un nodo del trie*, anche se non staremo continuamente a precisarlo.

po $O(1)$), proprio come nel trie. Tuttavia, quest'ultimo risultato vale solo usando array ad accesso diretto per memorizzare, in ogni nodo, i puntatori ai nodi figli. E ciò comporta, in generale, un enorme spreco di spazio, spotrattutto per alfabeti di elevata cardinalità. In alternativa, usando soluzioni mediante memoria dinamica, il costo di accesso aumenta di un fattore che può esser $O(\log|\Sigma|)$ o addirittura $O(|\Sigma|)$.

Il codice seguente illustra la funzione di codifica modificata in modo da prevedere l'uso di una tabella hash per l'implementazione del dizionario. L'alfabeto di riferimento utilizzato è generico.

```
LZW-H-ENCODE(f, g)
 1  ⟨Definizione della tabell hash H⟩
 2  c ← 0
 3  while c < |Σ|
 4        do H.INSERT(⟨−1, CHAR(c)⟩, c)
 5           c ← c + 1
 6  H.INSERT(⟨−1, END⟩, c)        ▷ Il codice assegnato a END e' |Σ|
 7  p ← −1
 8  while not end of file f
 9        do leggi un carattere x da f
10           q ← H.SEARCH(⟨p, x⟩)
11           if q ≠ NIL
12              then p ← q
13              else Scrivi code(p) su g usando lg(c) bit
14                   c ← c + 1
15                   H.INSERT(⟨p, x⟩, c)
16                   p ← H.SEARCH(⟨−1, x⟩)
17  Scrivi code(p) su g usando lg(c) bit
18  Scrivi |Σ| su g usando lg(c) bit
19  Chiudi f
20  Chiudi g
```

Passiamo ora ad analizzare gli aspetti legati all'implementazione dell'algoritmo di decompressione. In termini di operazioni primitive sul trie, ciò che serve è:

(i) una funzione per l'inserimento di un nodo r nel dizionario, di modo che questo sia raggiungibile da un nodo p già presente nel trie mediante un arco etichettato x;

(ii) una funzione che, dato un nodo q del trie, restituisce la path label di q (e dunque il segmento di testo il cui codice è associato al nodo q).

La seconda funzione serve per la decodifica del codice associato a q e per recuperare l'etichetta x sul primo arco del cammino dalla radice a q stesso. Il carattere x, come sappiamo[8], serve infatti per etichettare il nuovo arco che viene inserito nel trie (insieme ovviamente ad un nuovo nodo) subito dopo la decodifica di un codice (riga 24 di LZW-DECODE).

A differenza di quanto accade nell'algoritmo di compressione, il trie viene dunque attraversato dalle foglie in direzione della radice. Questa circostanza ha una fondamentale implicazione positiva sulla struttura dati concreta che utilizzeremo per l'implementazione del dizionario. Non serve, infatti, una tabella hash, bensì una semplice tabella T in cui memorizzare le informazioni che consentono, appunto, la risalita dell'albero. Più precisamente, ogni nodo dell'albero verrà rappresentato da un indice in T (e dunque da un numero intero, come già nel caso della compressione); inoltre, la generica posizione di indice p conterrà le seguenti informazioni:

1. un riferimento parent(p) all'intero che rappresenta il nodo genitore del nodo rappresentato da p.

[8]Si riveda eventualmente l'algoritmo LZW-DECODE.

2. un'etichetta label(p) per l'arco ($\texttt{parent}(p), p$).

Nel seguito, per semplicità, useremo lo stesso simbolo per identificare un nodo e il numero intero (indice) che lo rappresenta. È facile rendersi conto che non serve anche una proprietà code(p), per la semplice ragione che l'algoritmo può "sistemare" ogni nuovo nodo p proprio nella posizione della tabella di indice uguale al codice che p intende rappresentare. Ad esempio, se l'algoritmo deve allocare il nodo che rappresenta il codice (poniamo) 351, userà la cella 351 della tabella. Va da sè poi che, nel caso dell'alfabeto ASCII, ogni simbolo x verrà sistemato nella cella di indice ASCII(x). Si noti che, data la contiguità (oltre che l'unicità) dei codici usati, questa scelta non comporta alcuno spreco di spazio.

Con le scelte implementative adottate per il dizionario, le due operazioni sopra definite risultano quindi di immediata realizzazione. La funzione di inserimento, che indicheremo con, $T.\textsc{insert}(p, x)$, che inserisce un nuovo nodo "agganciandolo" come figlio del nodo p, è definita dai due assegnamenti seguenti, che fissano le proprietà del nuovo nodo inserito:

$$\begin{aligned} \text{parent}(c) &\leftarrow p \\ \text{label}(c) &\leftarrow x \end{aligned}$$

dove c è il solito contatore utilizzato per assegnari i codici. Naturalmente, l'implementazione vera e propria dipenderà da come verrà realizzata fisicamente la tabella. Ad esempio, nel caso di array T di strutture l'assegnamento parent(c) $\leftarrow p$ potrà avere la forma seguente: $T[c].parent \leftarrow p$. Viceversa, nel caso di array paralleli i due assegnamenti precedenti andranno riscritti tipicamente sostituendo le parentesi quadre a quelle tonde.

La funzione che restituisce la path label di un dato nodo p, e che indicheremo con $T.text(p)$, ha un'altrettanto semplice implementazio-

ne, che consiste nel risalire l'albero fino alla radice, concatenando (in ordine rovesciato) le etichette incontrate.

```
T.TEXT(p)
    ▷ Lo stack serve per rovesciare l'ordine delle etichette
1   S ← new stack (char)
2   while p ≠ −1                    ▷ Al solito, -1 rappresenta la radice
3         do S.push(label(p))
4            p ← parent(p)
5   α ← ϵ
6   while not S.empty()
7         do x ← S.pop()
8            α ← αx
9   return α
```

Siamo ora in grado di presentare il codice completo dell'algoritmo di decodifica.

LZW-H-DECODE(f, g)

```
 1  Definisci il dizionario T (ad esempio, come array paralleli)
 2  c ← 0
 3  while c < |Σ|
 4        do T.INSERT(−1, CHAR(c))
 5           c ← c + 1
 6  T.INSERT(−1, END)            ▷ Il codice assegnato a END e' |Σ|
 7  p ← primi lg(c) bit su f (interpretati come numero intero)
 8  α ← T.TEXT(p)
 9  Scrivi la stringa α su g
10  x ← α[0]                          ▷ α[0] e' il primo carattere di α
11  while true
12        do q ← successivi lg(c) bit su f
13           c ← c + 1
14           if q ≥ c                        ▷ Condizione di prolungamento
15              then T.INSERT(p, x)
16                   Scrivi αx su g
17              else α ← REVERSE(q)
18                   x ← α[0]
19                   if x ≠ END
20                      then T.INSERT(p, x)
21                           Scrivi la stringa α su g
22                      else break
23           p ← q
24  Chiudi f
25  Chiudi g
```