UNIVERSITÀ DI PISA

Metodi di compressione dati lossless

Esperienze di Programmazione

Francesco D'Izzia

Indice

1 Introduzione

Nell'utilizzo ormai quotidiano abbiamo a che fare, direttamente o indirettamente, con il tema della compressione di dati, senza il quale molti servizi o realtà odierne non sarebbero possibili o utilizzabili allo stesso modo, specialmente quando parliamo in termini di efficienza e velocità, due standard di cui non possiamo fare a meno.

Esistono svariati metodi e algoritmi dedicati all'argomento, è bene però dividere subito il tutto in due categorie specifiche: la *lossy compression* e la *lossless compression*. Come suggeriscono i rispettivi nomi, la prima tipologia si basa sul concetto di comprimere dei dati a discapito di qualche approssimazione o perdita di informazioni: un prezzo non banale da pagare, ma che li rende molto efficaci in ambito pratico, basti pensare al successo del formati JPEG e MPEG, che nonostante riducano di parecchio la mole di dati riescono comunque a garantire una certa fedeltà e riconoscibilità dei file compressi quando paragonati agli originali.

I secondi (lossless) invece vanno per una strada più "pura": la compressione avviene senza perdita di dati o informazioni, ed è su questa categoria che vogliamo focalizzare maggiormente l'attenzione. Prima, per quanto riguarda l'altra tipologia, i lossy, abbiamo accennato al fatto che il fulcro della compressione sia nell'azione di sostituire blocchi di dati con approssimazioni, più leggere, degli stessi.

Viene quindi spontaneo chiedersi quale sia invece il principio capace di supportare la categoria dei lossless: com'è possibile ridurre la quantità di dati evitando di perdere...dati per l'appunto? Come i vari algoritmi facciano nello specifico verrà approfondito più avanti, per ora basti pensare grossomodo all'idea che vi sta dietro, ovvero all'eliminazione o riduzione delle *ridondanze*. È molto più comune di quello che si pensi trovare più e più informazioni duplicate all'interno di un file, il compito di un algoritmo lossless è quello di codificare dati in un formato tale da ridurre o rimuovere completamente (in casi ottimali) le ridondanze associate a questi dati.

Al tempo stesso l'operazione di codifica/compressione non deve essere one-way, perché ovviamente bisogna che sia possibile ritornare al file originale mediante una relativa procedura di decodifica/decompressione dei dati.

Gli algoritmi presi in esame sono tre: il Runtime Length Encoding (RLE), la codifica di Huffman e l'algoritmo di Lempel-Ziv (LZ77). La scelta è ricaduta su questi in quanto funzionano in modo abbastanza diverso l'uno dall'altro ed è un buon modo per approcciare l'argomento in senso più generale: verrà analizzato il funzionamento dei singoli algoritmi, condurremo dei test in funzione di questi tre per valutare pregi e difetti di ciascun algoritmo e alla fine trarremo le giuste conclusioni. È bene sottolineare sin da subito che nel contesto che andremo ad affrontare non esiste un algoritmo "migliore" di un altro in senso assoluto: vedremo infatti che ognuno ha delle casistiche in cui può performare particolarmente bene o male rispetto ad altri. È dunque bene adoperarli nell'atto pratico scegliendo cosa usare in base al tipo di ridondanze davanti alle quale ci troviamo. È interessante anche evidenziare, pur non essendo argomento di questo articolo, come a volte possa essere particolarmente sensato combinare tra di loro questi algoritmi a più livelli, basti pensare per esempio al metodo **DEFLATE**, in cui vengono applicati sia LZ77 che la codifica di Huffman.

2 Il Run-Length Encoding (RLE)

Il primo metodo che vediamo è il Run-Length Encoding, il più semplice come funzionamento e implementazione.

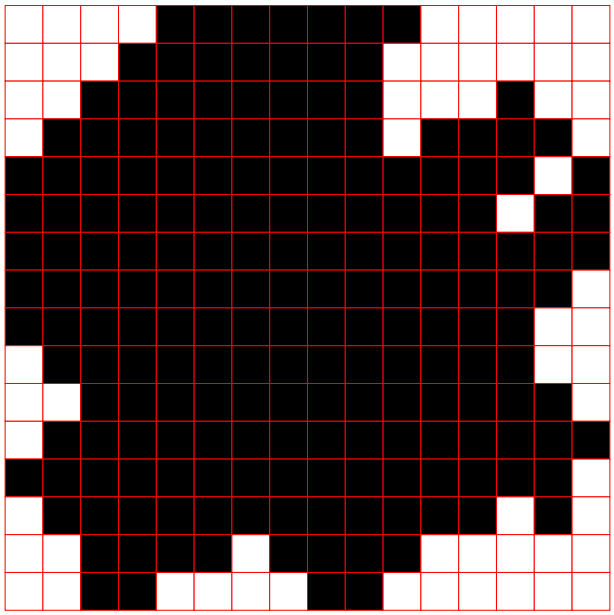
2.1 Funzionamento dell'algoritmo

RLE sfrutta le ridondanze di sequenze con simboli consecutivi (run) e nell'operazione di compressione sostituisce tali sequenze con delle coppie $\langle \#o, S \rangle$, dove $\#o$ rappresenta il numero di occorrenze consecutive del simbolo S.
Di seguito un piccolo esempio a titolo illustrativo.

Viene fornita in input la seguente stringa: AAAAABBBBBCCCDDDDEEFFFFGGGHHHH
L'output dell'algoritmo sarà dunque: $\langle 5, A \rangle\ \langle 5, B \rangle\ \langle 3, C \rangle\ \langle 4, D \rangle\ \langle 2, E \rangle\ \langle 4, F \rangle\ \langle 3, G \rangle\ \langle 4, H \rangle$

Il funzionamento di RLE risulta molto semplice e intuitivo: preso un insieme di dati in ingresso (in questo caso una stringa, quindi un insieme di caratteri) è sufficiente eseguire una scansione lineare, contando per ogni simbolo il numero di occorrenze in cui quest'ultimo viene riscontrato consecutivamente, generando le coppie viste in precedenza.

Un esempio più interessante è l'applicazione dell'algoritmo su immagini, nello specifico consideriamo il caso di bitmap a 1 bit (ovvero con i soli colori bianco e nero rappresentabili), dove il valore *0* identifica il colore bianco, mentre *1* il nero. Consideriamo un'immagine 16x16 pixel di questo tipo (figura in basso a sinistra) e associamo il corrispondente bit ad ogni pixel, ottenendo una matrice come quella nella figura accanto.



0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1	1	0	0	0	1	0	0
0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0
0	0	1	1	1	1	0	1	1	1	1	0	0	0	0	0
0	0	1	1	0	0	0	0	1	1	0	0	0	0	0	0

Figura 1: Bitmap "binarie" con relativo bit di codifica del colore

Una possibilità di memorizzazione dell'immagine potrebbe consistere nell'associare una matrice di bit 1:1, ma sarebbe poco furba e carente in quanto a ottimizzazione.
Nel nostro esempio sono presenti diverse sequenze (*run*) di pixel consecutivi con lo stesso colore: è in questi casi che RLE può tornarci molto utile! Infatti, leggendo la matrice riga per riga e da sinistra verso destra, possiamo trattarla come un array di bit, il che ci rende immediata e semplice l'applicazione dell'algoritmo. Partendo dalle prime quattro righe abbiamo:
000011111110000000011111110000000011111111000100011111111011110 ...
che in output diventerà
$\langle 4,0\rangle\ \langle 7,1\rangle\ \langle 8,0\rangle\ \langle 7,1\rangle\ \langle 8,0\rangle\ \langle 8,1\rangle\ \langle 3,0\rangle\ \langle 1,1\rangle\ \langle 3,0\rangle\ \langle 9,1\rangle\ \langle 1,0\rangle\ \langle 4,1\rangle\ \langle 1,0\rangle$ ...

2.2 I limiti di RLE

Abbiamo intuito che RLE si presta bene a collezioni di dati che tendono a ripetersi consecutivamente, ma come si comporta quando queste ripetizioni sono minime o addirittura inesistenti? La risposta è...male, molto male.
Prendiamo un caso limite: una bitmap 8x8 con una palette di 6 colori diversi, che mapperemo per comodità nel nostro esempio con lettere che vanno dalla A alla F (vedere le figure poco più avanti). L'output dell'algoritmo sarà dunque così composto: $\langle 1,A\rangle\ \langle 1,B\rangle\ \langle 1,A\rangle\ \langle 1,C\rangle\ \langle 1,D\rangle\ \langle 1,E\rangle\ \langle 1,B\rangle\ \langle 1,C\rangle\ \langle 1,C\rangle\ \langle 1,D\rangle\ \langle 1,C\rangle\ \langle 1,B\rangle\ \langle 1,F\rangle\ \langle 1,D\rangle\ \langle 1,C\rangle\ \langle 1,E\rangle$ ...
In questo caso particolare, volutamente esagerato ma non per questo completamente fuori dalla realtà, quello che l'algoritmo ci fornisce non è una versione ridotta del dato in ingresso, ma una versione di dimensione addirittura superiore all'originale!
Bisogna quindi tenere a mente questa possibilità e valutare se RLE sia effettivamente la scelta più adatta dinnanzi a certe tipologie di dati.

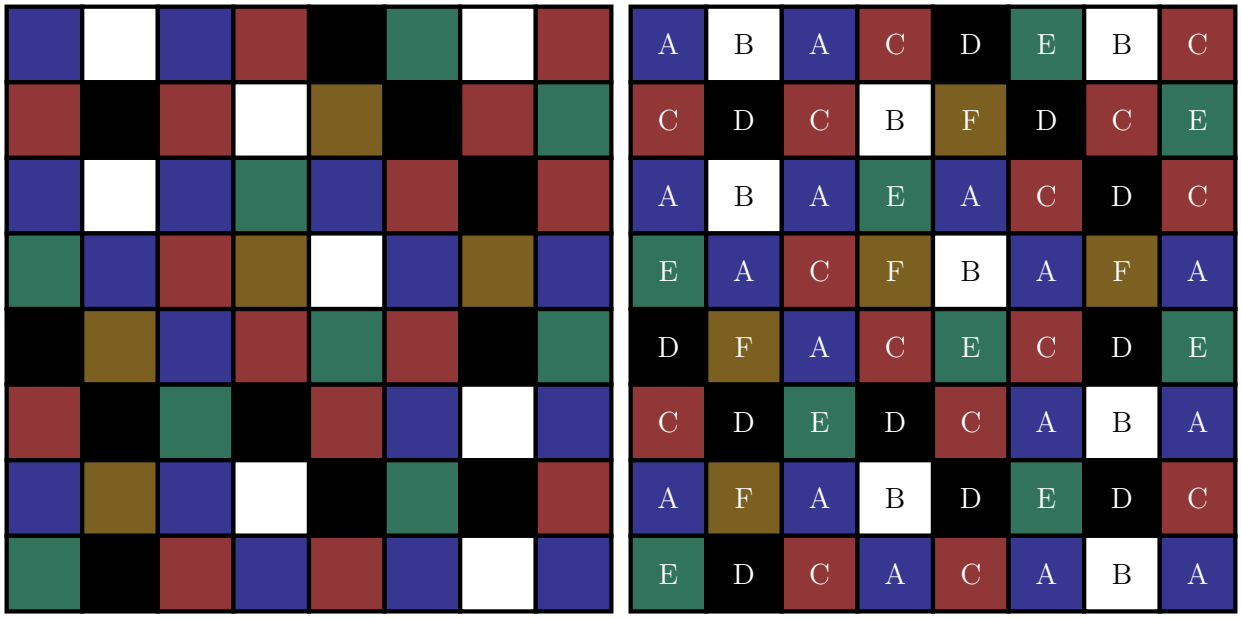
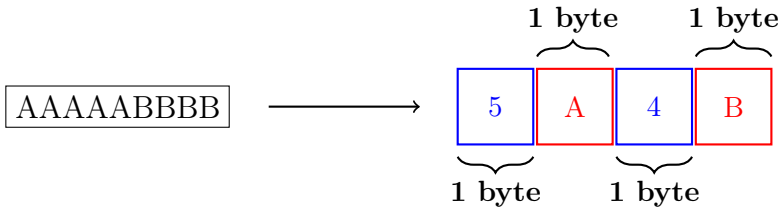


Figura 2: Bitmap a 6 colori, con relativa codifica

2.3 Formato di codifica



La codifica delle sequenze è stata organizzata in modo da associare per ogni coppia $\langle \#o, S \rangle$ due byte: il primo byte descriverà il numero di occorrenze consecutive mentre il secondo byte sarà il simbolo da ripetere quel determinato numero di volte.
È importante evidenziare che rappresentando dei numeri in 1 byte ci è concesso rappresentare esattamente 2^8 numeri (da 0 a 255), quindi nel caso in cui ci fosse un simbolo ripetuto un numero di volte superiore a 255, questo verrà gestito in più blocchi.

Supponiamo che un certo carattere C venga ripetuto 280 volte consecutivamente: verrà creato un primo blocco composto da 2 byte: nel primo byte, non potendo mettere un valore superiore a 255, ci fermeremo al valore massimo, mentre nel secondo byte metteremo ovviamente il carattere C.
Per quanto riguarda il secondo blocco metteremo nel primo byte la differenza tra il valore totale di occorrenze (280) e quello parziale consumato fino a questo momento (255), metteremo nel primo byte del secondo blocco il valore 25 e nel secondo byte di nuovo il valore C.
Se il valore della differenza fosse ancora superiore al nostro valore massimo rappresentabile con 8 bit, sarebbe necessario ripetere il processo più volte e andare avanti con più blocchi.

3 La codifica di Huffman

Analizziamo adesso la codifica di Huffman: si tratta di un processo che si occupa di trovare una codifica binaria basandosi sulla frequenza di ogni simbolo. La codifica generata è a prefissi di lunghezza variabile: l'algoritmo non codifica ogni simbolo con lo stesso numero di bit, ma esegue prima un conteggio delle occorrenze di ciascun simbolo, stila una lista ordinata in base al grado di frequenza di ciascuno e associa a quelli più frequenti una codifica con meno bit possibili. Essendo di dimensioni variabili, è assolutamente necessario che la codifica sia assente da ogni tipo di ambiguità e sia univoca per ogni simbolo.

Non può presentarsi una situazione in cui ad esempio ho il carattere a codificato come *10* e x codificato come *1010*, in quanto durante il processo di decodifica sarebbe impossibile risalire al messaggio originale data la situazione di ambiguità (il messaggio andrebbe decodificato come x o come aa?). L'algoritmo non presenta questo rischio in quanto per costruzione ci fornirà una rappresentazione univoca e priva di ambiguità attraverso la cosiddetta *prefix rule*, che asserisce che nessuna codifica deve essere un prefisso di un'altra possibile codifica.

3.1 Funzionamento della codifica

Il funzionamento dell'algoritmo può essere riassunto nei seguenti step:

- Si prende nota dei singoli simboli e delle loro rispettive occorrenze (o frequenze).
- Si crea un nodo foglia per ognuno dei simboli e si aggiungono ad una coda di priorità.
- Vengono estratti dalla coda i due nodi con le frequenze più basse (se più simboli hanno lo stesso numero di occorrenze, allora la selezione di questi può avvenire in maniera arbitraria).
- Viene creato un nuovo nodo a partire dai due simboli precedenti la cui frequenza è pari alla somma delle frequenze dei due simboli.
- Il nuovo nodo viene inserito nella coda di priorità.
- Si ritorna al terzo punto e si ripetono le operazioni successive fino a quando non rimane un solo elemento nella coda.
- Viene estratto l'ultimo nodo, che sarà la radice dell'albero, e si procede con l'etichettare gli archi dell'albero per poi creare la tabella che associa i codici con il simbolo rappresentato dal nodo.

Di seguito viene riportato un esempio di codifica di Huffman della parola ABRACADABRA.

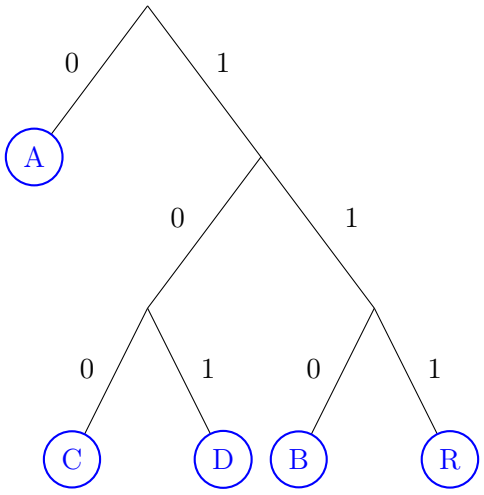
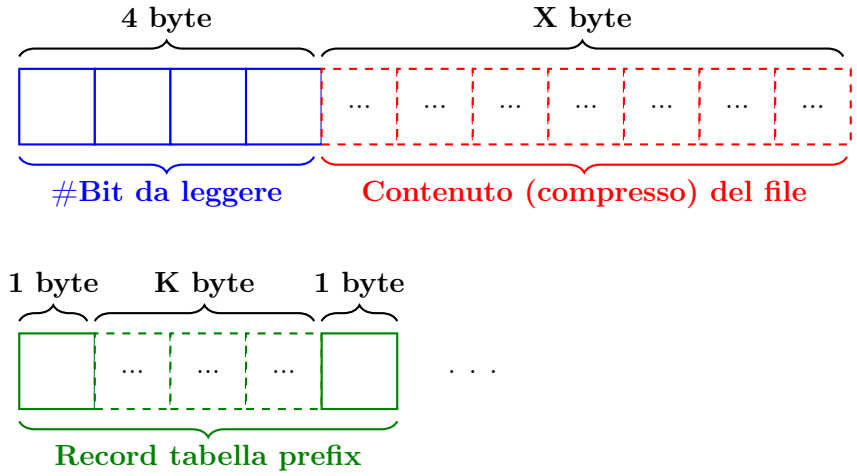


Tabella prefix	
Prefix code	Simbolo associato
0	'A'
100	'C'
101	'D'
110	'B'
111	'R'

$\longrightarrow$ 01101110100010101101110

3.2 Formato di codifica



Il file compresso sarà composto da 4 byte nel quale è descritto il numero di bit da leggere per decodificare il messaggio posto subito dopo. È stata fatta questa scelta perché se il numero di bit risultanti dalla compressione non fosse un multiplo di 8 allora sarebbe necessario capire quanti bit leggere, in quanto la lettura e scrittura dei file viene effettuata in byte (8 bit alla volta) e dunque eventuali bit rimanenti in modulo 8 andrebbero aggiunti per formare un byte. Essendo 4 byte, quindi 32 bit, il numero di bit leggibili massimo è di 2^{32}. Per i nostri scopi andrà più che bene, qualora però si volesse incrementare questa quantità è sufficiente modificare questa parte dell'intestazione, ad esempio si potrebbero sostituire con un blocco che indica quanti byte leggere (piuttosto che i bit) e un eventuale offset per dire quanti bit leggere dopo la lettura di N byte.

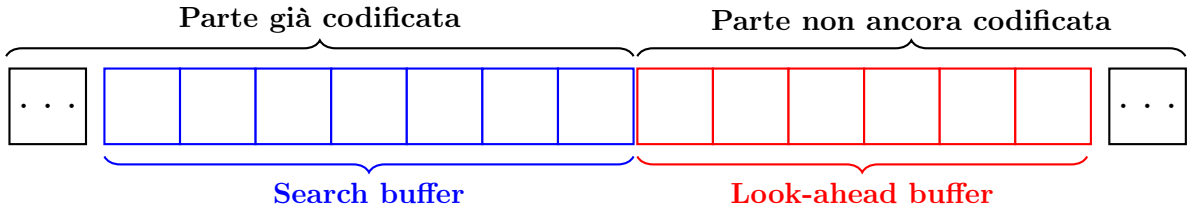
Durante la fase di codifica viene creata una "tabella prefix" che associa ad ogni simbolo la sua relativa codifica: tale informazione è fondamentale nel processo di decodifica, per questo durante la codifica questa tabella viene memorizzata ed esportata, nel nostro caso, in un file separato chiamato *table.huff* che verrà utilizzato al momento della decodifica. Un singolo record della tabella (figura in verde) viene memorizzato con il seguente formato: il primo byte contiene la lunghezza K del prefix code, i successivi K bit saranno il prefix code mentre il byte finale rappresenta il simbolo associato. Questa struttura va ripetuta per ogni riga della tabella.

4 Il metodo Lempel-Ziv (LZ77)

LZ77, creato da Lempev e Ziv nel 1977, è uno fra gli algoritmi di compressione più popolari ed è alla base di DEFLATE, quest'ultimo utilizzato in ZIP, gzip, zlib, PNG e molti altri.

4.1 Funzionamento dell'algoritmo

LZ77 funziona attraverso la cosiddetta "sliding window", una finestra che, come suggerito dal nome, continua a scorrere per tutto lo stream di dati. La finestra viene divisa per convenzione in due parti: il **search buffer**, che rappresenta la parte già esplorata e codificata dello stream e il **look-ahead buffer**, che contiene la parte successiva della sequenza che va ancora processata.

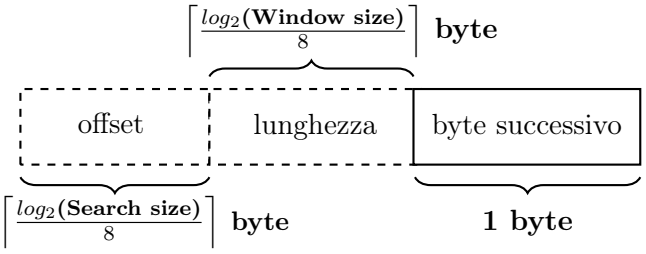


Viene mantenuto, durante l'operazione di codifica, un puntatore man mano che si scorre: questa finestra opera cercando il longest match tra il search buffer e la parte iniziale del look-ahead buffer. Quando viene trovata una corrispondenza viene generata una tripla $\langle o,\ l,\ c \rangle$, dove o rappresenta l'offset, ovvero la distanza tra la posizione attuale del look-ahead buffer e la corrispondenza trovata, l rappresenta la lunghezza della corrispondenza e c è il carattere/simbolo successivo nel look-ahead buffer. Nel caso in cui non venisse trovata una corrispondenza, la tripla generata vale semplicemente $\langle 0,\ 0,\ c \rangle$. L'output dell'intera fase di compressione sarà quindi dato dall'insieme di queste triple.

Si consideri, per esempio, nuovamente la parola ABRACADABRA con search window e look-ahead window pari rispettivamente a 7 e 4.
L'output generato da LZ77 sarà dato dal seguente insieme di triple:

- $\langle 0,\ 0,\ \text{'A'} \rangle$
- $\langle 0,\ 0,\ \text{'B'} \rangle$
- $\langle 0,\ 0,\ \text{'R'} \rangle$
- $\langle 3,\ 1,\ \text{'C'} \rangle$
- $\langle 2,\ 1,\ \text{'D'} \rangle$
- $\langle 7,\ 3,\ \text{'A'} \rangle$

4.2 Formato di codifica



Seguendo quanto detto in precedenza, è stato scelto di modellare il formato di codifica come un insieme di triple $\langle$*offset, lunghezza, byte successivo*$\rangle$.
È stato scelto di allocare $\left\lceil \frac{log_2(\text{Search buffer size})}{8} \right\rceil$ byte per l'offset in quanto quelli sono i byte necessari a descrivere il massimo valore per quel campo, non potendo l'offset superare i limiti dettati dalla grandezza della search window. Discorso simile per il campo lunghezza, dove però viene utilizzata la window size al posto della search size: questo perché esistono dei casi particolari in LZ77 dove è possibile che la lunghezza del longest match vada oltre la search size e si estenda anche "invadendo" il look-ahead buffer, di conseguenza è necessario prevedere una grandezza tale da poter supportare questa casistica. Infine l'ultimo campo (simbolo successivo) sarà ovviamente composto da un solo byte.

È conseguenza diretta di questa implementazione notare che aumentando di un certo quantitativo la search/window size vengano allocati sempre più byte, diventa quindi interessante trovare un buon compromesso tra quelli che sono i vantaggi di una window size relativamente grande e gli eventuali overhead introdotti dalla grandezza di quest'ultima.

5 Test e conclusioni

5.1 Esecuzione dei test

Finita la spiegazione dei tre metodi di compressione e delle relative scelte implementative, è tempo di dedicarsi ad una serie di test atti a valutare e confrontare le performance, eventuali problemi di utilizzo nell'atto pratico o scelta di parametri nel caso della window size per LZ77 (indicata tra parentesi nei vari test). In questi test la search buffer size e la look-ahead buffer size prendono valori pari a metà della window size.

Chess board

In questo primo test analizziamo una bitmap di 256x256 pixel, avente una color-depth da 4 bit. È composta da quadrati di 32x32 pixel che si alternano tra il nero e il bianco, formando difatti una scacchiera.

Dimensione del file originale: 32,1 KB (32.886 byte)

board.bmp	Dimensione compressione	Tempo compressione	Tempo decompressione	Rapporto di compressione	Spazio risparmiato
RLE	4,10 KB	0,005 secondi	0,001 secondi	7,83	87,23 %
Huffman	6,06 KB	0,006 secondi	0,05 secondi	5,30	81,12 %
LZ77 (256)	1023 byte	0,021 secondi	0,010 secondi	32,15	96,89 %
LZ77 (4096)	5,18 KB	0,925 secondi	0,027 secondi	6,20	83,87 %
LZ77 (32768)	5,13 KB	3,494 secondi	0,011 secondi	6,26	84,02 %

Lorem ipsum

Si tratta di un file testuale (.txt) generato attraverso `https://it.lipsum.com/` a cui sono stati successivamente rimossi i line/paragraph breaks.

Dimensione del file originale: 97,8 KB (100.168 byte)

lorem_ipsum.txt	Dimensione compressione	Tempo compressione	Tempo decompressione	Rapporto di compressione	Spazio risparmiato
RLE	192 KB	0,039 secondi	0,048 secondi	0,51	-96,66 %
Huffman	52,14 KB	0,022 secondi	1,727 secondi	1,88	46,69 %
LZ77 (256)	103 KB	1,835 secondi	0,056 secondi	0,94	-6,13 %
LZ77 (4096)	80,1 KB	14,158 secondi	0,041 secondi	1,22	18,11 %
LZ77 (65536)	54,3 KB	2,08 minuti	0,038 secondi	1,80	44,42 %

Words

Di nuovo alle prese con un file .txt, questa volta composto da varie parole in italiano.

Dimensione del file originale: 607 KB (622.345 byte)

words.txt	Dimensione compressione	Tempo compressione	Tempo decompressione	Rapporto di compressione	Spazio risparmiato
RLE	1,13 MB	0,24 secondi	0,272 secondi	0,52	-92,12 %
Huffman	314 KB	0,121 secondi	6,28 minuti	1,93	48,30 %
LZ77 (256)	243 KB	4,417 secondi	0,236 secondi	2,49	59,86 %
LZ77 (4096)	369 KB	1,18 minuti	0,249 secondi	1,65	39,27 %
LZ77 (65536)	351 KB	16,63 minuti	0,356 secondi	1,73	42,14 %

Covid spreadsheet

Un foglio elettronico (.csv) contenente le informazioni aggiornate dalla Protezione Civile sull'andamento nazionale giornaliero della pandemia COVID-19. (`https://github.com/pcm-dpc/COVID-19/blob/master/dati-andamento-nazionale/dpc-covid19-ita-andamento-nazionale.csv`)

Dimensione del file originale: 22,3 KB (22.872 byte)

covid.csv	Dimensione compressione	Tempo compressione	Tempo decompressione	Rapporto di compressione	Spazio risparmiato
RLE	40,9 KB	0,089 secondi	0,01 secondi	0,55	-83,17 %
Huffman	10,93 KB	0,007 secondi	0,081 secondi	2,04	51,06 %
LZ77 (256)	18,2 KB	0,316 secondi	0,011 secondi	1,23	18,51 %
LZ77 (4096)	22,5 KB	3,474 secondi	0,012 secondi	0,99	-0,78 %
LZ77 (16384)	20,3 KB	11,166 secondi	0,01 secondi	1,10	9,10 %
LZ77 (32768)	19,7 KB	15,948 secondi	0,01 secondi	1,13	11,42 %

Linux source code

File testuale contenente il codice (.c) del modulo *sysctl* implementato nel kernel Linux. (`https://github.com/torvalds/linux/blob/master/kernel/sysctl.c`)

Dimensione del file originale: 80,2 KB (82.127 byte)

sysctl.c	Dimensione compressione	Tempo compressione	Tempo decompressione	Rapporto di compressione	Spazio risparmiato
RLE	151 KB	0,032 secondi	0,038 secondi	0,53	-88,81 %
Huffman	53,6 KB	0,021 secondi	1,666 secondi	1,50	33,14 %
LZ77 (256)	76,7 KB	1,05 secondi	0,04 secondi	1,05	4,31 %
LZ77 (4096)	43,2 KB	6,980 secondi	0,031 secondi	1,85	46,03 %
LZ77 (65536)	30,1 KB	56,58 secondi	0,03 secondi	2,66	62,39 %

Firewatch

Bitmap di 2460x1514 pixel con color-depth di 8 bit tratta dal videogioco "Firewatch".

Dimensione del file originale: 3,55 MB (3.724.560 byte)

firewatch.bmp	Dimensione compressione	Tempo compressione	Tempo decompressione	Rapporto di compressione	Spazio risparmiato
RLE	77,6 KB	0,564 secondi	0,035 secondi	46,85	97,87 %
Huffman	818,82 KB	0,608 secondi	n.d.[1]	4,44	77,49 %
LZ77 (256)	141 KB	3,344 secondi	1,175 secondi	25,78	96,12 %
LZ77 (4096)	101 KB	19,144 secondi	1,35 secondi	35,75	97,20 %
LZ77 (32768)	68,4 KB	1,78 minuti	1,292 secondi	53,15	98,12 %
LZ77 (65536)	64,1 KB	3,29 minuti	1,257 secondi	56,69	98,24 %
LZ77 (131072)	85,1 KB	5,91 minuti	1,20 secondi	42,73	97,66 %

Space

Bitmap di 1920x1080 pixel con color depth di 24 bit.
È il mio attuale sfondo del desktop e vista la presenza di una tinta unica per buona parte dell'immagine sarebbe stato un peccato non includerlo nella fase di testing!

Dimensione del file originale: 5,93 MB (6.220.856 byte)

space.bmp	Dimensione compressione	Tempo compressione	Tempo decompressione	Rapporto di compressione	Spazio risparmiato
RLE	11,8 MB	2,795 secondi	3,154 secondi	0,50	-99,97 %
Huffman	1,67 MB	1,107 secondi	n.d.[1]	3,54	71,75 %
LZ77 (256)	463 KB	9,579 secondi	2,246 secondi	13,10	92,37 %
LZ77 (4096)	502 KB	1,47 minuti	2,23 secondi	12,08	91,72 %
LZ77 (32768)	311 KB	7,55 minuti	2,778 secondi	19,52	94,88 %
LZ77 (65536)	282 KB	13,39 minuti	2,143 secondi	21,48	95,34 %
LZ77 (131072)	367 KB	25,16 minuti	2,32 secondi	16,51	93,94 %

I valori presenti nelle ultime due colonne, il rapporto di compressione (compression ratio) e lo spazio risparmiato (space savings), sono così definiti:

$$\textbf{Rapporto di compressione} = \frac{\textbf{Dimensione file non compresso}}{\textbf{Dimensione file compresso}}$$

$$\textbf{Spazio risparmiato} = 1 - \frac{\textbf{Dimensione file compresso}}{\textbf{Dimensione file non compresso}}$$

Lo spazio risparmiato, per comodità, è stato indicato in percentuale (il valore di base è stato moltiplicato per 100).

[1] Non disponibile a causa delle grosse limitazioni della decodifica di Huffman, che scala molto male al crescere della dimensione dei dati, diventando difatti impraticabile considerando l'ordine di grandezza di vari MB.

5.2 Considerazioni finali

I test effettuati hanno confermato diversi aspetti prevedibili e al contempo hanno messo in luce dei problemi di attuabilità che non sempre erano così scontati.

Partendo da *RLE*, che era il più semplice da prevedere, è stato confermato come con file di uso quotidiano sia difficile ricadere nelle sue condizioni molto specifiche: l'algoritmo non è stato capace di primeggiare nemmeno una volta, il più delle volte ha introdotto il suo overhead da 1 byte per ogni simbolo, raddoppiando la dimensione originale del file. Si è comportato abbastanza bene però con le prime due bitmap in cui venivano ripetuti dei pixel consecutivamente (come facilmente intuibile), mentre nell'ultima bitmap (space) ha fallito nonostante la ridondanza concettualmente simile, questo perché l'ultima immagine ha una color-depth di 24 bit e di conseguenza gli elementi che vengono ripetuti consecutivamente sono a blocchi di 3 byte, e non di 1 byte come nei due casi precedenti.

Huffman si è dimostrato davvero efficace a gestire file di testo o comunque basati su caratteri, tuttavia gli ultimi due test hanno mostrato un suo grosso problema: è davvero molto pesante e lento a decodificare file di grosse dimensioni, tanto da diventare inutilizzabile con file dell'ordine di diversi MB. Nonostante l'ottima velocità di compressione è necessario trovare un'implementazione più efficace e furba per la decodifica di quella molto semplice e "naive" utilizzata qui.

LZ77 è il chiaro vincitore di questa batteria di test per quanto riguarda il rapporto di compressione e lo spazio risparmiato: si è dimostrato quasi sempre il più efficace e il più flessibile. Non è tutto oro quel che luccica però, in quanto il suo tempo di compressione si è dimostrato nettamente superiore agli altri, difatti giocando con la window size diventa molto interessante trovare il valore che massimizzi il rapporto di compressione/spazio risparmiato e al contempo minimizzi il tempo di compressione. Infine è interessante notare come non necessariamente ad un incremento della finestra corrisponda una dimensione di compressione minore o uguale a quelle precedenti, oltre certe soglie infatti, vista l'implementazione di LZ77, è naturale che venga introdotto un overhead dato da un maggiore numero di byte che devono rappresentare offset o lunghezze del match potenzialmente più grandi che non sarebbero rappresentabili con meno byte.

Riferimenti bibliografici

[1] Dhanesh Budhrani. *How data compression works: exploring LZ77.*
Towards Data Science, 12 settembre 2019.
`https://towardsdatascience.com/how-data-compression-works-exploring-lz77-3a2c2e06c097`

[2] Fabrizio Lana, Daniele Masato e Luca Polin. *Tecniche di compressione con dizionario.*
Università di Padova, 6 giugno 2006.
`http://www.dei.unipd.it/~capri/LDS/MATERIALE/lez0606.pdf`

[3] *Huffman Coding Compression Algorithm.*
Techie Delight. `https://www.techiedelight.com/huffman-coding/`

[4] *Run-length encoding.*
Wikipedia. `https://en.wikipedia.org/wiki/Run-length_encoding`

[5] *Data compression ratio.*
Wikipedia. `https://en.wikipedia.org/wiki/Data_compression_ratio`

A Appendice: Codice sorgente

A.1 test.py

```
from RLE import *
from Huffman import Huffman_encode, Huffman_decode
from LZ77 import LZ77_encode, LZ77_decode
import sys, os, time

#Verifica del numero di argomenti ed eventuale messaggio di usage
if len(sys.argv) < 3:
    print("\nUsage: python test.py fileInTestFolder COMPRESSION_METHOD WINDOW_SIZE")
    print("-COMPRESSION_METHOD = RLE | HUFFMAN | LZ77\n-WINDOW_SIZE is only required for LZ77, if not specified the default size is 256\n")
    sys.exit(1)
#Parsing di WINDOW_SIZE o eventuale scelta del valore di default
elif len(sys.argv) > 3:
    WINDOW_SIZE = int(sys.argv[3])
else:
    WINDOW_SIZE = 256

#Parsing degli argomenti da terminare e inizializzazione di variabili
FILENAME = sys.argv[1]
METHOD = sys.argv[2].upper()
WITHOUT_EXTENSION = FILENAME.split('.')[0]
WINDOW_SIZE_STRING = ''
EXTRA_SIZE = 0

#Apro il file in lettura
file = open('./test/' + FILENAME, 'rb')

#Faccio partire il timer
tic = time.perf_counter()

#Verifico il metodo di compressione e lo utilizzo a mia volta
if METHOD == 'RLE':
    encoded = RLE_encode(file)
elif METHOD == 'HUFFMAN':
    encoded = Huffman_encode(file)
else:
    encoded = LZ77_encode(file, WINDOW_SIZE)

#Fermo il timer
toc = time.perf_counter()

#Calcolo il tempo trascorso
compression_time = toc-tic

```

```
#Chiudo il file
file.close()


#Se sto usando Huffman devo considerare nella misura della dimensione anche
#la dimensione della tabella, se uso LZ77 il mio file decompresso avra' un nome diverso
#che indica la WINDOW_SIZE utilizzata
if METHOD == 'HUFFMAN':
    EXTRA_SIZE = os.stat('./test/table.huff').st_size
elif METHOD == 'LZ77':
    WINDOW_SIZE_STRING = '__' + str(WINDOW_SIZE)

#Procedo col salvataggio del file
compressed_filename = './test/' + WITHOUT_EXTENSION + WINDOW_SIZE_STRING + '.' + METHOD.lower()
output_compressed = open(compressed_filename, 'wb')
output_compressed.write(encoded)
output_compressed.close()


#Conclusa la fase di compressione, procedo con l'apertura del file compresso per
#iniziare i test di decompressione
input_compressed = open(compressed_filename, 'rb')

#Faccio partire il timer
tic = time.perf_counter()

#Verifico nuovamente il metodo di compressione e scelgo il relativo algoritmo
#di decodifica
if METHOD == 'RLE':
    decoded = RLE_decode(input_compressed)
elif METHOD == 'HUFFMAN':
    decoded = Huffman_decode(input_compressed)
else:
    decoded = LZ77_decode(input_compressed, WINDOW_SIZE)

#Fermo il timer
toc = time.perf_counter()

#Calcolo il tempo trascorso
uncompression_time = toc-tic

#Apro il file in scrittura e procedo con l'esportazione
decoded_file = open('./test/uncompressed_'+ METHOD + '__' + FILENAME, 'wb')
decoded_file.write(decoded)
decoded_file.close()
```

```
89  input_compressed.close()
90
91  #Calcolo la dimensione del file, contando eventualmente anche la ↩
        dimensione extra
92  #introdotta dalla tabella di Huffman
93  compressed_size = len(encoded) + EXTRA_SIZE
94
95
96  #Stampo il risultato dei test
97  print('_' * 100)
98  print('Original/uncompressed size: ' + str(len(decoded)) + ' bytes')
99  print('Compressed size (' + METHOD + '): ' + str(compressed_size) + ' ↩
        bytes\n' )
100 print('Compression ratio: ' + '{:.2f}'.format(len(decoded)/↩
        compressed_size))
101 print('Space savings: ' +  '{:.2f}'.format((1 - (compressed_size/len(↩
        decoded)))* 100) + ' %\n')
102 print('Compression time: ' + '{:.4f}'.format(compression_time) + " ↩
        seconds")
103 print('Uncompression time: ' + '{:.4f}'.format(uncompression_time) + " ↩
        seconds")
104 print('_' * 100)
```

A.2 RLE.py

```
1  #Funzione di codifica di RLE
2  def RLE_encode(data):
3      #Inizializzo il contatore di occorrenze
4      count = 1
5
6      #Inizializzo la variabile che indica l'ultimo byte letto
7      lastByte = data.read(1)
8
9      #Inizializzo l'array di byte
10     encoded = bytearray()
11
12     #Loop in cui continuo a leggere un byte alla volta
13     while(byte := data.read(1)):
14         #Ho raggiunto la fine del file: esco dal loop
15         if byte == b'':
16             break
17
18         #Se il byte appena letto e' uguale al precedente incremento il ↩
               contatore delle occorrenze
19         if byte == lastByte:
20             count += 1
21         else:
22             #Altrimenti verifico il contatore: se e' maggiore della ↩
                   soglia massima
23             #iterativamente aggiungo la coppia <occorrenze, simbolo> con↩
                    occorrenze
24             #al massimo pari a 255, fino al consumarle tutte
25             while(count > 255):
26                 count -= 255
27                 encoded += bytes([255])
28                 encoded += lastByte
29
30             encoded += bytes([count])
31             encoded += lastByte
32             count = 1
33         #Memorizzo l'ultimo byte letto
34         lastByte = byte
35
36     #Stessa cosa di sopra con le coppie rimanenti
37     while(count > 255):
38                 count -= 255
39                 encoded += bytes([255])
40                 encoded += lastByte
41
42     encoded += bytes([count])
43     encoded += lastByte
44     return encoded
45
```

```
46
47 #Funzione di decodifica di RLE
48 def RLE_decode(data):
49     #Inizializzo l'array di byte
50     output = bytearray()
51
52     #Loop in cui leggo 2 byte alla volta
53     while (byte := data.read(2)):
54         #Se ho raggiunto la fine del file termino
55         if byte[0] == b'':
56             break
57
58         #Aggiungo al mio array il simbolo appena letto dal secondo byte,↩
               ripetuto tante volte quante indicate
59         #nel valore del primo byte
60         #print("Byte: " + str(byte[0]) + " times " + str(byte[1]))
61         output += (bytes([byte[1]]) * int(byte[0]))
62     return output
```

A.3 Huffman.py

```
1  from queue import PriorityQueue
2  from bitarray import bitarray
3
4
5  #Funzione che si occupa di codificare la tabella in una stringa
6  def encodeTable(table):
7      tableString = ''
8      #Per ogni chiave nella tabella formatto a dovere la stringa:
9      #la lunghezza del prefix code espressa in 8 bit, il prefix code e ↩
           altri
10     #8 bit per il byte associato al prefix code
11     for k in table:
12         tableString += '{0:08b}'.format(len(k)) + k + '{0:08b}'.format(↩
               table[k])
13     return tableString
14
15
16 #Funzione che decodifica la tabella
17 def decodeTable():
18     #Apro il file contenente la tabella
19     file = open('./test/table.huff', 'rb')
20
21     #Inizializzo alcune variabili e leggo la tabella trasformandone il
22     #contenuto in una stringa binaria
23     dictionary = {}
24     byte = file.read()
25     file.close()
26     array = bitarray()
27     array.frombytes(byte)
28     text = array.to01()
29
30     #Fino a quando ho dei byte da leggere
31     while(len(text) >= 8):
32         #Parsing del numero di bit da leggere
33         n_bits = int(text[:8], 2)
34
35         #Parsing di chiavi e valori del dizionario
36         text = text[8:]
37         key = text[:n_bits]
38         text = text[n_bits:]
39         value = int(text[:8], 2)
40         text = text[8:]
41         dictionary[key] = value
42     return dictionary
43
44
45 #Funzione che converte la rappresentazione "ad albero" in una tabella
46 def treeToTable(root, prefix, lookup_table):
```

```
47      element = root[2]
48      #Se l'elemento attuale non rappresenta una ennupla
49      if type(element) != tuple:
50          #Se il prefisso e' vuoto lo imposto a zero
51          if prefix == '':
52              lookup_table[element] = '0'
53          #Altrimenti lo imposto direttamente
54          else:
55              lookup_table[element] = prefix
56      #Se abbiamo una ennupla chiamo ricorsivamente la funzione, una volta
57      #aggiungendo '0' al prefix e una volta '1', in questo modo tale da
58      #"srotolare" tutto l'albero
59      else:
60          treeToTable(element[0], prefix + '0', lookup_table)
61          treeToTable(element[1], prefix + '1', lookup_table)
62      return lookup_table
63
64
65  #Funzione che codifica il contenuto del file
66  def encodeBody(data):
67      if data == b'':
68          return ''
69
70      table = {}
71
72      # Aggiungo i caratteri alla table, associando ad ogni lettera il ↩
            numero di occorrenze
73      for char in data:
74          if char in table:
75              table[char] += 1
76          else:
77              table[char] = 1
78
79      #Creo una coda e un contatore che funge da identificatore
80      q = PriorityQueue()
81      counter_id = 0
82
83      #Per ogni chiave-valore della tabella inserisco la coppia
84      #nella coda
85      for k,v in table.items():
86          q.put((v, counter_id,k))
87          counter_id += 1
88
89      #Fino a quando la coda non e' vuota ottengo due nodi, creo
90      #il terzo nodo dato dalla somma delle frequenze dei due e
91      #aggiungo quest'ultimo alla coda
92      while(q.qsize() > 1):
93          x = q.get()
94          y = q.get()
95
96          sum_freq = (x[0]) + (y[0])
```

```
97          z = (sum_freq, counter_id,(x,y))
98          counter_id += 1
99
100         q.put(z)
101
102     #Estraggo l'ultimo nodo
103     root = q.get()
104
105     #Converto l'albero in una tabella
106     a = treeToTable(root, '', {})
107
108     #Effettuo la codifica del contenuto e preparo la tabella in un certo
109     #formato
110     t = {k:bitarray(v) for k,v in a.items()}
111     table2 = {v: k for k, v in a.items()}
112     encoded = bitarray()
113     encoded.encode(t, data)
114     return (encoded.to01(), table2)
115
116
117 #Effettua la decodifica del contenuto
118 def decodeBody(data, dictionary):
119     #Inizializzo e imposto i valori
120     res = bytearray()
121     subs = data[0]
122     l = 1
123
124     #Finch  ci sonod dati da leggere cerco iterativamente dei prefix
125     #code che siano presenti nella tabella
126     while data:
127         while subs not in dictionary:
128             l += 1
129             subs = data[:l]
130
131         #Aggiungo il valore associato al prefix code attraverso la
132         #tabella
133         res += bytes([dictionary[subs]])
134
135         #Scorro il buffer dei dati
136         data = data[len(subs):]
137
138         #Se non ci sono piu' dati concludo
139         if data == '':
140             break
141
142         #Ritorno alle impostazioni iniziali per la prossima iterazione
143         l = 1
144         subs = data[0]
145     return res
146
147
```

```
148
149
150 #Funzione della codifica di Huffman
151 def Huffman_encode(file):
152     #Leggo il contenuto del file
153     data = file.read()
154     file.close()
155
156     #Codifico il contenuto del file e preparo la tabella
157     (a, table) = encodeBody(data)
158     output = bytearray()
159
160     #Formatto l'output del body a dovere: 4 byte per il numero di bit da↩
            leggere
161     #e X byte contenenti il corpo codificato
162     bodyBits = bitarray(a)
163     bodyBytes = bodyBits.tobytes()
164     bitsLen = bodyBits.length()
165     output += bitsLen.to_bytes(4, "little")
166     output += bodyBytes
167
168     #Codifico anche la tabella e vado ad esportarla nel file 'table.huff↩
           '
169     encodedTable = encodeTable(table)
170     tableBits = bitarray(encodedTable)
171     tableBytes = tableBits.tobytes()
172     dictOutput = open('./test/table.huff', 'wb')
173     dictOutput.write(tableBytes)
174     dictOutput.close()
175     return output
176
177
178 #Funzione di decodifica di Huffman
179 def Huffman_decode(file):
180     #Decodifico la tabella
181     decodedTable = decodeTable()
182
183     #Parsing del numero di bit da leggere
184     bitLen = int.from_bytes(file.read(4), "little")
185
186     #Leggo il contenuto del file e procedo con la decodifica del corpo
187     data = file.read()
188     file.close()
189     bits = bitarray()
190     bits.frombytes(data)
191     bitString = bits.to01()[:bitLen]
192     res = decodeBody(bitString, decodedTable)
193     return res
```

A.4 LZ77.py

```
1  import math
2
3  #Funzione che restituisce il match pi  lungo e vicino alla fine del
4  #search buffer
5  def longestMatch(data, search_index, look_index, look_size):
6      #Imposto gli offset iniziali dei due buffer
7      offset_i = search_index
8      offset_j = look_index
9
10     #Inizializzo le mie variabili
11     length = 0
12     max_len = 0
13     index = 0
14     match = None
15     flag = False
16
17
18     #Loop che si arresta una volta soddisfatte le condizioni di ricerca
19     while (offset_i < look_index or flag) and (offset_j < (look_index+↩
           look_size) and offset_j < len(data)):
20         search_pos = data[offset_i]
21         look_pos = data[offset_j]
22
23         #Se la posizione del search buffer corrisponde a quella del look↩
               buffer
24         #incremento gli offset e la dilatazione della finestra
25         if search_pos == look_pos:
26             length += 1
27             offset_j += 1
28             offset_i += 1
29
30             #Se la lunghezza massima della finestra e' stata raggiunta
31             #memorizzo le informazioni raccolte nelle rispettive ↩
                   variabili
32             if length >= max_len:
33                 max_len = length
34                 index = offset_i-length
35                 match = data[offset_i-length:offset_i]
36
37             #Setto un flag nel caso in cui l'offset i superi la ↩
                   posizione del
38             #look buffer
39             if offset_i >= look_index:
40                 flag = True
41
42         #Se la posizione del search buffer non corrisponde e l'offset i ↩
               e'
```

```
43          #minore dell'indice del look buffer, allora continuo con la ↩
               ricerca
44          #del longest match
45          elif offset_i < look_index:
46              offset_j = look_index
47              length = 0
48              search_pos = data[offset_i]
49              look_pos = data[offset_j]
50
51              #Se ho una corrispondenza tra search e look buffer vado
52              #alla successiva iterazione senza incrementare l'offset i
53              if search_pos == look_pos:
54                  continue
55
56              offset_i += 1
57
58          #Se l'offset e' maggiore o uguale all'indice del look buffer ho
59          #finito e posso concludere la ricerca
60          else:
61              break
62      return (index, match)
63
64
65  #Funzione che ritorna il numero di byte minimo per rappresentare il ↩
       valore x
66  def getNeededBytes(x):
67      bitNeeded = math.ceil(math.log2(x))
68      return math.ceil(bitNeeded/8)
69
70
71  #Funzione di codifica LZ77, che prende come argomento una windowSize
72  def LZ77_encode(file, windowSize):
73      data = file.read()
74
75      #Inizializzo e imposto le variabili
76      current_search_size = 0
77      search_max_index = math.ceil(windowSize/2)
78      look_max_index = search_max_index
79      i = 0
80      j = 0
81      output = bytearray()
82
83      #Ottengo il numero di byte necessari a rappresentare il
84      #massimo numero richiesto per offset e length
85      byteLenO = getNeededBytes(search_max_index+1)
86      byteLenL = getNeededBytes(windowSize)
87
88      #Itero finch  l'indice j non ha raggiunto la fine dei dati
89      while j < len(data):
90          #Aggiorno la dilatazione del look buffer
91          look_buffer = data[j:j+look_max_index]
```

```
92
93          #Cerco il nuovo longest match
94          match = longestMatch(data, i, j, look_max_index)
95
96          #Se non ho trovato nessun match, ricado nel caso base dell'↩
               algoritmo
97          if match[1] is None:
98              o = 0
99              l = 0
100             c = look_buffer[0]
101         #Altrimenti devo calcolare opportunamente i parametri <offset, ↩
               length e
102         #next byte (c)>
103         else:
104             o = j - match[0]
105             sentry = len(match[1])
106             if j+sentry >= len(data):
107                 extra = len(data) - j
108                 l = extra-1
109                 c = data[len(data)-1]
110             else:
111                 l = len(match[1])
112                 c = data[j + l]
113
114
115         #Inserisco la tripla di valori in byte
116         output += o.to_bytes(byteLen0, 'little')
117         output += l.to_bytes(byteLenL, 'little')
118         output += bytes([c])
119
120         #Incremento l'indice j in funzione di quanto letto
121         j += l+1
122
123         #Eseguo varie operazioni per aggiustare a dovere
124         #la dimensione della finestra rispettando eventuali
125         #limiti
126         if (current_search_size+l+1) <= search_max_index:
127             current_search_size += (l+1)
128         else:
129             current_search_size = search_max_index
130
131         if j > (search_max_index-1):
132             current_search_size = search_max_index
133             i = j-search_max_index
134     return output
135
136
137 #Funzione di decodifica LZ77, prende anche lei come argomento una ↩
       windowSize
138 def LZ77_decode(data, windowSize):
139     output = bytearray()
```

```
140
141     #Calcolo i byte necessari in maniera speculare a quanto fatto prima
142     byteLen0 = getNeededBytes(math.ceil(windowSize/2)+1)
143     byteLenL = getNeededBytes(windowSize)
144
145     #Leggo, in relazione al numero di byte richiesti, la tripla di ↩
            valori
146     #<offset, length, c>
147     while (byte := data.read(byteLen0+byteLenL+1)):
148         o = byte[0:byteLen0]
149         l = byte[byteLen0:byteLen0+byteLenL]
150         c = byte[byteLen0+byteLenL:byteLen0+byteLenL+1]
151
152         #Se per qualche motivo (come la fine del file) trovo uno dei ↩
                valori
153         #nulli, allora posso concludere la fase di decodifica
154         if o == b'' or l == b'' or c == b'':
155             break
156
157         o = int.from_bytes(o, "little")
158         l = int.from_bytes(l, "little")
159
160         i = len(output) - o
161
162         #Fino a quando ho dei byte da elaborare trasferisco i byte ↩
                decodificati
163         #come output
164         while l > 0:
165             output += bytes([output[i]])
166             i += 1
167             l -= 1
168
169         output += c
170     return output
```