Guida alle compressioni

Aggiornata al 17/12/2007

by Phoenix

0. INTRODUZIONE

Lo scopo di questa guida è quello di far comprendere, in maniera più chiara possibile, il funzionamento delle compressioni basate sull'algoritmo LZSS. Per fare ciò ho allegato alla guida i codici sorgenti in c++ del decompressore (SoulExt.cpp) e del compressore (SoulCom.cpp) per i file di un gioco PSX (Soul Blade) e un file compresso (contenente il font di gioco) per fare delle prove. Se volete provare, potete aprire il file decompresso con un qualunque programma che supporti il formato TIM (consiglio TIM Viewer).
Ho letto un po' di guide riguardanti compressioni, ma alcune sono troppo teoriche e altre troppo sintetiche. Spero di riuscire a mettere insieme le qualità di entrambi i tipi di guida.

Le compressioni non sono altro che un sistema per far occupare ai nostri file meno spazio di quello che occuperebbero nel loro stato originale. Basti pensare a dei software come 7-Zip, WinRAR, WinZIP, WinACE ecc. Ognuno di questi programmi mette a disposizione un sistema di compressione più o meno soddisfacente...Ma questo lo sapevate già, vero?

Dopo quest' attimo di pazzia, possiamo iniziare!

Decompressione

1. SI COMINCIA

La compressione di tipo LZSS è molto semplice, basta solo capirne la logica.
In sostanza, questo tipo di compressione utilizza il metodo della coppia salto/recupero.
"Cosa sono?" direte voi, ecco un esempio:

Immaginiamo di avere un file con dentro scritto:

ciao_ciao_mamma,_mamma_mia!

N.B. Con "_" indico uno spazio.

Volendo ora comprimere questo file con il metodo LZSS ecco come apparirebbe il nostro file compresso:

ciao_**5,5**mamma,**7,6**mia!

Cerchiamo di capire cos'è successo.
Come potete notare, al posto della seconda parola ciao_ c'è qualcos'altro: la coppia salto/recupero. Questi due nuovi fattori stanno ad indicare che è stata trovata una sequenza di caratteri già presente (il primo ciao_) e quindi è possibile sostituirla con 2 numeri (la coppia): il salto, che indica dove andare a prendere la parola ciao_, e il recupero, che indica quanti caratteri prendere.

In questo modo, invece di scrivere per intero la parola ciao_, utilizziamo solo due caratteri, il che ci fa risparmiare un po' di spazio.

Per capire meglio come funziona la coppia, proviamo a decomprimere a mano la sequenza appena compressa (ciao_**5,5**mamma,**7,6**mia!):

Il primo carattere che incontriamo nella sequenza compressa è la lettera c. Non essendo parte di una coppia, ma un semplice carattere (che chiamerò normale), può essere scritto direttamente nel file decompresso in uscita:

c

Il prossimo carattere che vedremo sarà la lettera i. Anch'essa potrà essere scritta direttamente in uscita. Lo stesso procedimento varrà anche per le lettere successive, fino ad arrivare allo spazio (_). Otteniamo così:

ciao_

Dopo aver scritto i caratteri normali (ciao_), ci troveremo proprio sulla coppia **5,5**.
Il primo **5** della coppia indica il salto, mentre il secondo **5** indica il recupero.
Vediamo come bisogna comportarsi.
È necessario immaginare di posizionarsi alla fine dei dati decompressi fino a questo momento (ciao_), cioè subito dopo lo spazio.
Dopo esserci posizionati, sarà necessario andare indietro di 5 caratteri (salto). Ci troveremo quindi sulla lettera "c".
Fatto questo, preleviamo 5 (recupero) caratteri (ciao_), e scriviamoli alla fine dei dati decompressi fino ad ora, ottenendo:

ciao_ciao_

Si intuisce quindi che il salto ed il recupero sono riferiti ai dati appena decompressi, e non ai dati compressi.

N.B. Per quanto riguarda il recupero dei caratteri, il procedimento esposto non è del tutto corretto. Ho utilizzato questo sistema per semplificarne la comprensione, tuttavia consiglio caldamente, prima di continuare con la lettura, di leggere il paragrafo "e" del capito "ALCUNE CONSIDERAZIONI", in modo da capire quale sia il metodo corretto per il prelievo dei caratteri. Clicca qui per andare al paragrafo "e"

Fatto questo, notiamo che dopo la coppia **5,5** appena utilizzata, vi è un'altra sequenza di caratteri normali (mamma,), che andrà ad essere scritta direttamente nel file decompresso. Il risultato ottenuto sarà:

ciao_ciao_mamma,

Subito dopo notiamo un'altra coppia (**7,6).** Come fatto in precedenza, ci posizioneremo alla fine dei dati appena decompressi, cioè subito dopo la virgola, andremo indietro di 7 caratteri (salto), trovandoci così sullo spazio prima di mamma, e preleveremo 6 (recupero) caratteri (_mamma) che scriveremo alla fine dei dati decompressi, ottenendo così:

ciao_ciao_mamma,_mamma

Infine notiamo che subito dopo la coppia **7,6** appena utilizzata, vi sarà un'ulteriore sequenza di caratteri normali (mia!). Scriviamo anche questa sequenza direttamente, ed otterremo:

ciao_ciao_mamma,_mamma_mia!

Abbiamo completato la fase di decompressione!

Naturalmente la coppia salto/recupero non viene scritta così sul file ma utilizza soltanto due byte, a cui vengono applicate delle operazioni di cui parleremo a breve.

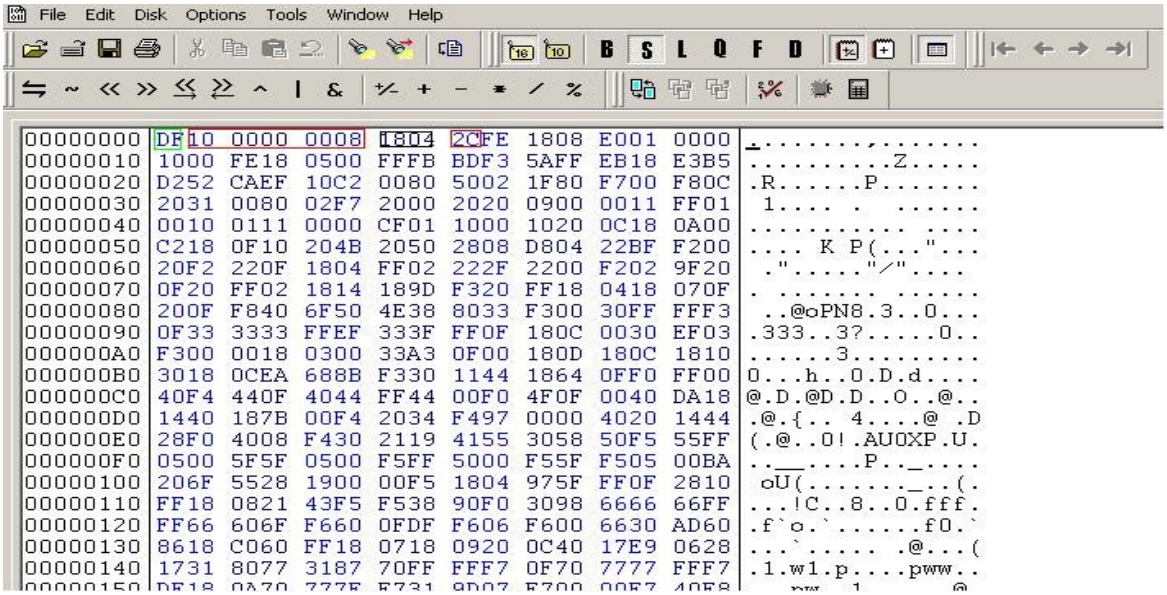
2. SI PASSA AI FATTI!

N.B. Tutti i valori preceduti da 0x sono da intendersi esadecimali.

Questa parte della guida servirà a far capire meglio come funziona una compressione basata su algoritmo LZSS servendosi di un esempio pratico:

Aprite con un editor Esadecimale (Consiglio Hex WorkShop) il file con estensione .sz presente in questo .zip.
Dovreste vedere una cosa del genere:

Figura (1)



Il primo byte (quello evidenziato in verde) è detto BYTE RICONOSCITORE.

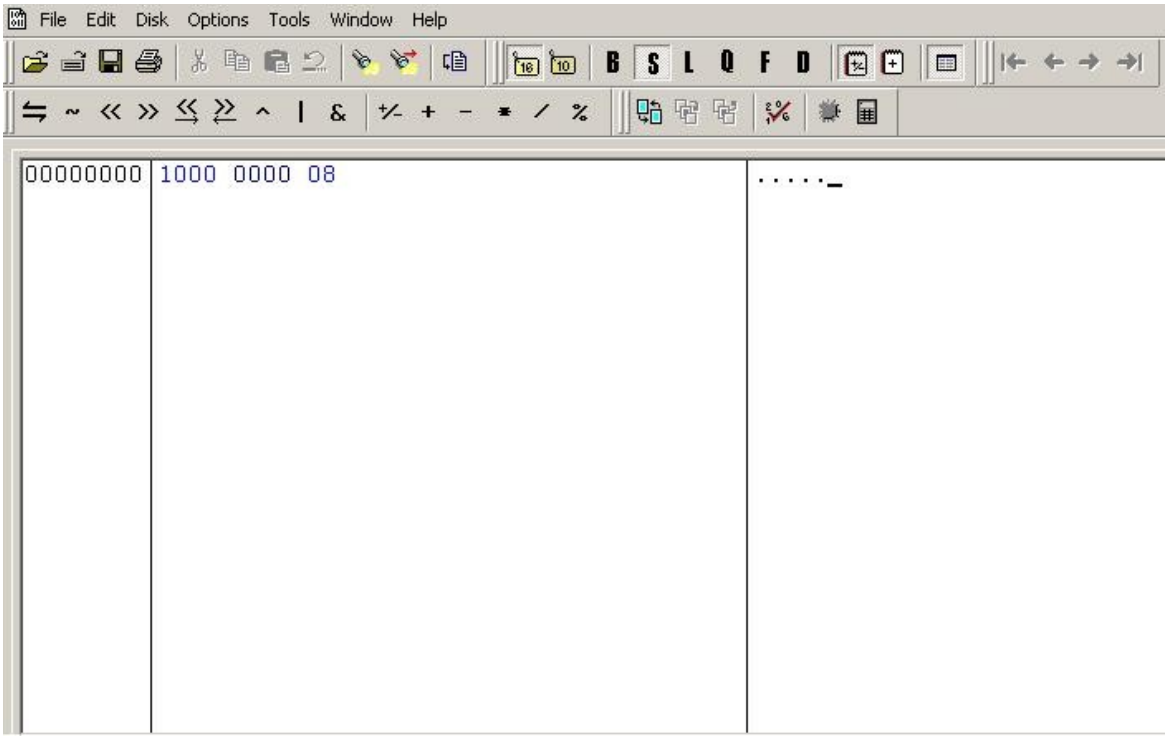
Il byte riconoscitore serve al decompressore per capire quali sono dati compressi e quali no.
Ecco come funziona:
Aprite la cacolatrice scientifica di Windows e convertite il byte riconoscitore (in questo caso 0xDF) in bit, e otterrete questa sequenza 11011111.

Questa sequenza di bit (detti flag) fa capire al decompressore quali, fra i byte successivi al riconoscitore, sono byte normali e quali invece coppie salto/recupero. I bit vanno letti da destra verso sinistra, quindi il primo bit che notiamo è 1. Questo 1 sta ad indicare che il primo byte dopo il riconoscitore (0x10) è un byte normale e quindi il decompressore può prenderlo così com'è e scriverlo nel file decompresso in uscita.

Il secondo bit da destra è un altro 1, quindi, il secondo byte subito dopo il riconoscitore (0x00) sarà un altro byte normale che andrà ad essere aggiunto al nostro file decompresso in uscita, e così via fin quando il decompressore non incontra un flag=0.

Il file in uscita, dopo che il decompressore ha scritto i byte "normali", dovrebbe avere un aspetto del genere:

Figura (2)



Quando il decompressore incontra il flag 0, che in questo caso è il sesto da destra, il decompressore capisce che il sesto byte subito dopo il riconoscitore NON è un byte normale, ma contiene informazioni sul salto ed il recupero:

Il byte in questione è **0x18**, ma un solo byte non può contenere due informazioni quali il salto e il recupero, ma è necessario un altro byte, quello che segue **0x18**, e cioè **0x04**.
"Il 0x18 indica il salto e lo 0x04 indica il recupero" direte voi. Non è detto, ogni compressione derivante dall'algoritmo LZSS utilizza il suo metodo di memorizzazione della coppia salto/recupero. Per quanto riguarda questo tipo di compressione, le operazioni da effettuare su questi due byte per ottenere il salto ed il recupero sono le seguenti:

Prendiamo il primo byte (0x18) e convertiamolo in binario. Otterremo 11000 dalla calcolatrice di Windows, però ogni byte è composto da 8 bit, quindi lì manca qualcosa...
il 0x18 può essere scritto in binario anche così: 00011000, senza trascurare quindi gli zeri iniziali, ed ottenendo così una sequenza di 8 bit completa.

Ora prendete i primi 5 bit (00011).
Nota per i programmatori: Questo equivale a fare uno shift a destra di 3 (vedi codice sorgente).

Convertiteli in decimale ed otterrete 3!
Ecco, questo è il nostro **recupero**, la quantità di byte da andare a recuperare. Ora non resta che calcolare il salto.

Prendete il secondo byte della coppia (0x04) e convertite anch'esso in bit, ottenendo così 100, che con la sequenza completa di 8 bit diventa 00000100.

Alla sequenza in bit ottenuta da 0x18 (00011000) accostate la sequenza ottenuta da 0x04, avendo come risultato 0001100000000100.
Nota per i programmatori: Questo equivale a fare uno SHIFT a SINISTRA di 8 al primo byte e poi fare un OR tra il risultato ottenuto e il secondo byte (vedi il codice sorgente).

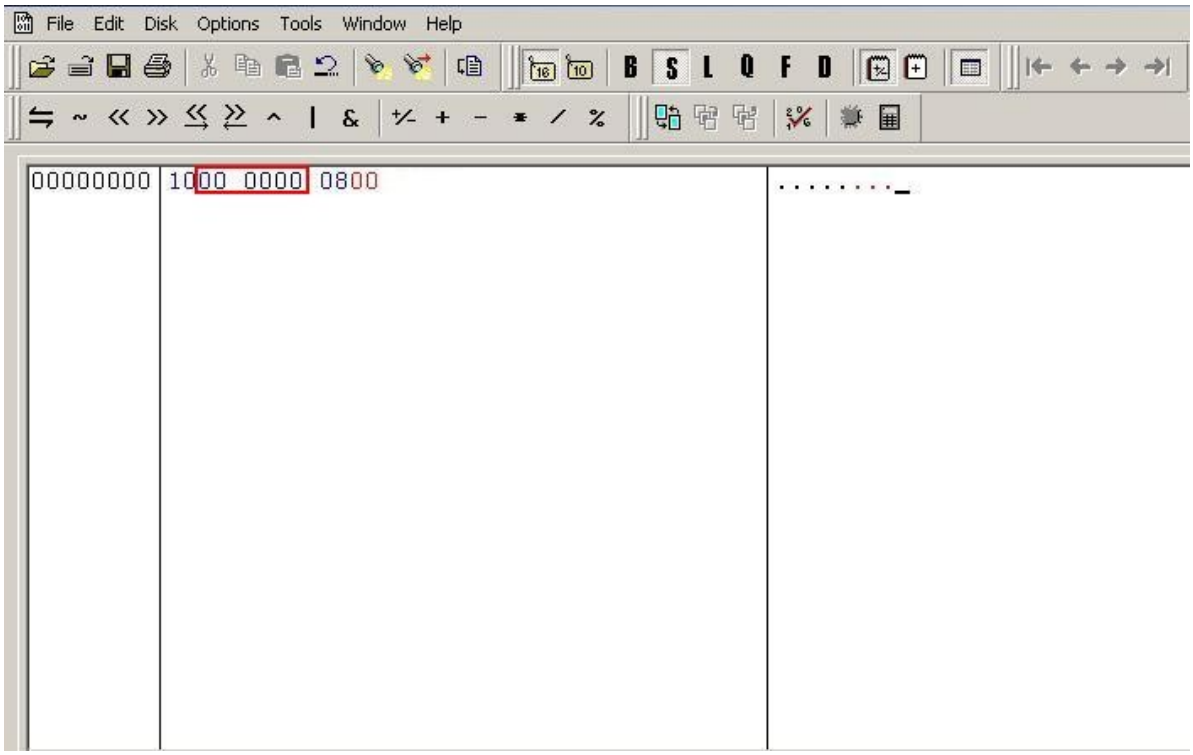
Da questa sequenza prendete solo gli utlimi 11 bit, ottenendo così 00000000100.
Nota per i programmatori: Questo equivale a fare un AND con 0x7FF (vedi codice sorgente).

Infine, convertite 00000000100 in decimale ed otterrete 4.
Questo è il nostro **salto**!

Il decompressore, dopo essersi calcolato il salto ed il recupero in questa maniera, li utilizzerà per recuperare i dati necessari come abbiamo visto nell'esempio del capitolo 1, **ricordando sempre che il salto è relativo ai dati appena decompressi,** nel nostro caso, i dati della figura (2).

Quindi, partendo dalla fine dei dati decompressi (fig. 2), il decompressore andrà indietro di 4 posizioni (salto) e si troverà sul byte 0x00 (quello subito dopo 0x10), prenderà 1 byte (il primo dei tre 0x00 evidenziati) e lo scriverà alla fine dei dati attuali, ottenendo come risultato una cosa del genere:

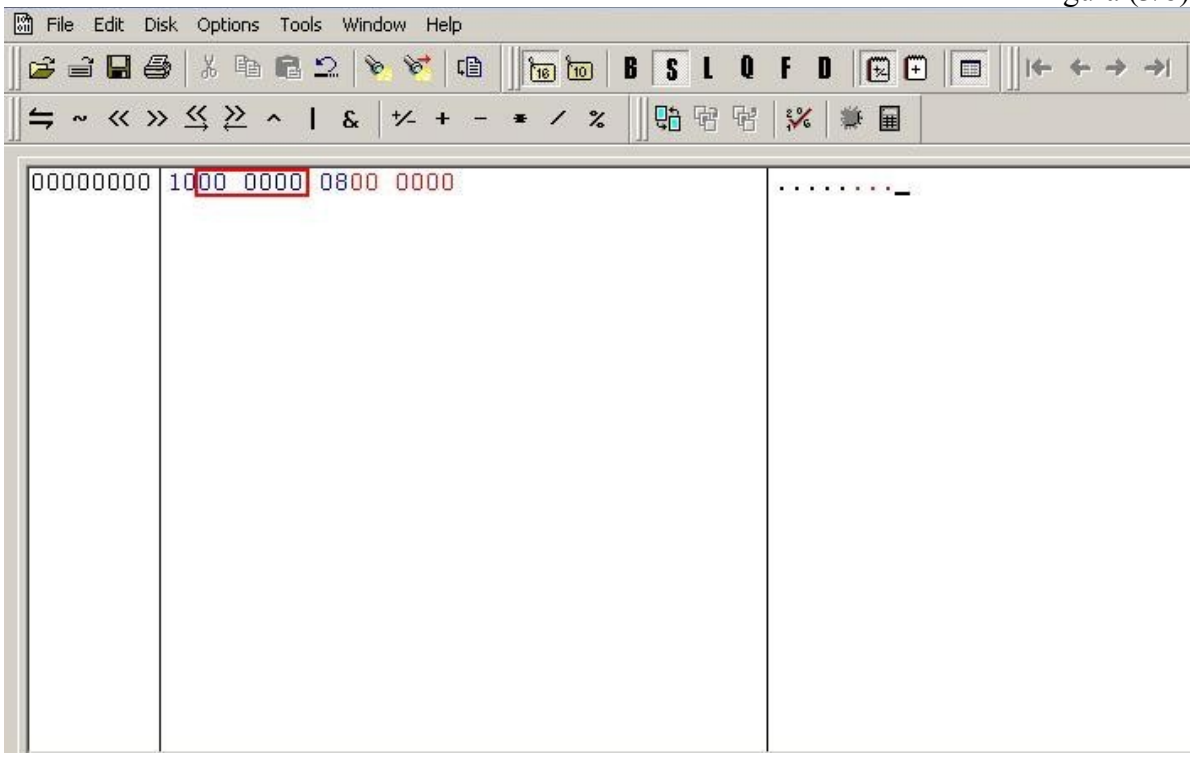
Figura (3/a)



Fatto questo, il decompressore prenderà il prossimo byte (il secondo dei tre 0x00) e lo

scriverà alla fine del file decompresso.
Infine, prenderà il byte successivo (il terzo dei tre 0x00), aggiungendolo alla fine del file, avendo effettuato così il recupero di **3** byte, come richiesto dalla coppia:

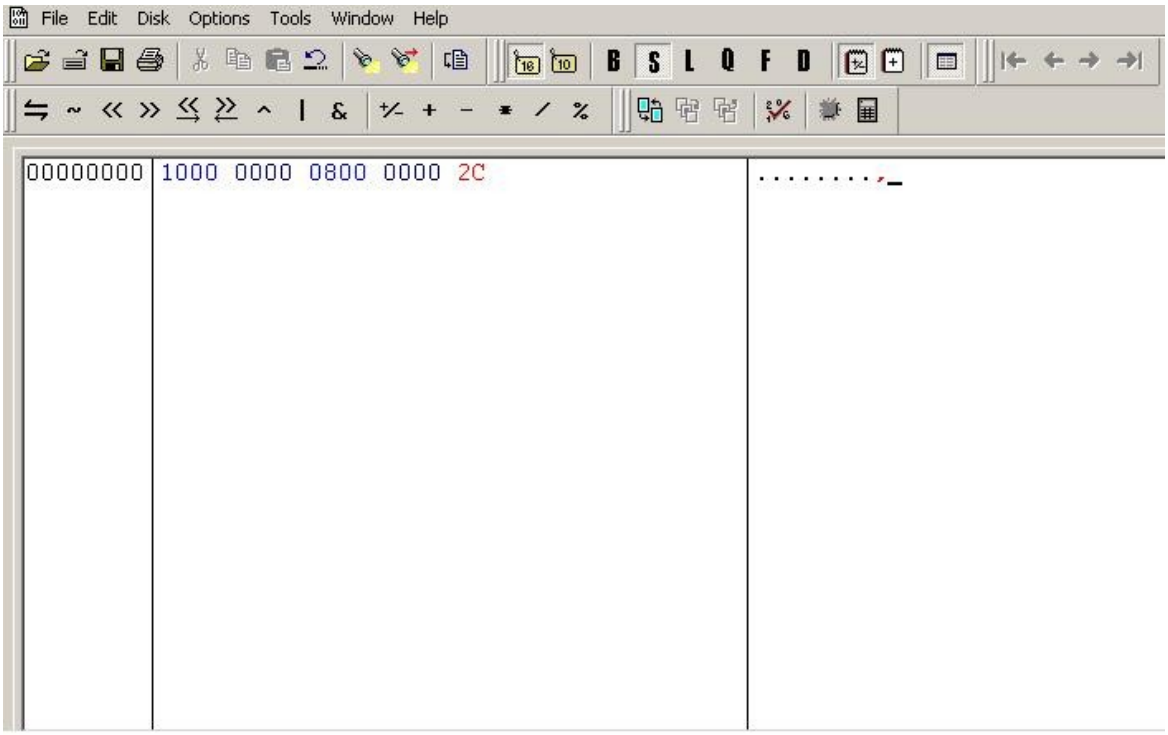
Figura (3/b)



N.B. Se avete notato, il recupero dei byte è stato effettuato un byte alla volta, proprio come descritto nel paragrafo "e" del capitolo "ALCUNE CONSIDERAZIONI" che avreste dovuto leggere prima di arrivare a questo punto :D.

Arrivati a questo punto, il decompressore può passare alla lettura del prossimo flag, cioè 1, e quindi scriverà il prossimo byte (0x2C) direttamente.
Subito dopo vediamo un altro flag, ma nel caso della nostra compressione, non viene utilizzato. Questa variante della LZSS si limita ai primi 7 flag da destra, mentre molte altre li utilizzano tutti e otto.
In questo caso avremo come risultato finale:

Figura (4)



Queste, in definitiva, sono le operazioni che compie il decompressore, per un singolo byte riconoscitore. Subito dopo ci sarà un altro byte riconoscitore, e quindi queste operazioni verranno ripetute per questo e per tutti i byte riconoscitori successivi.
Ad esempio, il prossimo byte riconoscitore sarà 0xFE. Infatti, si trova subito dopo tutti i byte che venivano indicati dal riconoscitore precedente.

3. ALCUNE CONSIDERAZIONI

a)
È importante sapere che gli 8 flag del byte riconoscitore non indicano che successivamente vi saranno SOLO 8 byte, ma ci saranno tanti byte normali quanti saranno i flag=1 e tante coppie quanti saranno i flag=0. Ad esempio, avendo questi flag 10011101, avremo 5 byte normali (visto che i flag=1 sono 5) e 3 coppie (visto che ci sono 3 zeri), quindi 6 byte.
In totale, dopo il riconoscitore, ci saranno 5+6=11 byte.
Subito dopo ci sarà il prossimo byte riconoscitore ecc. Ecc.

b)
In altre compressioni derivanti dall'LZSS (molte altre), anche dopo aver effettuato il calcolo del recupero, esso non sarà il recupero effettivo, ma gli si dovrà aggiungere un valore, molto spesso 3. Visto che la coppia occupa 2 byte, è normale che i dati da recuperare debbano essere almeno 3, altrimenti l'utilizzo della compressione risulterebbe inutile.
Se ad esempio, il recupero calcolato risultasse 2, il recupero effettivo sara 2+3=5.

c)
Questa variante della LZSS utilizza i flag=1 per indicare byte normali, mentre i flag=0 per indicare le coppie.
In genere, invece, è il contrario: flag=1 --> coppie ; flag=0 --> byte normali.

d)
Ci sono ancora due caratteristiche di questa variante che non ho menzionato, ma che sono presenti nel codice sorgente:

1) In caso il **recupero**, dopo aver effettuato i dovuti calcoli sopra menzionati, risultasse uguale a 0 (un po' improbabile, non vi pare?), il decompressore utilizzerà un recupero uguale a 32 (il massimo consentito dai 5 bit + 1, geniale eh!).
2) In caso il **salto**, dopo aver effettuato i dovuti calcoli sopra menzionati, risultasse uguale a 0 (un po' improbabile, non vi pare?), il decompressore utilizzerà un salto uguale a 2048 (il massimo consentito da 11 bit +1!).

e)
Prima di concludere, vorrei analizzare un caso particolare comune a tutte le compressioni di tipo LZSS.

Immaginiamo di avere una cosa del genere:

ah2,8

Seguendo alla lettera ciò che ci dice di fare la coppia, dovremmo tornare indietro di 2 posti e prendere 8 caratteri... Qualcosa non torna: Dove li piglio 8 caratteri se io ne ho solo 2!? ("ah").

Ecco la soluzione:

Uno degli errori più frequenti è quello di credere che i dati da recuperare vengano presi tutti in un colpo e successivamente scritti. Invece il recupero dei caratteri viene effettuato un carattere alla volta. Vediamo come.

Sicuramente nel file in uscita avremo la sequenza:

ah

In quanto abbiamo prima due caratteri normali (“ah”) e poi una coppia.

Adesso, facendo proprio come richiede la coppia, torniano indietro di 2 posizioni, partendo dalla fine dei dati appena decompressi ("ah") e prendiamo il primo carattere (in questo caso "a") e salviamolo alla fine del file, ottenendo in uscita:

aha

Spostandoci adesso di una posizione in avanti rispetto al carattere scritto in precendenza, ci troviamo sul carattere "h". Scriviamo anche questo carattere alla fine del file ottenendo:

ahah

Spostandoci nuovamente in avanti ci troveremo sulla seconda lettera "a" (quella fra le due "h"). Scriviamo anche questa ottenendo così:

ahaha

Si andrà avanti così fino quando non verrano scritti tutti i caratteri richiesti dalla coppia (recupero):

ahahahahah

Questo è il procedimento corretto per effettuare un recupero.

Clicca qui per tornare al capitolo “Decompressione”

f)
Può capitare a volte invece, che sia il salto ad essere troppo grande (si parla in questo caso di "buffer ad anello"). Mi spiego meglio:
Immaginiamo di avere:

Ciao7,3

Come faccio a tornare indietro di 7 posizioni se "Ciao" è lungo solo 4?
Basta tornare indietro di 4 (visto che non posso andare oltre), e mi trovo all'inizio di "Ciao".
Ho già esaurito quindi 4 delle 7 posizioni richieste dalla coppia, ne mancano 3.
Ora torno alla fine di "Ciao" e mi rimetto a contare, tornando indietro di 3, trovandomi così sulla "i". Completato il salto, non mi resta che prendere i caratteri richiesti (3), cioè "iao".
In definitiva abbiamo:

Ciaoiao

Compressione

1. **TECNICA DI COMPRESSIONE**
   compressione massima e non

A differenza del processo di decompressione, quello di compressione non viene effettuato utilizzando una procedura unica. Infatti, come molti di voi sapranno, i vari software di gestione archivi danno la possibilità di scegliere fra vari livelli di qualità di compressione (massimo, medio, basso o addirittura nullo).
In questo capitolo cercherò di spiegare le tecniche di compressione e di far capire da cosa dipendono i vari livelli di qualità.

Come nella prima parte della guida, ricorrerò ad un esempio per facilitare la comprensione:

Immaginiamo di avere la seguente sequenza di caratteri:

tre_treni_e_tre_trentini

N.B. Con il trattino in basso indico lo spazio

Prima di cominciare, facciamo un' osservazione:

Sapremo per certo che il primo carattere della stringa verrà sicuramente scritto nel file compresso, in quanto non potranno mai esserci delle corrispondenze prima di questo carattere, visto che è il primo carattere della sequenza. Quindi nel file compresso avremo sicuramente una cosa del genere:

t

Ora possiamo iniziare con la compressione (anche in questo caso, **qualunque operazione, eccetto la scrittura dei dati nel file compresso, viene eseguita sui dati decompressi**).

N.B. Per comodità del lettore, riscriverò, all'estrema destra della pagina per ogni passaggio, la sequenza di caratteri originale.

Dopo aver scritto il primo carattere, ci troveremo sul secondo carattere (r di tre).
Notiamo che prima di r c'è solo la lettera t. Non essendoci alcuna corrispondenza e non potendo quindi scrivere alcuna coppia, scriveremo il carattere così com'è:

tr

tre_treni_e_tre_trentini

Stesso dicasi per la terza lettera (e) e per la quarta (spazio). Avendo ottenuto così la sequenza:

tre_

tre_treni_e_tre_trentini

Il prossimo carattere è la t di treni. Possiamo notare che questa volta siamo riusciti a trovare una corrispondenza (la t di tre).
Potremmo semplicemente fermarci qui e scrivere una coppia che abbia come salto 4 (in quanto dalla t di treni è necessario andare indietro di quattro posizioni per arrivare alla t di tre) e come recupero 1 (prendendo cioè solo la t).

tre_**4,1**

tre_treni_e_tre_trentini

È intuibile che in questa maniera la compressione è inutile, anzi, invece di risparmiare spazio, ne occupiamo di più.

Quindi converrebbe continuare la ricerca di corrispondenze.

Notiamo infatti che non solo la t di treni è ripetuta, ma anche la r, quindi abbiamo trovato una corrispondenza maggiore (tr, recupero=2). Ma non basta, anche così la compressione risulterebbe inutile, perché la coppia (come visto in precedenza) occupa nel file compresso 2 byte, quindi sarà opportuno ricercare una corrispondenza tale che il recupero sia almeno 3.

Continuando quindi con la ricerca, troveremo che esiste una corrispondenza di caratteri lunga 3, cioè tre (scusate il gioco di parole :)).

In questo modo otteniamo un livello di compressione discreto, avendo così risparmiato un byte, sostituendo ad una sequenza di caratteri lunga 3 una coppia che occupa solo 2 byte.

Volendo, potremmo continuare con la ricerca e quindi migliorare la qualità di compressione. Per il momento non servirebbe a nulla, visto che prendendo la sequenza tren, questa non avrà alcuna corrispondenza prima di essa.

Il file compresso fino a questo punto sarà:

tre_**4,3**

tre_treni_e_tre_trentini

Cioè, torna indietro di 4 posizioni e preleva 3 caratteri **(facendo sempre riferimento alla stringa originale decompressa**). Avendo la coppia preso il posto della sequenza tre, il carattere da cui continuare la compressione sarà la n di treni, e come possiamo notare, non vi è alcuna n in precedenza, quindi dovremo scriverla direttamente:

tre_**4,3**n

tre_treni_e_tre_trentini

Stesso dicasi poi per la i di treni:

tre_**4,3**ni

tre_treni_e_tre_trentini

Il prossimo carattere sarà lo spazio dopo treni. Se notiamo, vi è una corrispondenza (lo spazio dopo tre), ma come visto prima, la corrispondenza di un solo carattere non è sufficiente. Anche continuando con la ricerca, non riusciremmo ad ottenere una corrispondenza sufficiente. Anche in questo caso siamo costretti a scrivere il carattere direttamente:

tre_**4,3**ni_

tre_treni_e_tre_trentini

Il prossimo carattere che andremo a prendere sarà la e congiunzione. Possiamo notare che questa volta vi è una corrispondenza, e anche molto lunga. Sto parlando della sequenza e_tre. Non essendo possibile ottenerne una più lunga, scriveremo la coppia relativa a questa:

tre_**4,3**ni_**8,5**

tre_treni_e_tre_trentini

Il carattere su cui ci troviamo adesso è lo spazio prima di trentini. Anche qui, si capisce benissimo che è possibile trovare una corrispondenza adeguata. Questa volta però, di corrispondenze ce ne sono due!

tre**_tren**i_e**_tre**_trentini

tre_treni_e_tre_trentini

Ecco un altro momento da cui dipenderà la qualità della nostra compressione. Il compressore deve essere in grado di capire quale, fra le due corrispondenze, è la migliore. In questo caso sarà la prima, ad ogni modo, se vogliamo ottenere la massima compressione sarà opportuno far ricercare al compressore tutte le corrispondenze possibili, senza limitarsi alla prima che viene trovata, in modo poi da poter scegliere quella ottimale.

Scegliendo la corrispondenza migliore (_tren), i dati compressi fino ad ora saranno:

tre_**4,3**ni_**8,5****12,5**

tre_treni_e_tre_trentini

N.B. Per evitare che i numeri vicini potessero essere letti in maniera errata, ho scritto le varie coppie con colori differenti

Infine, per l'ultima parte della sequenza da comprimere (tini) non vi sarà alcuna corrispondenza utile, ottenendo in definitiva:

tre_**4,3**ni_**8,5****12,5**tini

tre_treni_e_tre_trentini

2. LAVORIAMO UN PO' CON I FILE
facciamo quello che farebbe il compressore

N.B. Tutti i valori preceduti da 0x sono da intendersi esadecimali.

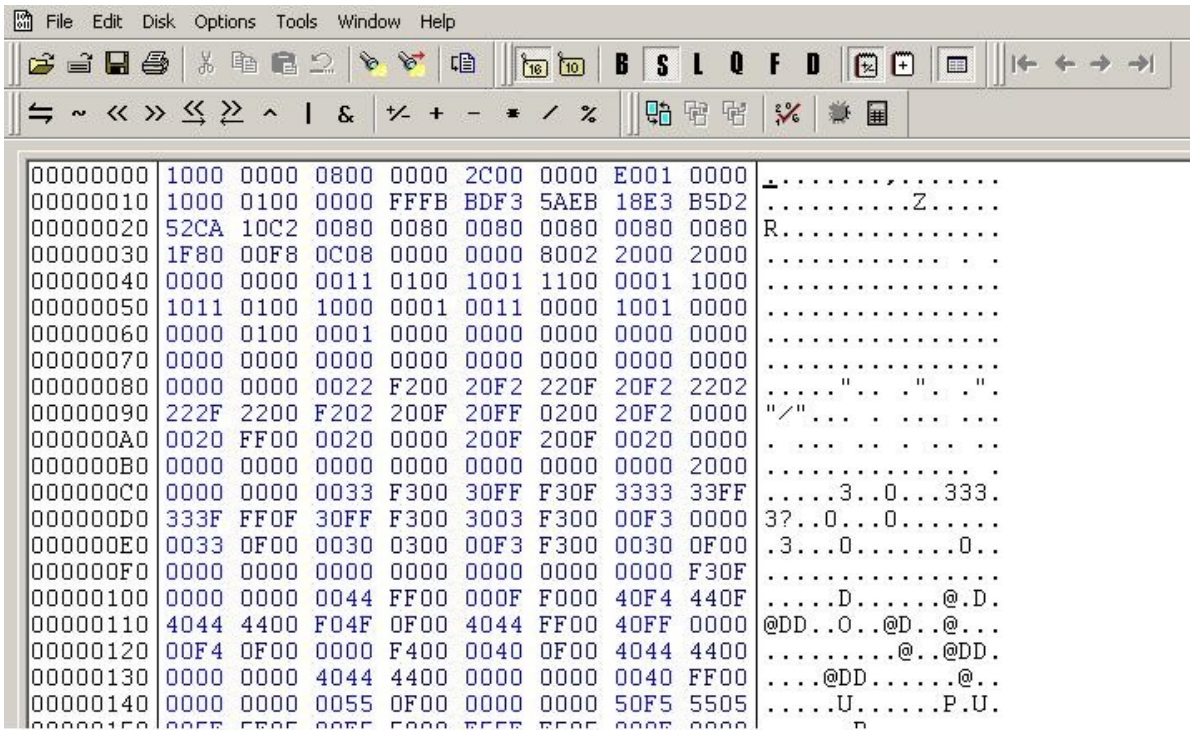
Questo capitolo ha l'obiettivo di chiarire gli ultimi (spero ultimi) dubbi sulla fase di compressione e di dare lo spunto hai programmatori per la creazione di un compressore funzionante.

Detto questo, cominciamo!

Il file “cavia” che tenteremo di comprimere, o almeno una parte, sarà lo stesso font decompresso dal mio tool. Quindi armatevi di pazienza e decomprimete il file con estensione .sz che trovate nel .zip.

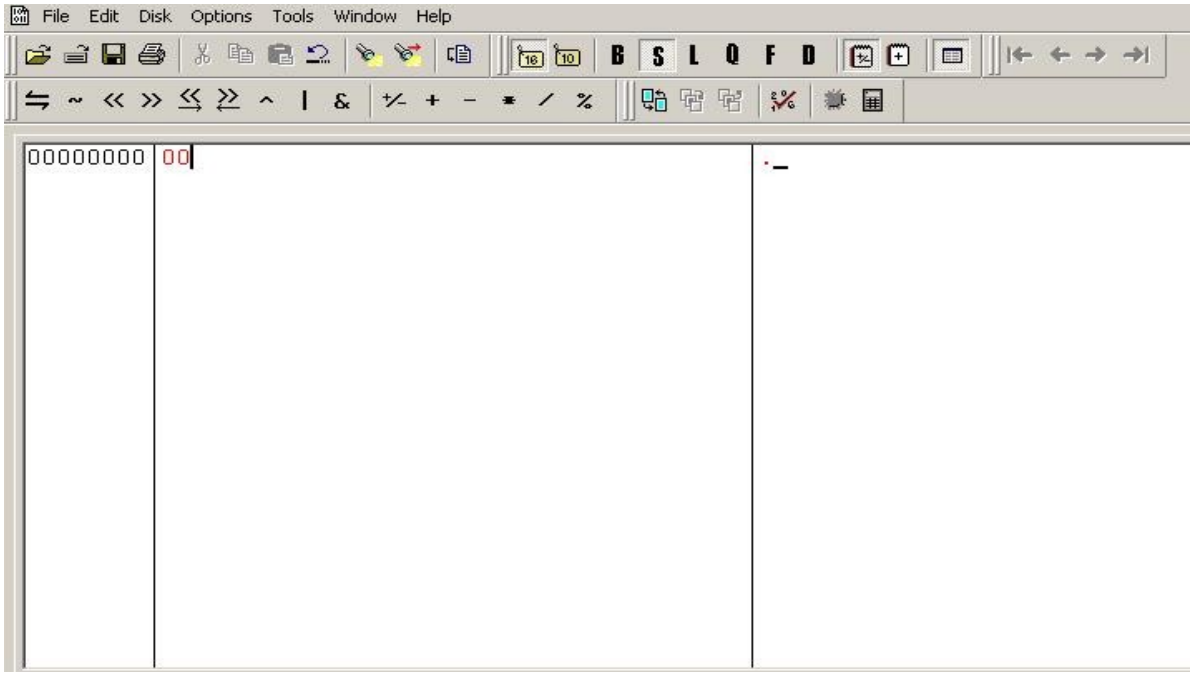
Dopo aver decompresso il file .sz aprite il file decompresso con un hex editor (come al solito consiglio Hex WorkShop). Dovremmo vedere una cosa del genere:

Figura (5)



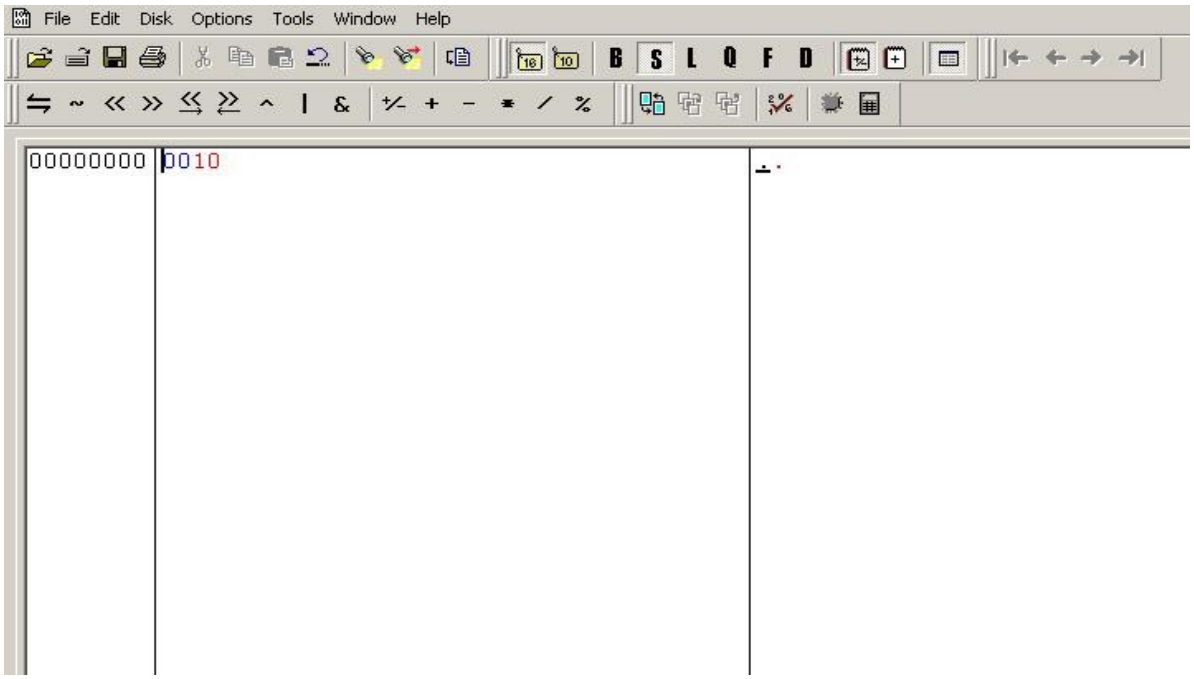
Se ricordate, il primo byte del file compresso visto in precedenza, era un byte riconscitore, il quale ci dava delle indicazioni sulle varie coppie e byte normali successivi.
Il primo passo per comprimere il nostro file è proprio quello di lasciare, nel file in uscita un byte vuoto (**0x00**) che successivamente dovrà essere sostituito con il riconoscitore.
Vediamo:

Figura (6)



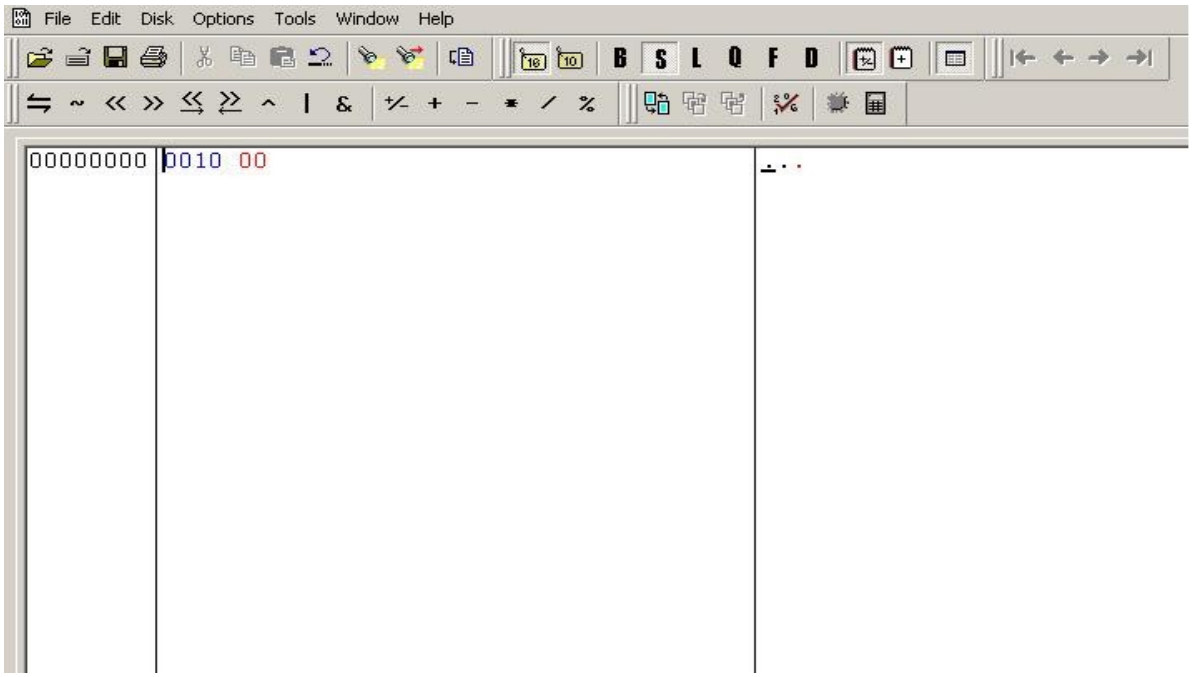
Ora possiamo iniziare con la compressione:
Il primo byte del file da comprimere sarà **0x10,** il quale, come visto nell'esempio teorico, andrà scritto direttamente in uscita, in quanto non avrà alcuna corrispondenza essendo il primo byte:

Figura (7)



Il prossimo byte sarà **0x00** (il byte subito dopo 0x10). Seguendo le operazioni descritte in precedenza, notiamo che l'unico byte che precede **0x00** è **0x10** quindi, anche qui non abbiamo alcuna corrispondenza utile per poter creare una coppia. Scriviamo il byte così com'è:

Figura (8)



Il byte successivo sarà un altro **0x00**. In questo caso possiamo notare che una corrispondenza c'è, ed è proprio il byte **0x00** subito prima. Ma non possiamo fermarci qui, continuiamo con la ricerca. Per effettuare la ricerca di una corrispondenza, si procede proprio come per il recupero dei byte, cioè un byte alla volta:

I dati da comprimere sono:

10 00 00 00 08 00 00 00 2C ecc. ecc.

In questo momento, ci troviamo sul byte indicato dalla freccia azzurra e abbiamo trovato la prima corrispondenza (recupero=1), indicata dalla freccia gialla:

10 00 00 00 08 00 00 00 2C

Trovata la prima corrispondenza si procede col trovarne un'altra, immaginando di spostare le due frecce in avanti:

10 00 00 00 08 00 00 00 2C

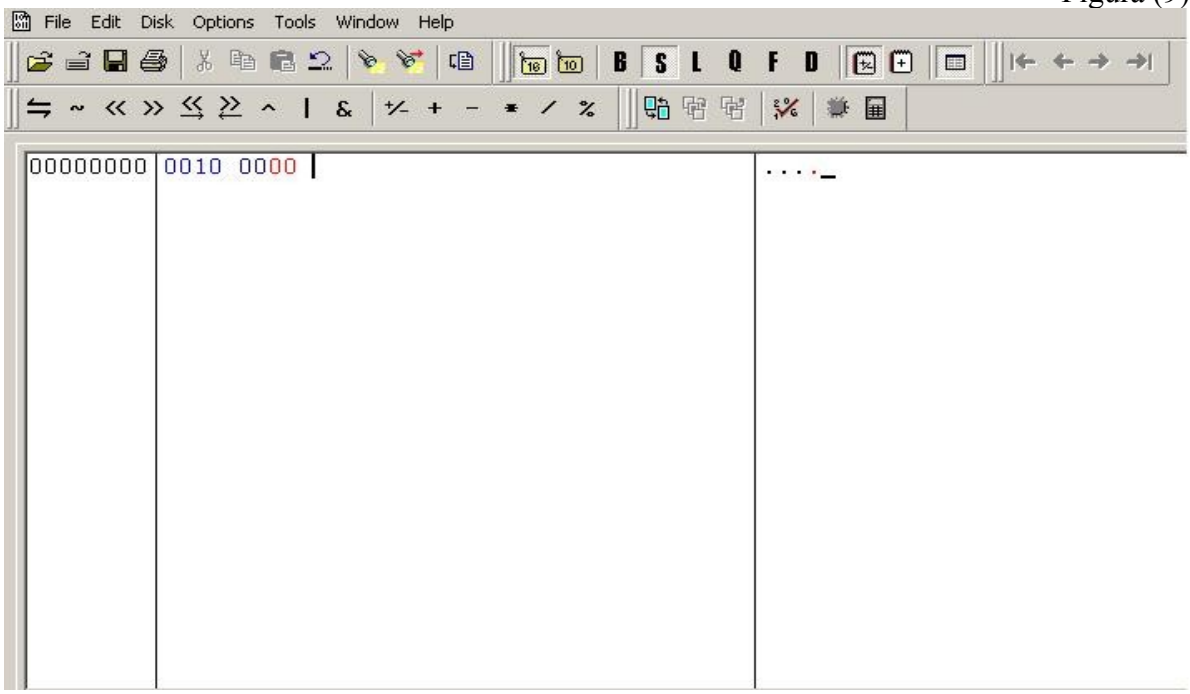
Anche in questo caso abbiamo una corrispondenza in quanto le due frecce indicano lo stesso carattere (0x00). Fino a questo momento abbiamo ottenuto un recupero pari a 2.

Si procederà con la ricerca, spostando di volta in volta le frecce, fin quando le due frecce immaginarie non punteranno più a dei caratteri uguali (non vi è più corrispondenza).
In questo caso le frecce si fermeranno qui:

10 00 00 00 08 00 00 00 2C

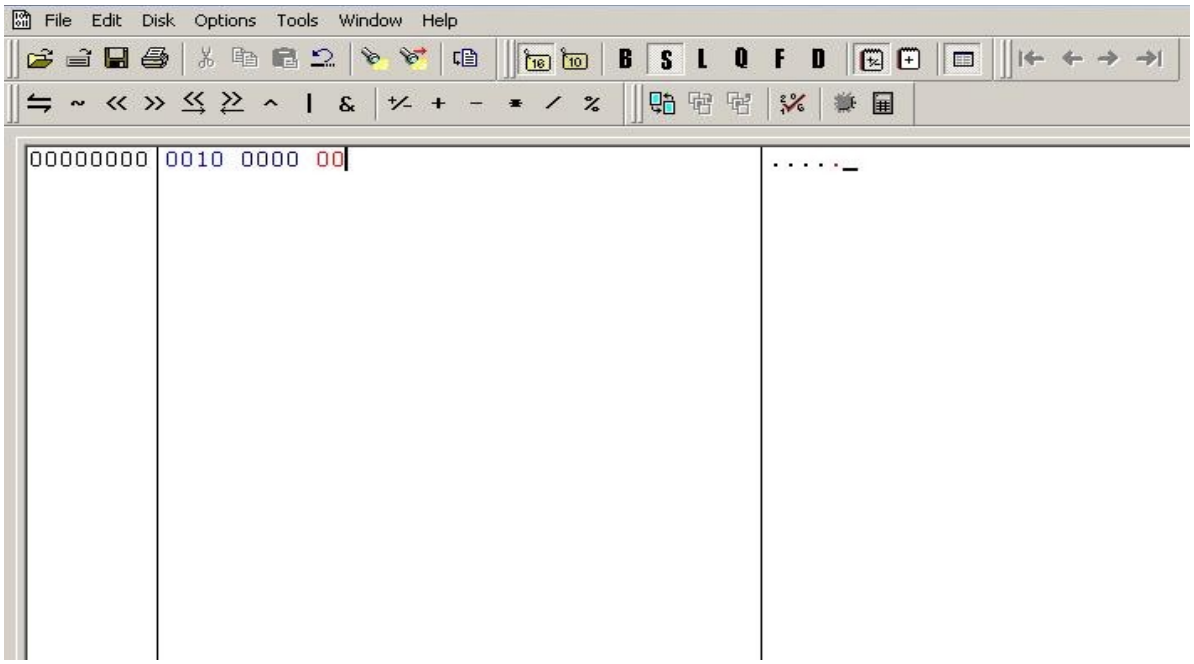
Si può notare come, in totale, è stata trova una corrispondenza di soli 2 byte, il che non è sufficiente per poter scrivere una coppia. Scriveremo quindi alla fine del file in uscita direttamente il byte da cui avevamo cominciato la nostra ricerca (il secondo dei tre 0x00). Il file compresso fino a questo punto sarà:

Figura (9)



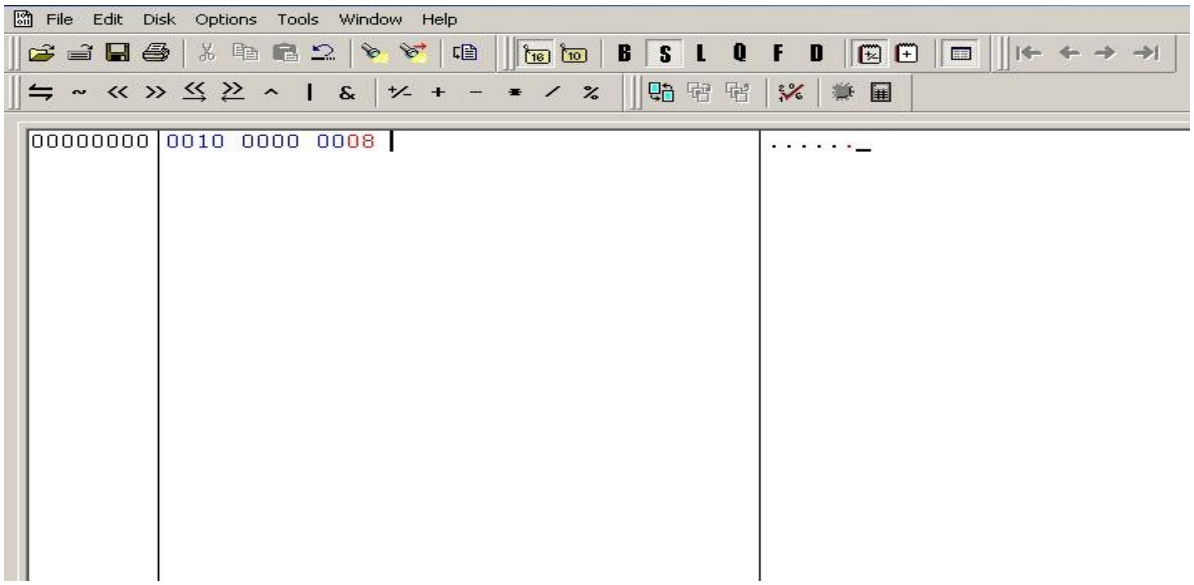
Anche per il prossimo byte (il terzo dei tre 0x00), avremo che la corrispondenza trovata non sarà sufficiente, infatti scriveremo anch'esso direttamente:

Figura (10)



Il prossimo byte da cui partiremo per effettuare la ricerca di una corrispondenza sarà **0x08**, e come possiamo notare non vi è alcuna corrispondenza fra i byte che lo precedono.
Scrivendolo direttamente otterremo:

Figura (11)



Arrivati a questo punto, ci troviamo sul byte **0x00** (quello subito dopo 0x08).
Notiamo subito che vi è una prima corrispondenza:

10 00 00 00 08 00 00 00 2C

Procediamo alla stessa maniera di prima. Spostiamo in avanti le due frecce, fino a quando esse non punteranno più a caratteri uguali. Quindi le frecce scorreranno in avanti fino a fermarsi qui:

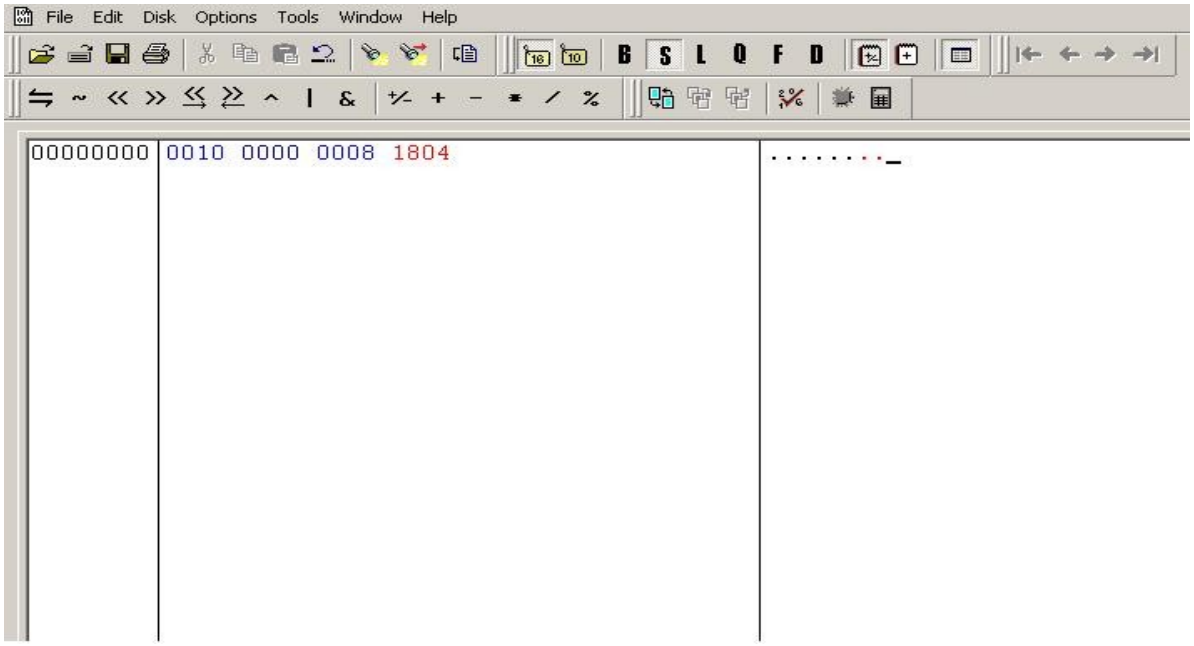
10 00 00 00 08 00 00 00 2C

Notiamo che questa volta le frecce si sono spostate in avanti ben 3 volte, più che sufficiente per poter scrivere una coppia.

Per sapere quali saranno i due byte che rappresenteranno la coppia, procediamo come visto nel capitolo sulla decompressione, ma al contrario.
Sappiamo che il nostro recupero è 3 (numero di spostamenti delle frecce), mentre il salto sarà la distanza fra le due frecce (4 byte ---> salto=4).

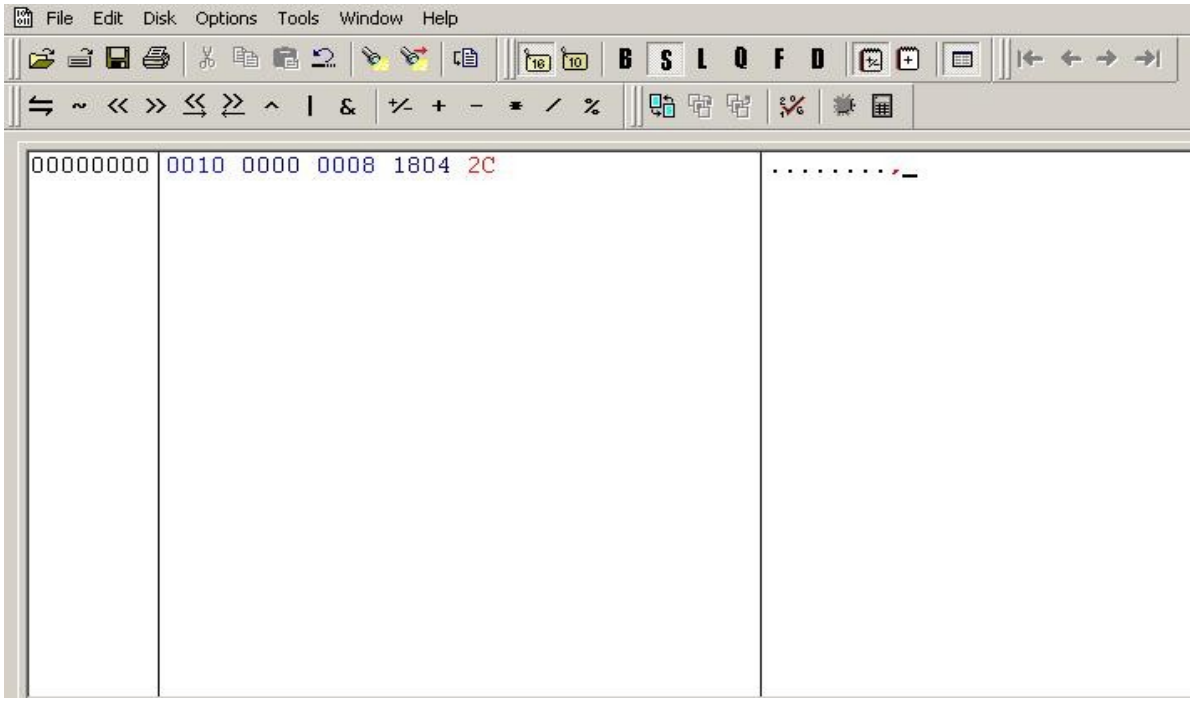
Convertiamo il recupero ed il salto in bit, ottenendo rispettivamente 11 e 100. Come visto prima, questa compressione utilizza 5 bit per memorizzare il recupero e 11 bit per il salto, quindi dobbiamo completare il recupero aggiungendo tre zeri iniziali e il salto aggiungendo 8 zeri iniziali ottenendo 00011 e 00000000100. Mettiamoli insieme ed otteniamo 0001100000000100, che diviso in due byte (8 bit alla volta) risulterà 00011000 00000100. Convertiamoli in esadecimale, ed otterremo: **0x18** e **0x04**.
Ora possiamo scrivere questi due byte nel file in uscita, i quali rappresenteranno la coppia:

Figura (12)



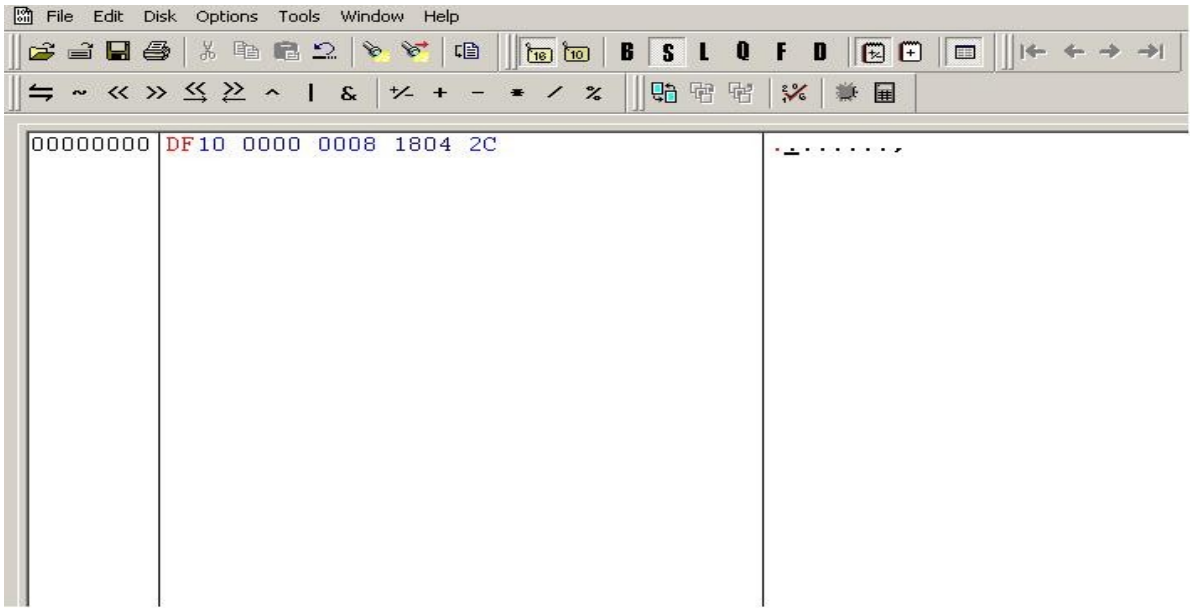
Visto che la coppia non ha fatto altro che prendere il posto della sequenza di cui abbiamo trovato la corrispondenza (0x00 0x00 0x00), il prossimo byte da cui cominciare una nuova ricerca sarà il byte successivo alla sequenza trovata (**0x2C**), il quale non avrà alcuna corrispondenza e verrà scritto direttamente:

Figura (13)



Se ci facciamo caso, fino a questo momento, abbiamo scritto ben 6 byte normali (0x10, 0x00, 0x00, 0x00, 0x08, 0x2C) ed 1 coppia (0x18 0x04), per un totale di 7 unità. Questo ci fa capire che siamo pronti a scrivere il byte riconoscitore, in quanto, in questa particolare compressione, utilizza solo 7 bit, e quindi può fare riferimento a 7 unità alla volta. Il byte riconoscitore relativo ai dati scritti fino ad ora sarà (in bit) 11011111.
Infatti, leggendo i bit da destra verso sinistra, come visto prima, abbiamo 5 bit uguali a 1 uno dopo l'altro, infatti dopo il riconoscitore vediamo scritti 5 byte normali; poi un bit uguale a 0 che indica la coppia 0x18 0x04 e poi il settimo bit che indica un altro byte normale (0x2C). Convertendo quindi il riconoscitore in esadecimale, otteniamo0xDF, che potremo scrivere nello spazio che gli avevamo riservato all'inizio della fase di compressione:

Figura (14)



Completato il primo byte riconoscitore, si potrà procedere con l'aggiunta di uno spazio libero per il prossimo riconoscitore, si effettueranno nuovamente le ricerche e una volta terminate si scriverà il riconoscitore nello spazio lasciato libero in precedenza. Si andrà avanti in questa maniera fino a quando non si raggiungerà la fine del file.

ULTIME PAROLE

Tempo fa, avvalendomi di alcune guide raccimolate per la rete, tentai di capire cosa fossero le compressioni e come funzionassero. Devo dire che, non avendo per niente esperienza a riguardo, trovai quest'impresa alquanto ardua. Infatti molte volte lasciai stare, poi ripresi, lasciai ancora, fino a quando non riuscii, più o meno, a capire come stessero le cose. Ho scritto questa guida nell'intento di far comprendere "al primo colpo" le compressioni, senza doversi sbattere nell'interpretare espressioni a volte incomprensibili, oppure tentare di capire concetti espressi troppo sinteticamente.

Spero di essere riuscito nel mio intento. ;)

P.S. Oltre al mio sorgente scritto in C++ del de/compressore dei file di Soul Blade è presente (vedi cartella "Morpher\sorgenti vb6") un modulo per la decompressione della grafica di DejaVU I&II scritto in Visual Basic da Morpher con allegato un file compresso (sono presenti ulteriori chiarimenti fra i commenti del suddetto modulo) e un De/Compressore (vedi cartella "Morpher\Tools") per i file di Soulblade scritto da lui completo di GUI. Grazie Morpher ;)

5. RINGRAZIAMENTI

Un grazie va a coloro che mi hanno aiutato nel migliorare la guida e nel correggere gli ultimi errori di battitura.
Ringrazio in particolare (in ordine alfabetico):

Gemini, Geo, il mio amico Katzenjammer, Morpher, Sephiroth 1311 e Vash.

That's all Folks!

Phoenix.