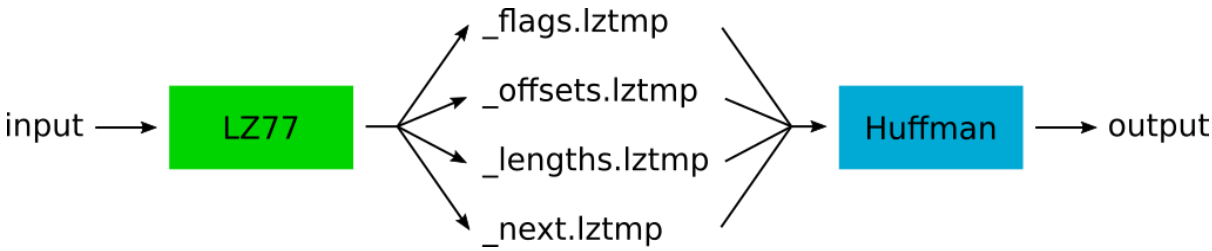
# Progetto compressore

Kuhn - Paoliello

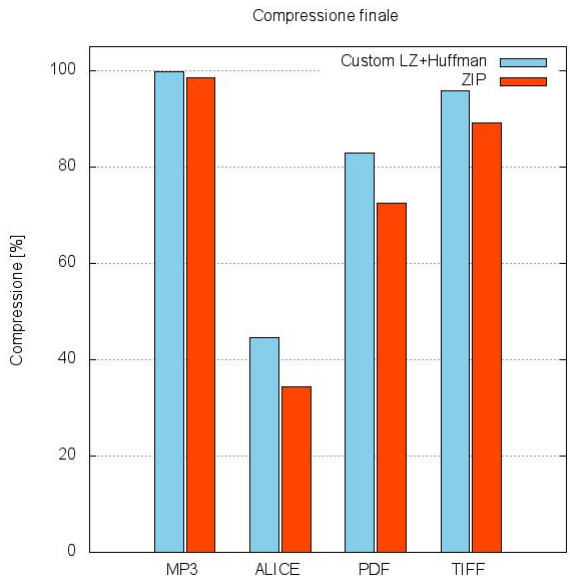
## Descrizione algoritmo

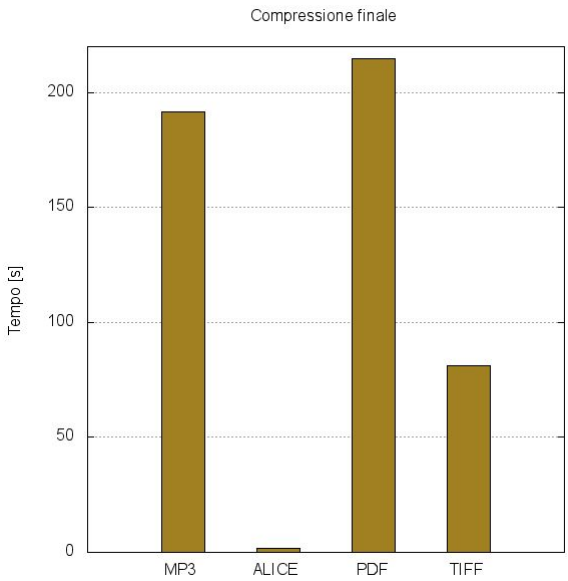
### Descrizione generale

L'algoritmo di compressione finale consiste in una combinazione di una versione ottimizzata di LZ77 combinata con un'implementazione di Huffman a bit variabili.
Il file da comprimere viene inizialmente processato con LZ77, che separa l'output in quattro parti (*_flags.lztmp*, *_offsets.lztmp*, *_lengths.lztmp* e *_next.lztmp*), e ognuna di queste viene ulteriormente compressa con Huffman a bit variabili. L'output finale viene scritto su un unico file.
Di seguito una schematizzazione del processo appena descritto e i dettagli relativi alle implementazioni di Huffman e LZ77.



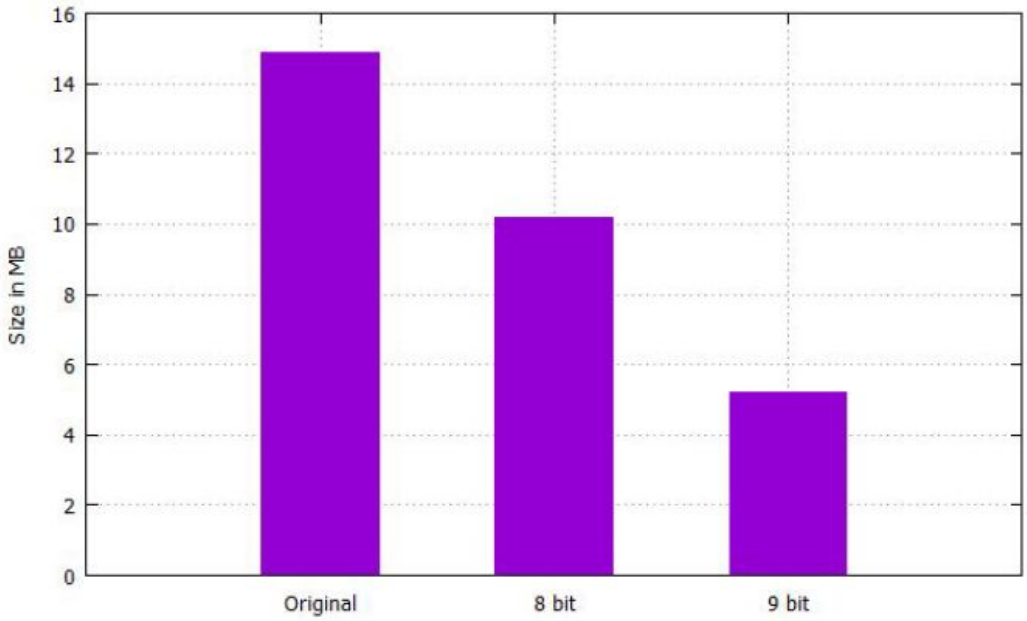

Alcuni rapporti di compressione messi a confronto con quelli di ZIP sono visibili nel seguente grafico. È inoltre visibile il tempo necessario per effettuare la compressione per gli stessi files.
Benché il rapporto di compressione sia buono, è chiaro che il principale limite dell'algoritmo è rappresentato dal tempo di compressione, limite determinato quasi esclusivamente da LZ77.




# Huffman bit variabili

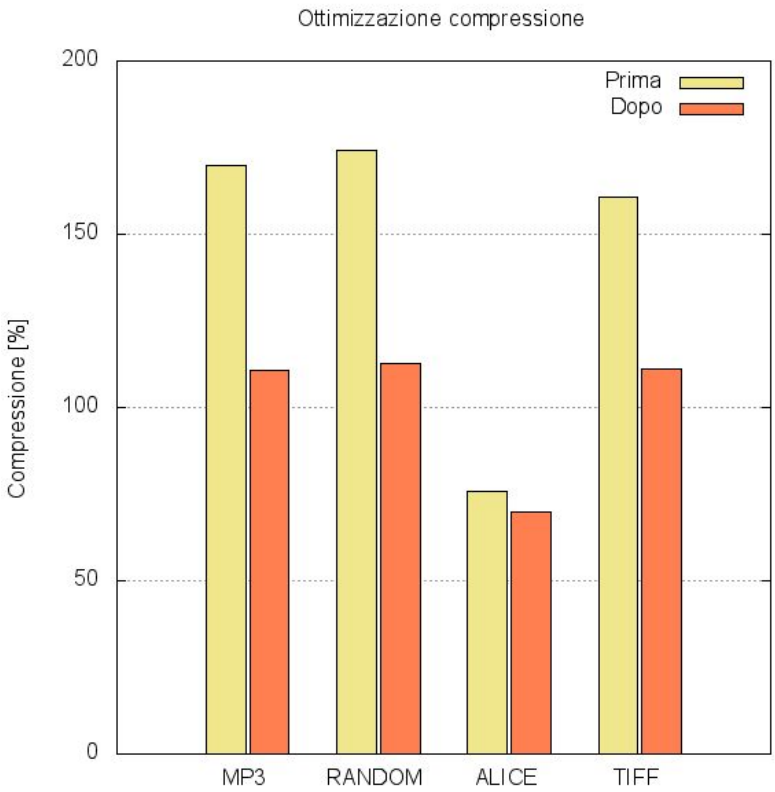
L'idea che Huffman codifichi "caratteri" di un file potrebbe essere fuorviante, è più corretto dire che esso codifica dei simboli, questi possono essere associati ad un numero binario la cui lunghezza non ha limiti in termini di compatibilità con Huffman.
L'idea iniziale era quella di comprimere con Huffman in ugual modo i quattro output di LZ77, ma il fatto che alcuni di questi file contengono dati che sono alla fine del loro calcolo rappresentati con un numero di bit diverso da 8, ha reso necessario al fine di raggiungere una compressione migliore un'implementazione di Huffman che codifica un numero generico di bit per volta. Questo numero generico viene scritto all'inizio di uno dei file di output di LZ77 (file *_flags.lztmp*), e ovviamente anche nel file finale.
Tale generalizzazione ha reso necessario un buffer in lettura il cui contenuto veniva interpretato come un simbolo nel momento in cui superava una certa lunghezza binaria, la dimensione dell'input scelto appunto. È stato inoltre necessario tener conto del fatto che questo buffer potrebbe non raggiungere la soglia di interpretazione nel momento in cui si è raggiunta la fine del file, di conseguenza il buffer viene riempito con degli zeri fino a raggiungimento di un numero intero di byte e successivamente scritto. Il numero di zeri usati è scritto nella parte iniziale del file compresso.
Nel file compresso viene salvata solo la lunghezza canonica delle codifiche, questo è particolarmente vantaggioso con un Huffman generico dato che simboli a n bit vorrebbe dire salvare 2^n simboli differenti. Per ottimizzare ulteriormente la scrittura della mappa è stato scelto di scrivere ogni lunghezza con un numero di bit pari al numero di bit necessari per rappresentare la lunghezza binaria della codifica canonica più lunga.
L'immagine sottostante è uno degli esempi più lampanti che dimostra quanto la compressione migliori se si rispetta l'integrità iniziale del dato, il file in questione era un *_lengths.lztmp* in cui la lunghezza dei pattern matching era codificata con 9 bit.




# LZ77 ottimizzato

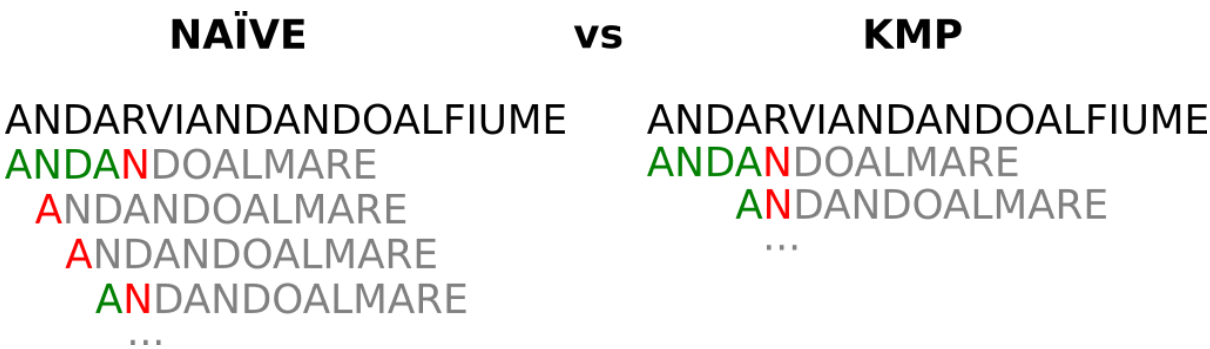
## Flag bit

Una prima idea nasce dall'osservazione che scrivere un match "breve" (di seguito verrà definito quanto) può prendere più spazio che non scrivere direttamente i singoli caratteri del match.
Nell'implementazione dell'algoritmo si è optato per un dizionario di taglia 4kB e un look ahead buffer di 512B, dunque per scrivere l'informazione di offset sono necessari 12bit, mentre per la lunghezza del match 8bit. Scrivere un match su file consuma dunque 12 + 8 + 8 = 28bit e per un match lungo 3 caratteri o meno (<= 24bit) non vale la pena scrivere il match in forma estesa.
Introduciamo quindi un bit in più che segnala se i valori che seguono sono relativi ad un match o ad un singolo carattere.
Il seguente grafico illustra il miglioramento del rapporto di compressione per alcuni tipi di files.

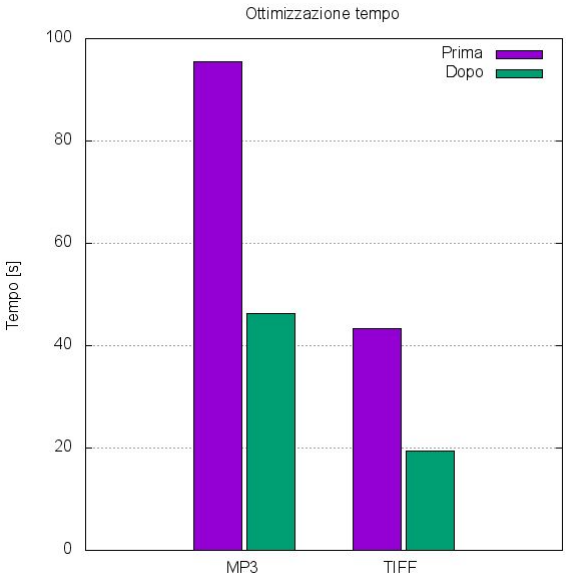


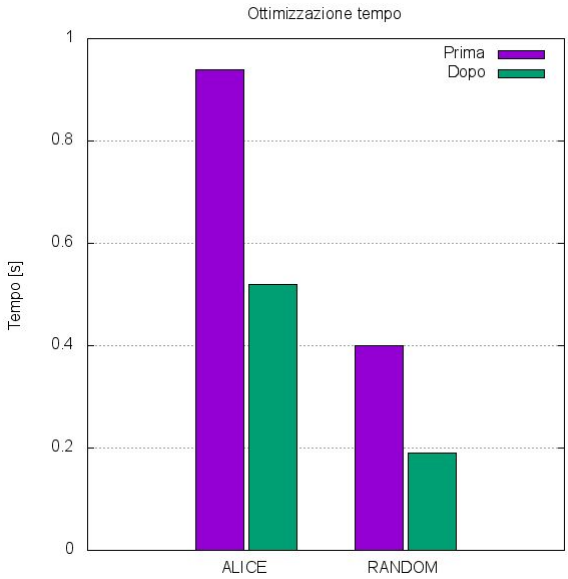

## Ricerca KMP

L'algoritmo Knuth-Morris-Pratt permette di ridurre il tempo di ricerca di una stringa rispetto alla banale ricerca sequenziale, sfruttando alcune informazioni riguardo alla stringa da cercare così da poter saltare alcuni passi avanti quando in presenza di un match parziale.
Nell'esempio illustrato qui di seguito, si ricerca la stringa "ANDANDOALMARE". L'implementazione *naïve* (banale) cerca il match più lungo a partire dalla posizione corrente e quando il match termina, la posizione avanza **sempre** di un solo carattere e la ricerca ricomincia. Secondo la logica KMP invece, il match parziale contiene già sufficiente informazioni per determinare quale sarà la prossima posizione in cui si troverà il prossimo potenziale match, ovvero al prossimo carattere 'A', permettendo così di saltare due iterazioni che sarebbero in ogni caso risultate superflue.




Il miglioramento in termini di tempo computazionale si aggira attorno al 50%, come illustrato dai seguenti grafici.




# Struttura codice sorgente

Il codice sorgente è suddiviso nei seguenti files:

- **run.c** : programma principale che interpreta e verifica i dati in input dalla linea di comando e che lancia i due algoritmi in sequenza, sia nel caso della compressione che della decompressione
- **sf_lz77.c** : contiene tutto il codice relativo all'algoritmo di compressione LZ77
- **sf_lz77d.c** : contiene tutto il codice relativo all'algoritmo di decompressione LZ77
- **compHuff.c** : contiene tutto il codice relativo all'algoritmo di compressione Huffman
- **decHuff.c** : contiene tutto il codice relativo all'algoritmo di decompressione di Huffman
- **testScript.sh** : uno script usato per eseguire compressioni/decompressioni di tutti i files presenti nella cartella testInput e registrare dati realtivi a tempo e rapporto di compressione.

# Strutture utilizzate

## Huffman

Invece di usare un albero, per creare la mappa di Huffman è stato usato un array multidimensionale dinamico definito come doppio puntatore a int. Inizialmente ha un numero di righe pari al numero di simboli diversi letti e due colonne, la prima contiene il valore decimale del simbolo mentre la seconda il valore -1 che serve come delimitatore di fine riga.
L'array contiene i simboli in ordine di frequenza, ogni iterazione le ultime due righe vengono unite appendendo uno '0' alla codifica di tutti i simboli nella riga superiore e un '1' a tutti quelli nella riga inferiore. Per risparmiare memoria viene eseguito un realloc alla riga superiore e un free a quella inferiore.

Per comodità e chiarezza le codifiche e i simboli associati si trovano in un array di struct che ha questi unici due campi. Nel decompressore invece è stato deciso di implementare un albero i cui nodi foglia contengono i simboli originali, il percorso per raggiungere questi nodi rappresenta la codifica compressa.

## LZ77

La finestra a scorrimento (*SlidingWindow*) è stata implementata come buffer circolare, in modo da evitare la necessità di effettuare degli *shift* di tutto il buffer ad ogni iterazione dell'algoritmo.
Per la scrittura su file si è optato per l'uso di un buffer provvisorio di scrittura (*WritingBuffer*), anch'esso circolare, contenente valori *char* '0' e '1' delle informazioni che si intende scrivere su file. Ad ogni chiamata alla funzione di riempimento di questo tipo di buffer, se sono disponibili sufficienti dati da poter scrivere uno o più caratteri sul relativo file di output, ciò viene effettuato convertendo 8 valori consecutivi del buffer in un char e scrivendo quest'ultimo su file (ripetendo la procedura finché il buffer ha dimensione minore di 8). Terminato l'algoritmo di compressione, tutti i buffers di questo tipo vengono svuotati, scrivendo su file gli ultimi dati ed eventualmente appendendo alcuni '0' per completare l'ultimo carattere.

# Compilazione codice sorgente

Per compilare il codice sorgente nell'eseguibile finale chiamato *byteComp*, eseguire il seguente comando (è una sola riga) nella cartella in cui si trovano i files sorgente (src):

```
gcc -o byteComp -std=c99 -O3 -w compHuff.c decHuff.c sf_lz77.c
sf_lz77d.c run.c -lm
```

# Procedure di test effettuate

Abbiamo utilizzato il file *testScript.sh* (presente tra i files della consegna) per verificare che comprimendo e decomprimendo ognuno dei files presenti nella cartella *testInput* non vi fossero divergenze. Un esempio di output è il seguente:

```
kuax@dyloo:~/prog/bC$ ./testScript.sh
20sylKodama.mp3
I  file  testInput/20sylKodama.mp3 e  testOutput/20sylKodama.mp3 sono
identici
32k_ff
I file testInput/32k_ff e testOutput/32k_ff sono identici
32k_random
I file testInput/32k_random e testOutput/32k_random sono identici
alice.txt
I file testInput/alice.txt e testOutput/alice.txt sono identici
empty
```

```
I file testInput/empty e testOutput/empty sono identici
ff_ff_ff
I file testInput/ff_ff_ff e testOutput/ff_ff_ff sono identici
fisica.pdf
I file testInput/fisica.pdf e testOutput/fisica.pdf sono identici
immagine.tiff
I file testInput/immagine.tiff e testOutput/immagine.tiff sono
identici
swiss-flag.bmp
I file testInput/swiss-flag.bmp e testOutput/swiss-flag.bmp sono
identici
kuax@dyloo:~/prog/bC$
```

Per estrarre informazioni riguardo ai tempi e ai rapporti di compressione è stata utilizzata la funzione *testMain* presente nel file *run.c* (commentata nella consegna finale).

# Problemi noti

Per quanto riguarda la parte di lavoro di Huffman, il problema di maggior rilevanza è rappresentato dai file di piccola grandezza, sotto la decina di KB. Questo perchè la mappa di Huffman nonostante sia stata ottimizzata contiene le lunghezze anche dei simboli con frequenza zero, quindi più il file è piccolo più le dimensioni della mappa influenzano negativamente il rapporto con il file finale. Una possibile soluzione è, nel caso di un Huffman a n bit, quella di dedicare 2^n bit per specificare quali simboli ci sono(‘0’ o ‘1’) e quindi scrivere solo la lunghezza dei simboli presenti.