Sommario

Teoria dell'Informazione e Compressione Dati
-Portfolio A.A. 2022/2023-

Domenico Giosuè

Introduzione

Il presente documento nasce con lo scopo di descrivere le implementazioni, e i risultati raggiunti tramite queste, degli Algoritmi proposti dal docente.

Il documento sarà suddiviso in 6 capitoli, ognuno dei quali tratterà un esercizio di portfolio differente.

Cap. 1) Sardinas-Patterson

Il primo esercizio di portfolio vede l'implementazione dell'Algoritmo di Sardinas-Patterson per verificare se un codice risulta essere Univocamente Decodificabile (UD); ovvero, se questo, durante la decodifica di una stringa codificata, non porta ad alcuna ambiguità.

L'Algoritmo procede creando dei sottoinsiemi S_i, generati a partire dal sottoinsieme S_0 di tutto il Codice, i quali sono composti dai soli suffissi generati a partire dalle word dei due sottoinsiemi precedenti a quello corrente.

Non appena non si potranno più generare nuovi sottoinsiemi, si potrà verificare che nessuno suffisso generato corrisponde a una codeword del codice.
Se ciò non accade, allora, il Codice sarà Univocamente Decodificabile.

L'implementazione si basa su due semplici funzioni:

- *create_set_n(code, n)*
    - Funzione ricorsiva che crea l'insieme *n-esimo*

- *create_all_sets(code)*
    - Funzione che costruisce l'insieme unione di tutti gli insiemi generabili

Il tutto viene poi sfruttato da un'unica funzione finale '*sardinas_patterson()*' che verificherà se l'intersezione tra il Codice e l'insieme Unione generato risulta in un insieme vuoto.
In tal caso, verrà determinato che il codice è Univocamente Decodificabile.

Per dettagli implementativi è possibile visionare il codice commentato presente nel file '*SardinasPatterson.py*'

Cap. 2) Huffman

Il secondo esercizio per il portfolio prevede l'implementazione della Codifica di Huffman, la quale genera sempre Codici Ottimi; ovvero, codici con Average Code Length minima.

L'Algoritmo è implementato con l'ausilio di un Albero Binario che verrà costruito, iterativamente, seguendo una politica bottom-up.
Di fatto, verranno creati sempre nuovi nodi a partire da quelli associati ai due simboli della sorgente aventi minore probabilità.

Questo processo verrà ripetuto fino ad arrivare al Nodo Radice.
A questo punto, il codice potrà essere costruito percorrendo l'Albero dal Nodo Radice fino al Simbolo da codificare, concatenando alla codeword un determinato simbolo per ogni ramo attraversato verso il Nodo Foglia.

L'implementazione vede, inoltre, l'utilizzo di una Coda con Priorità per consentire l'estrazione del simbolo della sorgente a cui è associata la probabilità minore.

Possiamo suddividere l'implementazione Python nei seguenti elementi:

- Node
    - Oggetto che rappresenta un Nodo avente diversi attributi, tra cui i Nodi figli.
      Quest'ultimo consentirà la rappresentazione dei due sottoalberi a partire dal Nodo in questione.
      Si nota come è possibile rappresentare l'intero albero costruendo il Nodo Radice

- *prob_calc(text)*
    - Funzione che, a partire dal testo di input, calcola la probabilità di occorrenza dei simboli che lo compongono

- *create_queue(char_prob)*
    - Funzione che costruisce la Coda con Priorità basandosi sulle probabilità di occorrenza dei simboli

- *create_nodes(prio_queue)*
    - Funzione che, a partire dalla Coda con Priorità, si occupa di costruire i Nodi dell'Albero

- *make_codes(node, code, codes)*
    - Funzione che si occupa di costruire la Codeword associata a ogni Simbolo della Sorgente

- *create_huffman(text)*
    - Funzione che costruisce il Codice di Huffman per Simboli presenti nel testo fornito

- *huffman_encoding(text)*
    - Funzione che effettua la Codifica di Huffman percorrendo l'Albero

- *huffman_decoding(encoded_text, tree_root)*
    - Funzione che decodifica un testo codificato con Huffman e che necessita dell'intero Albero utilizzato per costruire il Codice

Cap. 3) Codifica Universale per Interi

Questo capitolo richiede l'implementazione di alcuni Algoritmi di Codifica Universale per numeri Interi.

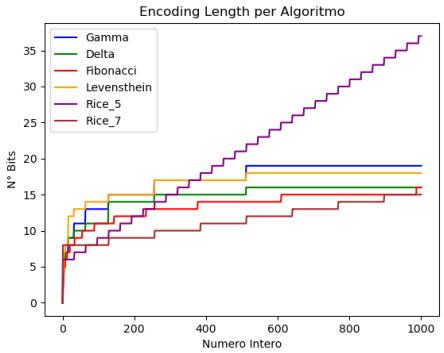
In breve, un Codice Intero Universale non è altro che un Codice Prefisso che associa una codifica binaria a ogni numero intero.

Tralasciando le singole funzioni python utilizzate per l'implementazione, il file *'IntegerEncoding.py'* conterrà:

- Gamma Encoding
- Gamma Decoding
- Delta Encoding
- Delta Decoding
- Fibonacci Encoding
- Fibonacci Decoding
- Levenshtein Encoding
- Levenshtein Decoding
- Rice Encoding
- Rice Decoding

Infine, sono presenti alcuni esperimenti descritti a partire dai corrispettivi Grafici mostrati di seguito…

Esperimenti

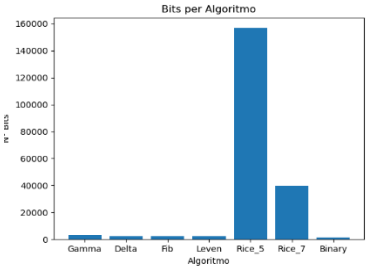


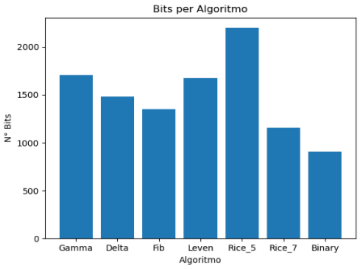
Il Grafico mostra l'andamento del numero di bits utilizzati per codificare i numeri che vanno da '1' a '1000'.

Si può notare come Rice Encoding, per un valore di ***k*** pari a '5', richieda un numero di bits che cresce velocemente al crescere di ***n***.

Diversamente, la Rice Encoding con ***k*** pari a '7' risulta mantenersi a valori bassi sull'asse delle ordinate.

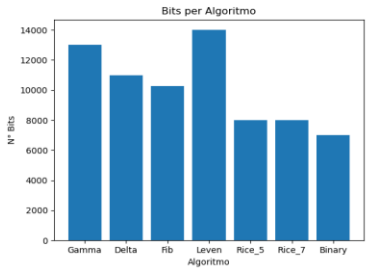
100 interi
$1 \leq x \leq 100000$
Con step di 1010

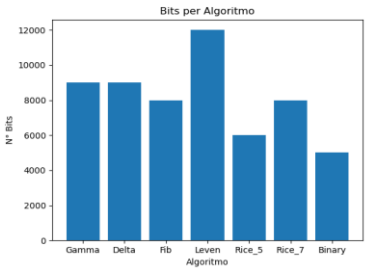




100 interi casuali
$1 \leq x \leq 1000$

1000 interi estratti da una distribuzione normale
$N(\mu = 25, \sigma = 2)$





1000 interi estratti da una distribuzione normale
$N(\mu = 86, \sigma = 3)$

Cap. 4) Codifica su Dizionari

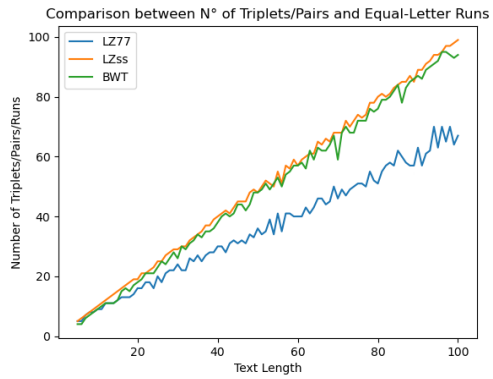
La Codifica basata su Dizionari consente di Codificare dei testi senza la necessità di conoscere la probabilità dei simboli emessi dalla sorgente.

L'esercizio per il portfolio presenta le seguenti implementazioni:

- LZ77 Encoding
- LZ77 Decoding
- LZss Encoding
- LZss Decoding
- Variante di LZ77 Encoding
    - Implementata per essere utilizzata nella ricerca degli String Attractor
- Burrows-Wheeler Transform

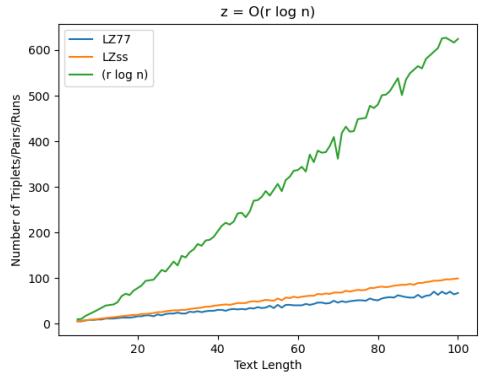
Sono presenti, inoltre, alcune funzioni ausiliarie utilizzate in altri esercizi o utilizzate al fine di portare a termine gli esperimenti mostrati di seguito…

Esperimenti



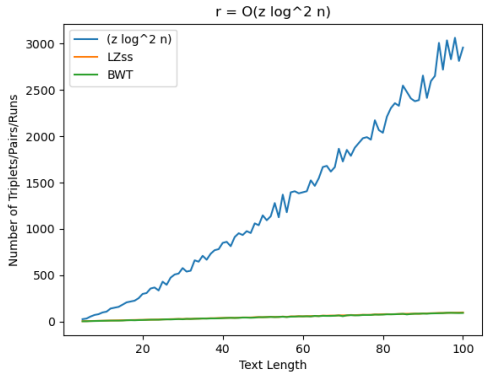
Confronto del numero di Terzine/Coppie con il numero di Run di stessi simboli presenti nell'output della BWT.

Si può notare come il numero di Terzine prodotte da LZ77 tenda a crescere molto più lentamente rispetto agli altri due parametri (Coppie LZss e Equal-Letter Runs).



Dimostrazione dell'esistenza di un limite superiore per il numero ***z*** di Terzine prodotte dall'algoritmo LZ77 (o LZss).

In particolare, il limite superiore è dato da $r \log n$, dove ***r*** è il numero di Equal-Letter Runs prodotto dalla BWT ed ***n*** è la lunghezza del testo.



Dimostrazione dell'esistenza di un limite superiore per il numero di Equal-Letter Runs ***r*** prodotte dalla BWT.

In particolare, il limite superiore è dato da $z \log^2 n$, dove ***z*** è il numero di Terzine prodotte dall'algoritmo LZ77 (o LZss) ed ***n*** è la lunghezza del testo.

Cap. 5) String Attractor

Il quinto esercizio per il portfolio richiede di lavorare con gli String Attractor; ovvero, delle sequenze di posizioni tali per cui tutte le sottostringhe possibili, appartenenti a un dato testo, hanno almeno un'occorrenza che interseca almeno una delle posizioni che compongono l'Attractor.

Questa implementazione è stata svolta solamente in parte; di fatto, il codice Python vede la presenza delle seguenti funzioni:

- *generate_substring(T)*
    - Genera tutte le possibili sottostringhe di un testo ***T***

- *isAttractor(T, Gamma)*
    - Verifica se l'insieme di posizioni ***Gamma*** è uno String Attractor ammissibile per ***T***

- *find_attractor_lz77(T)*
    - Definisce uno String Attractor a partire dalle Terzine generate dall'Algoritmo LZ77
    - Viene sfruttata una variante di LZ77 implementata in *'LZEncoding.py'*

- *find_attractor_bwt(T)*
    - Funzione che è stata suddivisa in due diverse funzioni che operano in modi differenti
    - *find_attractor_bwt_1(T)*
        - Cerca di memorizzare le posizioni del primo simbolo di ogni run prodotta dalla BWT, costruendo così un insieme di posizioni che dovrebbero rappresentare lo String Attractor
        - NOTA: Il funzionamento non è quello attesto
    - *find_attractor_bwt_2(T)*
        - Sfrutta il processo per la ricostruzione di un testo ***T*** a partire da ***BWT(T)*** per identificare le posizioni, nel testo ***T***, dei primi simboli di ogni run prodotta dalla BWT
        - NOTA: Il funzionamento non è quello atteso in quanto non sempre riesce a produrre uno String Attractor

Per quanto riguarda le altre funzionalità, queste non sono state implementate.
Per questo motivo, è stata mia premura strutturare anche l'esercizio alternativo del Portfolio che verrà illustrato nel corso del capitolo successivo.

Cap. 6) RE-PAIR

L'ultimo esercizio del portfolio prevede l'implementazione dell'Algoritmo RE-PAIR per la costruzione di una Grammatica Context-Free in grado di codificare un testo T di input.

Tutto l'esercizio è svolto implementando le seguenti funzionalità:

- *repair_enc(T, lower=False, n_char=0)*
    - Funzione che costruisce una Grammatica Context-Free, a partire dal testo ***T***, codificando quest'ultimo sfruttando i simboli Non Terminali presenti nella Grammatica generata

- *repair_dec(enc_T, dict)*
    - Funzione che ricostruisce il testo ***T*** a partire dalla sua codifica ***enc_T*** e dal dizionario ***dict*** che rappresenta la Grammatica Context-Free utilizzata per la codifica

- *repair_cnf(T)*
    - Funzione che effettua una codifica con RE-PAIR producendo, però, una Grammatica nella Forma Normale di Chomsky (CNF)

TEST

Per poter effettuare i Test opportuni, è stato implementato un menù che consente di spostarsi tra le diverse categorie di esercizi.
È bene precisare che, per ridurre i tempi, non sono stati implementati controlli particolari per evitare eccezioni durante l'utilizzo del codice; dunque, è bene sempre inserire gli input in maniera corretta seguendo le indicazioni mostrate nel menù.

L'intero programma è eseguibile attraverso il file 'Test.py'.

Esempi di Test

Le seguenti immagini mostrano l'interazione con il menù e lo svolgimento di alcuni test d'esempio.

Menù Iniziale

```
Seleziona la categoria da visitare
        1) Sardinas-Patterson
        2) Huffman Encoding
        3) Integer Encoding
        4) LZ Encoding
        5) String Attractor
        6) RE-PAIR
        7) Exit
```

Sardinas-Patterson

```
Inserisci le codewords separate da spazio
0 11 00 01

Codice non UD
```

Huffman Encoding

```
abccddabbcdefab
Il Codice di Huffman generato è il seguente:

{'a': '111', 'b': '10', 'c': '00', 'd': '01', 'e': '1100', 'f': '1101'}


Il testo
'abccddabbcdefab'
viene codificato nel testo seguente:

111100000010111110100001110011011110


La Decodifica del testo codificato
'111100000010111110100001110011011110'
viene decifrata nel testo seguente:

abccddabbcdefab
```

Levenshtein Encoding

```
Inserisci il numero da codificare: 90100
La codifica del numero scelto è: 11111000000000101111111110100
La decodifica è: 90100
```

LZss Encoding

```
Inserisci il testo da codificare: aabbabbaabba


Inserisci la dimensione della Sliding Window (Premi Invio per Default): 12
La codifica è rappresentata come:
[(0, 'a'), (1, 1), (0, 'b'), (1, 1), (3, 3), (7, 5)]
La decodifica è: aabbabbaabba
```

String Attractor from LZ77

```
Inserisci la stringa:
aaabbaaab

Lo String Attractor identificato è il seguente:
[0, 1, 2, 3, 4, 8]
```

RE-PAIR CNF

```
Seleziona la stringa da Codificare con una Grammatica CNF:
aabbbaabbaaababbaa

|A->a

|B->b

|C->AA

|D->BB

|E->CD

La stringa codificata con la suddetta Grammatica CNF è la seguente:
EBECABADC

-----

La corrispettiva Decodifica è la seguente:

aabbbaabbaaababbaa
```