



**UNIVERSITÀ DEGLI STUDI DI TORINO**

DIPARTIMENTO DI INFORMATICA

TESI DI LAUREA TRIENNALE

**Strutture dati compresse:  
implementazione grafica di un Wavelet Tree in Java**

Primo Relatore: Viviana Bono

Secondo Relatore: Ferruccio Damiani

Correlatore: Giorgio Audrito

Candidato: Luca Serena

Anno Accademico 2016/17

# Indice

1.	Algoritmi per la compressione .....	5
1.1.	Entropia .....	5
1.2.	Codifica di Huffman .....	7
1.3.	Codifica aritmetica .....	10
1.4.	Prediction by Partial Matching .....	12
1.5.	Trasformata di Burrows Wheeler .....	14
1.6.	Move to front .....	16
1.7.	Run lenght encoding .....	18
2.	Algoritmi per la compressione .....	19
2.1.	Array di suffissi .....	19
2.2.	Albero di suffissi .....	20
2.3.	FM index .....	22
2.4.	Rank/Select .....	24
3.	Wavelet Tree .....	25
3.1.	Struttura e creazione .....	25
3.2.	Rank .....	27
3.3.	Select .....	29
4.	Implementazione e rappresentazione di un wavelet tree in Java .....	32
4.1.	package wavelet .....	32
4.2.	package graphic .....	35

# Introduzione

Ci si trova in un'epoca in cui lo spazio digitale non è uno dei problemi principali dell'informatica, basti pensare quanto piccole e poco costose siano diventate le schede di memoria. Tuttavia essere in grado di risparmiare oltre il 90% dello spazio per memorizzare un file audio semplicemente tramite algoritmi è qualcosa di estremamente vantaggioso, soprattutto perché minimizzare lo spazio comporta un risparmio nei tempi di accesso, elaborazione e trasferimento dei dati.

Comprimere dati significa ridurre le dimensioni fisiche di un blocco di informazioni. La maggior parte dei documenti presenti sui nostri computer personali, come file testuali, video, musica ed immagini, è immagazzinata con una serie di formati che ottimizzano lo spazio fisico necessario a riprodurre il documento. Questo è possibile grazie ad algoritmi di compressione che modificano i dati in maniera reversibile. Una proprietà necessaria per il funzionamento di queste tecniche è infatti la possibilità di riottenere la sequenza di bit dei dati originali tramite algoritmi di decompressione, che operano in maniera inversa rispetto alle corrispondenti funzioni di compressione.

Le tecniche di compressione sono diverse a seconda del tipo di file di cui si vuole ottimizzare lo spazio: gli algoritmi che si utilizzano per comprimere un video saranno diversi rispetto a quelli necessari per compiere la stessa operazione su documento di word. Ad esempio, in un file di testo non compresso un carattere è rappresentato utilizzando un numero fisso di bit, 8 nella codifica ASCII, 16 in Unicode. L'obiettivo è quello di minimizzare il numero di bit necessari a rappresentare univocamente un simbolo, tenendo anche d'occhio l'efficienza di codifica e decodifica del testo.

La compressione di un file può essere con perdita di dati (lossy) o senza perdita (lossless), a seconda che si verifichi o meno una parziale perdita di informazioni.

L'informazione che viene trasmessa può essere di tre tipi:

- ridondante, ovvero ripetitiva e prevedibile.

- irrilevante, ovvero contenente dettagli che non possiamo apprezzare e la cui eliminazione quindi non influenza il contenuto percettibile del messaggio. L'udito umano, ad esempio, è in grado di rilevare solo i suoni con una frequenza compresa tra i 16/20 Hz ed i 16 000/20 000

Hz. Le frequenze che si trovano al di fuori di questo intervallo costituiscono perciò un'informazione irrilevante.

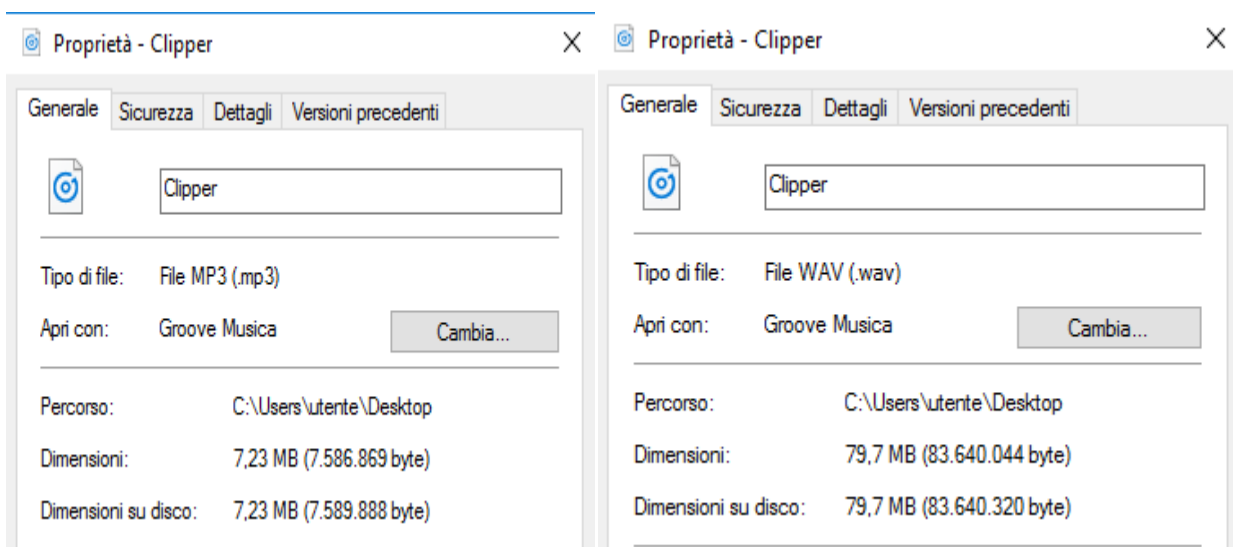
-fondamentale, ovvero costituita da dati che non sono né ridondanti né irrilevanti e che quindi devono essere obbligatoriamente codificati per ricostruire l'oggetto originale.

Nelle tecniche di compressione lossy le informazioni irrilevanti vengono trascurate e quindi, a differenza degli algoritmi lossless, è impossibile una ricostruzione esatta dei dati originali a partire da quelli compressi. Formati come JPEG, mp3 e Flash sono di tipo lossy [5], mentre PNG, RAR e ZIP sono di tipo lossless [6].

La compressione con perdita di dati è molto frequente soprattutto nei documenti audiovisivi, dove sarebbe inutile codificare diverse informazioni non captabili dai sensi umani. I problemi che si affrontano in questo tipo di tecniche non sono quindi di natura esclusivamente algoritmica, ma richiedono anche conoscenze fisiche per poter determinare quali sono i dati irrilevanti.

È possibile verificare quanto sia importante la compressione dei dati con un semplice esperimento. Di solito la musica che ascoltiamo dai nostri computer e cellulari è in formato mp3 e lo spazio occupato per un minuto di canzone ha solitamente un valore nell'ordine di 1Mb.

Si prenda ad esempio in considerazione un file mp3 contenente il pezzo Clipper degli Autechre, della durata di 7 min e 54 sec. Le dimensioni su disco in questo formato sono di appena 7,23 MB.



Trasformando lo stesso file in WAV, un formato audio non compresso di codifica digitale, le dimensioni del file si decuplicano, arrivando a ben 79.7 MB.

# 1. Algoritmi per la compressione

In questa sezione si vedranno alcuni algoritmi che, usati in combinazione, permettono di risparmiare una consistente quantità di spazio per raffigurare univocamente un oggetto. Alcuni di questi permutano semplicemente i dati in modo da ridurre la casualità di un flusso di informazione, altri invece consentono direttamente di rappresentare il documento in un minor numero di bit. Gli algoritmi di compressione in generale si basano sul presupposto che in qualsiasi tipo di file ci siano dei dati ridondanti, superflui o mal disposti.

## 1.1. Entropia

Un concetto correlato con la compressione di dati è quello di entropia. Il concetto di entropia è in comune con altri campi scientifici, come la termodinamica e la meccanica statistica, ed in entrambi i casi si può descrivere come una sorta di “misura del disordine”. In particolare nella teoria dell’informazione è un indicatore dell’incertezza in una sorgente di dati e può essere definita come l’informazione media contenuta in tali messaggi.

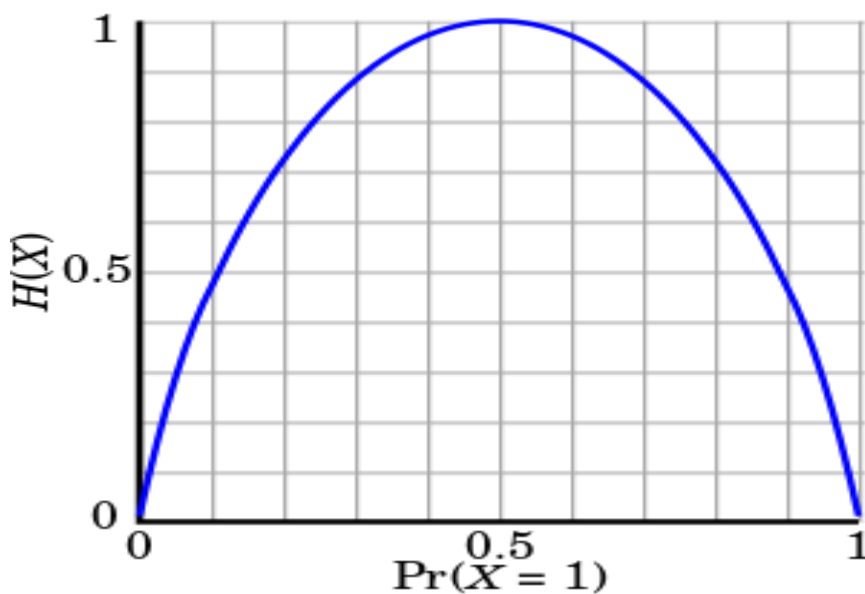
In un testo in italiano alcune lettere, come la ‘w’ o la ‘q’, non sono così frequenti come la ‘a’ o la ‘e’. Considerato che simboli diversi hanno diverse probabilità di comparire, la sequenza di caratteri non è completamente casuale e quindi l’entropia sarà l’entità di misura di questa casualità.

In una stringa composta da caratteri alfanumerici casuali, come “8bXla953nkwPbsl3” ci sarà una maggiore entropia rispetto ad una frase di senso compiuto come “il gatto ci guarda da sotto il tavolo”. A sua volta questa stringa avrà un valore più alto di entropia rispetto ad una sequenza binaria come “11111000001111”.

Sono i simboli meno frequenti a portare una maggiore quantità di informazione. Facendo un’analogia con il nostro linguaggio, in un articolo sull’agricoltura saranno parole come “pomodori”, “semina” e “raccolto” a farci capire qual è l’argomento del documento. Termini come

“che”, “il” o “quindi” al contrario portano poca informazione, e se anche venissero eliminati dal testo probabilmente non si avrebbero comunque problemi a comprendere quale sia l’argomento trattato.

Per capire meglio la relazione tra entropia e probabilità di un evento consideriamo un caso elementare, ovvero una sorgente binaria  $X$  che ha probabilità  $p$  di produrre un 1, e probabilità  $1-p$  di produrre uno 0.



Come si può osservare dal grafico, man mano che la probabilità di produrre un 1 si avvicina ad uno dei due estremi, l’entropia diminuisce. Invece l’entropia raggiunge il massimo valore possibile quando la probabilità di generare l’1 è del 50%, condizione in cui il sorteggio di un 1 piuttosto che di uno 0 è completamente casuale.

La misurazione dell’entropia può essere molto utile perché consente di capire qual è il limite massimo di compressione di una fonte di informazione. Si può dunque dire che questo valore è inversamente proporzionale con il tasso di compressibilità di un messaggio.

## 1.2. Codifica di Huffman

La codifica di Huffman è un algoritmo per la compressione di dati che prende nome dall'omonimo creatore David A. Huffman.

L'obiettivo è abbinare ad ogni carattere un corrispettivo codice binario, assegnando ai simboli che appaiono più frequentemente una sequenza di bit più corta e viceversa.

Si assuma di avere come input un alfabeto  $A = \{a_1, a_2, \dots, a_n\}$  di grandezza  $n$  e l'insieme  $W = \{w_1, w_2, \dots, w_n\}$  composto dai pesi dei vari simboli in cui vale  $w_i = \text{peso } a_i$  con  $1 \leq i \leq n$ . La quantità di informazione  $h$  presente in ogni simbolo  $a_i$  con probabilità non nulla è tale che  $h(a_i) = \log_2 1/w_i$ .

L'entropia, espressa in bit, è la somma ponderata della quantità di informazione di tutti i simboli  $a_i$  con probabilità non nulla.

$$H(A) = \sum_{w_i > 0} w_i h(a_i) = \sum_{w_i > 0} w_i \log_2 \frac{1}{w_i} = - \sum_{w_i > 0} w_i \log_2 w_i.$$

La codifica di Huffman usa un metodo specifico per scegliere la rappresentazione di ciascun carattere, originando un codice prefisso, ovvero una sequenza di bit che rappresenta univocamente un simbolo e che non è un prefisso di nessun'altra sequenza di bit. Questa caratteristica è fondamentale perché consente di capire senza ambiguità in che posizione finisce la rappresentazione di un carattere all'interno della successione di bit, anche se i vari codici hanno una lunghezza differente.

La tecnica utilizzata per ottenere questa struttura dati è l'algoritmo di Huffman, che consiste nella creazione di un albero binario a partire da una lista di nodi.

Nella lista inizialmente vengono inseriti tutti i nodi foglia, che sono contraddistinti da due elementi: il simbolo ed il peso, un valore che sarà proporzionale alla frequenza con cui si trova il simbolo. Ci saranno tante foglie quanto è il valore della cardinalità dell'alfabeto in considerazione. Il peso dei nodi interni corrisponderà alla somma dei pesi dei nodi figli.

Il processo di costruzione dell'albero comincia creando un nodo interno che unisce le due foglie con peso minore. Questo nuovo nodo avrà quindi come figli le due foglie e come peso la somma del peso dei due figli. A questo punto si eliminano i figli dalla lista, e vi si inserisce il nuovo nodo creato. Questo procedimento si ripete fino a che rimane un solo nodo nella lista, che diventerà la radice dell'albero.

In generale si può così riassumere il processo di creazione:

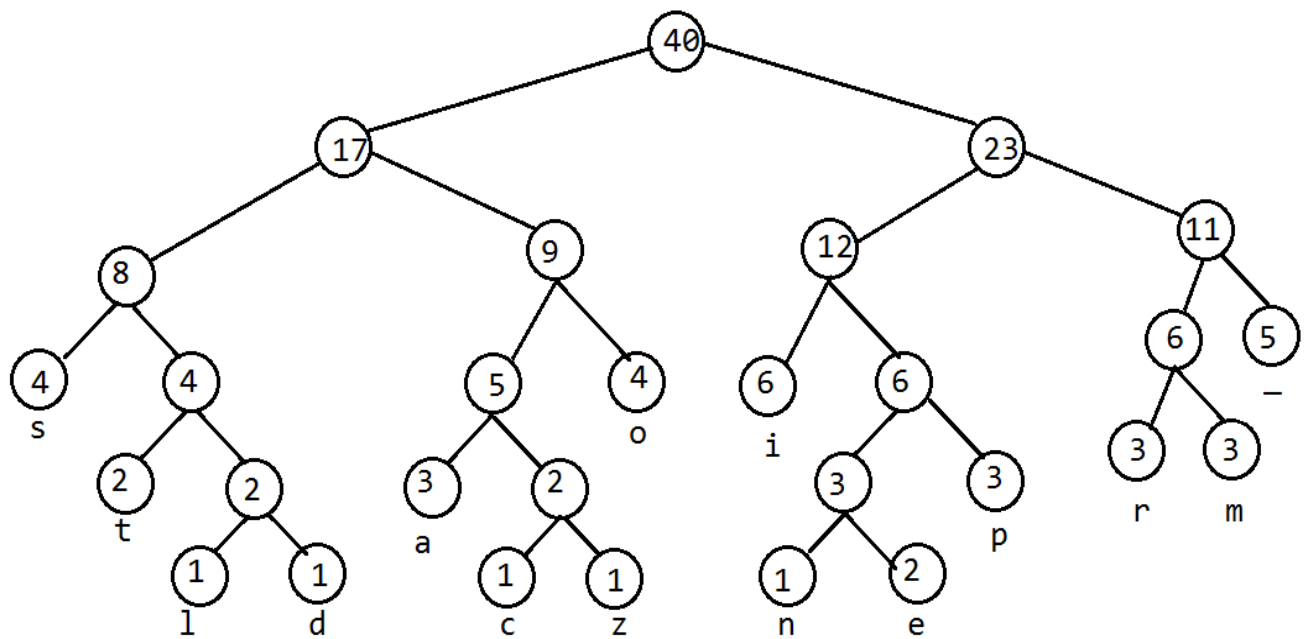
- 1) Creare un nodo foglia per ogni simbolo, associando il peso in base alla frequenza di apparizione e inserirlo in una lista ordinata in modo crescente.
- 2) Fino a che si hanno nodi nella lista:
  - Eliminare i due nodi con peso minore.
  - Creare un nodo interno che colleghi i nodi precedenti, assegnandogli come peso la somma dei pesi dei figli.
  - Inserire il nuovo nodo nella giusta posizione della lista.
- 3) Il nodo che rimane diventa la radice dell'albero.

Ecco ora un esempio con la stringa "comprimendo\_il\_testo\_si\_risparmia\_spazio"

Carattere	Frequenza	Codice
I	6	00
_	5	100
O	4	1100
S	4	1110
M	3	0100
A	3	0101
R	3	1010
P	3	1011
T	2	0110
E	2	0111
Z	1	11010
N	1	110110
C	1	110111
D	1	111100
L	1	111101

Una volta ultimata la tabella si può procedere alla creazione dell'albero.





Si noti che numero delle foglie, 15, corrisponde alla cardinalità dell'alfabeto e che il peso della radice ha lo stesso valore della lunghezza complessiva della stringa.

La lunghezza necessaria per rappresentare la fonte in input equivale all'entropia con la possibile aggiunta di un bit per ogni simbolo compresso.

Per la decodifica di ogni simbolo bisogna attraversare l'albero da radice a foglia, operazione che richiede mediamente la visita di  $O(\log n)$  nodi. La complessità temporale complessiva sarà dunque di  $O(k \log n)$ , attribuendo a  $k$  il valore della lunghezza totale del testo in considerazione. La complessità spaziale invece è  $O(k)$  per il testo codificato ed  $O(n)$  per l'albero, avendo questo una dimensione che è direttamente proporzionale al numero di caratteri di cui è costituito l'alfabeto da rappresentare.

## 1.3. Codifica aritmetica

La codifica aritmetica è una forma di codifica entropica utilizzata nella compressione di dati lossless. Se normalmente una stringa si rappresenta con un numero fisso di bit per carattere, con la codifica aritmetica i caratteri che appaiono più frequentemente sono rappresentati da un numero minore di bit rispetto a quelli che ricorrono in maniera più sporadica. La codifica aritmetica si differenzia dalle altre forme di codifica entropica perché invece di suddividere l'input in simboli e sostituire ognuno di essi con un codice univoco, codifica l'intero messaggio in un solo numero decimale  $S$  compreso tra 0 e 1.

L'uso di queste tecniche in campo commerciale è però poco diffuso perché molte delle implementazioni della codifica a partire dagli anni 80 sono state protette da brevetti molto costosi e restrittivi [8]. Le immagini di tipo JPEG, ad esempio, possono usare internamente sia la codifica di Huffman che quella aritmetica. Quest'ultima è però quasi inutilizzata perché, anche se molti dei brevetti sono ormai scaduti, ci sono ancora pochi programmi e sistemi operativi che supportano automaticamente questa tecnica, che pure consentirebbe di risparmiare quasi il 15% dello spazio rispetto alla codifica di Huffman.

Il numero di bit usati dalla codifica aritmetica su una fonte di input  $S$  di lunghezza  $n$  è al massimo  $nH_0 + 2$ , dove  $H_0$  rappresenta l'entropia di ordine 0 di  $S$ . Il suo differenziale rispetto all'entropia è dunque costante, mentre nella codifica di Huffman questo valore dipende dalla lunghezza del testo. Di qui scaturisce la differenza di compressibilità tra i due algoritmi.

Data una stringa con alfabeto  $|\Sigma|=n$  si suddivide quindi l'intervallo  $[0,1)$  in  $n$  sottointervalli, la cui ampiezza corrisponde alla stima della probabilità che il carattere corrente sia uguale al simbolo corrispondente all'intervallo. Il numero  $S$  cadrà in uno degli  $n$  intervalli permettendo così di decodificare il primo carattere. Terminato il primo passo si suddividerà l'intervallo preso precedentemente in considerazione in altri  $n$  sottointervalli utilizzando il modello già usato in precedenza. L'algoritmo continuerà ricorsivamente in questa maniera. Per sapere quando terminare la ricorsione è necessario o conoscere in anticipo il numero totale di caratteri della stringa o utilizzare un ulteriore intervallo che denoti il simbolo di *end of file*.

Si prenda ad esempio la decodifica del numero binario 0.0011 (0.1875 in decimale ) con i seguenti intervalli di probabilità  $A = 0.3$ ,  $B = 0.5$ ,  $C = 0.2$ . Supponiamo che la stringa abbia la lunghezza pari a 4 caratteri.

Intervallo corrente	A	B	C	Output
[0 , 1)	[0 , 0.3)	[0.3 , 0.8)	[0.8 , 1)	A
[0 , 0.3)	[0 , 0.09)	[0.09 , 0.24)	[0.24 , 0.3)	B
[0.09 , 0.24)	[0.09 , 0.135)	[0.135, 0.21)	[0.21 , 0.24)	B
[0.135, 0.21)	[0.135 , 0.1575)	[0.1575 , 0.195)	[0.195 , 0.21	B

Sarà quindi sufficiente scrivere il numero binario 0.0011 per codificare la stringa ABBB. In generale, in ogni passo del processo di decodifica il cifrario deve considerare solamente tre elementi:

- Il seguente simbolo da cifrare
- L'intervallo attuale, sempre impostato a  $[0 , 1)$  ma aggiornato ad ogni passo
- Le probabilità che il modello dà ad ognuno dei diversi simboli che è possibile incontrare

Il cifrario divide l'intervallo attuale in sottointervalli, ognuno dei quali rappresentato da una frazione dell'intervallo corrente proporzionale alla probabilità del simbolo nel contesto.

Nell'esempio precedentemente riportato, 0.1875 è racchiuso al primo passaggio nel primo intervallo (quello con il 30% di probabilità) e nei restanti tre passaggi dal secondo intervallo, quello con probabilità maggiore. In questo caso ovviamente si suppone di conoscere all'inizio la lunghezza del testo.

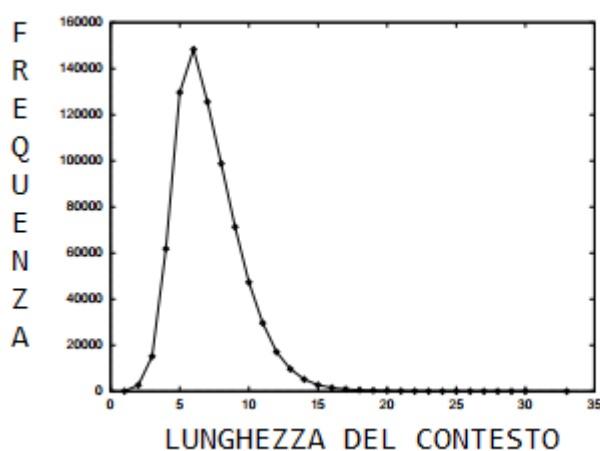
## 1.4. Prediction by Partial Matching

Prediction by Partial Matching (PPM) è un algoritmo adattivo di compressione dei dati basato su un modello di previsione statistica. Inventato da John G. Cleary e Ian H. Witten nel 1984, l'algoritmo iniziale fu oggetto di numerose modifiche e miglioramenti e oggi fa parte degli standard della compressione lossless.

I modelli PPM eseguono le loro predizioni in base al contesto, ovvero fanno un'analisi su un flusso di simboli non compressi che sono già stati tradotti in precedenza. L'algoritmo analizza più modelli, ognuno caratterizzato da un ordine  $k$ , in cui  $k$  rappresenta il numero di caratteri consecutivi presi precedentemente in considerazione.

Questa tecnica consente di migliorare l'efficienza di compressione, che inizialmente è caratterizzata da un'entropia di ordine 0. Si può passare quindi ad un'entropia di ordine  $k$ , chiaramente più precisa perché dipendente dalla probabilità di comparire non di un carattere, bensì di un blocco di  $k$  caratteri.

Se  $k = 0$ , il valore abbinato ai simboli sarà dato esclusivamente dalla sua frequenza del testo. Il numero massimo di ordini a cui si può arrivare è stabilito empiricamente prima di iniziare l'algoritmo, e di solito non si supera il limite di 5, che - come si evince dal grafico sottostante - è di solito il valore massimo più efficiente.



Ai vari simboli è assegnato un valore numerico, che sarà direttamente proporzionale alla frequenza con cui tale simbolo appare in un determinato contesto. Quando si analizza il carattere successivo *next*, ci sono due possibilità:

- *next* è conosciuto, in questo caso si aggiorna solamente la probabilità di incontrare *next* nel dato contesto.
- *next* non è mai apparso in precedenza. Occorre quindi scrivere nella tabella delle probabilità il cosiddetto simbolo di fuga, che permette di regredire al sistema di ordine minore. Questa operazione, se necessario, viene ripetuta fino ad arrivare al modello di ordine 0, dove il simbolo viene codificato senza alcuna rielaborazione.

Ecco ora un esempio di come funziona il Prediction by Partial Matching dopo aver elaborato la stringa BANANA\_BANALE.

Ordine k = 2	Ordine k = 1	Ordine k = 0
BA → <i>esc</i> = 1/3 → N = 2/3  AN → <i>esc</i> = 1/3 → A = 2/3  NA → <i>esc</i> = 3/6 → N = 1/6 → _ = 1/6 → L = 1/6  A_ → <i>esc</i> = 1/2 → B = 1/2  _B → <i>esc</i> = 1/2 → A = 1/2  BA → <i>esc</i> = 1/3 → N = 2/3  AL → <i>esc</i> = 1/2 → E = 1/2	B → <i>esc</i> = 1/3 → A = 2/3  A → <i>esc</i> = 3/8 → N = 3/8 → _ = 1/8 → L = 1/8  N → <i>esc</i> = 1/4 → A = 3/4  _ → <i>esc</i> = 1/2 → B = 1/2  L → <i>esc</i> = 1/2 → E = 1/2	→ <i>esc</i> = 6/19  → B = 2/19  → N = 3/19  → A = 5/19  → _ = 1/19  → L = 1/19  → E = 1/19

L'ordine massimo è stato impostato a 2 e si è indicato con "*esc*" il simbolo di fuga (escape character).

## 1.5. Trasformata di Burrows-Wheeler

La trasformata di Burrows-Wheeler (BWT) è un algoritmo usato nelle tecniche di compressione. La sua finalità non è direttamente quella di comprimere i dati, ma di permutare in maniera reversibile le sequenze di bit, in modo tale che altri algoritmi di compressione funzionino in modo più efficace [2]. Si noti quindi che questa trasformata non riduce l'entropia del messaggio. L'output della BWT non è altro che una permutazione della stringa originaria e potrà contenere lunghe sequenze di caratteri uguali posizionati adiacentemente. Se la stringa originale contiene molte sottostringhe che appaiono frequentemente allora la stringa, dopo la trasformazione, avrà diverse posizioni nelle quali lo stesso carattere sarà ripetuto varie volte di fila. Per ottenere la permutazione desiderata di una stringa è sufficiente eseguire tutte le possibili permutazioni del testo e ordinarle in una colonna in ordine alfabetico. L'unione dell'ultimo carattere di tutte le stringhe presenti nella colonna costituisce l'output.

Ecco un esempio di trasformazione con la parola MACACO\$ (viene usato \$ come simbolo di fine della parola):

Input	Tutte le permutazioni	Permutazioni ordinate	Output
MACACO\$	MACACO\$ ACACO\$M CACO\$MA ACO\$MAC CO\$MACA O\$MACAC \$MACACO	ACACO\$M ACO\$MAC CACO\$MA CO\$MACA MACACO\$ O\$MACAC \$MACACO	MCAA\$CO

A questo punto entra in azione una caratteristica fondamentale della BWT: la reversibilità. Se infatti l'unico scopo dell'algoritmo fosse permutare la stringa originale in modo da comprimere i dati in modo più semplice, basterebbe allora ordinare i caratteri in ordine alfabetico. Con la trasformata di Burrows-Wheeler invece è possibile ricostruire l'input iniziale a partire solamente dai caratteri dell'ultima colonna.

Per compiere questa operazione su una stringa  $s$  di lunghezza  $n$  è innanzitutto necessario creare una tabella vuota. Poi si inserisce  $s$  verticalmente nella tabella (nel senso che si aggiunge un carattere a ognuna delle  $n$  righe della colonna) e si ordina la colonna in senso alfabetico crescente. Questa operazione si ripete  $n$  volte, ovvero fino a quando si ottengono  $n$  stringhe di lunghezza  $n$ . A questo punto si cerca la stringa che ha come ultimo carattere il simbolo di *end of file* e tale stringa costituirà il valore di ritorno della funzione.

Input:	A	AC	ACA	ACAC	ACACO	ACACO\$	ACACO\$M	Output:
	A	AC	ACO	ACO\$	ACO\$M	ACO\$MA	ACO\$MAC	
	C	CA	CAC	CACO	CACO\$	CACO\$M	CACO\$MA	
MCAA\$CO	C	CO	CO\$	CO\$M	CO\$MA	CO\$MAC	CO\$MACA	MACACO\$
	M	MA	MAC	MACA	MACAC	MACACO	MACACO\$	
	O	O\$	O\$M	O\$MA	O\$MAC	O\$MACA	O\$MACAC	
	\$	\$M	\$MA	\$MAC	\$MACA	\$MACAC	\$MACACO	

Si possono fare delle ottimizzazioni a questo algoritmo, in modo da farlo funzionare più efficientemente. Nella BWT non c'è necessità di rappresentare la tabella né nel processo di codifica né in quello di decodifica. Nella codifica ogni linea della tabella può essere rappresentata attraverso un unico puntatore tra stringhe e si può realizzare l'ordinamento mediante gli indici.

## 1.6. Move To Front

Move to front (MTF) è una trasformata usata per diminuire l'entropia di un input, prima che esso venga codificato.

In questo algoritmo, pubblicato per la prima volta da Ryabko Boris Yakovlevich nel 1980, i caratteri dei simboli in input sono sostituiti da indici. Inizialmente si crea una lista contenente i simboli dell'alfabeto del testo. Ad ogni passaggio si sostituisce il carattere in input con il relativo indice della lista e successivamente si modifica l'ordine dell'alfabeto spostando il simbolo in considerazione nella prima posizione. In questo modo i caratteri che sono apparsi di recente saranno nelle prime posizioni dell'alfabeto, quindi anche i simboli che occorrono più frequentemente dovrebbero trovarsi all'inizio della lista. È chiaro che se il testo è costituito da caratteri alfanumerici casuali non vi è alcuna riduzione dell'entropia e l'unico effetto della trasformata è un aumento dell'overhead complessivo.

Anche nei testi di senso compiuto un uso non adeguato di questo algoritmo può essere inutile o addirittura dannoso. Ad esempio, nel celebre monologo di Amleto sono stati calcolati 7033 bit di entropia. Eseguendo direttamente la trasformata, l'entropia aumenta a 7807 bit, più del 10% rispetto a prima [9]. Applicando invece prima la trasformata di Burrows-Wheeler e poi MTF, l'entropia si riduce a 6187 bit. Questo algoritmo infatti si combina particolarmente bene con BWT, essendo questa trasformata in grado di permutare il messaggio in input in modo che abbia diversi caratteri uguali posizionati adiacentemente.

Ecco ora un esempio con la stringa "macaco\_cattivo":



Simbolo in input	Indice	Alfabeto	Sequenza di indici
M	3	{a, c, i, m, o, t, v, _}	3
A	1	{m, a, c, i, o, t, v, _}	31
C	2	{a, m, c, i, o, t, v, _}	312
A	2	{c, a, m, i, o, t, v, _}	3122
C	1	{a, c, m, i, o, t, v, _}	31221
O	4	{c, a, m, i, o, t, v, _}	312214
_	7	{o, c, a, m, i, t, v, _}	3122147
C	2	{_, o, c, a, m, i, t, v}	31221472
A	3	{c, _, o, a, m, i, t, v}	312214723
T	6	{a, c, _, o, m, i, t, v}	3122147236
T	0	{t, a, c, _, o, m, i, v}	31221472360
I	6	{t, a, c, _, o, m, i, v}	312214723606
V	7	{i, t, a, c, _, o, m, v}	3122147236067
O	5	{v, i, a, t, c, _, o, m}	31221472360675
\	\	{o, v, i, a, t, c, _, m}	<b>31221472360675</b>

L'output della trasformata è 31221472360675.

MTF è facilmente reversibile. Per riottenere la stringa originale dalla sequenza di interi è sufficiente ripetere in modo inverso le operazioni, a partire dall'output e dall'alfabeto finale. La stringa verrà riprodotta recuperando carattere per carattere, cominciando dall'ultimo simbolo fino ad arrivare al primo.

Ad ogni passo il carattere recuperato sarà il primo nell'alfabeto. Dopodiché si toglie l'ultimo indice dalla sequenza di interi e si modifica l'ordine dell'alfabeto facendo sì che il simbolo recuperato abbia tale indice all'interno della lista. Ripetendo questa operazione fino che ci sono interi nella lista si ottiene la stringa originale.

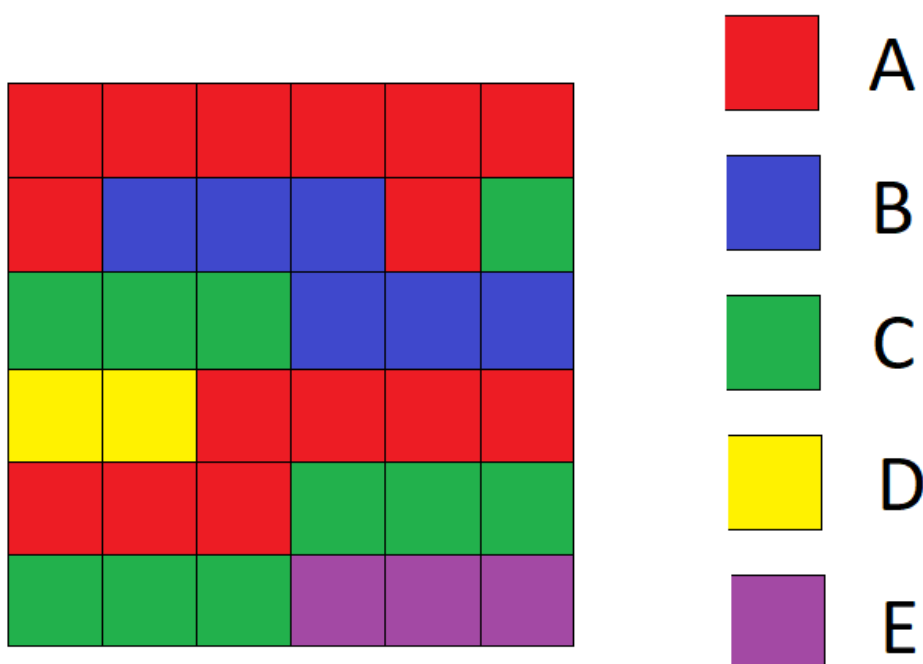
La complessità temporale di questo algoritmo è nel peggior caso  $O(nk)$ , dove si intende con  $n$  la lunghezza complessiva del testo e con  $k$  la cardinalità dell'alfabeto.

## 1.7. Run-length encoding

Run-length encoding (RLE) è un algoritmo lossless di compressione in cui sequenze di dati equivalenti sono codificate con un valore unico, unito ad un intero che rappresenta quante volte questo si ripete consecutivamente. È usato principalmente per contenuti audiovisivi e immagini ed è particolarmente efficiente nelle icone, dove capita sovente di trovare vari pixel adiacenti con lo stesso colore.

L'uso di questo algoritmo non è affatto recente, già a partire dal 1967 infatti veniva utilizzato per la trasmissione di segnali televisivi [10].

Prendiamo come esempio la stringa `AAAAAABBBACCCBBBDDAAAAAACCCCEEE` in cui le diverse lettere maiuscole rappresentano i colori dei pixel di un'immagine 6x6.



La stringa sarà codificata come `7A3B1A4C3B2D7A6C3E`

Gli algoritmi illustrati in precedenza vengono sovente combinati. Ad esempio in bzip2, un formato molto utilizzato in cui i dati vengono compressi, si applicano in sequenza al messaggio iniziale RLE, BWT, MTF e nuovamente RLE. L'output di queste trasformazioni viene poi sottoposto alla codifica di Huffman.

## 2. Strutture dati per la compressione

In questa sezione si vedranno alcune strutture dati per la compressione di un testo come l'array di suffissi, l'albero di suffissi e l'FM-index. Una delle caratteristiche più importanti di queste strutture è la possibilità di eseguire operazioni come la ricerca di una sottostringa senza dover ricorrere alla decompressione della stringa. Questo è possibile grazie a funzioni che, operando su un codice binario, consentono di contare il numero di occorrenze di un carattere (Rank) e di individuarne la posizione (Select).

### 2.1. Array di suffissi

Un array di suffissi è l'array ordinato di tutti i suffissi di una stringa. Sia  $S = S_1, S_2, \dots, S_n$  una stringa di lunghezza  $n$ , e  $S[i,j]$  la sottostringa di  $S$  compresa tra la  $i$ -esima e la  $j$ -esima posizione di  $S$ .

L'array dei suffissi  $A$  della stringa  $S$  è costituito da un array di interi, che forniscono le posizioni iniziali dei suffissi di  $S$  in ordine alfabetico. Quindi  $A[i]$  contiene la posizione iniziale dell' $i$ -esimo suffisso più piccolo di  $S$  e perciò  $\forall i, 1 \leq i \leq n, S[A[i-1], n] < S[A[i], n]$ .

Consideriamo un esempio con  $S = \text{MACACO}\$$

Suffissi	Indice	Suffissi ordinati	Indice
MACACO\$	1	ACACO\$	2
ACACO\$	2	ACO\$	4
CACO\$	3	CACO\$	3
ACO\$	4	CO\$	5
CO\$	5	MACACO\$	1
O\$	6	O\$	6
\$	7	\$	7

L'array di suffissi A contiene le posizioni iniziali di tutti i suffissi ordinati.

<b>i</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>
<b>A[i]</b>	2	4	3	5	1	6	7

Ad esempio A[2] contiene il valore intero 4, quindi si riferisce al suffisso che inizia in posizione 4 all'interno di S, ovvero la sottostringa ACO\$.

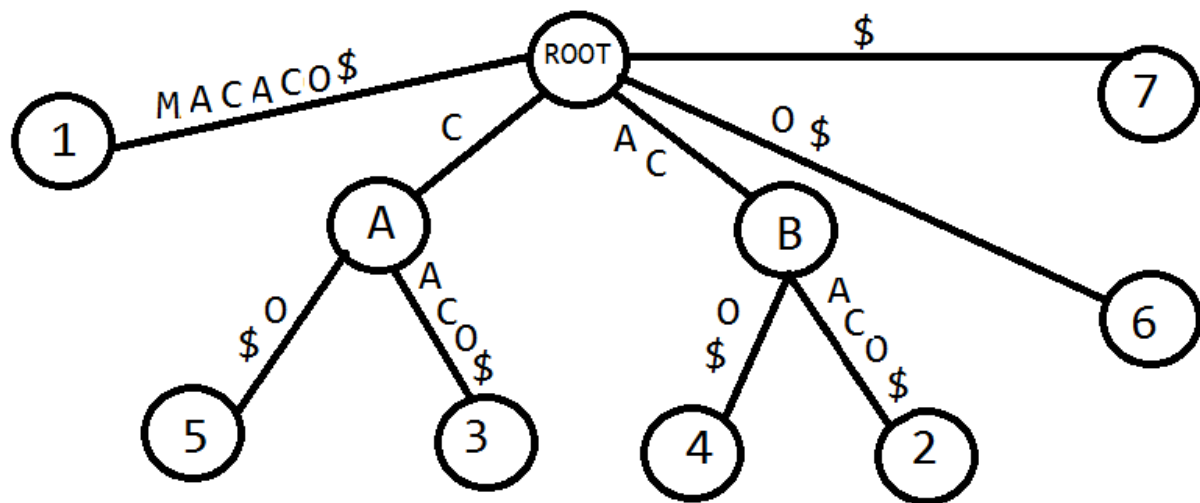
## 2.2. Albero di suffissi

L'albero di suffissi è una struttura particolarmente efficiente nella ricerca di una sottostringa, in quanto è in grado di risolvere questa operazione in tempo lineare. Infatti, dato un messaggio di lunghezza n, è necessario un tempo  $O(n)$  per la costruzione dell'albero ed un tempo  $O(k)$  per la ricerca di una stringa di lunghezza k all'interno dell'albero.

L'albero per poter svolgere le funzionalità descritte deve possedere le seguenti proprietà:

- devono esserci esattamente n foglie, che vengono numerate con i relativi valori ottenuti dopo aver ordinato l'array di suffissi.
- ogni collegamento fra nodi (arco) è etichettato con una sottostringa del messaggio.
- ogni nodo (eccetto la radice) ha sempre più di un figlio e gli archi che partono verso questi non possono iniziare con lo stesso carattere.
- la concatenazione delle stringhe con cui sono etichettati gli archi lungo il cammino che parte dalla radice ed arriva alla foglia i corrisponde al suffisso del messaggio che si trova in posizione i all'interno dell'array di suffissi.

Anche in questo caso uso la stringa MACACO\$ come esempio di costruzione della struttura dati.



Si noti che le sottostringhe che hanno un carattere iniziale differente da ogni altro suffisso (in questo caso la 1, la 6 e la 7), sono collegate alla radice da un solo arco. Per trovare la sottostringa T una volta costruito l'albero A è necessario percorrere A dalla radice, confrontando i caratteri di T con i possibili percorsi. Se si incontra un simbolo di T che non fa parte dei cammini possibili, significa che non è possibile trovare T all'interno del messaggio. In caso contrario si continua fino all'esaurimento di T, che assumiamo avvenga nel nodo N. A questo punto l'algoritmo si ferma: le foglie raggiungibili da N contengono le posizioni di inizio delle sottostringhe T all'interno del messaggio.

Ad esempio, se dobbiamo cercare le occorrenze di "C" all'interno di MACACO\$ ci si sposta da ROOT al nodo A. Avendo già eguagliato la sottostringa cercata sappiamo che le foglie raggiungibili da A indicheranno le posizioni di inizio della sottostringa, in questo caso 3 e 5.

## 2.3. FM-Index

Un FM-index è una struttura dati inventata da Paolo Ferragina e Giovanni Manzini. Si basa sulla trasformata di Burrows-Wheeler e presenta alcune similarità con l'array di suffissi. Il suo obiettivo è quello di comprimere un messaggio consentendo in modo efficiente di svolgere alcune query, come la ricerca di una sottostringa.

Come nella BWT il primo passo dell'algoritmo è ordinare tutte le permutazioni della stringa. Ad esempio con la stringa  $S = \text{COCOMERO\_ROSSO}$ :

Index	First	Stringa	Last
1	<b>C</b>	COCOMERO_ROSSO	<b>O</b>
2	<b>C</b>	COMERO_ROSSOCO	<b>O</b>
3	<b>E</b>	ERO_ROSSOCOCOM	<b>M</b>
4	<b>M</b>	MERO_ROSSOCOCO	<b>O</b>
5	<b>O</b>	OCOCOMERO_ROSS	<b>S</b>
6	<b>O</b>	OCOMERO_ROSSOC	<b>C</b>
7	<b>O</b>	OMERO_ROSSOCOC	<b>C</b>
8	<b>O</b>	OSSOCOCOMERO_R	<b>R</b>
9	<b>O</b>	O_ROSSOCOCOMER	<b>R</b>
10	<b>R</b>	ROSSOCOCOMERO_	<b>_</b>
11	<b>R</b>	RO_ROSSOCOCOME	<b>E</b>
12	<b>S</b>	SOCOCOMERO_ROS	<b>S</b>
13	<b>S</b>	SSOCOCOMERO_RO	<b>O</b>
14	<b>_</b>	_ROSSOCOCOMERO	<b>O</b>

Identifichiamo con  $F$  la sequenza dei vari simboli First dopo la trasformata di Burrows-Wheeler e con  $L$  quella dei Last. Con  $L[i]$  e  $F[i]$  si intenderà il carattere nella lista in posizione  $i$ . Si creano quindi due tabelle, una che indica in che posizione all'interno di  $F$  comincia la sequenza di un determinato carattere ed un'altra per sapere quante occorrenze di un simbolo ci sono fino alla posizione indicata in  $L$ .

char	<b>C</b>	<b>E</b>	<b>M</b>	<b>O</b>	<b>R</b>	<b>S</b>	<b>_</b>
$C(c)$	0	2	3	4	9	11	13

Indichiamo con  $C(c)$  la funzione che restituisce l'indice in cui iniziano le occorrenze di  $c$  in CCEMOOOOORRSS\_. Ci saranno tante colonne nella tabella quanto è il valore della cardinalità dell'alfabeto.

char/ind	C	E	M	O	R	S	_
1	0	0	0	1	0	0	0
2	0	0	0	2	0	0	0
3	0	0	1	2	0	0	0
4	0	0	1	3	0	0	0
5	0	0	1	3	0	1	0
6	1	0	1	3	0	1	0
7	2	0	1	3	0	1	0
8	2	0	1	3	1	1	0
9	2	0	1	3	2	1	0
10	2	0	1	3	2	1	1
11	2	0	1	3	2	1	1
12	2	1	1	3	2	2	1
13	2	1	1	4	2	2	1
14	2	1	1	5	2	2	1

Indichiamo con  $Occ(char, ind)$  la funzione che restituisce quante occorrenze di  $char$  si trovano dalla posizione 1 fino alla posizione  $ind$  all'interno di OOMOSCCRR\_ESOO.

Consultando le tabelle è possibile stabilire la posizione del carattere  $L[i]$  all'interno di  $F$  creando una relazione biunivoca  $L[i] = F[j]$ . Ci riferiamo a questa operazione con  $LF[i]$  (last – to – first).

$$LF[i] = C(L[i]) + Occ(L[i], i).$$

$$\text{Nell'esempio svolto } L[13] = C(O) + Occ(O, 13) = 4 + 4 = 8.$$

Inoltre si noti che in ogni permutazione di  $S$  Last è il carattere che precede First. Questa caratteristica consente di ricostruire  $S$  a partire da  $L$ .

Si fissi inizialmente  $i = 1$  ed  $S[n] = L[1]$ . Per ogni  $k = n, \dots, 2$   $S[k-1] = L[LF(i)]$

Nell'esempio proposto  $L[1] = S[14] = O$ . Per ottenere  $S[13]$ , si effettua il seguente calcolo:

$$S[13] = L[LF(1)] = L[C(i) + Occ(i, 1)] = L[5] = S.$$

Gli FM index sono particolarmente efficienti nella ricerca di sequenze di caratteri all'interno del

testo originale e questa operazione si separa in due fasi: una per contare il numero di occorrenze della sottostringa (Count) e una per individuare la posizione di partenza (Locate).

Per svolgere la funzione Count si calcolano passo per passo degli intervalli ed il risultato equivale alla grandezza dell'intervallo. Questo viene ridotto ad ogni passaggio. Nella ricerca di una sottostringa  $M$  di lunghezza  $m$  ci saranno  $m$  iterazioni nella funzione, perciò la complessità temporale dell'algoritmo è  $O(m)$ .

Ogni iterazione ha come input un simbolo di  $M$ : si comincia dall'ultimo carattere,  $S[m]$ , e si scala di uno all'indietro fino all'esaurimento della sottostringa. L'intervallo iniziale è dato da  $\{C(S[m]) + 1 ; C(S[m]+1)\}$  ovvero i suoi estremi (con cui mi riferisco con start ed end) sono la posizione iniziale e quella finale delle occorrenze di char all'interno di  $F$ .

Si prende poi il secondo carattere a partire dal fondo,  $S[m-1]$ , ed il nuovo intervallo sarà  $\{C(S[m-1]) + Occ(S[m-1], start-1) + 1 ; C(S[m-1]) + Occ(S[m-1], end)\}$ . Si esegue ad aggiornare i valore di end e start fino ad arrivare ad esaminare  $S[1]$ , dopodiché  $end-start + 1$  sarà il risultato della funzione. Per trovare le occorrenze di "RO" all'interno della stringa di esempio, essendo RO di lunghezza 2 ci saranno due passaggi:

- Inizialmente l'intervallo sarà  $\{C(O) + 1, C[R]\}$  quindi  $start = 5$  ed  $end = 9$ .
- Poi diventa  $\{C(R) + Occ(R, 4) + 1 ; C(R) + Occ(R, 9)\} = \{10 ; 11\}$ .
- Il valore di ritorno è  $end-start+1 = 11-10+1 = 2$ .

## 2.4. Rank/Select

Ci si può riferire alle operazioni Count e Locate dell'FM-index come Rank e Select, definendole in questo modo:

$rank_x(S, i)$ : restituisce il numero di  $x$  nel prefisso  $S[1, i]$

$select_x(S, i)$ : restituisce la posizione dell' $i$ -esimo  $x$  in  $S$

Combinando queste funzioni, dato del codice binario, si è in grado di effettuare la ricerca di una sottostringa su un alfabeto arbitrario in tempo costante, senza che ci sia bisogno della decompressione della struttura.



## 3. Wavelet Tree

In questa sezione verranno esaminati i wavelet tree, una struttura dati inventata nel 2003 da Roberto Grossi, Ankur Gupta e Jeffrey Scott Vitter. Il nome deriva dalla trasformata wavelet, uno strumento matematico per l'analisi e l'elaborazione dei segnali, nonostante le due cose non siano assolutamente collegate.

Ci si può riferire a questa struttura come ad una rappresentazione di una sequenza, di una permutazione di elementi o di un reticolo di punti [4].

Una delle proprietà più interessanti dei wavelet tree è che consentono di effettuare le operazioni rank e select in tempo costante [1].

### 3.1. Struttura e creazione

Quando si utilizza questa tecnica la stringa in input viene convertita in un wavelet tree, ovvero un albero binario bilanciato dove ogni nodo contiene un vettore di bit che rappresenta parte del messaggio originale.

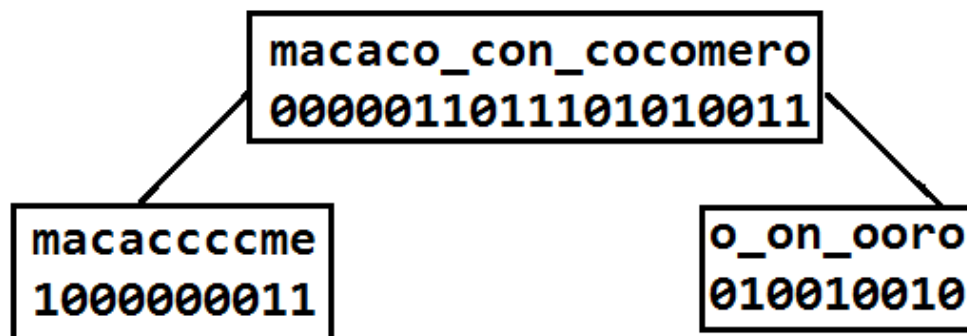
All'interno della sequenza di bit di un nodo l'ennesimo valore binario indica se il carattere rappresentato dal bit in posizione  $n$  appartiene alla prima o alla seconda metà dell'alfabeto costituito dall'insieme dei simboli rappresentati nel nodo. Si supponga per convenzione di raffigurare con "0" i simboli che appartengono alla prima metà dell'alfabeto nodo e con "1" i rimanenti. L'insieme di tutti i caratteri etichettati come 0 costituirà il contenuto della sequenza di bit del nodo figlio sinistro, analogamente tutti gli 1 comporranno il nodo figlio destro. Se il nodo è costituito da soli 1 allora c'è un solo carattere rappresentato. Ci si trova quindi in una foglia, ovvero in un nodo che non ha figli.

Assumiamo che la cardinalità dell'alfabeto in questione sia  $|\Sigma|=n$ : il dimezzamento della dimensione dell'alfabeto ad ogni passaggio garantisce che l'altezza dell'albero corrisponda al logaritmo in base due di  $n$ , sempre arrotondato per eccesso.

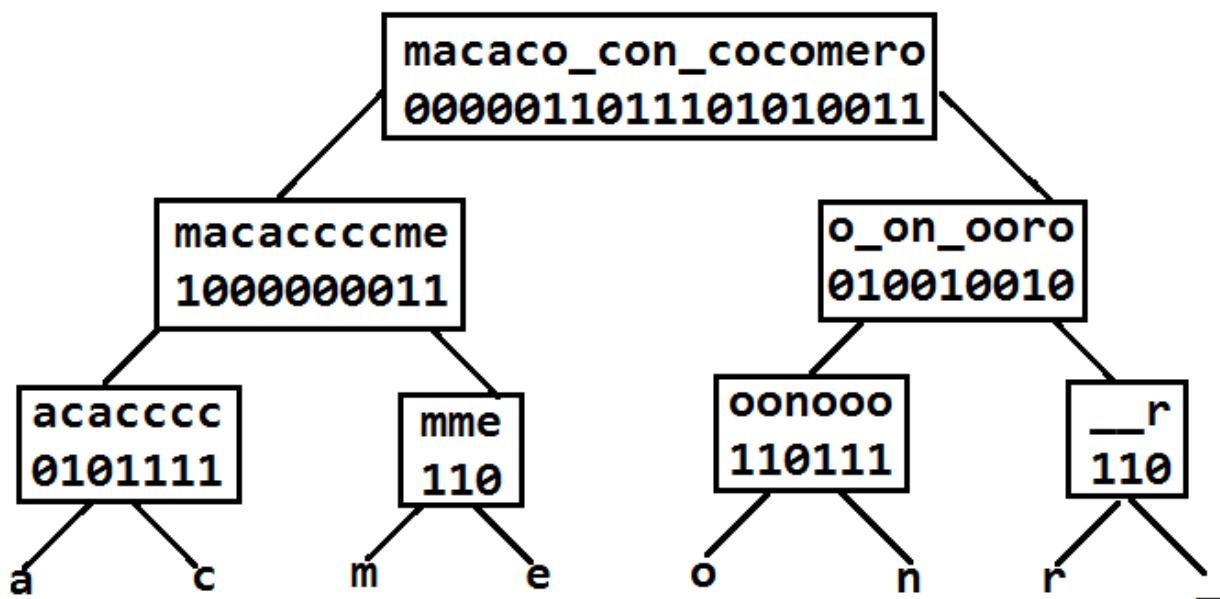
Ecco ora un esempio dove "macaco\_con\_cocomero" è la stringa in input da rappresentare.

L'alfabeto della stringa, costituito da 8 caratteri {a, c, m, e, o, n, r, \_}, deve essere ordinato, generalmente usando come criterio per questa operazione il valore numerico ASCII o Unicode associato ai simboli in questione.

La stringa "macaco\_con\_cocomero" viene quindi inizialmente mappata come "0000011011101010011", in quanto i caratteri {a, c, m, e} verranno sostituiti con uno 0, mentre {o, n, r, \_} saranno rimpiazzati con un 1.



Si continua ricorsivamente con questo modus operandi fino ad arrivare alle foglie. L'altezza dell'albero quindi sarà  $\log_2(8) = 3$ . Ecco il risultato finale:



I bit sono l'unica informazione contenuta nel nodo, oltre ai puntatori per raggiungere i due nodi figli ed il primo carattere della seconda metà dell'alfabeto.

Il motivo fondamentale però per cui si usano i wavelet tree è la possibilità di interrogare l'albero con le query accesso,  $\text{rank}_q$  e  $\text{select}_q$  in modo efficiente. Spesso queste funzioni vengono combinate per effettuare query più complesse, come ad esempio la ricerca di una sottostringa.

## 3.2. Rank

Per Rank (char ch, int index) si intende la funzione che ha come valore di ritorno il numero di occorrenze del carattere ch fino all'indice index della stringa in input.

Per sapere il numero di occorrenze in un intervallo, ad esempio [3 ; 11], sarà sufficiente sottrarre al valore di Rank ( ch, 11) il valore di Rank (ch, 3).

Ottenere il risultato di questa funzione sarebbe molto facile se la stringa S fosse rappresentata tramite un normale array di caratteri. Ad esempio in java basterebbe scorrere sequenzialmente l'array nel seguente modo.

```
static int countOccurrences (String S, char ch, int index){  
    int counter = 0;  
    for (int i=0; i <= index; i++) {  
        if ( S.charAt(i) == ch ) {  
            counter ++;  
        }  
    }  
    return counter;  
}
```

Il costo dell'operazione però è troppo alta, in quanto scorrere sequenzialmente l'array non è una soluzione efficiente.

L'operazione di rank viene eseguita ricorsivamente tante volte quanta è l'altezza dell'albero aumentata di uno: infatti il primo nodo a chiamare la funzione è la radice e le ricorsioni continuano dal padre verso i figli fino a quando si giunge ad una foglia. In ogni passaggio, per sapere il valore di index da inserire come parametro, si esegue la sottofunzione  $\text{rank}_1(\text{index})$ , che conta il numero di occorrenze di 1 all'interno del vettore di bit, permettendoci di scoprire qual è il numero di 1 che sono presenti nella sequenza di bit del nodo corrente fino all'indice index. Si contano ovviamente il numero di 1 se il carattere in questione è rappresentato con 1 in quella sequenza, viceversa si calcolano gli 0.

Si può quindi descrivere la funzione secondo il seguente pseudocodice, indicando con this il nodo che esegue il metodo:

```
function Rank (char, index) : int
```

```
    if (this è una foglia) :
```

```
        return index+1                // si ritorna la posizione, ovvero l'indice aumentato di 1
```

```
    bitVal  $\leftarrow$  valore binario con cui char è rappresentato nella sequenza di bit del nodo locale
```

```
    nextIndex  $\leftarrow$  RankbitVal(index)
```

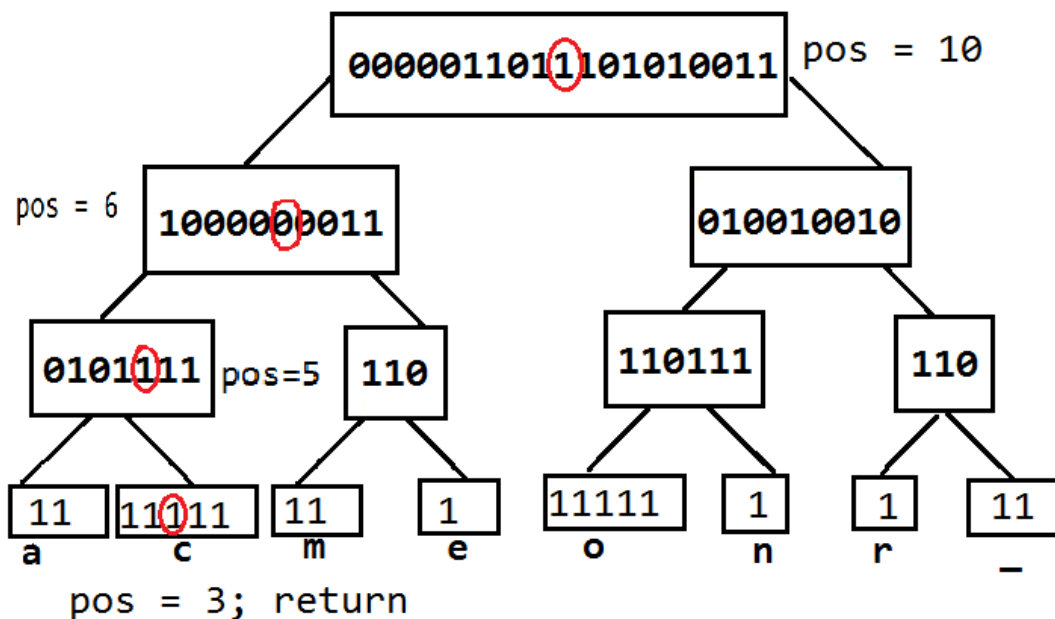
```
    if (bitVal = 1):
```

```
        return this.rightChild.Rank (char, nextIndex)
```

```
    else
```

```
        return this.leftChild.Rank (char, nextIndex)
```

Ad esempio si può provare ad eseguire con Rank (C, 9) all'interno di COCOMERO\_ROSSO. I numeri cerchiati in rosso indicano la posizione che costituisce i valori di ritorno della varie sottofunzioni  $\text{rank}_{\text{bit}}(\text{char})$ .



I bit cerchiati di rosso saranno i risultati delle varie operazioni  $\text{Rank}_{\text{bitVal}}(\text{index})$ .

### 3.3. Select

Select (char ch, int occ) è una funzione che restituisce la posizione in cui si trova l'occorrenza numero occ del carattere ch all'interno della stringa. Anche in questo caso se l'input fosse rappresentato come un array di caratteri Unicode l'operazione, anche se meno efficiente, sarebbe molto più intuitiva: sarebbe infatti sufficiente scorrere sequenzialmente l'array fermandosi dopo aver trovato l'occorrenza richiesta e restituendo la posizione in cui ci si trova.

Una prima soluzione per risolvere questa query è di invocare la funzione ricorsivamente a partire dalla foglia in cui si trova il carattere passato come parametro. In questo caso il primo passo dell'algoritmo sarà dunque trovare la foglia corretta.

```
function getLeaf (char ch) : Node
    Node n = root;
    while (n non è una foglia) :
        if (ch è rappresentato con 1 all'interno della sequenza di bit di n):
            n = n.right
        else :
            n = n.left
    return n
```

La foglia restituita dalla funzione descritta dal precedente pseudocodice sarà dunque il primo nodo dell'albero su cui eseguire l'operazione select vera e propria. Select (char ch, int occ) sarà dunque eseguita ricorsivamente sul cammino dalla foglia ad un albero. Si può descrivere così l'operazione:

```
function select (char ch, int occ) : int
    if ( this è la radice) :
        return occ
    int position
    if (char è rappresentato con 1 nella sequenza di bit) :
        position = select1(occ)
    else :
        position = select0(occ)
    return this.parent.select (ch, position)
```

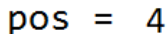
Nel primo passaggio essendo le foglie costituite solamente da 1, il valore di occ non varierà. Risalendo invece l'albero dalla foglia alla radice occ assumerà un valore che sarà maggiore o uguale rispetto alla sua precedente versione.

Nella versione proposta invece le due funzioni precedenti vengono accorpate in un unico metodo select ricorsivo, che opera secondo il seguente pseudocodice:

```
return select1(newIndex)
```

ricorsive. Si risale così fino alla radice, dove viene restituito il valore finale dell'operazione.

solita stringa MACACO CON COCOMERO.



costituisce il risultato delle varie sottofunzioni  $\text{select}_{\text{bit}}(\text{ch})$ .

# 4. Implementazione e rappresentazione di un wavelet tree in Java

Nell'implementazione che ho realizzato si fanno uso di due differenti package, uno in cui si trova la parte logica dell'algoritmo (wavelet) ed uno in cui si gestiscono gli aspetti relativi alla visualizzazione grafica(graphic). Il progetto è reperibile su git, con il seguente URL: <https://github.com/luca-Serena/tesi-strutture-dati-compressione>

## 4.1. Package wavelet

wavelet contiene i seguenti file:

- BitRankSelect, l'interfaccia in cui vengono definite le operazioni che sono necessarie per trattare la sequenza di bit. Queste funzioni sono:
  - BitRankSelect fromArray(ArrayList <Boolean> v), che restituisce la classe implementante l'interfaccia che viene utilizzata.
  - int rank(boolean bit, int index), che restituisce il numero di occorrenze nella sequenza di bit di 1 o 0 fino all'indice index compreso.



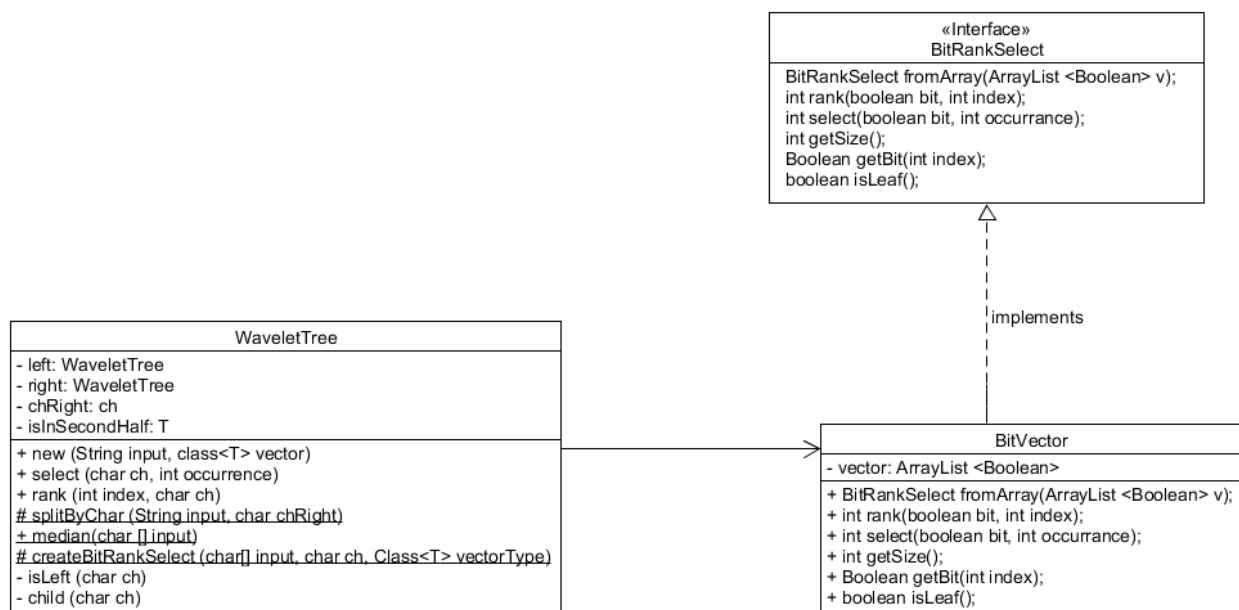
- `int select(boolean bit, int occ)`, che restituisce la posizione dell'occorrenza numero `occ` del bit richiesto.
  - `int getSize()`, che restituisce la dimensione del vettore di bit.
  - `Boolean getBit(int index)`, che restituisce il valore binario del bit che ha come indice `index`.
  - `boolean isLeaf()`, che indica se il nodo da cui parte la chiamata della funzione è una foglia oppure no.
- `BitVector`, la classe concreta che implementa l'interfaccia `BitRankSelect` e che può venire utilizzata per gestire la sequenza di bit di ogni nodo. Ha come unica variabile di istanza un `arraylist` di `Boolean`, che rappresenta la successione di bit del nodo. Il codice proposto non rappresenta una soluzione efficiente, in quanto avviene una scansione dell'`arraylist`, che viene percorso sequenzialmente. Questa versione potrebbe però essere migliorata: il senso di avere una classe come parametro di `WaveletTree` è di lasciare all'ipotetico utilizzatore del package la possibilità di sostituire `BitVector` con una classe più efficiente.
- Per capire se un nodo è una foglia invece si esamina l'`arraylist`: se c'è almeno un elemento nella lista contraddistinto dallo 0 significa che ci sono almeno due caratteri, quindi il nodo ha dei figli.
- `WaveletTree`, il file in cui risiedono tutte le operazioni principali del progetto. Le variabili di istanza di questa classe sono i puntatori per raggiungere i nodi figli (`left` e `right`), il carattere che indica qual è il primo simbolo della seconda metà dell'alfabeto ed `isInSecondHalf`, variabile di tipo generico `T`, in cui `T` è un oggetto appartenente ad una classe che implementa l'interfaccia `BitRankSelect`. `isInSecondHalf` contiene quindi tutte le informazioni sulla sequenza di bit. L'albero binario viene creato nel costruttore, i cui parametri sono la stringa in input e la classe da utilizzare per rappresentare il vettore di bit. A partire dalla radice vengono creati tutti i nodi, di cui vengono anche inizializzate le istanze.

Tramite i metodi pubblici `rank` e `select` è possibile eseguire le omonime operazioni. Queste

possono restituire il risultato o lanciare un'eccezione, `CharacterNotFoundException` nel caso il carattere passato come parametro non appartenga al testo oppure `IndexOutOfBoundsException` se viene selezionato un indice superiore alla lunghezza della stringa o se viene ricercata un'occorrenza inesistente di un simbolo.

Nella classe sono inoltre presenti alcuni metodi statici:

- `median`, che restituisce il carattere medio (`chRight`) dell'alfabeto del nodo
- `createBitRankSelect`, che inizializza il vettore di bit di tipo `T`
- `splitByChar`, che restituisce la stringhe da utilizzare come parametro del costruttore dei nodi figli.



Il codice scritto è stato testato tramite JUnit test dalla classe `WaveletTreeTest`, che verifica la correttezza delle operazioni su alcune stringhe di esempio.

## 4.2. Package graphic

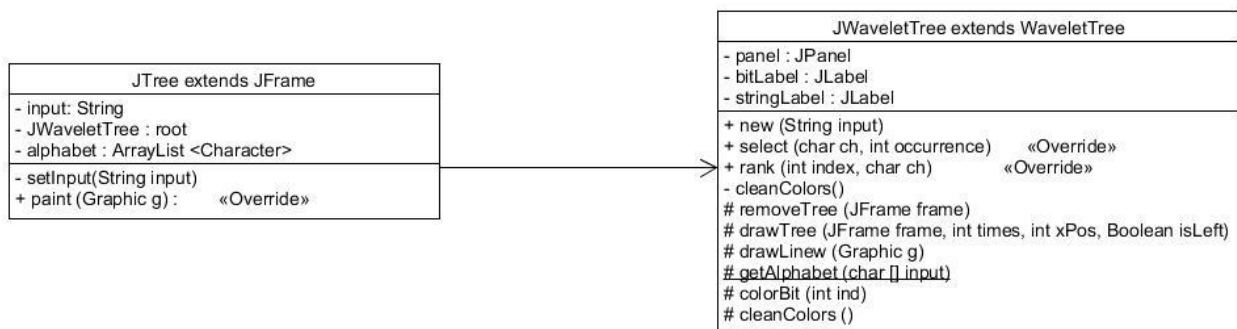
Oltre alle classi precedentemente descritte, è stato implementato anche un ulteriore package, `graphic`, che consente di visualizzare graficamente un wavelet tree ed i risultati delle query rank e select, data una stringa inserita dall'utente.

Tutte le componenti grafiche utilizzate fanno parte della libreria `javax.swing`.

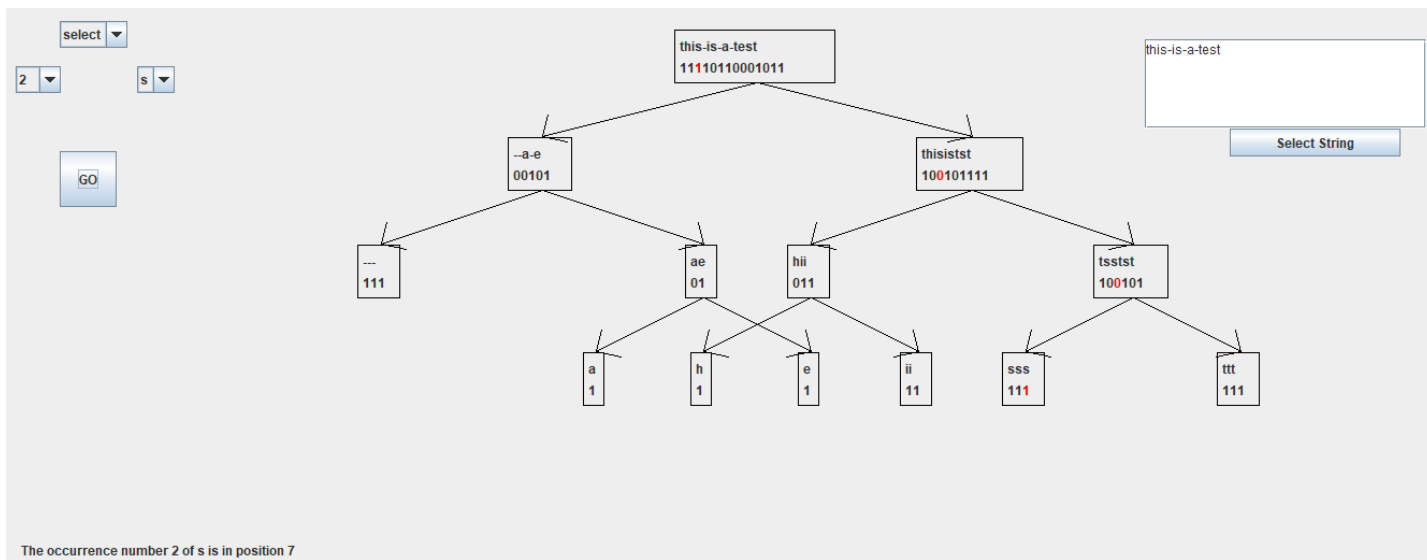
In `graphic` sono presenti due file:

- `JTree`, una classe che estende `JFrame`. Contiene una `JTextArea` dentro cui inserire la stringa in input, una `JComboBox` in cui è possibile scegliere quale tra le due query eseguire ed altre due `JComboBox` che consentono di selezionare i parametri per le operazioni rank e select. In questo file le uniche funzioni presenti sono gli `actionListener` delle componenti grafiche.
- `JWaveletTree`, classe che estende `WaveletTree`. Ogni oggetto di questa classe rappresenta un nodo dell'albero che viene visualizzato sullo schermo.

La grafica consiste in un pannello che contiene due `Label`, una per rappresentare la sequenza di bit e l'altra per raffigurare i caratteri Unicode corrispondenti. Le funzioni `drawTree` e `drawLines` servono a creare l'albero ed i collegamenti tra i nodi, il metodo `removeTree` a rimuovere l'albero quando l'utente decide di cambiare la stringa in input. Viene inoltre fatto `override` dei metodi rank e select in modo da colorare i bit che costituiscono il risultato di una loro sottoquery. Ci sono poi metodi `colorBit` e `cleanColors` per gestire nello specifico colorazione e la rimozione dei colori all'interno dei nodi.



Questo è l'aspetto dell'interfaccia grafica implementata:



# Bibliografia

- 1) Navarro, Gonzalo, and Eliana Provedel. "Fast, Small, Simple Rank/Select on Bitmaps." *SEA* 7276 (2012): 295-306
- 2) Ferragina, Paolo, and Giovanni Manzini. "Indexing compressed text." *Journal of the ACM (JACM)* 52.4 (2005): 552-581.
- 3) Kärkkäinen, Juha, Dominik Kempa, and Simon J. Puglisi. "Linear time Lempel-Ziv factorization: Simple, fast, small." *Annual Symposium on Combinatorial Pattern Matching*. Springer, Berlin, Heidelberg, 2013.
- 4) Navarro, Gonzalo. "Wavelet trees for all." *Journal of Discrete Algorithms* 25 (2014): 2-20
- 5) "Lossy Compression". Wikipedia. [https://en.wikipedia.org/wiki/Lossy\\_compression](https://en.wikipedia.org/wiki/Lossy_compression)
- 6) "Lossless Compression". Wikipedia.  
[https://en.wikipedia.org/wiki/Lossless\\_compression](https://en.wikipedia.org/wiki/Lossless_compression)
- 7) "Arithmetic coding". Wikipedia. [https://en.wikipedia.org/wiki/Arithmetic\\_coding](https://en.wikipedia.org/wiki/Arithmetic_coding)
- 8) "Bzip2". Wikipedia. <https://en.wikipedia.org/wiki/Bzip2>
- 9) "Move to front Transform". Wikipedia. [https://en.wikipedia.org/wiki/Move-to-front\\_transform](https://en.wikipedia.org/wiki/Move-to-front_transform)
- 10) "Run-length encoding". Wikipedia. [https://en.wikipedia.org/wiki/Run-length\\_encoding](https://en.wikipedia.org/wiki/Run-length_encoding)