Algorithm 1 getSplittedICFL(w)

```
1: splittedFactors = []
2: for ∀ factor f in ICFL(w) do
3:     while true do
4:         if length(f) > 1 ∧ borderLength(f) > 0 then
5:             if borderLength = length then
6:                 Split f in length factors of length 1
7:                 Add each factor to splittedFactors
8:             else
9:                 factor1 ← substring(f,0,length-border)
10:                factor2 ← substring(f,length-border, length)
11:                Add factor1 to splittedFactors
12:                f ← factor2
13:            end if
14:        else
15:            Add factor f to splittedFactors
16:            break
17:        end if
18:    end while
19: end for
20: return splittedFactors
```

La funzione substring(f, i, len) restituisce la sottostringa di f che parte dall'indice i e consta di len caratteri.
La funzione borderLength(w) restituisce il bordo massimale di una parola w.

Questo pseudocodice restituisce una lista di fattori spezzati, dove il criterio per spezzare differisce in base alle caratteristiche del fattore:

Qualora il bordo sia lungo quanto la stringa stessa, il fattore viene decomposto in stringhe di un singolo carattere, aggiunte alla lista dei fattori spezzati.

Qualora il bordo non sia lungo quanto la stringa stessa, ma più grande di 0, c'è ancora da scomporre. La scomposizione spezza la stringa in una metà sinistra, che corrisponde alla stringa meno i caratteri che costituiscono il bordo visto come suffisso, e una metà destra, che corrisponde al bordo visto come suffisso. Poi, si cerca di scomporre ulteriormente il bordo nel caso in cui il bordo stesso sia bordered, facendolo diventare il nuovo fattore corrente. Il ciclo si ripeterà all'infinito, finché:

Quando il fattore considerato non ha più un bordo, si aggiunge tale fattore alla lista dei fattori spezzati e si rompe il ciclo infinito, passando al fattore successivo. Di fatto questa funzione rappresenta la differenza principale, a livello strutturale, di questa variante rispetto alla BWT di Scott. Si tratta di una sorta di "mediatore": utilizza la ICFL invece della CFL e svolge delle

operazioni aggiuntive di preparazione dei fattori non presenti nell'algoritmo di Scott. Il resto dell'algoritmo, e della sua trasformazione inversa, dettagliati successivamente, non sono nient'altro che, con le dovute differenze dovute all'ordine discendente invece che ascendente, analoghe all'algoritmo di partenza.

Algorithm 2 inverseLyndonConjugates (w)

```
1: splittedFactors = getSplittedICFL(w)
2: conjTable = []
3: i ← 0
4: while i < length(w) ∧ ∀ factor f in splittedFactors: do
5:     for j = 0, ··· length(f)-1 do
6:         conjTable[i+j] ← substring(f, f+length(f)-j, j ) · substring(f, 0, length(f)-j)
7:     end for
8:     i ← i+length(f)
9: end while
```

Il simbolo · indica la concatenazione di due stringhe.

Algorithm 3 BWTZ(w)

```
1: if length(w) ≤ 1 then return w
2: end if
3: inverseConjugates = inverseLyndonConjugates(w)
4: sort(inverseConjugates)
5: bwt = []
6: for i = 0, ··· length(w) do
7:     bwt ← bwt · last(inverseConjugates[i])
8: end for
9: return bwt
```

La funzione last(w) restituisce l'ultimo carattere di una stringa w.
La funzione sort() ordinerà in modo decrescente. L'ordinamento è quello all'infinito descritto nel paper di riferimento.

Algorithm 4 IBWTZ(bwt)

```
1: if length(bwt) ≤ 1 then return w
2: end if
3: permutation = standardPermutationDescending(w)
4: cyclesList = findCycles(permutation)
5: ibwt = []
6: i ← 0
7: while  i ≤ length(bwt) ∧ ∀ cycle in cycleList do
8:     for ∀ elements in current cycle do
9:         ibwt[i] ← bwt[element]
10:        i ← i + 1
11:    end for
12: end while
13: return ibwt
```

La funzione standardPermutationDescending(w) restituisce la permutazione standard degli indici delle lettere della parola w, in ordine decrescente.
La funzione findCycles(permutation) restituisce una lista di liste (implementata nel codice sorgente come una lista di puntatori a liste). Il codice assume che la lista dei cicli venga restituita ordinata, dall'ultimo ciclo al primo. Esistono diversi modi per implementare una funzione di questo tipo, nell'implementazione ne usiamo una possibile versione, che punta alla semplicità.

Algorithm 5 findCycles(standardPermutation)

```
1: cyclesList = []
2: for i = 0, ··· length(standardPermutation) do
3:     currentIndex ← i
4:     currentCycle = []
5:     while standardPermutation[currentIndex] ≠ -1 do
6:         nextIndex ← standardPermutation[currentIndex]
7:         add nextIndex to currentCycle
8:         standardPermutation[currentIndex] ← -1
9:         currentIndex ← nextIndex
10:    end while
11:    if currentCycle is not empty then
12:        add currentCycle to cycleList
13:    end if
14: end for
```

Questa funzione si limita a marcare l'elemento appena letto del ciclo corrente come -1, in maniera tale da delimitare quando abbiamo finito di leggere l'intero ciclo e passare al successivo.

Algorithm 6 standardPermutation(bwt)

```
1: indexArray = []
2: for i = 0, ··· length(bwt) do
3:     indexArray[i].index ← i
4:     indexArray[i].value ← bwt[i]
5: end for
6: sort(indexArray)
7: standardPermutation = []
8: for i = 0, ··· length(bwt) do
9:     standardPermutation[i] ← indexArray[i].index
10: end for
11: return standardPermutation
```

Questa funzione fa uso di un array di indici, e ordina questo array in base al valore di ogni casella (carattere associato della BWT) in senso decrescente. La funzione sort(indexArray) si limita solo a ordinare. L'ordinamento può essere implementato con qualunque algoritmo di ordinamento. Infine, restituisce l'array che rappresenta la permutazione standard.