Ne' weitere Perle unter den Algorithmen:

Burrows Wheeler Transformation

Einleitung.

- Ordnet gleiche Buchstaben möglichst hintereinander an.
  - Problem: Text muss wiederherstellbar sein.
  - Zielkonflikt:

Sortiert → besser komprimierbar

Reproduzierbar → schlechter komprimierbar

Einleitung.

- Wozu dient die Transformation?
  - Vorstufe des Komprimierens.
  - Weitere Verarbeitung durch: Huffman, Arithmetische Codierung möglich.

```
>transform "Lorem
ipsum dolor sit
amet, consetetur
sadipscing elitr,
sed diam nonumy
eirmod tempor...“
```

```
>("...tt..ttaattt
ttt,,ddddmmmmttoo
oottyyggoommssssse
essmmddrrmmmmttaa
ttee,,mmaarraaoo,
,,,rrrr...“, 115)
```

Ein- und Ausgaben.

- Hintransformation:

$$transform :: Ord\ a \Rightarrow [a] \rightarrow ([a], Int)$$

- Rücktransformation:

$$untransform :: Ord\ a \Rightarrow ([a], Int) \rightarrow [a]$$

Funktionsweise: Hin-Transformation.

- Beispiel Zeichenkette: „.Wedel.“:

  (Punkte markieren das Ende und den Beginn des Strings)

- Rotation der Zeichenkette:
  - 1. Schritt:
    - „.Wedel.“

Funktionsweise: Hin-Transformation.

- Beispiel Zeichenkette: „.Wedel.“:

- Rotation der Zeichenkette:
  - 2. Schritt:
    - „.Wedel.“
    - „..Wedel“

Funktionsweise: Hin-Transformation.

- Beispiel Zeichenkette: „.Wedel.“:

- Rotation der Zeichenkette:
  - 3. Schritt:
    - „.Wedel.“
    - „..Wedel“
    - „l..Wede“

Funktionsweise: Hin-Transformation.

- Beispiel Zeichenkette: „.Wedel.“:

- Rotation der Zeichenkette:
  - 4. Schritt:
    - „.Wedel.“
    - „..Wedel“
    - „l..Wede“
    - „el..Wed“

Funktionsweise: Hin-Transformation.

- Beispiel Zeichenkette: „..Wedel.“:

- Rotation der Zeichenkette:
  - 5. Schritt:
    - „..Wedel.“
    - „...Wedel“
    - „l..Wede“
    - „el..Wed“
    - „del..We“

Funktionsweise: Hin-Transformation.

- Beispiel Zeichenkette: „.Wedel.“:

- Rotation der Zeichenkette:
  - 6. Schritt:
    - „.Wedel.“
    - „..Wedel“
    - „l..Wede“
    - „el..Wed“
    - „del..We“
    - „edel..W“

Funktionsweise: Hin-Transformation.

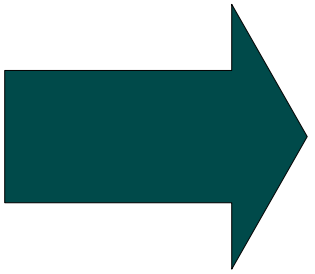
- Beispiel Zeichenkette: „.Wedel.“:

- Rotation der Zeichenkette:
  - 7. Schritt:
    - „.Wedel.“
    - „..Wedel“
    - „l..Wede“
    - „el..Wed“
    - „del..We“
    - „edel..W“
    - „Wedel..“

Funktionsweise: Hin-Transformation.

- Beispiel Zeichenkette: „..Wedel.“:

- Auflistung in einer Tabelle + Sortierung
  - „..Wedel.“
  - „..Wedel“
  - „l..Wede“
  - „el..Wed“
  - „del..We“
  - „edel..W“
  - „Wedel..“

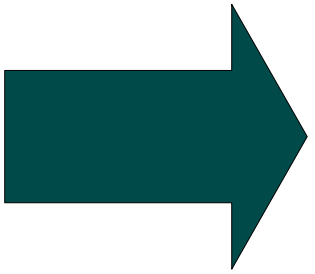


  - „..Wedel“
  - „.Wedel.“
  - „Wedel..“
  - „del..We“
  - „edel..W“
  - „el..Wed“
  - „l..Wede“

Funktionsweise: Hin-Transformation.

- Beispiel Zeichenkette: „.Wedel.“:

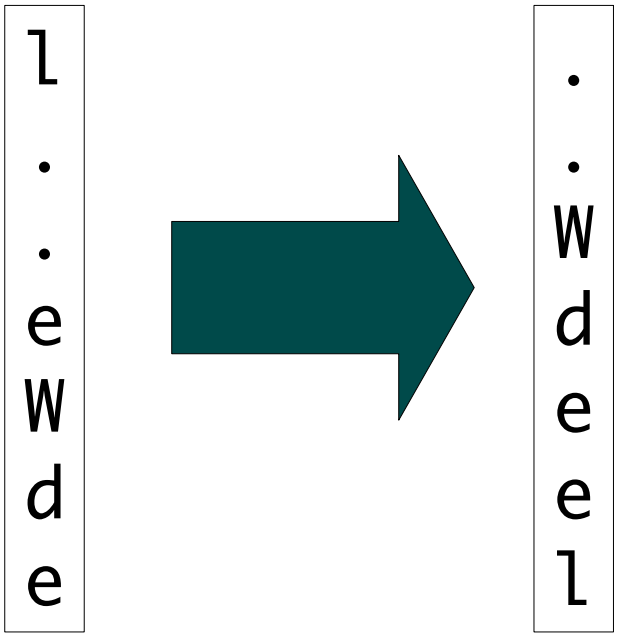
- Extraktion des letzten Zeichens:
  - „..Wedel“
  - „.Wedel.“
  - „Wedel..“
  - „del..We“
  - „edel..W“
  - „el..Wed“
  - „l..Wede“



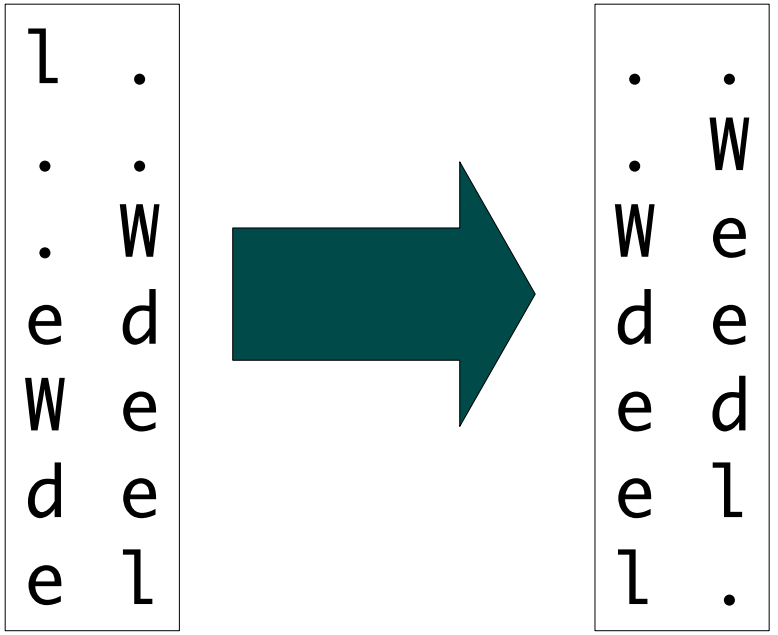
„l..eWde“, 1

Funktionsweise: Rückwärts-Transformation.

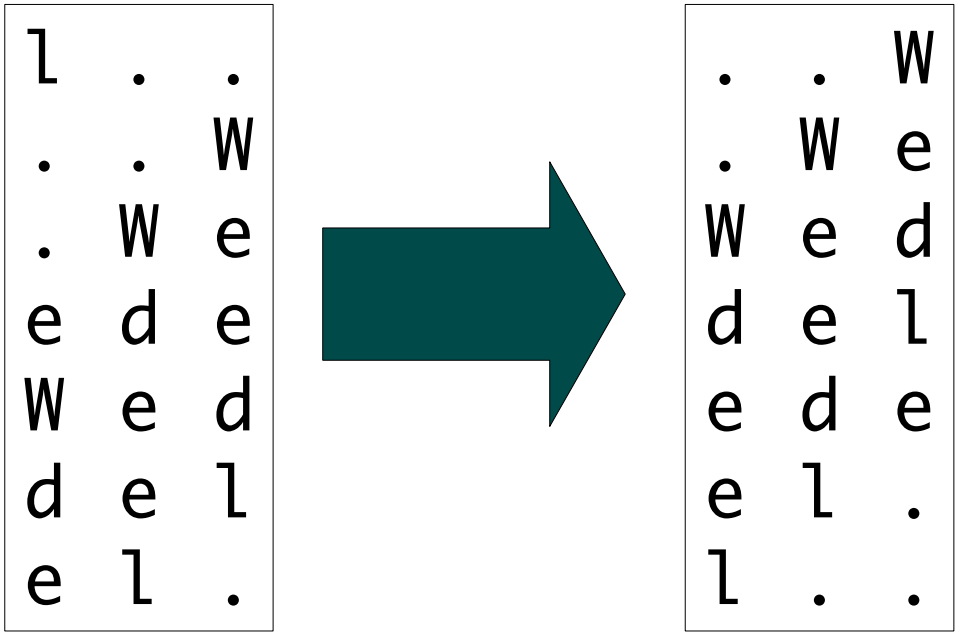
- Beispiel Zeichenkette: „**.Wedel.**“ bzw. inzwischen: **„l..eWde“** und Index 1.



Funktionsweise: Rückwärts-Transformation.

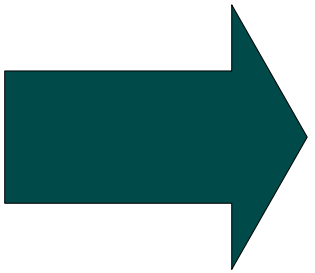


Funktionsweise: Rückwärts-Transformation.



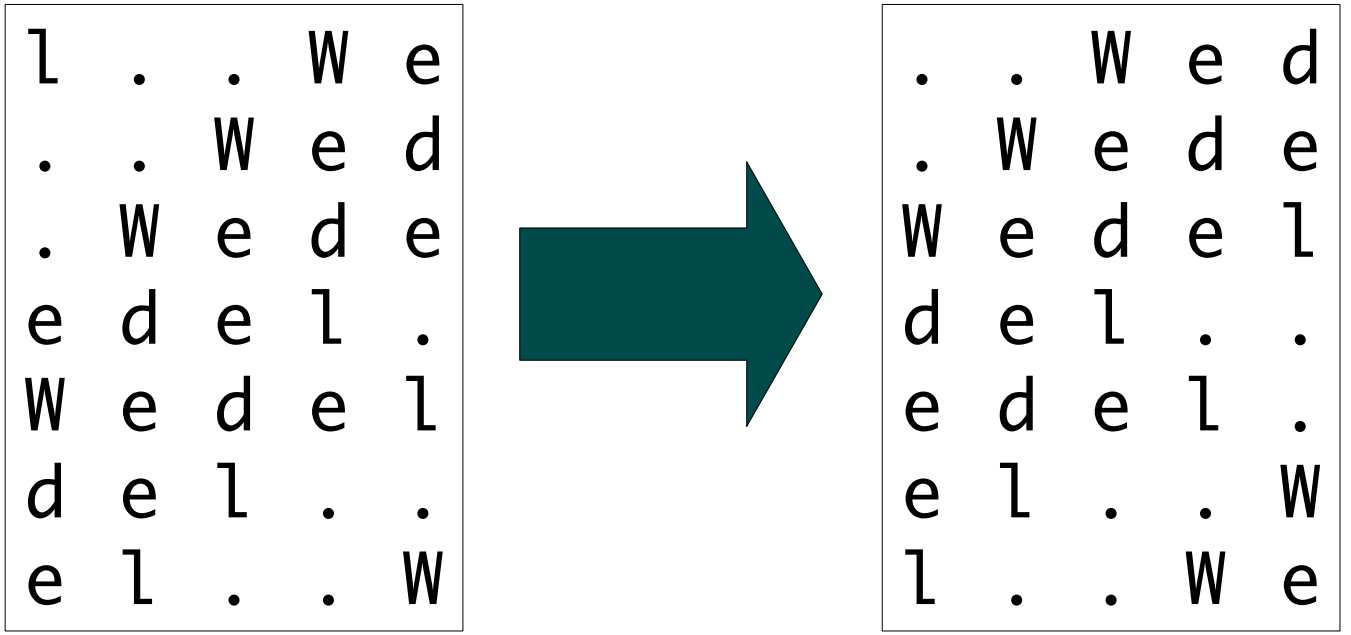
Funktionsweise: Rückwärts-Transformation.

l	.	.	W
.	.	W	e
.	W	e	d
e	d	e	l
W	e	d	e
d	e	l	.
e	l	.	.



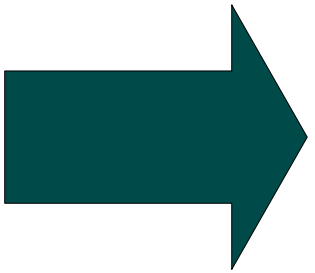
.	.	W	e
.	W	e	d
W	e	d	e
d	e	l	.
e	d	e	l
e	l	.	.
l	.	.	W

Funktionsweise: Rückwärts-Transformation.



Funktionsweise: Rückwärts-Transformation.

```
l . . W e d
. . W e d e
. W e d e l
e d e l . .
W e d e l .
d e l . . W
e l . . W e
```

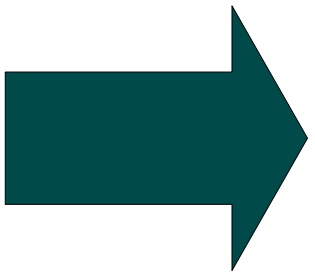


```
. . W e d e
. W e d e l
W e d e l .
d e l . . W
e d e l . .
e l . . W e
l . . W e d
```

Funktionsweise: Rückwärts-Transformation.

- In Zeile 1 steht das gesuchte Wort (→ Index).

l	.	.	W	e	d	e
.	.	W	e	d	e	l
.	W	e	d	e	l	.
e	d	e	l	.	.	W
W	e	d	e	l	.	.
d	e	l	.	.	W	e
e	l	.	.	W	e	d



.	.	W	e	d	e	l
.	W	e	d	e	l	.
W	e	d	e	l	.	.
d	e	l	.	.	W	e
e	d	e	l	.	.	W
e	l	.	.	W	e	d
l	.	.	W	e	d	e

Code: Hin-Transformation

Code.

```
--BW-Transformation
transform :: Ord a => [a] -> ([a],Int)
transform [] = ([],0)
transform xs = (map last xss, position xs xss)
                where xss = sort (rots xs)

--Ermittelt den Index des ersten Vorkommens in einer Liste
position :: Eq a => a -> [a] -> Int
position xs xss = length ( takeWhile (/=xs) xss )

--Erstellt eine Liste mit allen Rotationen der Eingabe
rots :: Ord a => [a] -> [[a]]
rots xs = take (length xs) (iterate lrot xs)

--Rotiert die Eingabe linksrum
lrot :: Ord a => [a] -> [a]
lrot (x:xs) = xs ++ [x]
```

Code: Rück-Transformation

Code.

```
--BW-Rücktransformation
untransform :: Ord a => ([a],Int) -> [a]
untransform (ys,k) = (recreate n ys) !! k
                     where n = length ys

--Berechnet die Ursprungsmatrix Spalte für Spalte
recreate :: Ord a => Int -> [a] -> [[a]]
recreate 0     = map (const [])
recreate (j+1) = hdsort.consCol.fork (id,recreate j)
                 where
                 fork (f,g) x = (f x,g x)

--Sortiert eine Liste von Listen nach dem ersten Element
hdsort :: Ord a => [[a]] -> [[a]]
hdsort = sortBy cmp
         where
         cmp (x:xs) (y:ys) = compare x y

--Fügt eine Spalte zur Matrix hinzu
consCol :: ([a],[[a]]) -> [[a]]
consCol (xs,xss) = zipWith (:) xs xss
```

Code: Optimierung I der Rück-Transformation

Code.

```
untransform2 :: Ord a => ([a],Int) -> [a]
untransform2 (ys,k) = (recreate2 n p ys) !! k
                    where
                    n = length ys
                    p = permutation ys

--Verbesserung: hdsort durch apply ersetzt
recreate2 :: Ord a => Int -> [Int] -> [a] -> [[a]]
recreate2 0 _ xs    = map (const []) xs
recreate2 (j+1) p xs = consCol.fork(apply p, apply p . recreate2 j p) $ xs
                  where
                  fork (f,g) x = (f x,g x)

--sort ys = apply p ys
apply :: Ord a => [Int] -> [a] -> [a]
apply px xs   = [ xs !!(px !! i) | i <- [0..(length(xs)-1)]]

--Erstellt eine Liste mit den zugehörigen Positionen der Elemente und
--sortiert diese
permutation :: Ord a => [a] -> [Int]
permutation xs = map snd (sort (zip xs [0..]))
```

Code: Optimierung II der Rück-Transformation

Code.

```
untransform3 :: Ord a => ([a],Int) -> [a]
untransform3 (ys,k) = (recreate3 n ys) !! k
                    where n = length ys

--Verbesserung: Rekursion aufgelöst
recreate3 :: Ord a => Int -> [a] -> [[a]]
recreate3 j xs = transpose.(take j).tail.iterate (apply p) $ xs
                 where
                 p = permutation xs

--sort ys = apply p ys
apply :: Ord a => [Int] -> [a] -> [a]
apply px xs   = [ xs !!(px !! i) | i <- [0..(length(xs)-1)]]

--Erstellt eine Liste mit den zugehörigen Positionen der Elemente und
--sortiert diese
permutation :: Ord a => [a] -> [Int]
permutation xs = map snd (sort (zip xs [0..]))
```

Code: Optimierung III der Rück-Transformation

Code.

```
--Verbesserung: Indexoperator (!!) durch Arrays ersetzt
untransform4 :: Ord a => ([a],Int) -> [a]
untransform4 (ys,k) = take n (tail (map (ya!) (iterate (pa!) k)))
                      where
                      n  = length ys
                      ya = listArray (0,n-1) ys
                      pa = listArray (0,n-1) (map snd (sort (zip ys [0..])))
```

Code: Optimierung IV der Hin-Transformation

Code.

```
--Verbesserung: Listen durch Arrays ersetzt (EOF muss im Alphabet kleiner als
--alle im Text vorkommenden Zeichen sein)
transform2 :: Ord a => a -> [a] -> ([a],Int)
transform2 eof xs = ([ xa ! (pa ! i) | i <- [0..n-1]],k)
                where
                n = length xs
                k = length (takeWhile ((/=)0) ps)
                xa = listArray (0,n-1) (rrot xs)
                pa = listArray (0,n-1) ps
                ps = map snd (sort (zip (tails (tag xs))[0..(n-1)]))
                tag xs = xs ++ [eof]
```

Weitere Optimierungen.

- Buchstaben müssen nicht in Tabellen dargestellt werden, sondern können mit *pointern* umgesetzt werden.

Danke. Fragen?