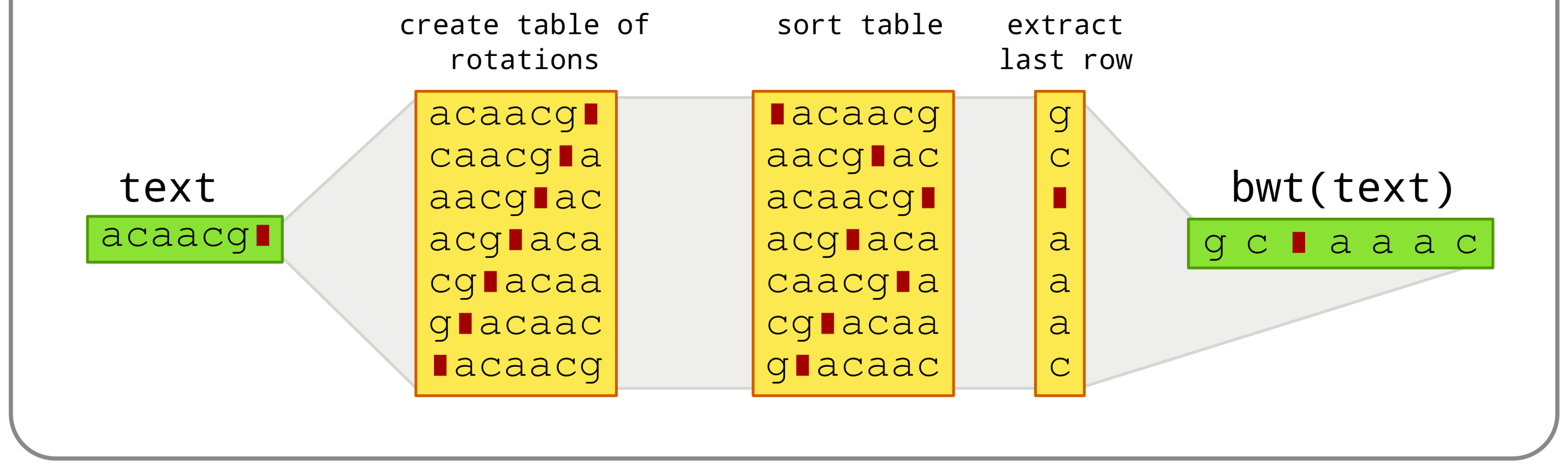
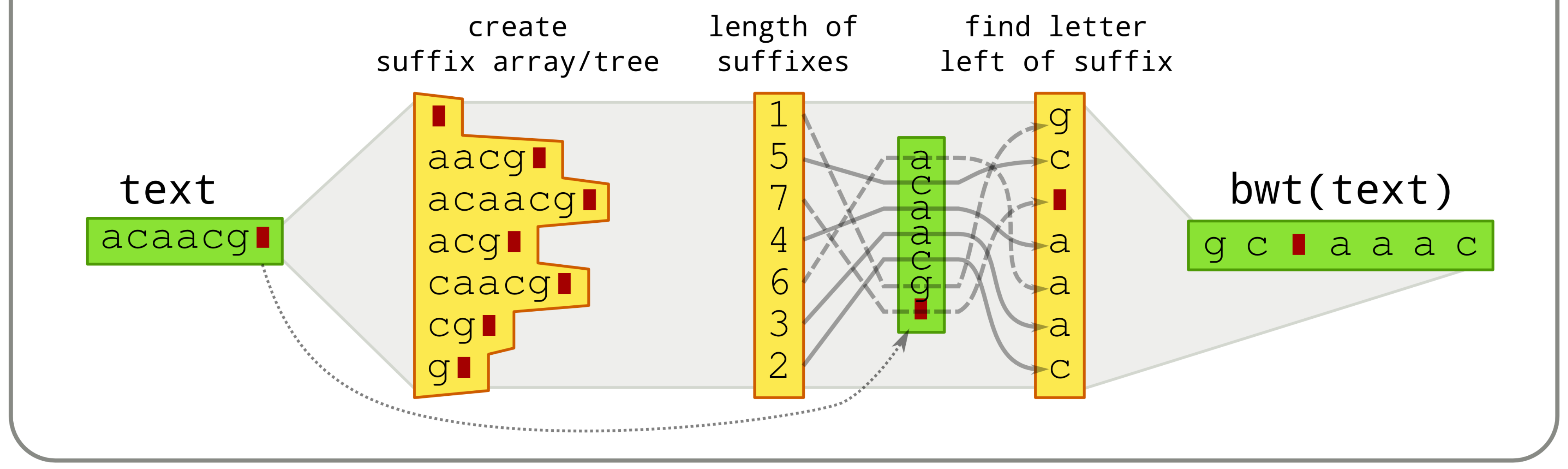
Burrows-Wheeler Transform

text -> bwt(text)

naïve

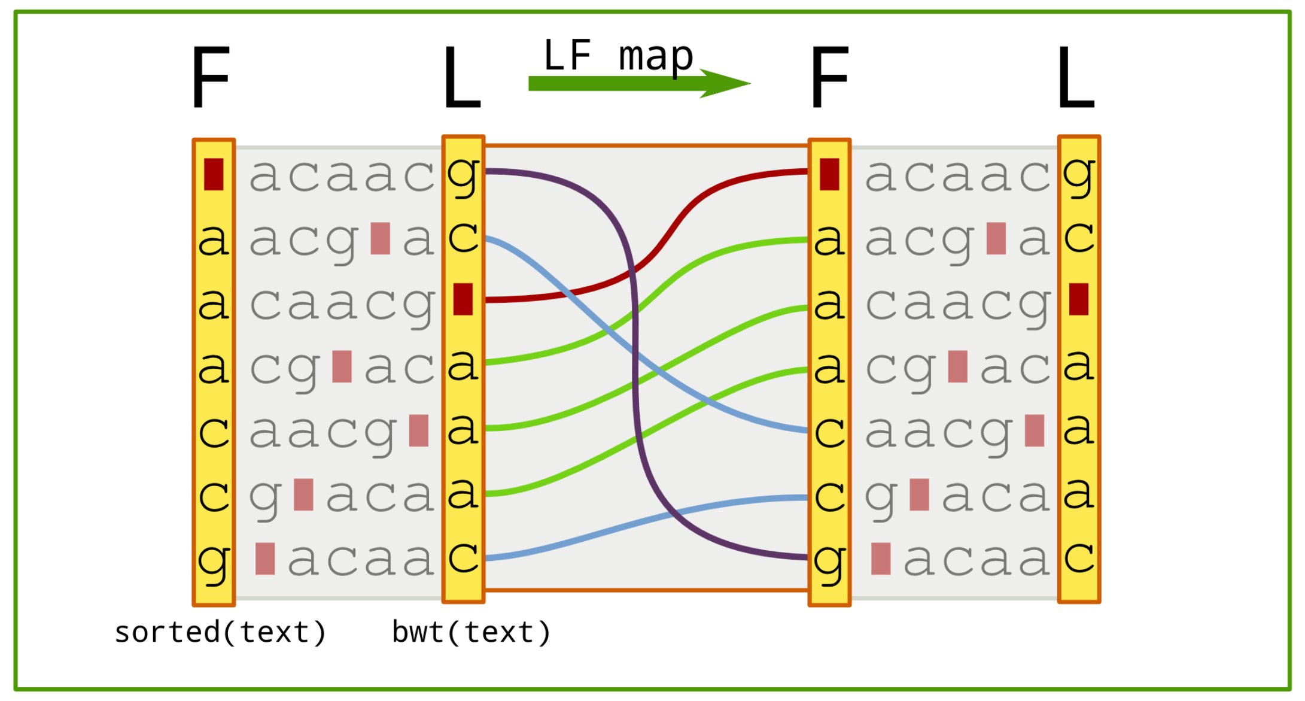


suffix array/tree



LF - mapping

shows where the row will be after a rotation



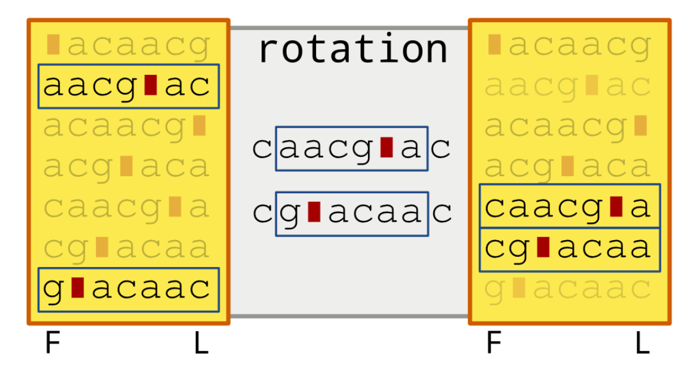
'C[c]' is the first occurance of letter 'c' in the F column

LF can be calculated with:

LF(i,c) = C[c] + Occ(c,i)

for transformation 'c = bwt[i]'

rank of letters in L and F are the same



Rows ending with 'c' are in the same order as rows starting with 'c' because:

* before rotation the rows are sorted
* after rotation rows beginning with 'c' will be sorted same way, because they were sorted without 'c'

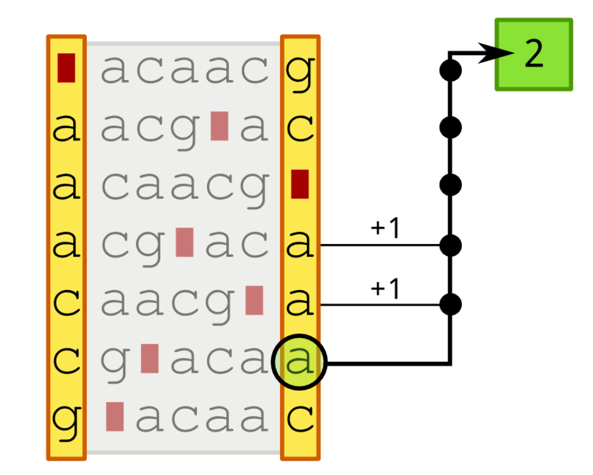
Therefore we can calculate LF, if we know letter rank for a row in L and F

'Occ(c,i)' or rank of letter 'c' at position 'i' is the count of the letter 'c' upto position 'i'.

The Occ function is the most important factor in the speed of LF mapping implementation.

Occ - naïve

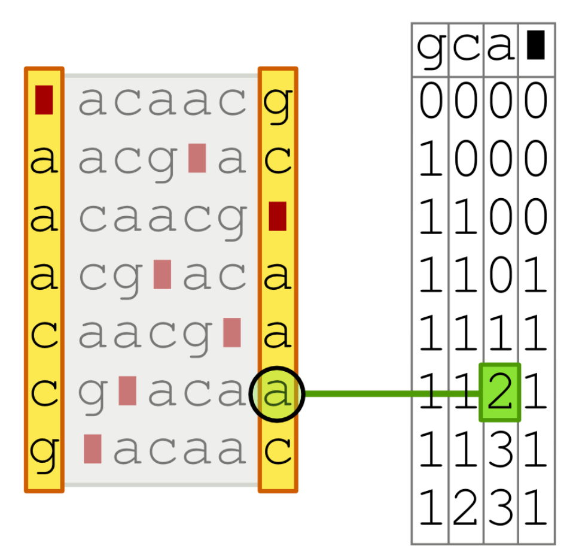
count the occurances



time - O(n)
space - O(1)

Occ - full

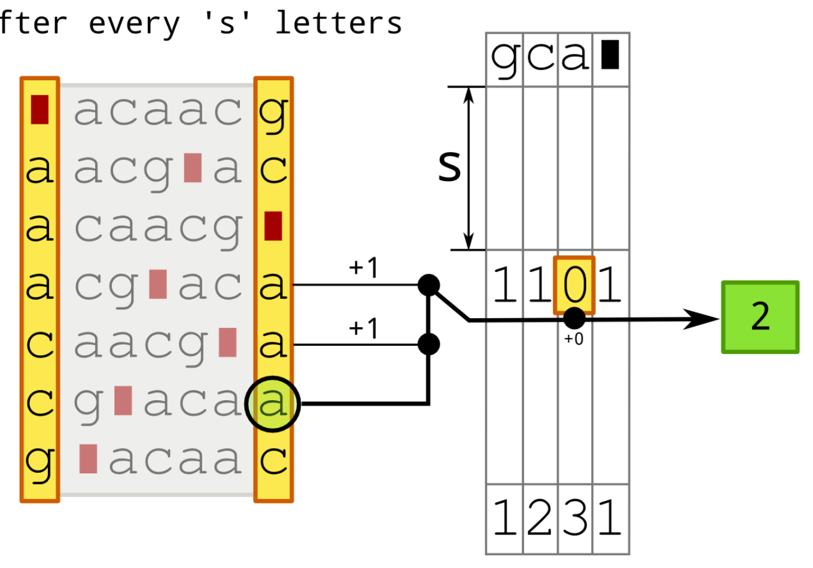
store letter count



time - O(1)
space - O(n*Σ)

Occ - checkpoints

store some letter counts
after every 's' letters



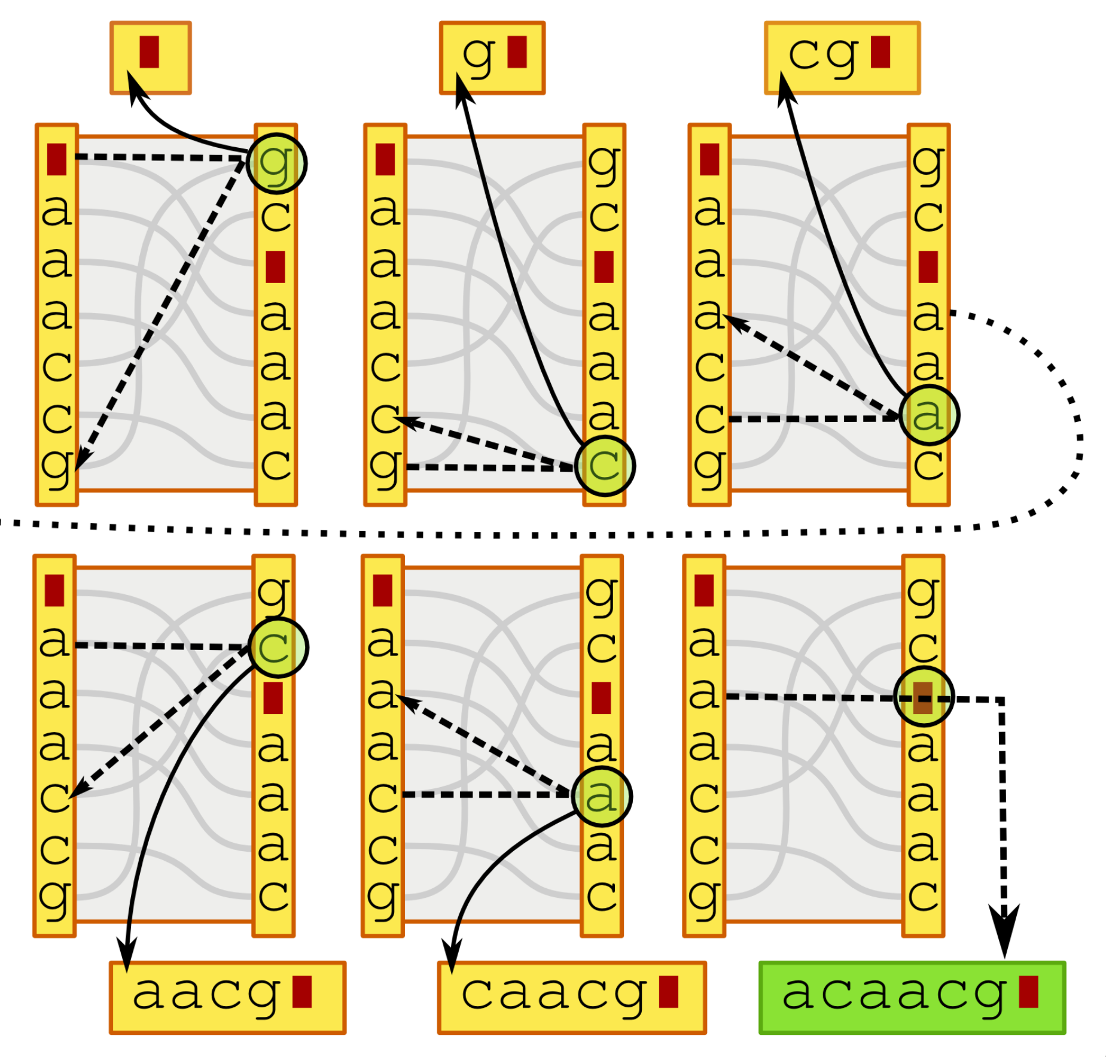
time - O(s)
space - O($\frac{n}{s}$Σ)

bwt(text) -> text

inverse

follow LF-mapping on bwt(text)

```
i = 0
T = ▮
while( BWT[i] != ▮ )
    T = BWT[i] + T
    i = LF(i)
```



FM-index

uses bwt(text) for full text search

FM-index is a memory efficient full text index

it uses bwt(text) as an index

it uses LF mapping property to search in the index

it requires memory linear to the size of the text

if compression is used then its memory use can be sublinear to the size of the text

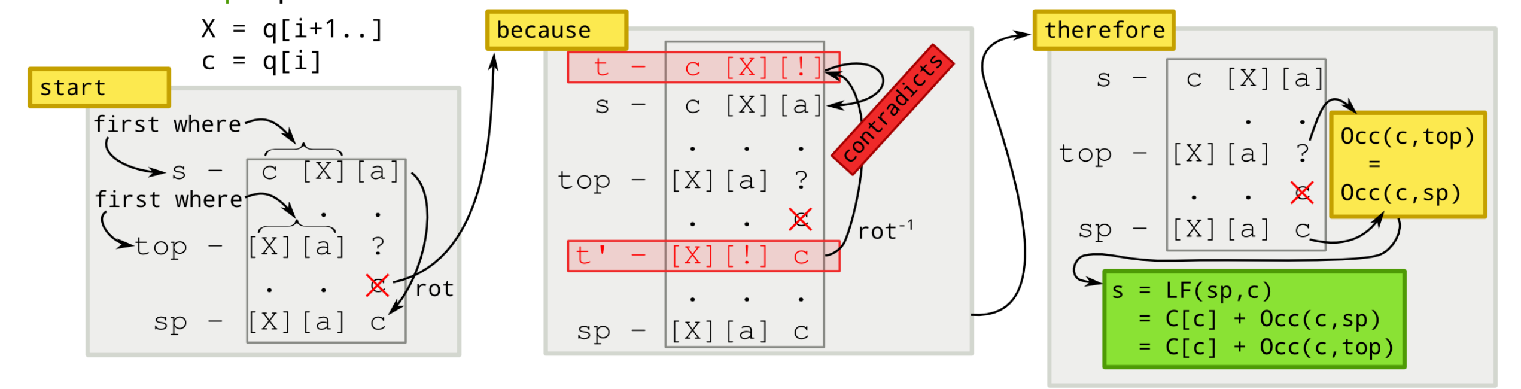
searching

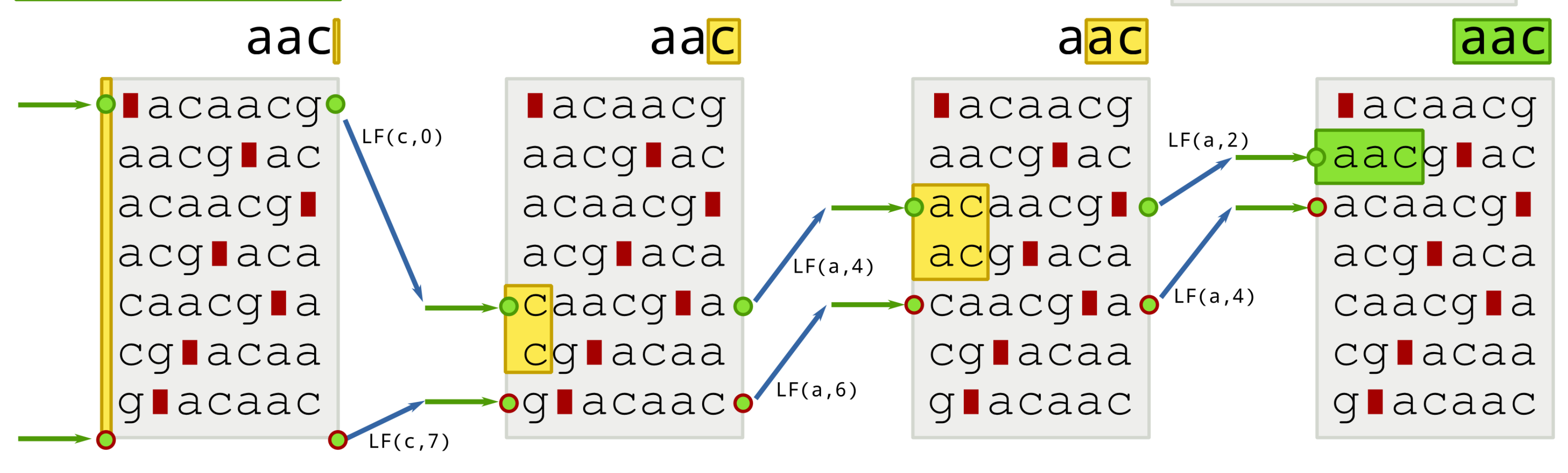
```
top = 0
bot = last + 1

for i= len(q)-1 .. 0
  c   = q[ i ]
  top = LF(top, c)
  bot = LF(bot, c)
  if ( top >= bot )
     return none
return [top..bot)
```

Proof of top update

X = q[i+1..]
c = q[i]





count

```
top, bot = search(q)
return bot - top
```

because top points to the first suffix and bot, to the last + 1

location

```
top, bot = search(q)
for i in range(top, bot)
  c = 0
  while( BWT[i] != ▮ )
    i = LF(i)
    c++
  query is at pos c
```

count how far query is from the beginning of the string using inverse bwt

similarly to naïve Occ it is slow

can be sped up with checkpointing similarly to Occ

```
while( BWT[i] != ▮ )
  if check[i]:
    c += check[i]
    break
  i = LF(i)
  c++
```