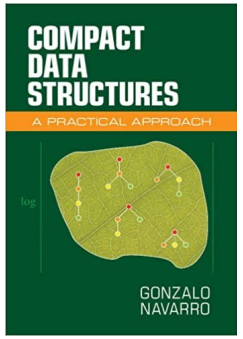
rank/ *select* functions

- given a string T, efficiently answer queries *rank*(a, i) on the number of a's in T$[1..i]$
- *rank* function (on bit vectors) turns out to be a fundamental algorithmic block for building succinct data structures [Jacobson 89]



- on bit vectors *rank* can be supported in time $O(1)$ using $o(n)$ additional *bits* of memory
- complementary function *select*(a, j): output the position of the j-th occurrence of a in T. *select* can also be supported in $O(1)$ time
- on large alphabets, *rank*/*select* can be supported in $O(\log |A|)$ time using *wavelet trees* with $O(n \log |A|)$ additional bits

Implementing *rank* on bitmaps

- consider a bitmap B of size n
- *idea*:
  - split B into small blocks
  - tabulate rank queries "inside blocks" for all possible blocks
  - store "cumulative rank" for block borders
- tabulate *rank* within all blocks of size $(\log n)/2$

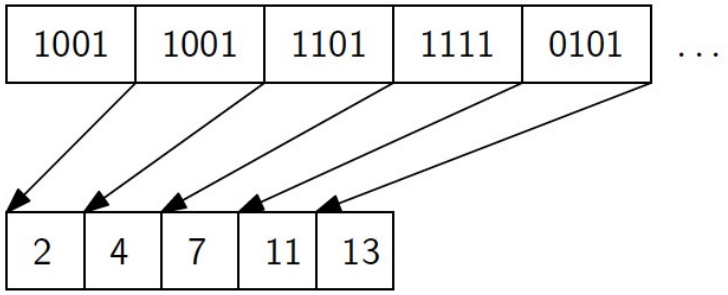
  there are $2^{(\log n)/2} = \sqrt{n}$ different blocks, and $(\log n)/2$ possible queries, with the result taking $(\log \log n)$ bits.

  Overall space: $O(\sqrt{n} \cdot \log n \cdot \log \log n) = o(n)$
- pre-compute "cumulative rank" for block borders

takes $2n/\log n \cdot \log n = 2n$ bits

too much!

trick: introduce two levels of blocks



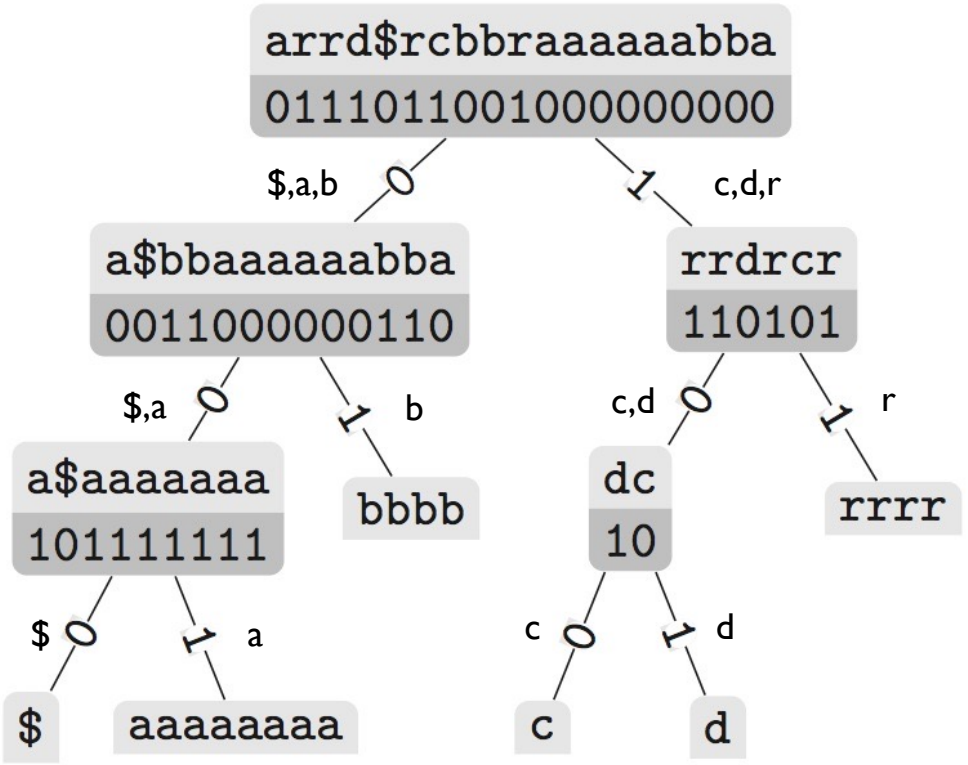
Implementing *rank* on bitmaps

- consider a bitmap B of size n
- *idea*:
  - split B into small blocks
  - tabulate rank queries "inside blocks" for all possible blocks
  - store "cumulative rank" for block borders
- tabulate *rank* within all blocks of size $(\log n)/2$

  there are $2^{(\log n)/2} = \sqrt{n}$ different blocks, and $(\log n)/2$ possible queries, with the result taking $(\log \log n)$ bits.

  Overall space: $O(\sqrt{n} \cdot \log n \cdot \log \log n) = o(n)$
- split B into $n/(\log^2 n)$ superblocks of size $(\log^2 n)$, compute cumulative rank. This takes $n/(\log^2 n) \cdot \log n = n/(\log n) = o(n)$ bits
- split each superblock into blocks of size $(\log n)/2$, compute cumulative rank *inside superblock;* the result takes $O(\log \log n)$ bits. Overall this takes $\frac{n}{\log^2 n} \cdot (2 \log n) \cdot \log \log n = o(n)$

rank for large alphabets: wavelet trees



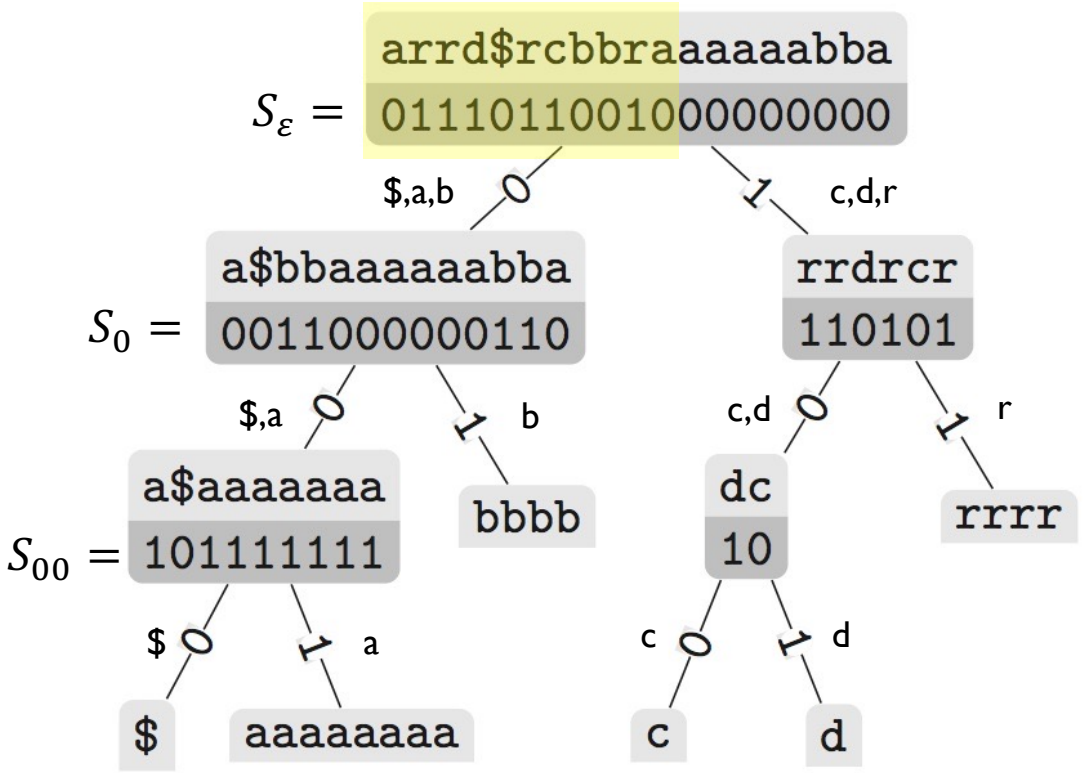
For each bitmap, compute a data structure supporting $rank_0()$ *and* $rank_1()$

Space: $n \cdot \log(|A|)$ bits

rank(a, 11)=?

a: 001

rank for large alphabets: wavelet trees



For each bitmap, compute a data structure supporting $rank_0()$ *and* $rank_1()$

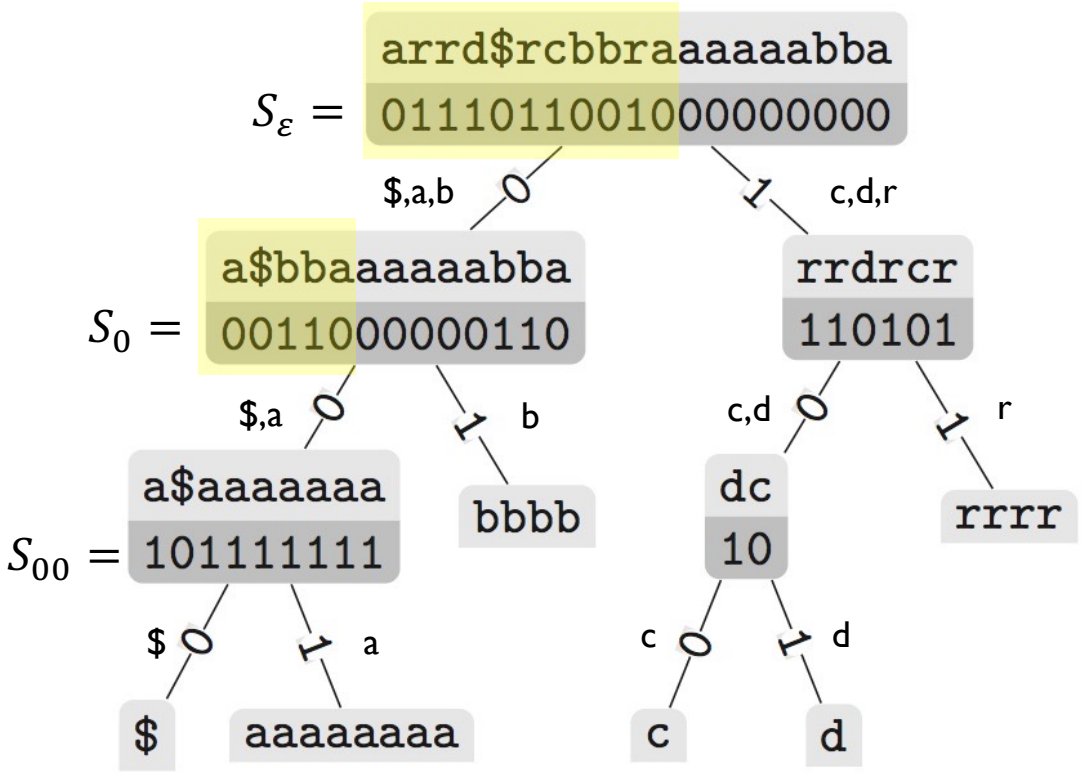
Space: $n \cdot \log(|A|)$ bits

rank(a, 11)=?

a: 001

$rank_0(11, S_\varepsilon) = 5$

rank for large alphabets: wavelet trees



For each bitmap, compute a data structure supporting $rank_0()$ *and* $rank_1()$

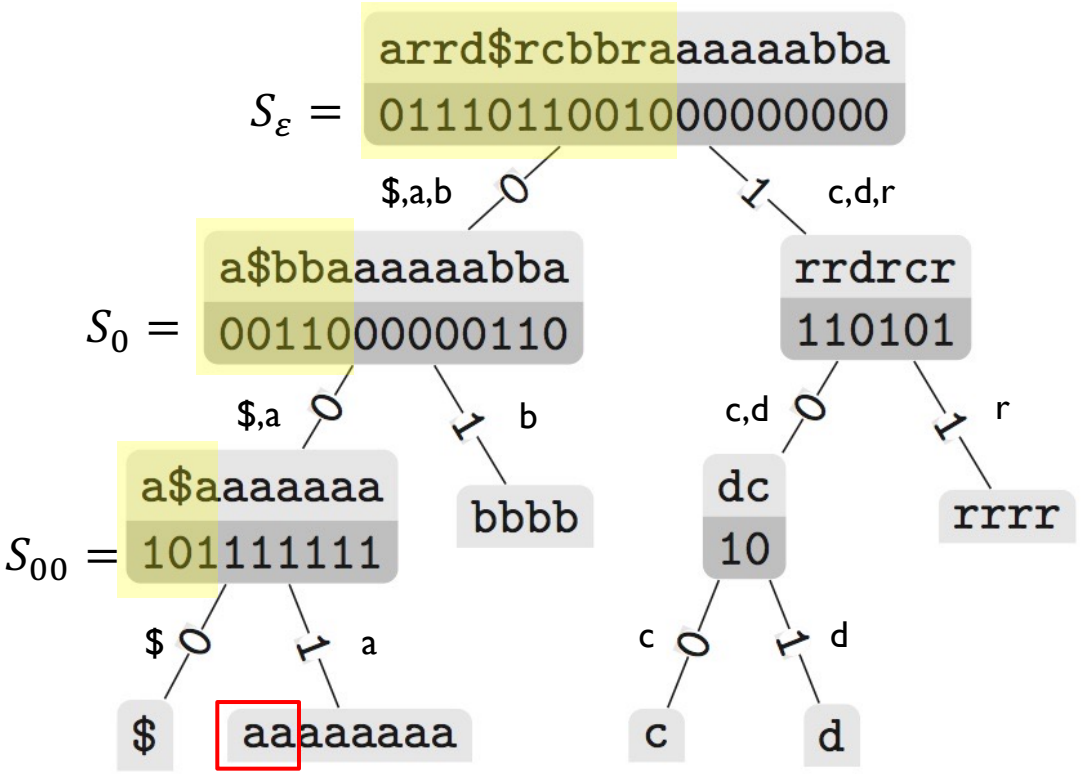
Space: $n \cdot \log(|A|)$ bits

rank(a, 11)=?

a: 001

$rank_0(11, S_\varepsilon) = 5$
$rank_0(5, S_0) = 3$

rank for large alphabets: wavelet trees



For each bitmap, compute a data structure supporting $rank_0()$ *and* $rank_1()$

Space: $n \cdot \log(|A|)$ bits

rank(a, 11)=?

a: 001

$rank_0(11, S_\varepsilon) = 5$
$rank_0(5, S_0) = 3$
$rank_1(3, S_{00}) = 2$

rank is computed in $O(\log(|A|))$ time

String matching with BWT

T=acatacagatg$

P=taca

T[SA[i]]		BWT
$	acatacagat	g
a	cagatg$aca	t
a	catacagatg	$
a	gatg$acata	c
a	tacagatg$a	c
a	tg$acataca	g
c	agatg$acat	a
c	atacagatg$	a
g	$acatacaga	t
g	atg$acatac	a
t	acagatg$ac	a
t	g$acatacag	a
F		L

String matching with BWT

T=acatacagatg$

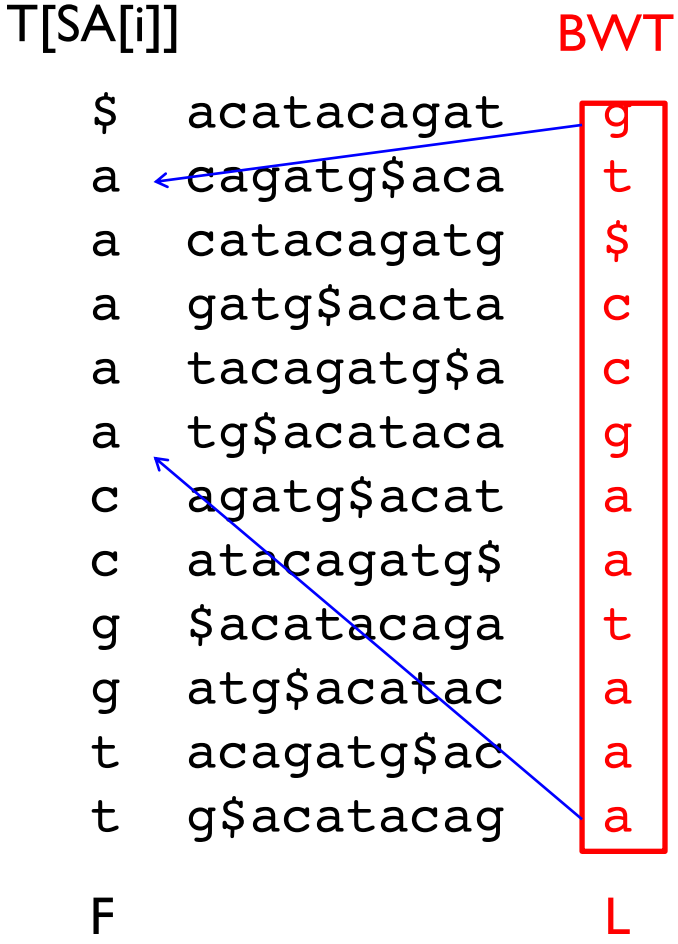
P=taca

T[SA[i]]		BWT
$	acatacagat	g
a	cagatg$aca	t
a	catacagatg	$
a	gatg$acata	c
a	tacagatg$a	c
a	tg$acataca	g
c	agatg$acat	a
c	atacagatg$	a
g	$acatacaga	t
g	atg$acatac	a
t	acagatg$ac	a
t	g$acatacag	a
F		L

String matching with BWT

T=acatacagatg$

P=taca



String matching with BWT

T=acatacagatg$

P=tac**a**

T[SA[i]]		BWT	
$	acatacagat	g	
a	cagatg$aca	t	e
a	catacagatg	$	
a	gatg$acata	c	
a	tacagatg$a	c	
a	tg$acataca	g	f
c	agatg$acat	a	
c	atacagatg$	a	
g	$acatacaga	t	
g	atg$acatac	a	
t	acagatg$ac	a	
t	g$acatacag	a	
F		L	

[e,f] : current interval

x : letter

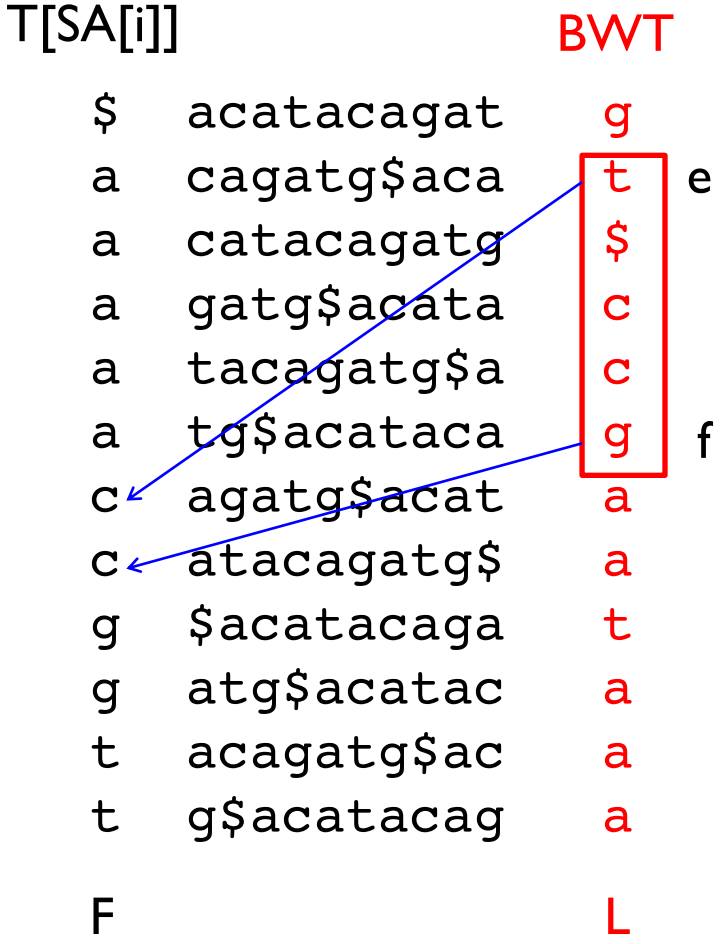
e:= C[x]+*rank*[x,e]+1

f:= C[x]+*rank*[x,f]

String matching with BWT

T=acatacagatg$

P=taca



[e,f] : current interval

x : letter

e:= C[x]+*rank*[x,e]+1

f:= C[x]+*rank*[x,f]

String matching with BWT

T=acatacagatg$

P=taca

T[SA[i]]		BWT	
$	acatacagat	g	
a	cagatg$aca	t	
a	catacagatg	$	
a	gatg$acata	c	
a	tacagatg$a	c	
a	tg$acataca	g	
c	agatg$acat	a	e
c	atacagatg$	a	f
g	$acatacaga	t	
g	atg$acatac	a	
t	acagatg$ac	a	
t	g$acatacag	a	
F		L	

[e,f] : current interval

x : letter

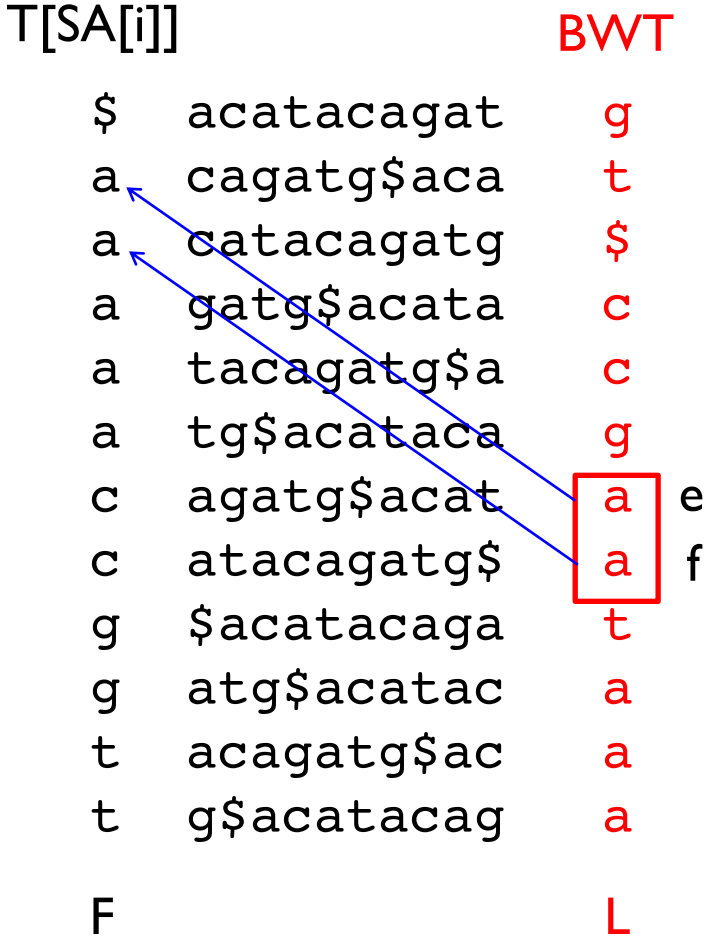
e:= C[x]+*rank*[x,e]+1

f:= C[x]+*rank*[x,f]

String matching with BWT

T=acatacagatg$

P=taca



[e,f] : current interval

x : letter

e:= C[x]+*rank*[x,e]+1

f:= C[x]+*rank*[x,f]

String matching with BWT

T=acatacagatg$

P=taca

T[SA[i]]		BWT	
$	acatacagat	g	
a	cagatg$aca	t	e
a	catacagatg	$	f
a	gatg$acata	c	
a	tacagatg$a	c	
a	tg$acataca	g	
c	agatg$acat	a	
c	atacagatg$	a	
g	$acatacaga	t	
g	atg$acatac	a	
t	acagatg$ac	a	
t	g$acatacag	a	
F		L	

[e,f] : current interval

x : letter

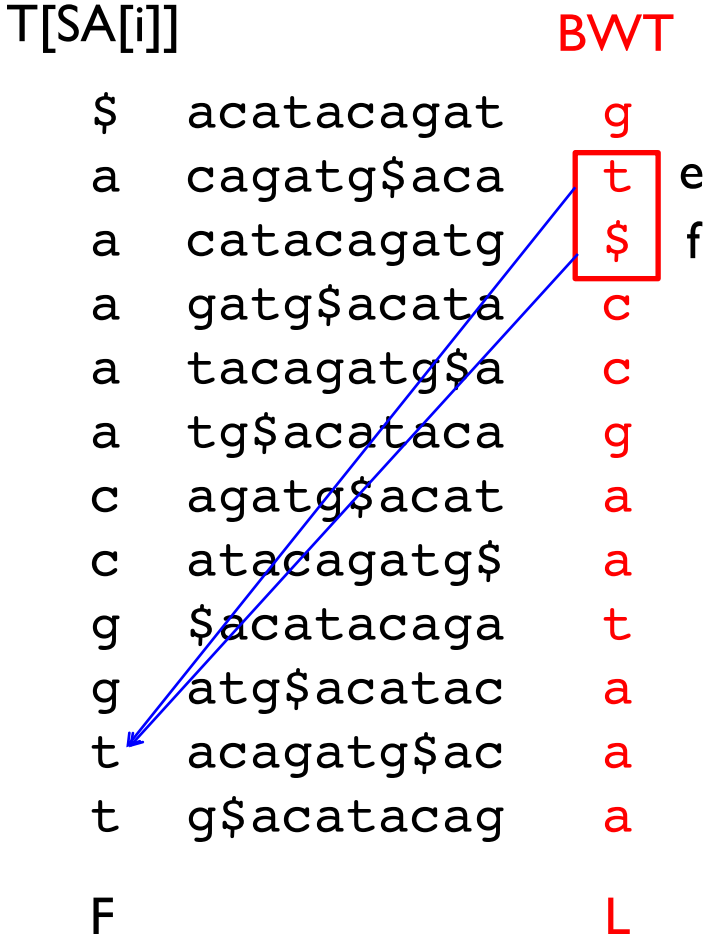
e:= C[x]+*rank*[x,e]+1

f:= C[x]+*rank*[x,f]

String matching with BWT

T=acatacagatg$

P=taca



[e,f] : current interval

x : letter

e:= C[x]+*rank*[x,e]+1

f:= C[x]+*rank*[x,f]

String matching with BWT

T=acatacagatg$

P=taca

T[SA[i]]		BWT
$	acatacagat	g
a	cagatg$aca	t
a	catacagatg	$
a	gatg$acata	c
a	tacagatg$a	c
a	tg$acataca	g
c	agatg$acat	a
c	atacagatg$	a
g	$acatacaga	t
g	atg$acatac	a
t	acagatg$ac	a
t	g$acatacag	a
F		L

S : current string (pattern suffix)

[e,f] : current interval

x : letter

compute new interval for xS

e:= C[x]+*rank*[x,e]+1

f:= C[x]+*rank*[x,f]

String matching with BWT

T=acatacagatg$

P=taca

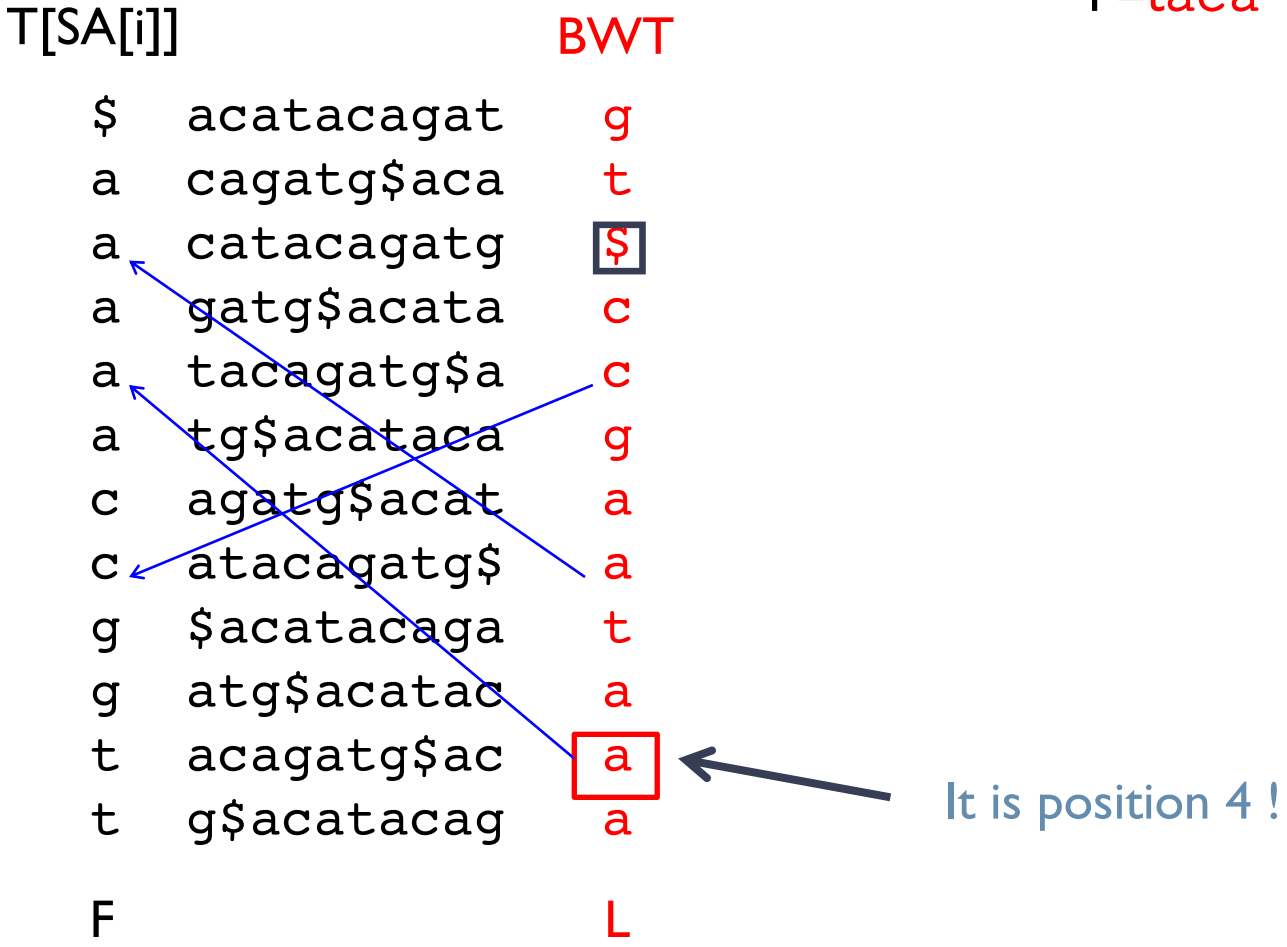
T[SA[i]]		BWT
$	acatacagat	g
a	cagatg$aca	t
a	catacagatg	$
a	gatg$acata	c
a	tacagatg$a	c
a	tg$acataca	g
c	agatg$acat	a
c	atacagatg$	a
g	$acatacaga	t
g	atg$acatac	a
t	acagatg$ac	a
t	g$acatacag	a
F		L

What position is it??

String matching with BWT

T=acatacagatg$

P=taca



BWT-index (FM-index)

T=acatacagatg$

SA	T[SA[i]]		BWT
12	$	acatacagat	g
5	a	cagatg$aca	t
1	a	catacagatg	$
7	a	gatg$acata	c
3	a	tacagatg$a	c
9	a	tg$acataca	g
6	c	agatg$acat	a
2	c	atacagatg$	a
11	g	$acatacaga	t
8	g	atg$acatac	a
4	t	acagatg$ac	a
10	t	g$acatacag	a
	F		L

Solution: store only a fraction of values of SA.

Storing one value over log(n) leads to $O(n \cdot \log(n)/\log(n))=O(n)$ *bits*

BWT-index (FM-index)

T=acatacagatg$

SA	T[SA[i]]		BWT
12	$	acatacagat	g
	a	cagatg$aca	t
	a	catacagatg	$
	a	gatg$acata	c
	a	tacagatg$a	c
	a	tg$acataca	g
6	c	agatg$acat	a
2	c	atacagatg$	a
	g	$acatacaga	t
8	g	atg$acatac	a
4	t	acagatg$ac	a
10	t	g$acatacag	a
	F		L

Solution: store only a fraction of values of SA.

Storing one value over log(n) leads to $O(n \cdot \log(n)/\log(n))=O(n)$ *bits*

Search time becomes $O(|P|+occ \cdot \log(n))$

[Ferragina, Manzini 00]

FM-index includes:

- BWT
- selection of SA values
- auxiliary structures: array C, *rank*, position marking …

BWT-index: practical issues

- BWT-index can be implemented using ~3 bits/char (!!)
- BWT-index is now *widely* used in practical bioinformatics software: `BWA`, `bowtie`, `SOAP2` (mapping), `CGA` (assembly)
- `BWA` and `bowtie` mentioned in Milestones in Genomic Sequencing (Nature, Feb 2021)
- Variants: *r*-index, bi-directional BWT-index
- other succinct data structure exist (including *compact suffix array*) and continue to appear
- construction may require much more space than the resulting structure
- external memory algorithms, algorithms specialized to multi-core or GPU processor architectures
- dynamic indexes