High-throughput read mapping with Burrows-Wheeler indexes

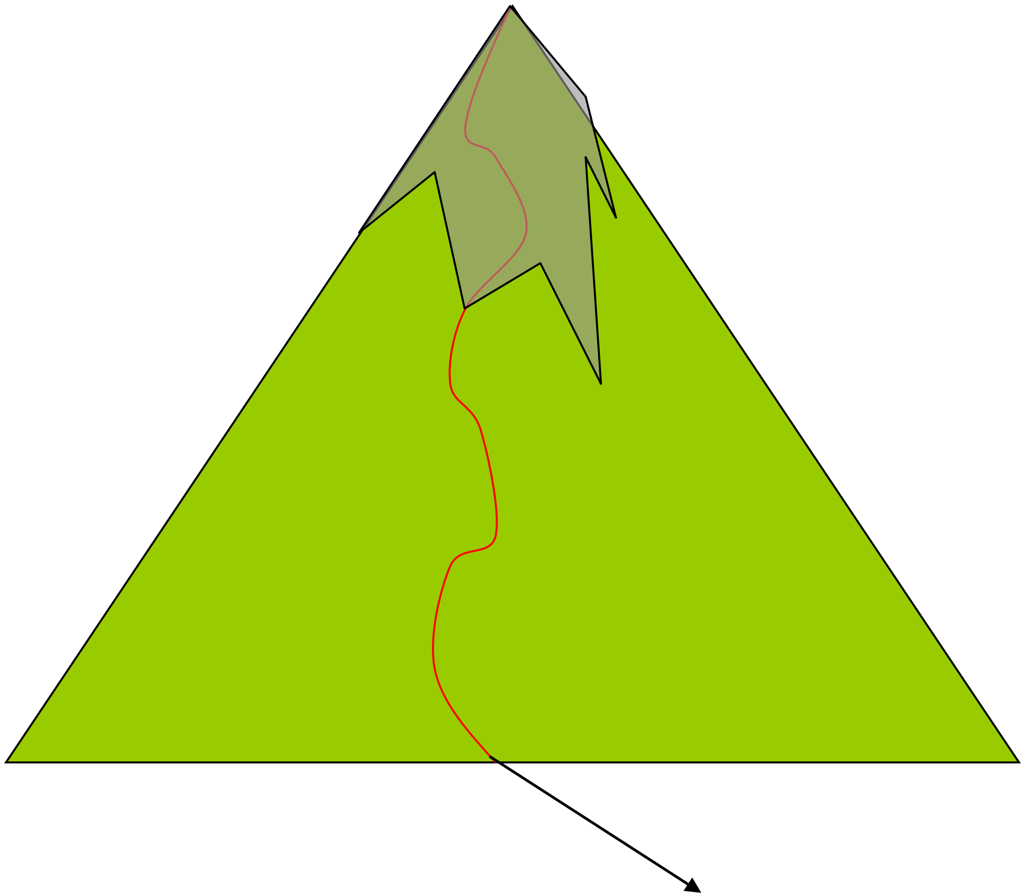
Veli Mäkinen

Biological Sequence Analysis, Spring 2013
Lecture 8

Read mapping

- Input: Short reads extracted from donor DNA.
- Output: Alignment of the reads to their locations in reference genome.
- Some errors (but not many) need to be allowed in the mapping.

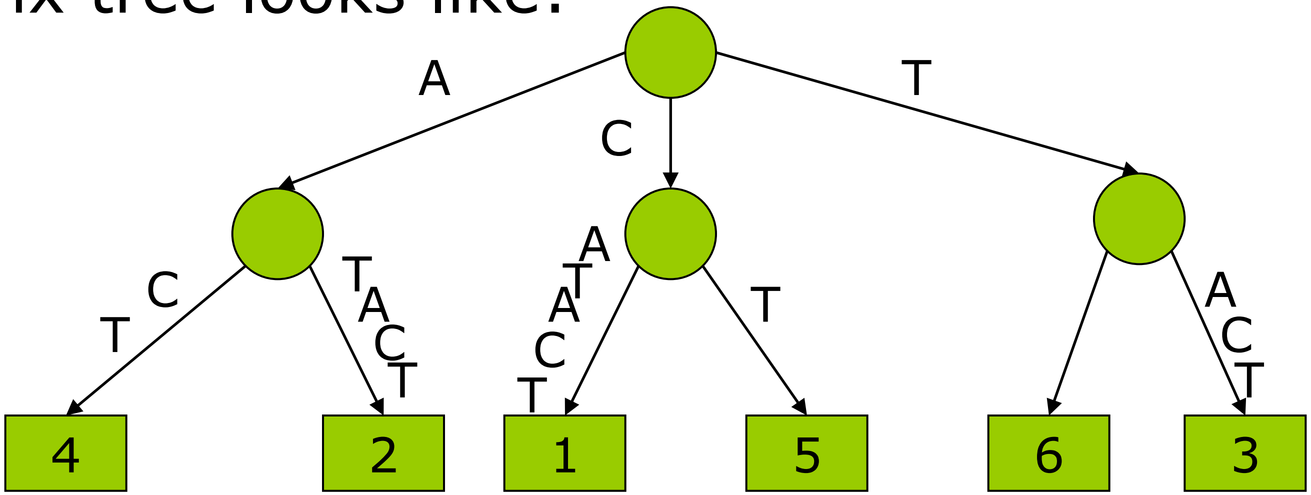
Solution: backtracking with suffix tree



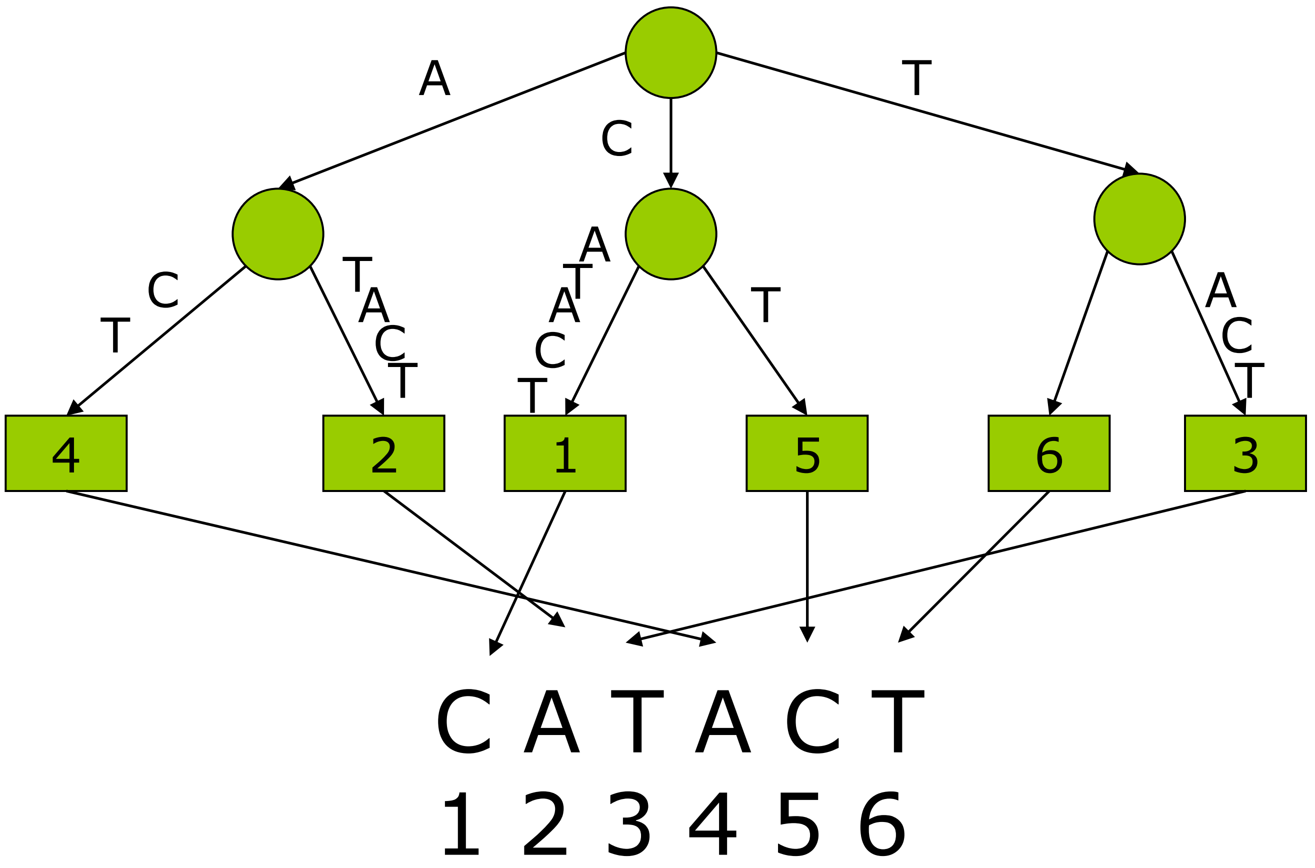
...ACACATTATCACAGGCATCGGCATTAGCGATCGAGTCG.....

Suffix tree

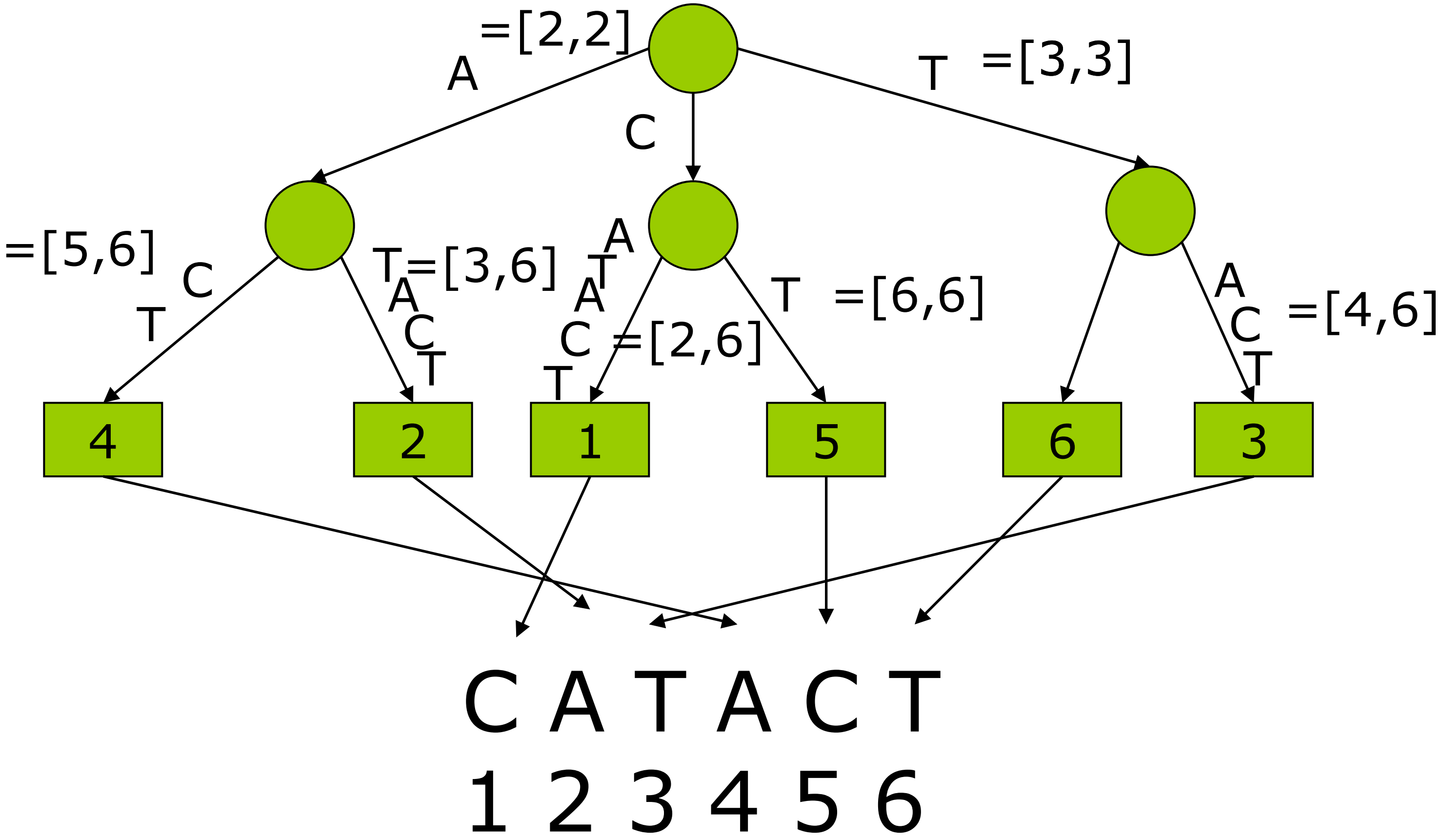
- Suffix tree is a compressed keyword trie of all *suffixes* of a sequence
- E.g. suffixes of sequence CATACT are CATACT, ATACT, TACT, ACT, CT, T.
  - suffix tree looks like:



Suffix tree

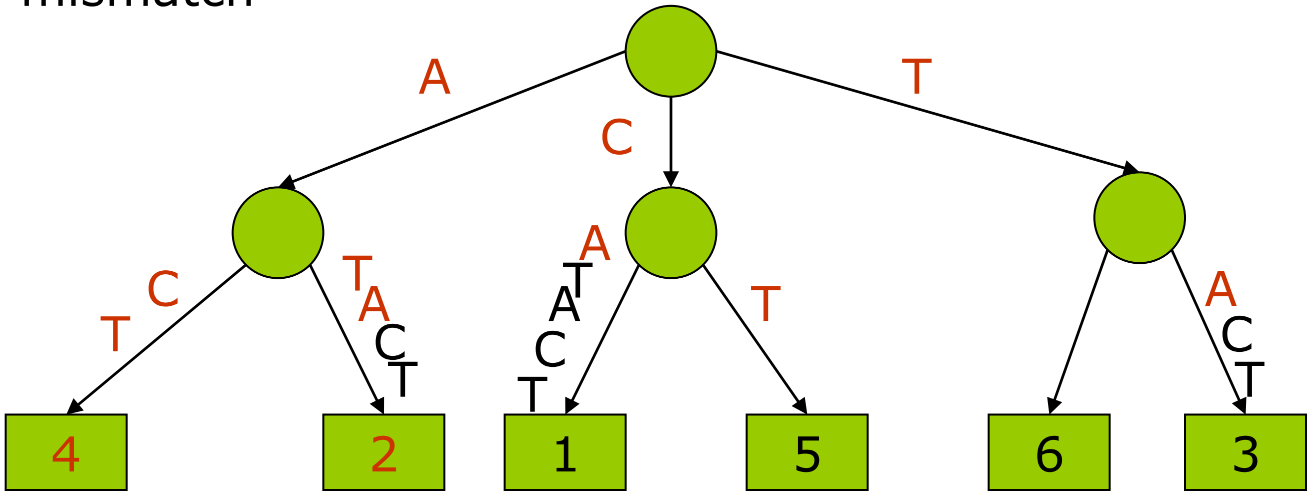


Suffix tree


A
=[2,2]
C
T
=[3,3]
C
T
=[5,6]
T
A
C
T
=[3,6]
A
T
A
C
T
=[2,6]
T
=[6,6]
A
C
T
=[4,6]
4
2
1
5
6
3
C A T A C T
1 2 3 4 5 6

Back to backtracking

ACA, 1 mismatch



Same idea can be used to many other forms of approximate search, like Smith-Waterman, position-restricted scoring matrices, regular expression search, etc.

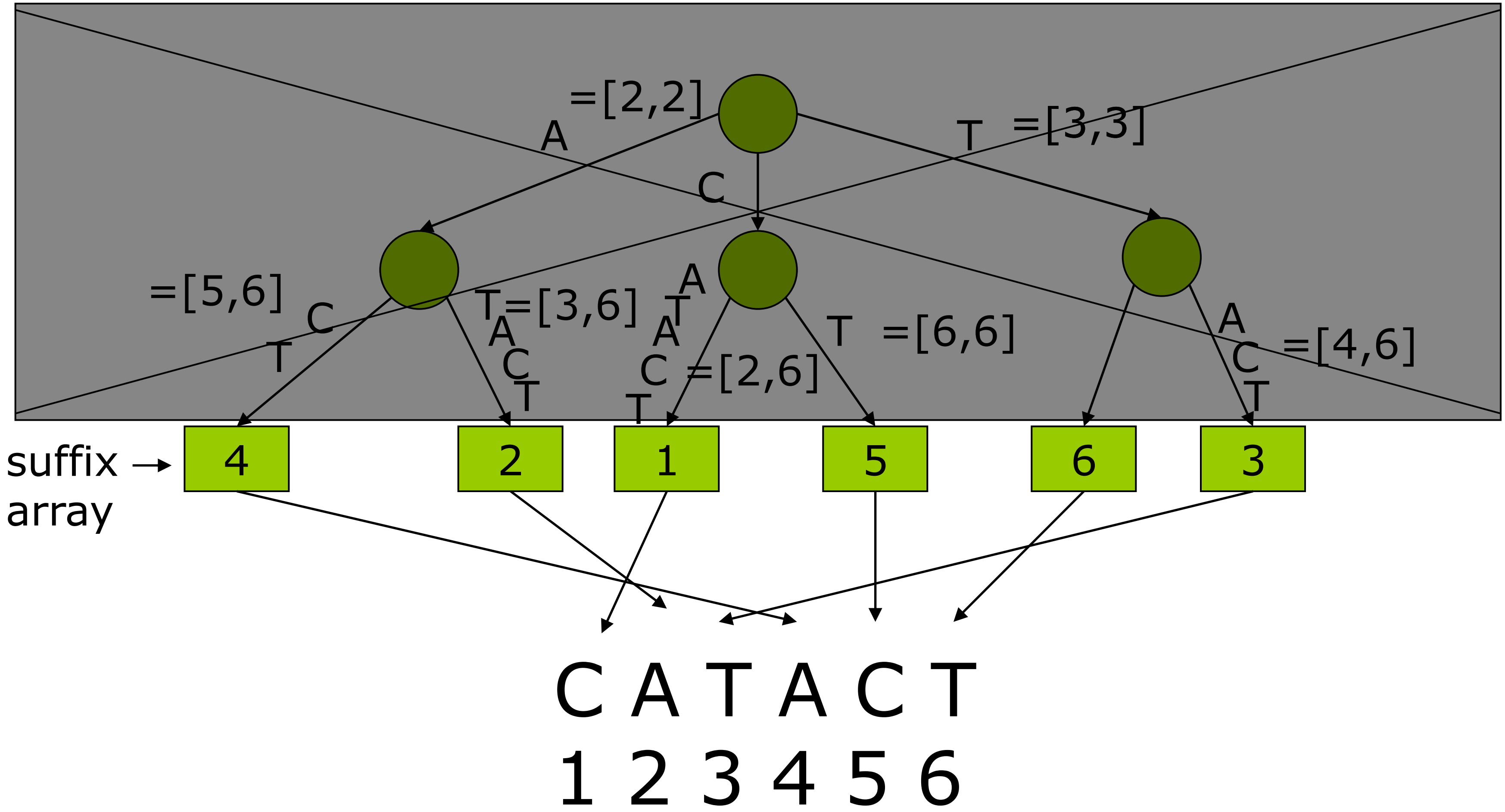
Properties of suffix tree

- Suffix tree has n leaves and at most n-1 internal nodes, where n is the total length of all sequences indexed.
- Each node requires constant number of integers (pointers to first child, sibling, parent, text range of incoming edge, statistics counters, etc.).
- Can be constructed in linear time.

Properties of suffix tree... in practice

- Huge overhead due to pointer structure:
  - Standard implementation of suffix tree for human genome requires over 200 GB memory!
  - A careful implementation (using log n -bit fields for each value and array layout for the tree) still requires over 40 GB.
  - Human genome itself takes less than 1 GB using 2-bits per bp.

Reducing space: suffix array



suffix array →

C A T A C T

1 2 3 4 5 6

Suffix array

- Many algorithms on suffix tree can be simulated using *suffix array*...
  - ... and couple of additional arrays...
  - ... forming so-called *enhanced suffix array*...
  - ... leading to the similar space requirement as careful implementation of suffix tree
- Not a satisfactory solution to the space issue.

What we learn today?

- We learn that *backtracking* can be done using *compressed suffix arrays* requiring only 2.1 GB for the human genome.

Burrows-Wheeler transform (BWT)

- Compute a matrix M whose rows are cyclic shifts of sequence $S=s_1s_2..s_n$: $s_1s_2..s_n$, $s_2s_3..s_ns_1$, $s_3s_4..\ s_ns_1s_2$, … , $s_{n-1}s_n..\ s_{n-3}s_{n-2}$, $s_ns_1..\ s_{n-2}s_{n-1}$.
- Sort the rows in the lexicographic order in M.
- Let L be the last column and F the first column of M.
- bwt(T)=(L,i), where i is the row number in M containing $s_1s_2..s_n=S$.

BWT vs. suffix array

- The lexicographic order of the cyclic shifts of S is essentially *suffix array* sa(S).

sa F M L

1:7 #CATACT
2:4 ACT#CAT
3:2 ATACT#C
4:1 CATACT#
5:5 CT#CATA
6:6 T#CATAC
7:3 TACT#CA

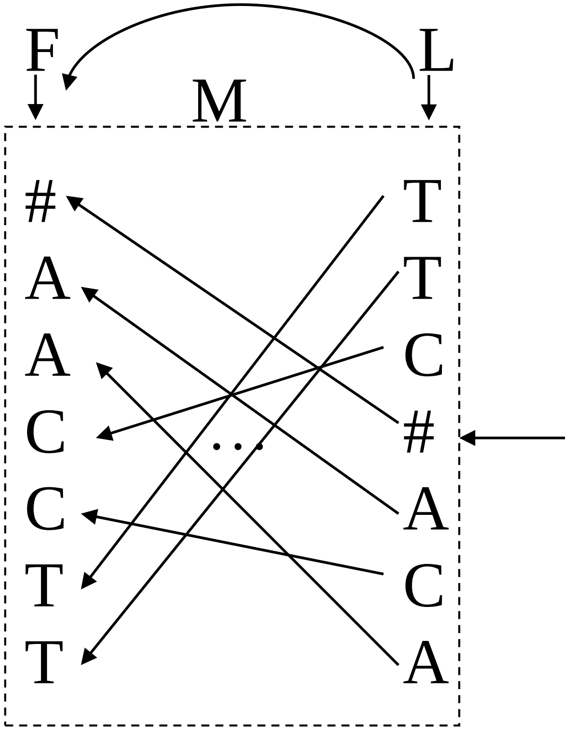
1234567
S = CATACT#

BWT=
(L = TTC#ACA, row 4)

Exercise: Given L and the row number, how to compute S and sa(S)?

S^{-1}= # TCATAC

stable sort



F

L

M

F		L
#		T
A		T
A		C
C	. . .	#
C		A
T		C
T		A

sa(S)

1: 7

2: 4

3: 2

4: 1

5: 5

6: 6

7: 3

i	1	2	3	4	5	6	7
LF[i]	6	7	4	1	2	5	3

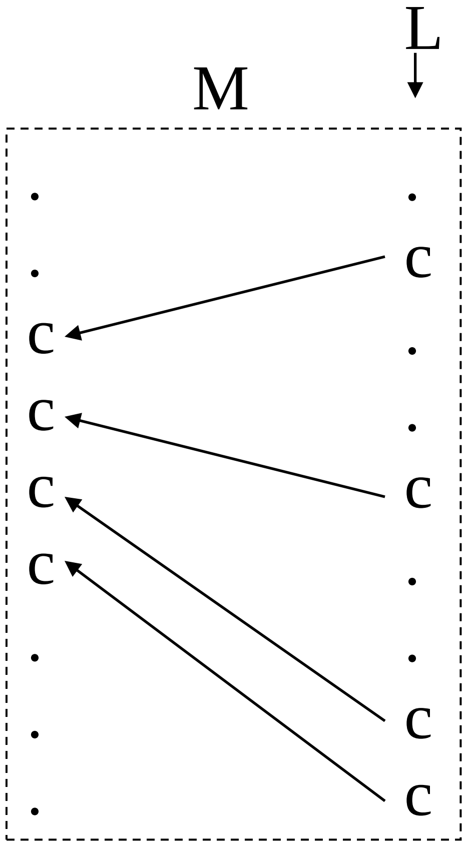
LF-mapping

- Let the i-*th* row of M contain cyclic shift fXl, and j-*th* row cyclic shift lfX.
- LF[i] =j.
- Hence, L[i] L[LF[i]] L[LF[LF[i]]]… gives the original text in reverse order, where L[1,n] is the transformed text.
- **Exercise:** Why the previous sorting algorithm to compute LF-mapping works correctly?

LF-mapping...

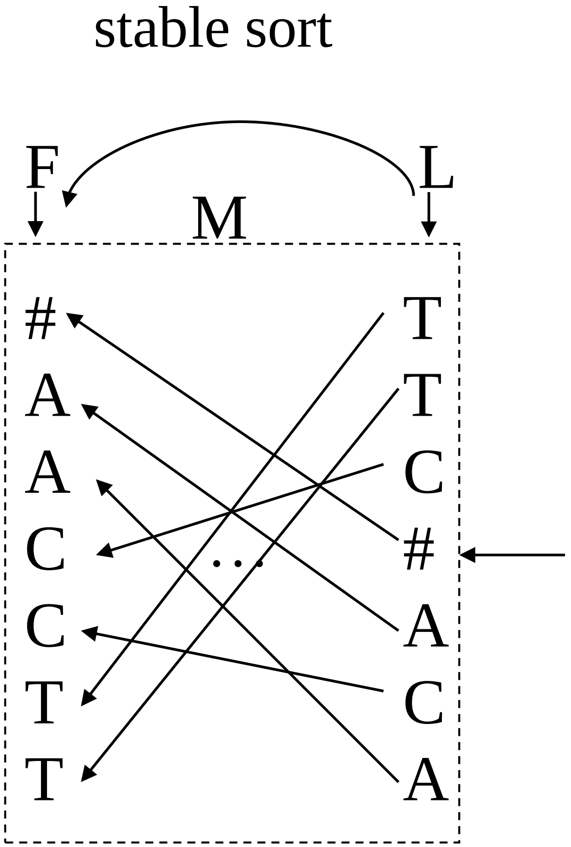
- Let $C[c]$ be the amount of symbols smaller than c in T, $c \in \{1,2,\ldots,\sigma\}$.
- **Lemma 1**: $LF[i] \in [\ C[L[i]]+1, C[L[i]+1]\ ]$
- Let $rank_c(L,i)$ be the amount of symbols c in the prefix $L[1,i]$.
- **Lemma 2**: $LF[i] = C[L[i]]+rank_{L[i]}(L,i)$.
- **Lemma 3**: When L is stable sorted into L', then $L[i]$ is mapped to $L'[LF[i]]$.

Proving Lemmas 2 and 3



- Let $Xc < Yc$. Then $X < Y$ and so is $cX < cY$.
- Let $M[i] = Yc$, $c=L[i]$.
- Let $J = \{j \mid M[j]=Xc\}$.
- $LF[i] = C[c] + |\{j \mid j \in J, j \leq i\}|$
  $= C[c] + \mathrm{Rank}_c(L,i)$.
- It is easy to see that sorting $\{(L[i],i)\}_i$ gives the same mapping.

S^{-1}= # TCATAC



sa(S)

1:	7
2:	4
3:	2
4:	1
5:	5
6:	6
7:	3

i	1	2	3	4	5	6	7
LF[i]	6	7	4	1	2	5	3

$LF(6)=C[C]+rank_C(L,6)$

$=3+2=5$

Suffix array & BWT construction

- One solution is to first build suffix tree using e.g. McCreight‘s or Ukkonen‘s suffix tree construction algorithm and then read suffix array from its leaves. This takes time $O(n \log \sigma)$.
- There are also new direct constructions for suffix arrays that take linear time, when $\sigma = O(n)$.
- BW-transform L is then given by L[i]=S[sa[i]-1], where S[0]=S[n].

Rank function

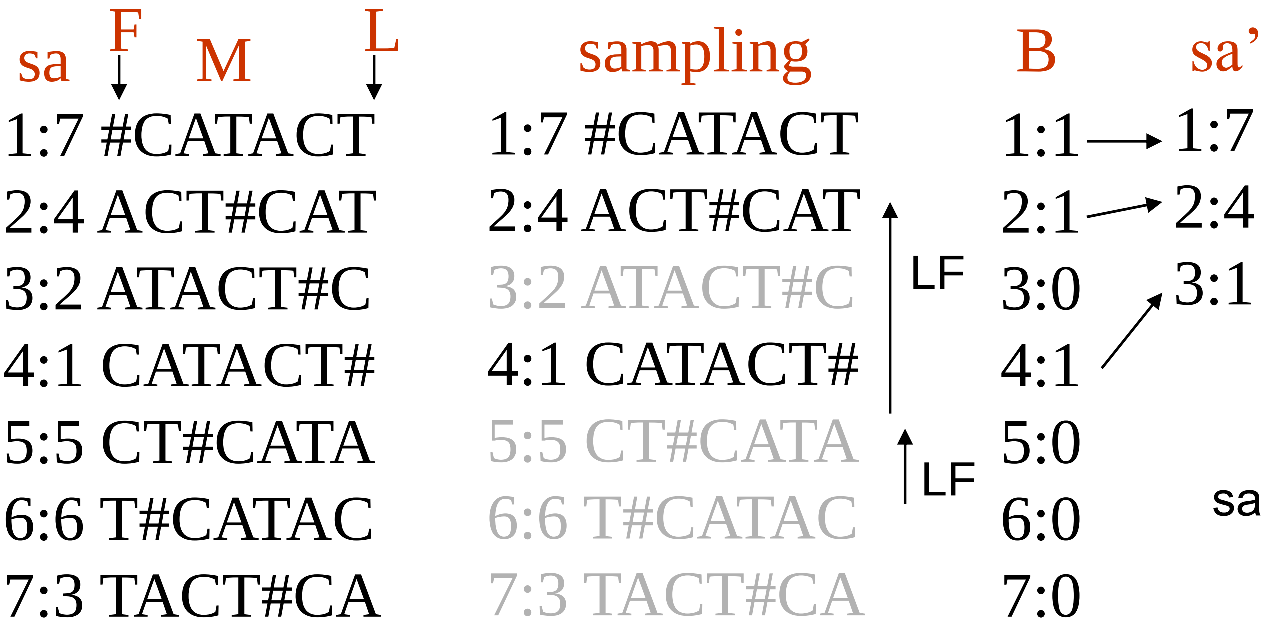
- **Lemma 4**. Given a bitvector B[1,n], there is a data structure occupying o(n) bits that supports $rank_1(B,i)$ and $rank_0(B,i)=i-rank_1(B,i)$ in constant time.
- **Lemma 5**. Sequence L[1,n] can be replaced by a data structure (called *wavelet tree*) occupying $n \log \sigma (1+o(1))$ bits and supporting $rank_c(L,i)$ for all $c \in \Sigma$ in $O(\log \sigma)$ time.
- *Proofs*. See end of these lecture slides.

Compressed suffix array

- Suffix array sa(S) occupies $|S| \log |S|$ bits.
- Next we will develop a *compressed suffix array* csa(S), which occupies $2|S|+\sigma \log |S|+|S| \log \sigma(1+o(1))$ bits, and simulates SA[i] computation in $O(\log \sigma \log |S|)$ time.
- Idea:
  - Store only every $\log n$:th suffix array value.
  - Use LF-mapping locally to find the nearest sampled value.

Compressed suffix array

log n = 3



sa F M L	sampling	B	sa’
1:7 #CATACT	1:7 #CATACT	1:1	1:7
2:4 ACT#CAT	2:4 ACT#CAT	2:1	2:4
3:2 ATACT#C	3:2 ATACT#C	3:0	3:1
4:1 CATACT#	4:1 CATACT#	4:1	
5:5 CT#CATA	5:5 CT#CATA	5:0	
6:6 T#CATAC	6:6 T#CATAC	6:0	
7:3 TACT#CA	7:3 TACT#CA	7:0	

sa[4]=sa’[rank(B,4)]

SA[6]=SA[5]+1= SA[2]+2=sa’[rank(B,2)]+2=4+2=6

Compressed suffix array

- For LF-mapping table $C[1,\sigma]$ and wavelet tree of BW-transform are enough:
  - $LF[i]=C[L[i]]+rank_{L[i]}(L,i)$
  - Space $\sigma \log |S|+|S| \log \sigma(1+o(1))$ bits.
  - $LF[i]$ computation takes time $O(\log \sigma)$.
- In addition, the bitvector B takes $|S|+o(|S|)$ bits, as it needs to support constant time rank.
- sa' takes $(|S|/\log |S|)\log |S|=|S|$ bits.
- Computation of one $SA[i]$ value requires at most $\log |S|$ LF-mappings, so the overall time is $O(\log |S| \log \sigma)$.

Backward search

Search for ”ala”

i	SA[i]	suffix $T_{SA[i],n}$
1:	21	$alabar_a_la_alabarda
2:	7	_a_la_alabarda$alabar
3:	12	_alabarda$alabar_a_la
4:	9	_la_alabarda$alabar_a
5:	20	a$alabar_a_la_alabard
6:	11	a_alabarda$alabar_a_l
7:	8	a_la_alabarda$alabar_
8:	3	abar_a_la_alabarda$al
9:	15	abarda$alabar_a_la_al
10:	1	alabar_a_la_alabarda$
11:	13	alabarda$alabar_a_la_
12:	5	ar_a_la_alabarda$alab
13:	17	arda$alabar_a_la_alab
14:	4	bar_a_la_alabarda$ala
15:	16	barda$alabar_a_la_ala
16:	19	da$alabar_a_la_alabar
17:	10	la_alabarda$alabar_a_
18:	2	labar_a_la_alabarda$a
19:	14	labarda$alabar_a_la_a
20:	6	r_a_la_alabarda$alaba
21:	18	rda$alabar_a_la_alaba

Step 1: search for ”a”

i	SA[i]	L	suffix $T_{SA[i],n}$
1:	21	a	$alabar_a_la_alabarda
2:	7	r	_a_la_alabarda$alabar
3:	12	a	_alabarda$alabar_a_la
4:	9	a	_la_alabarda$alabar_a
5:	20	d	a$alabar_a_la_alabard
6:	11	l	a_alabarda$alabar_a_l
7:	8	_	a_la_alabarda$alabar_
8:	3	l	abar_a_la_alabarda$al
9:	15	l	abarda$alabar_a_la_al
10:	1	$	alabar_a_la_alabarda$
11:	13	_	alabarda$alabar_a_la_
12:	5	b	ar_a_la_alabarda$alab
13:	17	b	arda$alabar_a_la_alab
14:	4	a	bar_a_la_alabarda$ala
15:	16	a	barda$alabar_a_la_ala
16:	19	r	da$alabar_a_la_alabar
17:	10	_	la_alabarda$alabar_a_
18:	2	a	labar_a_la_alabarda$a
19:	14	a	labarda$alabar_a_la_a
20:	6	a	r_a_la_alabarda$alaba
21:	18	a	rda$alabar_a_la_alaba

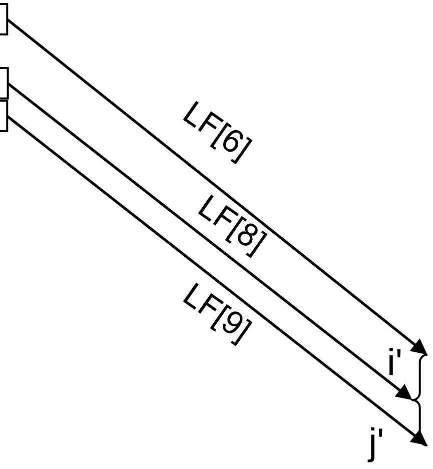
Step 2: search for ”la”

i	SA[i]	L	suffix $T_{SA[i],n}$
1:	21	a	$alabar_a_la_alabarda
2:	7	r	_a_la_alabarda$alabar
3:	12	a	_alabarda$alabar_a_la
4:	9	a	_la_alabarda$alabar_a
5:	20	d	a$alabar_a_la_alabard
6:	11	l	a_alabarda$alabar_a_l
7:	8	_	a_la_alabarda$alabar_
8:	3	l	abar_a_la_alabarda$al
9:	15	l	abarda$alabar_a_la_al
10:	1	$	alabar_a_la_alabarda$
11:	13	_	alabarda$alabar_a_la_
12:	5	b	ar_a_la_alabarda$alab
13:	17	b	arda$alabar_a_la_alab
14:	4	a	bar_a_la_alabarda$ala
15:	16	a	barda$alabar_a_la_ala
16:	19	r	da$alabar_a_la_alabar
17:	10	_	la_alabarda$alabar_a_
18:	2	a	labar_a_la_alabarda$a
19:	14	a	labarda$alabar_a_la_a
20:	6	a	r_a_la_alabarda$alaba
21:	18	a	rda$alabar_a_la_alaba

Step 3: search for ”ala”

Backward search with LF-mapping

	i	SA[i]	L	suffix $T_{SA[i],n}$
	1:	21	a	$alabar_a_la_alabarda
	2:	7	r	_a_la_alabarda$alabar
	3:	12	a	_alabarda$alabar_a_la
	4:	9	a	_la_alabarda$alabar_a
i	5:	20	d	a$alabar_a_la_alabard
	6:	11	l	a_alabarda$alabar_a_l
	7:	8	_	a_la_alabarda$alabar_
	8:	3	l	abar_a_la_alabarda$al
	9:	15	l	abarda$alabar_a_la_al
	10:	1	$	alabar_a_la_alabarda$
	11:	13	_	alabarda$alabar_a_la_
	12:	5	b	ar_a_la_alabarda$alab
j	13:	17	b	arda$alabar_a_la_alab
	14:	4	a	bar_a_la_alabarda$ala
	15:	16	a	barda$alabar_a_la_ala
	16:	19	r	da$alabar_a_la_alabar
	17:	10	_	la_alabarda$alabar_a_
	18:	2	a	labar_a_la_alabarda$a
	19:	14	a	labarda$alabar_a_la_a
	20:	6	a	r_a_la_alabarda$alaba
	21:	18	a	rda$alabar_a_la_alaba



i	SA[i]	L	suffix $T_{SA[i],n}$
1:	21	a	$alabar_a_la_alabarda
2:	7	r	_a_la_alabarda$alabar
3:	12	a	_alabarda$alabar_a_la
4:	9	a	_la_alabarda$alabar_a
5:	20	d	a$alabar_a_la_alabard
6:	11	l	a_alabarda$alabar_a_l
7:	8	_	a_la_alabarda$alabar_
8:	3	l	abar_a_la_alabarda$al
9:	15	l	abarda$alabar_a_la_al
10:	1	$	alabar_a_la_alabarda$
11:	13	_	alabarda$alabar_a_la_
12:	5	b	ar_a_la_alabarda$alab
13:	17	b	arda$alabar_a_la_alab
14:	4	a	bar_a_la_alabarda$ala
15:	16	a	barda$alabar_a_la_ala
16:	19	r	da$alabar_a_la_alabar
17:	10	_	la_alabarda$alabar_a_
18:	2	a	labar_a_la_alabarda$a
19:	14	a	labarda$alabar_a_la_a
20:	6	a	r_a_la_alabarda$alaba
21:	18	a	rda$alabar_a_la_alaba

$i'=LF[6]=C['l']+\mathrm{Rank}_{'l'}(L,6)=C['l']+\mathrm{Rank}_{'l'}(L,i-1)+1=16+0+1=17$

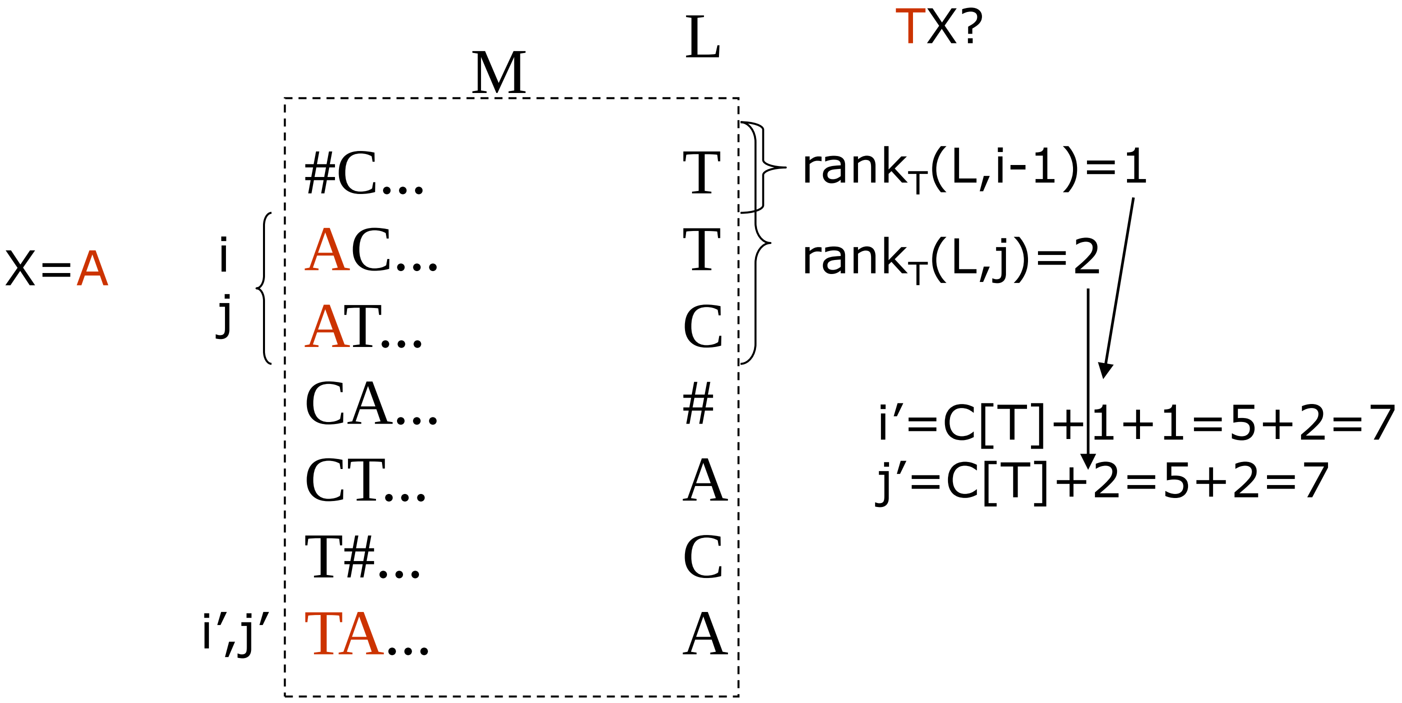
$j'=LF[9]=C['l']+\mathrm{Rank}_{'l'}(L,9)=C['l']+\mathrm{Rank}_{'l'}(L,j)=16+3=19$

Backward search with rank-queries

- **Observation**: If [i,j] is the range in BW-matrix M, where all rows start with X, then range [i’,j’], where all rows start with cX can be computed using:

$$i' := C[c]+Rank_c(L,i-1)+1,$$
$$j' := C[c]+Rank_c(L,j).$$

Backward search with rank-queries



TX?

X=A

	M	L
	#C...	T
i	AC...	T
j	AT...	C
	CA...	#
	CT...	A
	T#...	C
i',j'	TA...	A

$\mathrm{rank}_T(L,i-1)=1$

$\mathrm{rank}_T(L,j)=2$

i'=C[T]+1+1=5+2=7
j'=C[T]+2=5+2=7

Backward search - pseudocode

```
Algoritmi Count(P[1,m], L[1,n],C[1,$\sigma$])
c = P[m]; k = m;
i = C[c]+1; j = C[c+1];
while (i ≤ j ja k>1) do begin
    c = P[k-1]; k = k-1;
    i = C[c]+$\mathrm{Rank}_c$(L,i-1)+1;
    j = C[c]+$\mathrm{Rank}_c$(L,j); end;
if (j<i) then return 0 else return (j-i+1);
```

Backward search...

- Algorithm Count makes O(m) queries to function $Rank_c(L,i)$.
- Depending on the underlying structure to support $Rank_c(L,i)$, different time/space tradeoffs can be obtained.

Compressed suffix array as a self-index

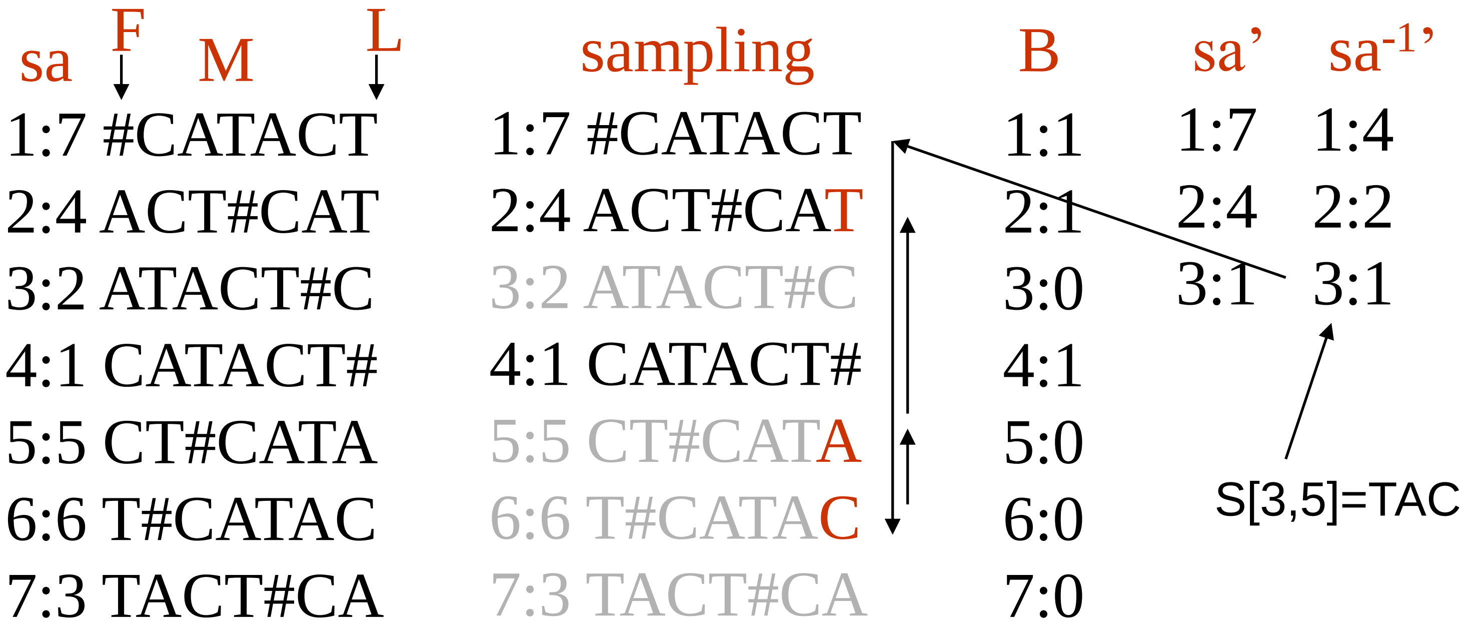
- Let us define a self-index csa(S) as a structure that replaces a sequence S with a compressed representation that supports:
  - Count(P): Compute the number of occurrences of a given pattern P in S.
  - Range(P): Return the suffix array range [i,j] containing the suffixes prefixed by the pattern.
  - Locate(i): Return value SA[i].
  - Display(k,l): Return substring S[k,l].

Compressed suffix array as a self-index

- Combining the earlier compressed suffix array with backward search supports directly Count(), Range() ja Locate() operations.
- Display() can be supported by sampling inverse suffix array values and using again LF-mapping.

Display()

log n = 3



sa	F M L	sampling	B	sa’	sa^{-1}’
1:7	#CATACT	1:7 #CATACT	1:1	1:7	1:4
2:4	ACT#CAT	2:4 ACT#CAT	2:1	2:4	2:2
3:2	ATACT#C	3:2 ATACT#C	3:0	3:1	3:1
4:1	CATACT#	4:1 CATACT#	4:1		
5:5	CT#CATA	5:5 CT#CATA	5:0		
6:6	T#CATAC	6:6 T#CATAC	6:0		
7:3	TACT#CA	7:3 TACT#CA	7:0		

S[3,5]=TAC

Compressed suffix array self-index

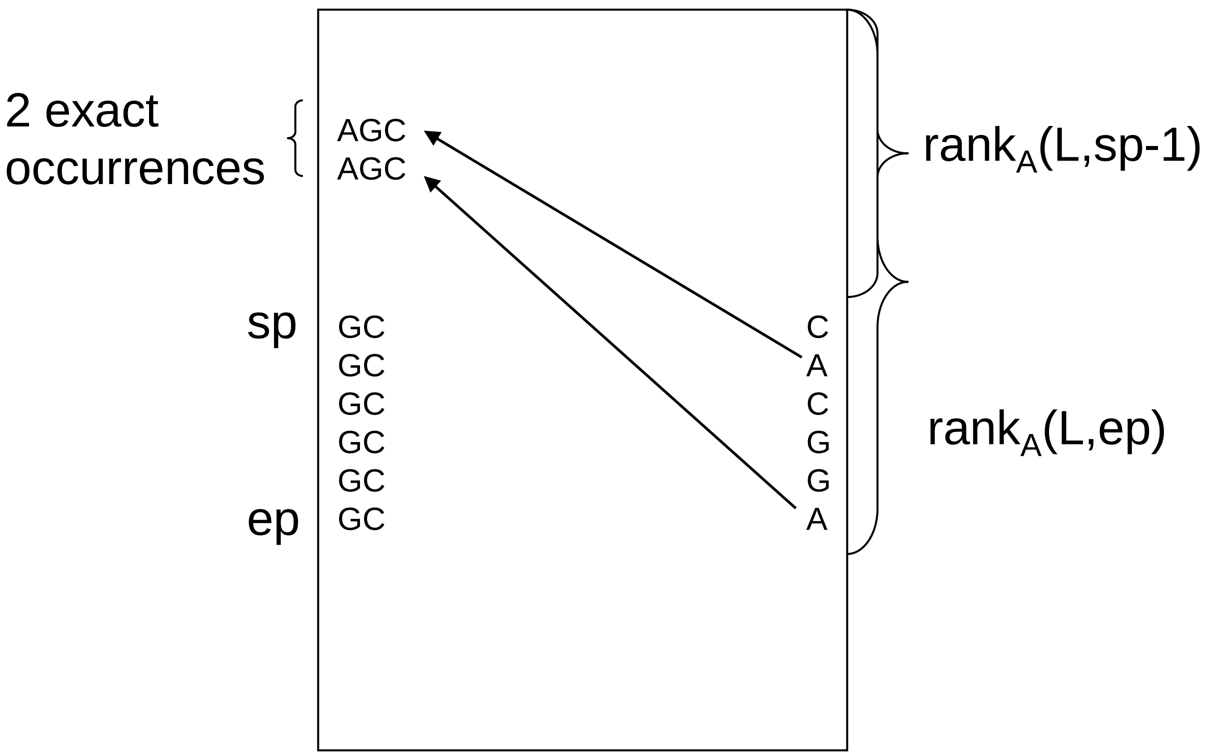
- Overall space is $n \log \sigma (1+o(1))$ bits, when:
  - Each $(\log n)^{1+\varepsilon}/ \log \sigma$:th value is sampled, making sa' and $sa^{-1}{}'$ tables occupy $o(n \log \sigma)$ bits, with $\varepsilon>0$.
  - Bitvector B can be compressed into $o(n \log \sigma)$ bits (we omit the details here).
  - Locate(i) takes time $O((\log n)^{1+\varepsilon})$.
  - Display(i,j) takes time $O((\log n)^{1+\varepsilon}+(j-i)\log \sigma)$.
  - Count() / Range() take time $O(m \log \sigma)$.

High-throughput mapping in practice

- Several tools exist for sequence mapping, e.g. Maq, BWT-SW, BWA, SOAP2, and Bowtie.
- Most are based on *backtracking on BWT*.
- Let us consider the k-mismatches problem for simplicity.
- Recall the backward search algorithm.

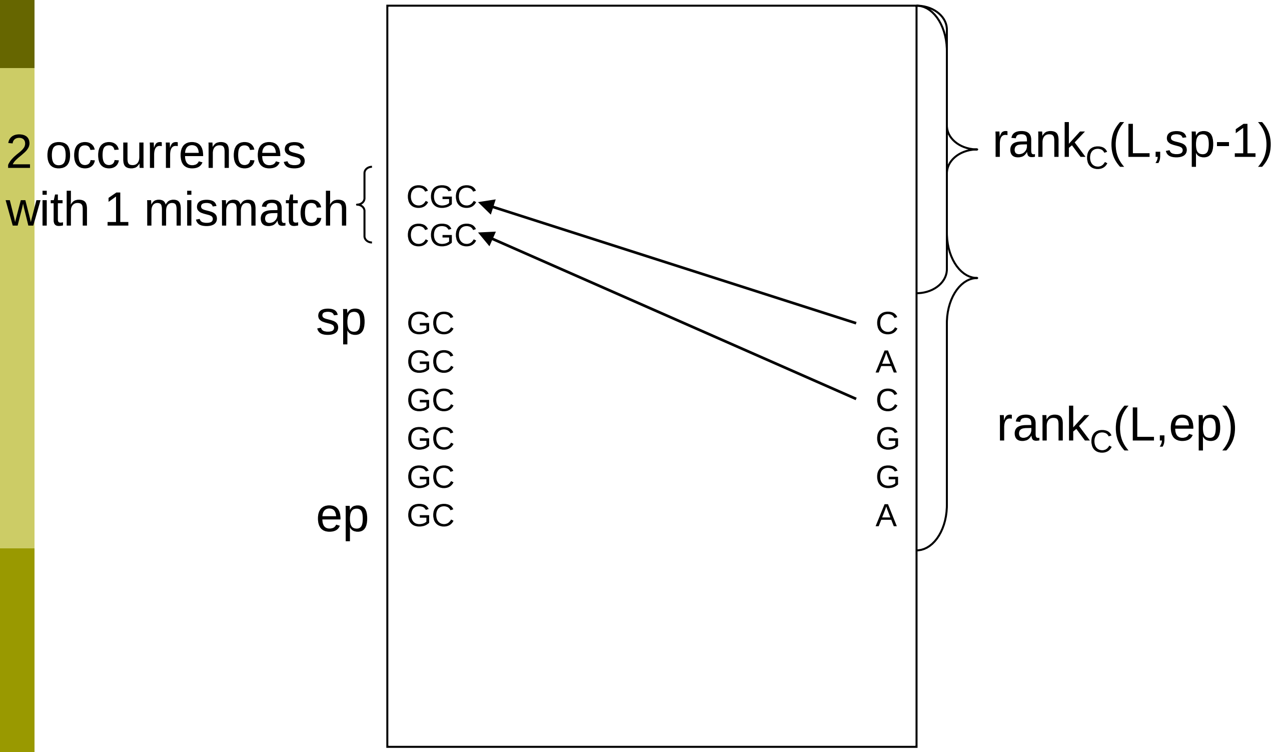
Backward backtracking, one step

P=AGC, k=1



Backward backtracking, one step

P=AGC, k=1



2 occurrences
with 1 mismatch

CGC
CGC

sp
GC
GC
GC
GC
GC
ep
GC

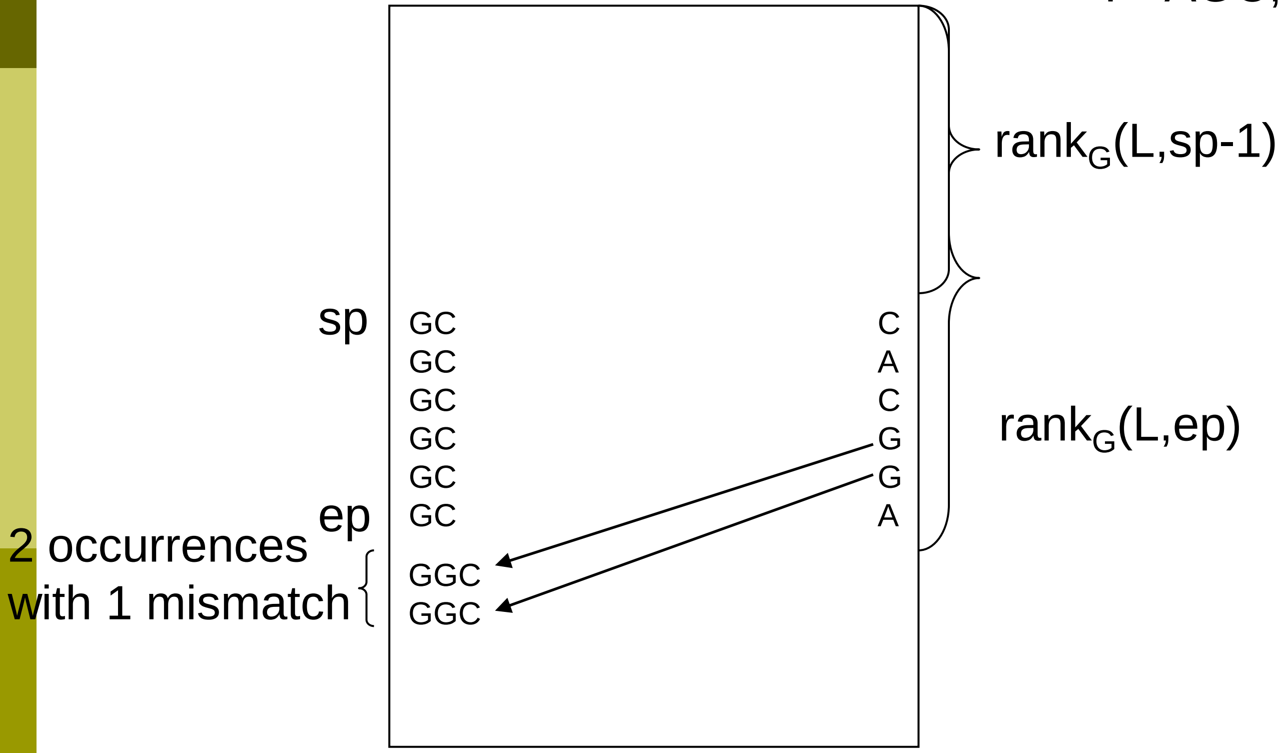
C
A
C
G
G
A

$\text{rank}_C(L,sp-1)$

$\text{rank}_C(L,ep)$

Backward backtracking, one step

P=AGC, k=1



Backward backtracking – pseudocode

```
Algorithm kmismatches(P,L, k, j, sp, ep)
if ($j = 0$) then
    Report occurrences Pos[sp], . . . , Pos[ep]; return;
for each $s \in \Sigma$ do
    $sp' \leftarrow C[s] + rank_s(L, sp - 1)+1$;
    $ep' \leftarrow C[s] + rank_s(L, ep)$;
    if ($P[j] != s$) $k' \leftarrow k - 1$; else $k' \leftarrow k$;
    if ($k' \geq 0$) kmismatches(P,L, $k'$, $j - 1$, $sp'$, $ep'$);
```

First call: kmismatches(P,L,k,m,1,n)

Example test run

- Compressed suffix array for human genome occupied 2.1 GB.
- 10000 patterns of length 32 searched for with k=0,1,2 mismatches.
- Average search times (finding the ranges) were 0.3, 8.2, and 121 milliseconds per pattern, for k=0,1,2, respectively.
- Locating one occurrence took 0.9 milliseconds on average.

Search space pruning: BWA

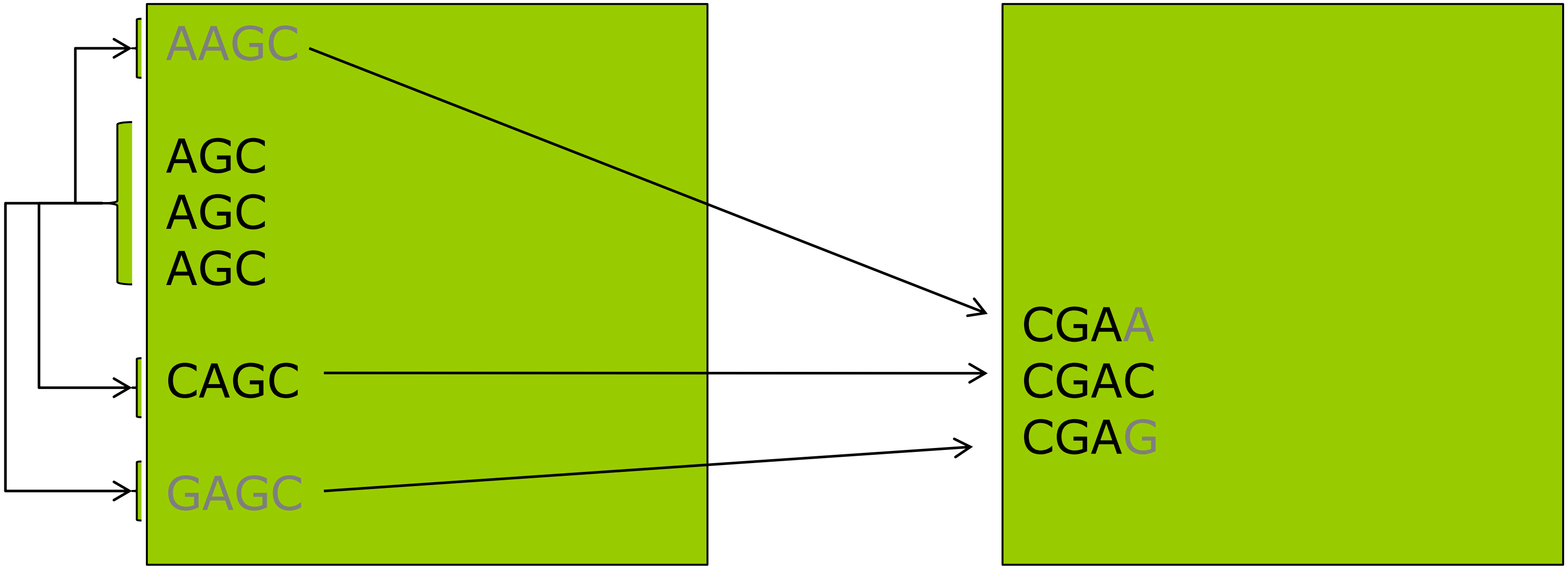
- Build compressed suffix array (aka FM-index) for S and its reverse S^r. Call these *forward FM-index* and *reverse FM-index*.
- Compute a table D[1,m] such that D[i]=$\kappa(\alpha)$ gives a lower bound for the minimum amount of errors needed to match α = P[1,i] in T.
  - E.g. D[i] = minimum number of times P[1,i] need to be split such that each piece occurs exactly in T.
  - Initialize D[0]=0. With reverse FM-index, backward search P^r from i=m to i=1 setting D[m-i]=D[m-i-1] until empty interval, say at P^r[i'], then set D[m-i']=D[m-i'-1]+1 and continue in the same way.
- Consider search space state corresponding to suffix P[j,m], interval [sp,ep], and k' mismatches. If k'+D[j-1]>k, no need to continue search down from current state.

Search space pruning: Bowtie

- Also uses *forward FM-index* and *reverse FM-index*.
- Splits possible occurrences in different categories and searches each category separately. For example, with 1-mismatches search, an occurrence can have the error (i) in P[1,m/2] or (ii) in P[m/2+1,m]. For (i) category occurrences use reverse FM-index for the search; no branching in first m/2 characters. For (ii) category occurrences use forward FM-index for the search; no branching in first m/2 characters.
- For more errors, there is always a bad category, e.g. in 2-mismatches search, pattern can be split to 3 pieces and categories are all different ways to distribute 2 erros in 3 pieces. Distribution 101 is a bad one, as one has to start the search allowing branching.

Search space pruning: SOAP2

- Also uses *forward FM-index* and *reverse FM-index*.
- Solves the bad category (e.g. 101) case of bowtie: Is able to search P[1,2m/3] using forward FM-index and continue directly the search from reverse FM-index with P[2m/3+1,m] (see lecture script for details).



Search space pruning: suffix filter

- Extension of the simple pattern splitting filter.
- ACACAGAGCTAGCT, k=2
- Search e.g. suffixes (many variations of the theme)

```
ACAC|AGAG|CTAGCT
     AGAG|CTAGCT
          CTAGCT
```

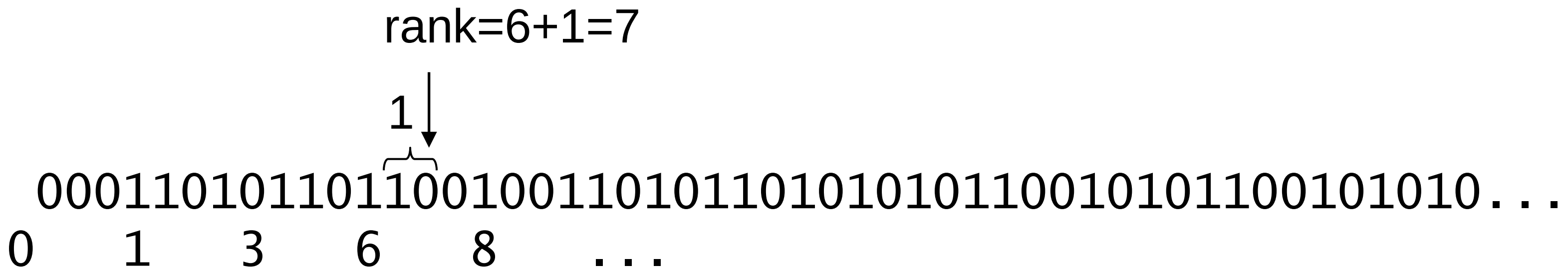
- At | increase the allowed number of errors by 1.
- Check all candidate occurrences. Can be shown to be lossless filter.
- Using instead prefixes, the search can be implemented on top of forward FM-index.
- Extends to finding approximate overlaps between set of reads (de novo assembly precomputation).

Appendix

PROVING RANK LEMMAS

Constant time rank using o(n) extra bits

- <O(n) *space*, O(log n) *time*>: Store answers at each log n:th position.
  - Read the rest from the original bitvector.



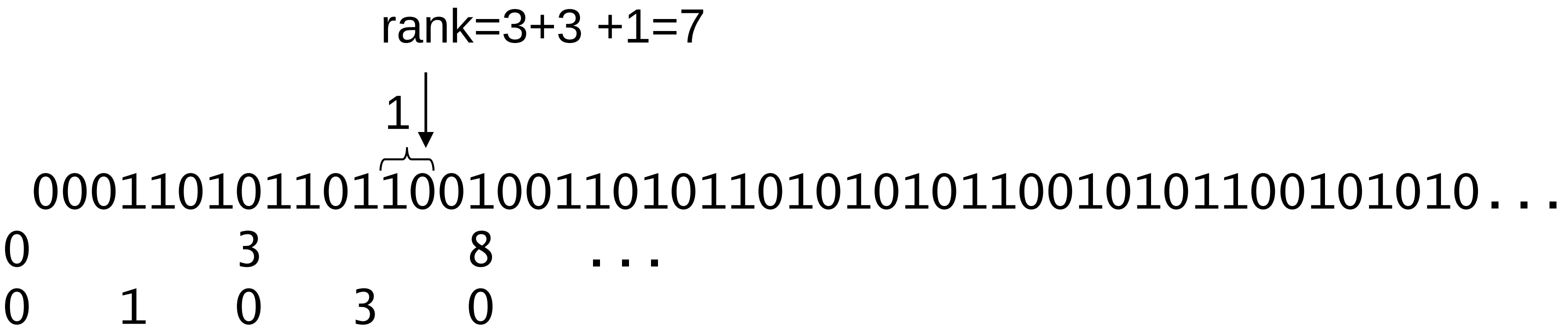
rank=6+1=7

1

```
  000110101101110010011010110101010110010101100101010...
0    1    3    6    8    ...
```

Constant time rank using o(n) extra bits

- <o(n) *space,* O(log n) *time*>: Store answers at each $\log^2 n$:*th* position. Store relative answers at each log n:*th* position. - Read the rest from the original bitvector.

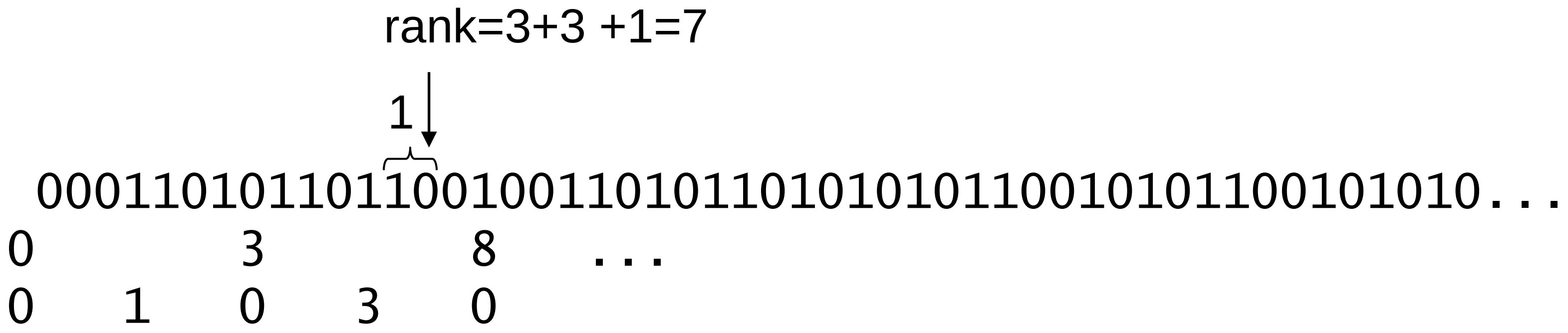


rank=3+3 +1=7

```
                1
 00011010110110010011010110101010110010101100101010...
0        3        8    ...
0   1    0    3   0
```

Constant time rank using o(n) extra bits

- <o(n) *space*, O(1) *time*>: Same as before, but read the last answer from a precomputed table of size o(n).



rank=3+3 +1=7

```
                  1
  00011010110110010011010110101010110010101100101010...
0       3       8   ...
0   1   0   3   0
```

Precomputed table

- We would need to answer rank in a block of length log n bits in constant time.
- Let us divide the block into two (log n) / 2 bits parts.
- There are $2^{(\log n) / 2} = \sqrt{n}$ bitvectors of length (log n) / 2.
- We can store in $\sqrt{n} \log n \log \log n$ bits the answers to all possible rank-queries for all bitvectors of length (log n) / 2 .

Example of rank-computation

B 0 1 0 1 1 0 1 0 **0 1 1 1** 0 1 0 0

superblockrank 0 8

blockrank 0 2 **4** 7 0

smallrank	0	1
00	0	0
01	0	1
10	1	1
11	1	2

rank$_1$(B,11) = superblockrank[0] + blockrank[2] +
smallrank[**01**,1] + smallrank[**11**,0]
= 0 + 4 + 1 + 1 = 6

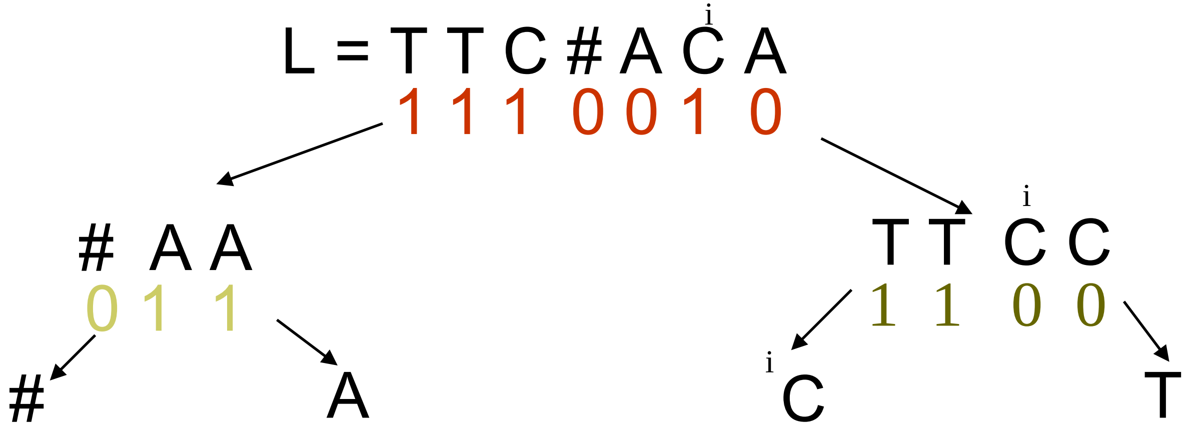
Wavelet tree

- Rank/Select for sequences.
- Recall LF-mapping of BW-transform:
  $LF[i] = C[L[i]]+Rank_{L[i]}(L,i)$
- Wavelet tree of represents $L=bwt(S)$ in $n \log \sigma (1+o(1))$ bits, such that each $Rank_c(L,i)$ query takes $O(\log \sigma)$ time, where $n=|S|$.

Wavelet tree, example 1

Σ={#,A,C,G,T}

i = 6



i

L = T T C # A C A

1 1 1 0 0 1 0

A A

0 1 1

i

T T C C

1 1 0 0

#

A

i

C

T

B=1110010, B[6]=1 ->right

i = rank$_1$(1110010,6)=4

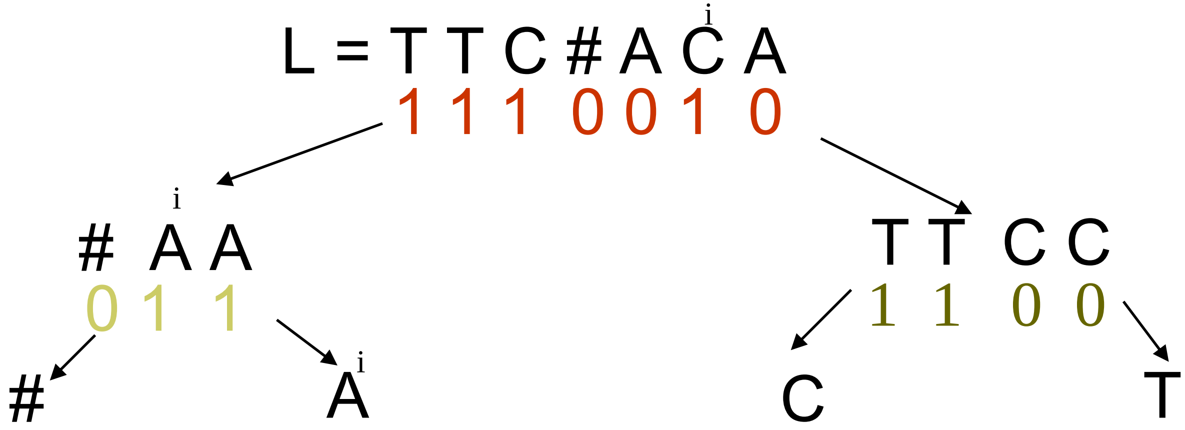
B=1100, B[4]=0 ->left

i = rank$_0$(1100,4)=2

L[6] = C

Wavelet tree, example 2

Σ={#,A,C,G,T}



i

L = T T C # A C A

1 1 1 0 0 1 0

i

A A

0 1 1

T T C C

1 1 0 0

i

# A C T

i = 6

A ∈{#,A}-> left

i = rank_0(1110010,6)=2

A ∈{A}-> right

i = rank_1(011,2)=1

$\mathrm{Rank}_A(L,6) = 1$

Rank() function space/time

- The tree has $\log \sigma$ levels, each consuming constant time for $\mathrm{rank}_{0/1}$-queries:
  - $O(\log \sigma)$ time for $\mathrm{Rank}_c(L,i)$.
- Each level has at most n bits. After preprocessing each level for $\mathrm{rank}_{0/1}$-queries, the whole tree occupies $n \log \sigma\ (1+o(1))$ bits.

Other alternatives

- Indicator-bitvector for each symbol:
  - $O(1)$ time in $\sigma n(1+o(1))$ bits.
- Instead of balanced tree, use Huffman tree:
  - $O(\log n)$ time in $n(H_0+1)(1+o(1))$ bits.
- Compress the bitvectors still supporting rank-queries:
  - $O(\log \sigma)$ time in $nH_0(1+o(1))$ bits, or $nH_k(1+o(1))$ bits if the input is BW-transform.
  - (Details omitted here)