

High-throughput read mapping with Burrows-Wheeler indexes



Veli Mäkinen

Department of Computer Science

University of Helsinki

History

- This lecture was first presented as a tutorial in ISMB 2009.
- Then given several times in Biological Sequence Analysis course at the University of Helsinki.
- All details and many extensions can now found in a textbook:
 - Veli Mäkinen, Djamel Belazzougui, Fabio Cunial, Alexandru I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press 2015.

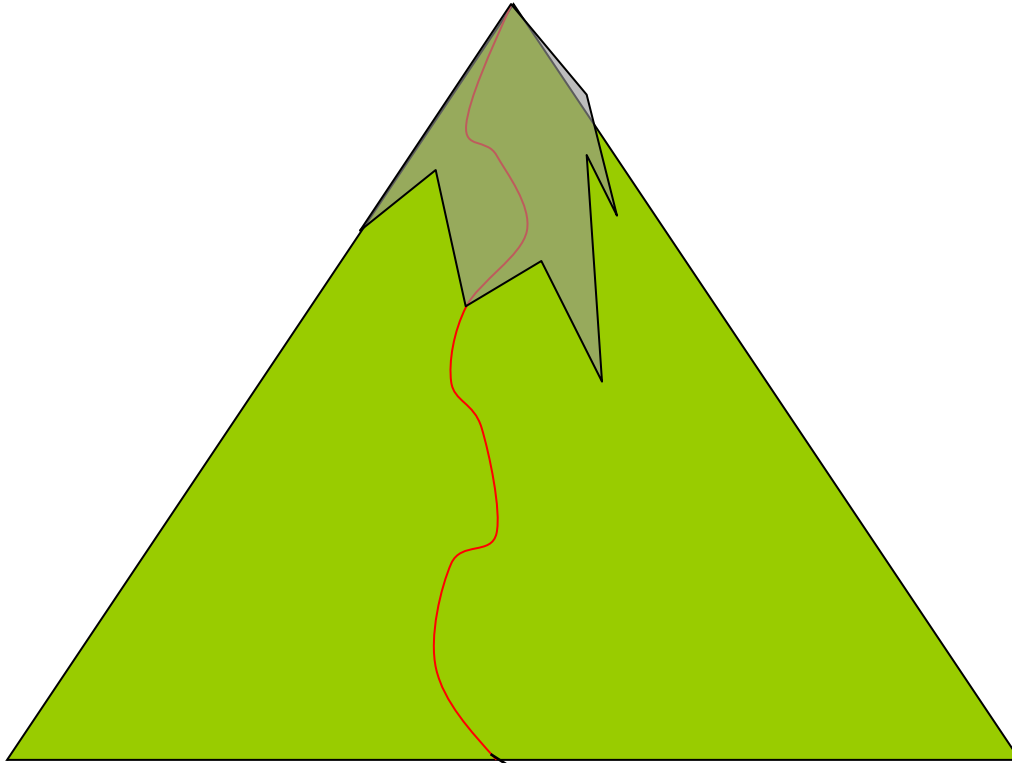
Towards practical read mapping

BACKTRACKING WITH SUFFIX TREE

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.

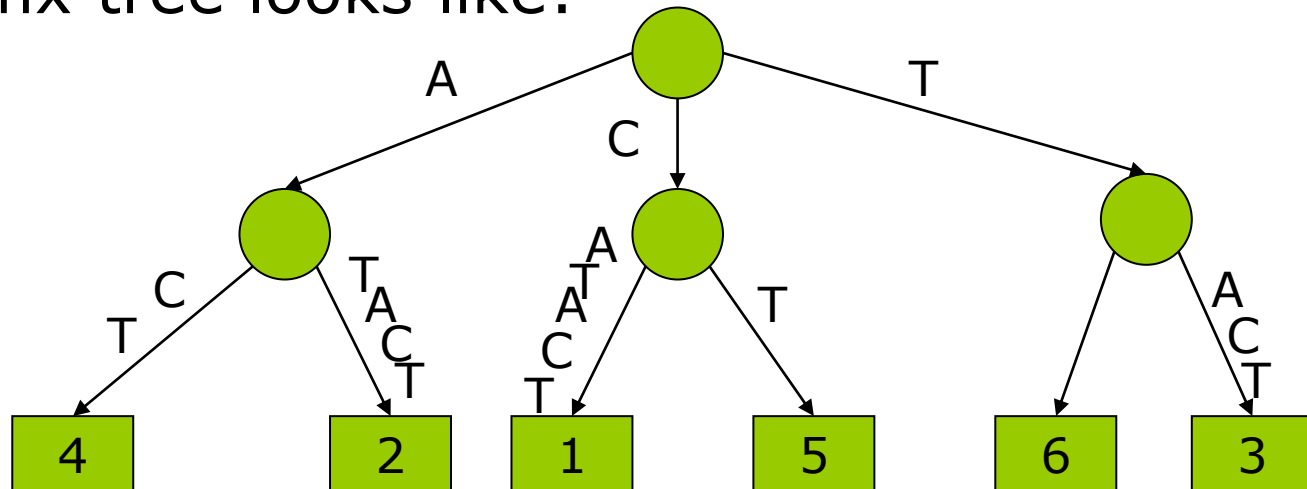
Solution: backtracking with suffix tree



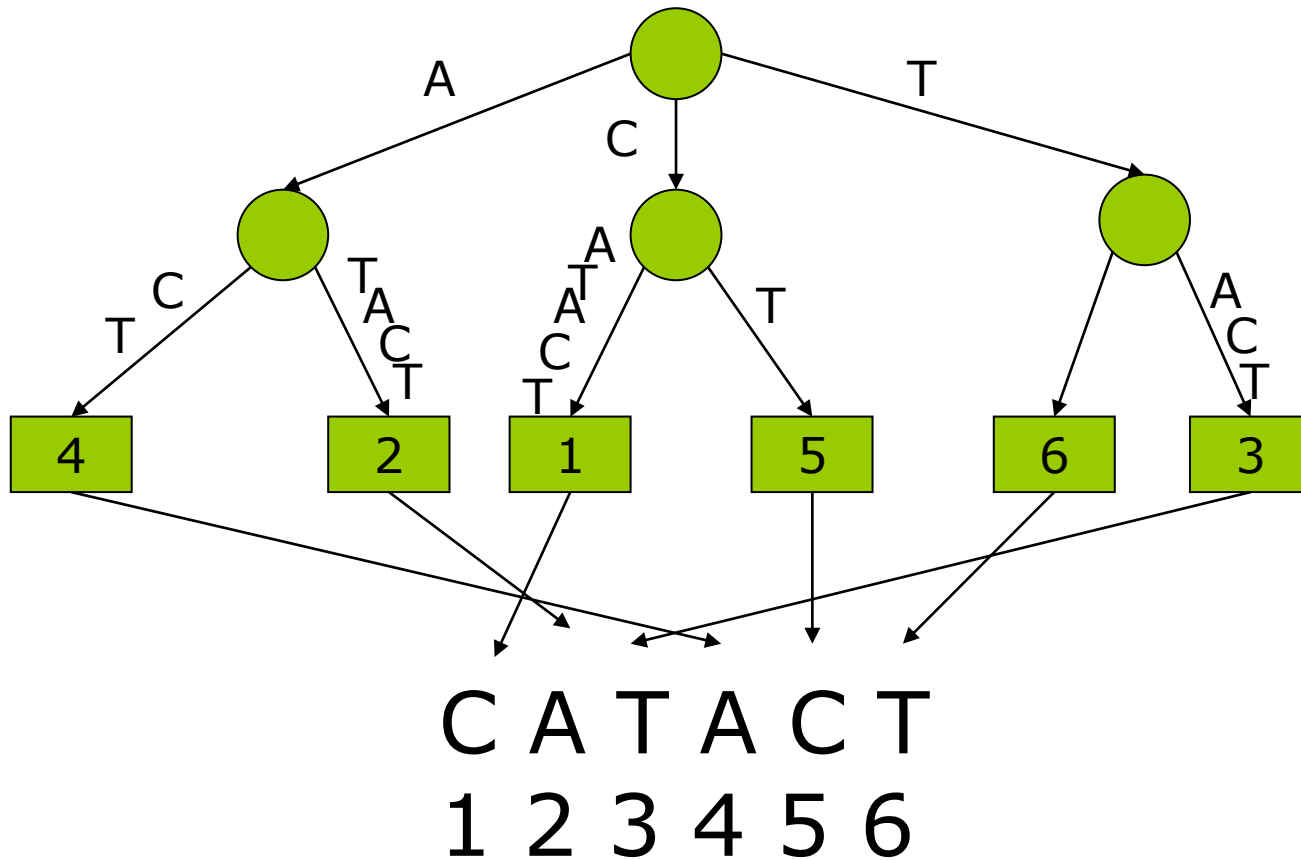
...ACACATTATCACAGGCATCGGCATTAGCGATCGAGTCG.....

Suffix tree

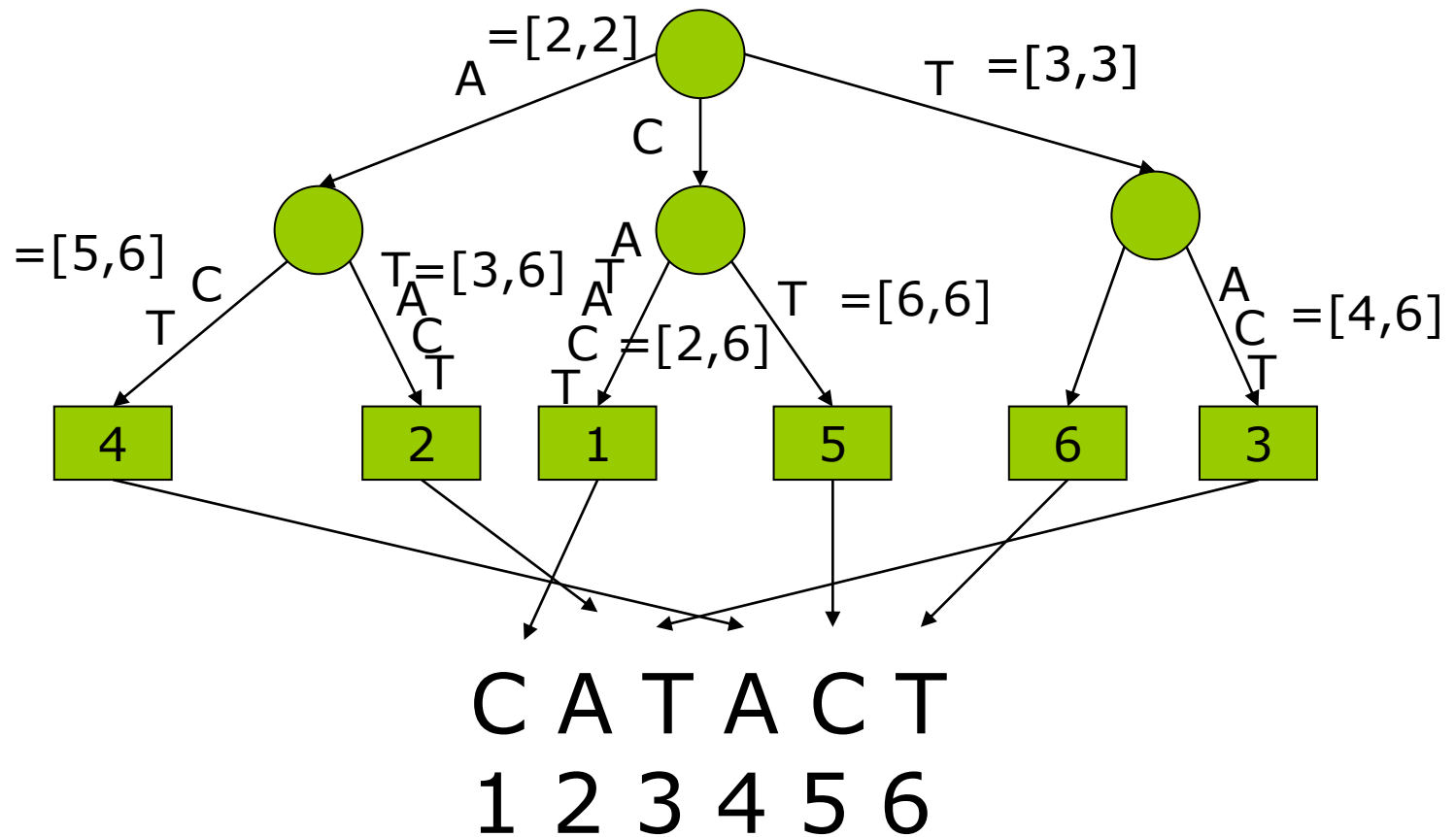
- Suffix tree is a compressed keyword trie of all *suffixes* of a sequence
- E.g. suffixes of sequence CATACT are CATACT, ATRACT, TACT, ACT, CT, T.
 - suffix tree looks like:



Suffix tree

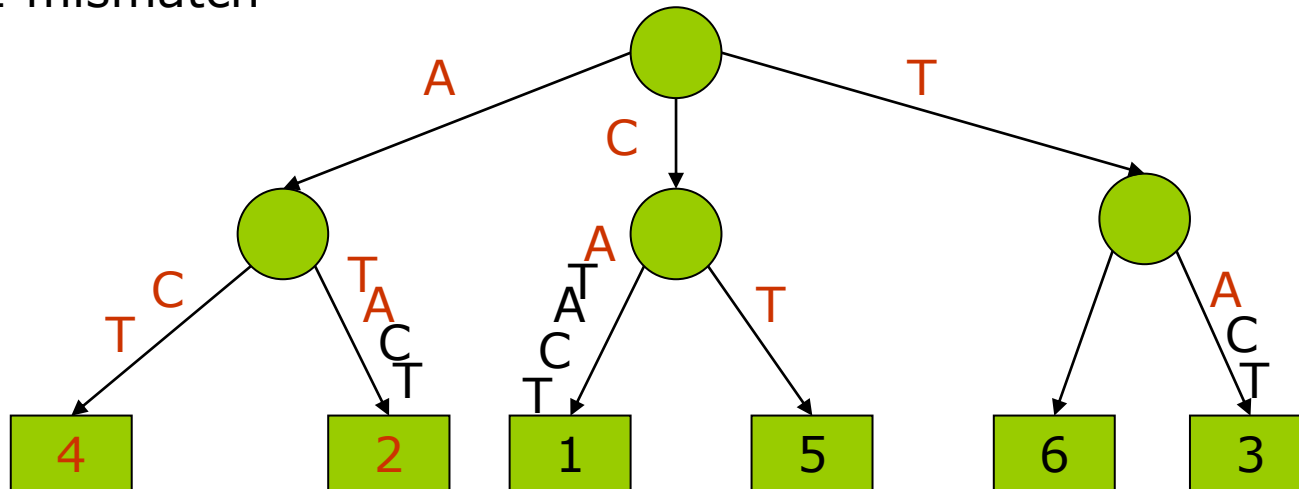


Suffix tree



Back to backtracking

ACA, 1 mismatch



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

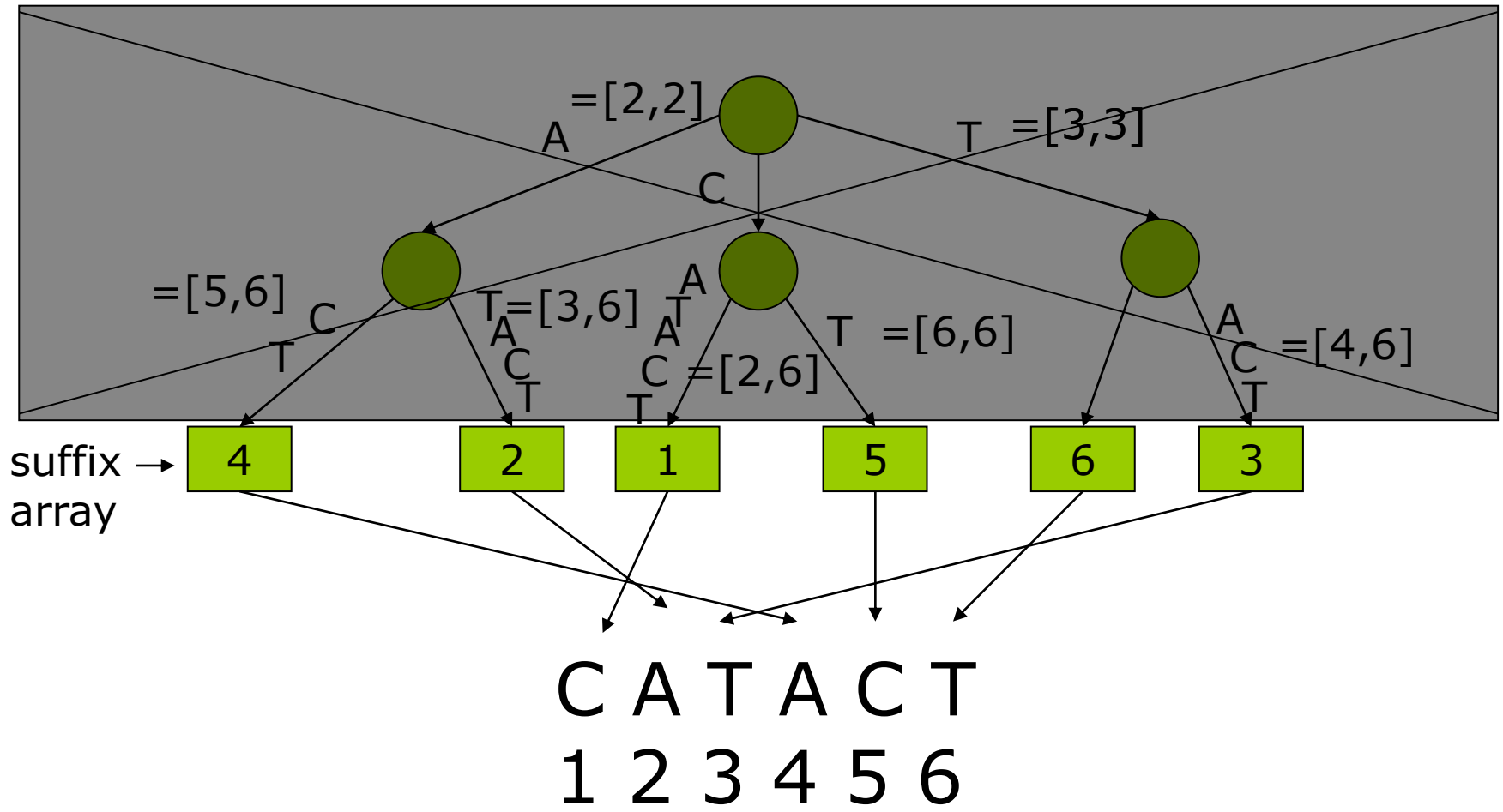
Properties of suffix tree

- 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

- 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.

Towards backtracking in smaller space

BURROWS-WHEELER TRANSFORM

Burrows-Wheeler transform (BWT)

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

BWT vs. suffix array

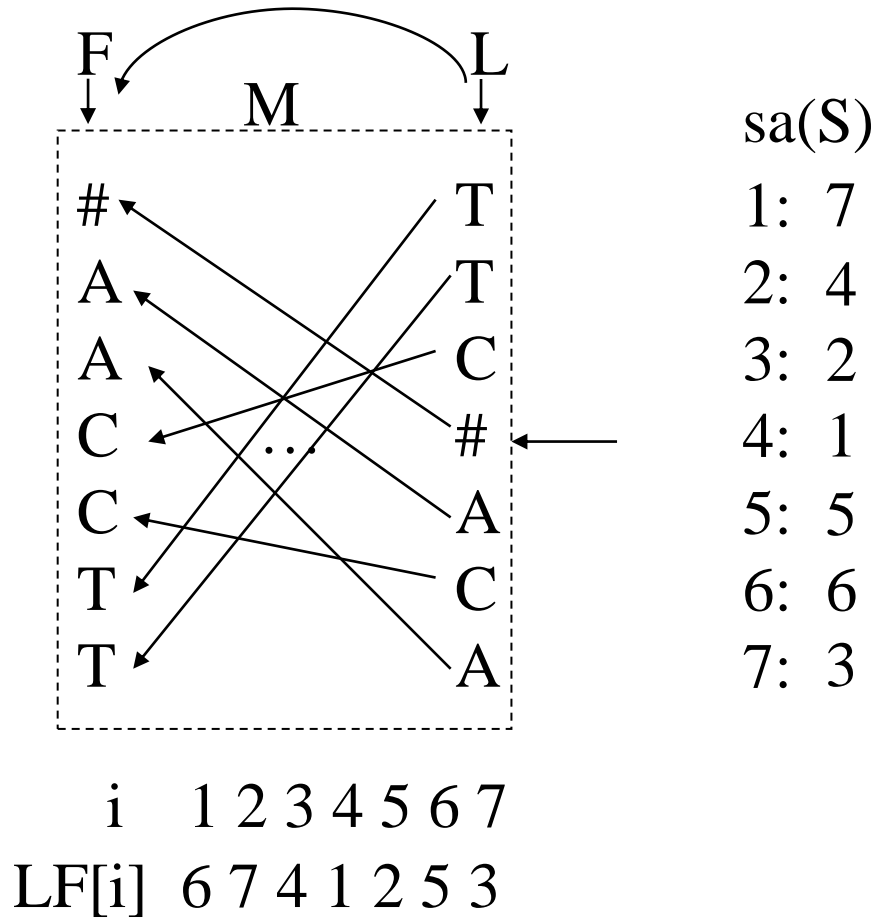
- The lexicographic order of the cyclic shifts of **S** is essentially *suffix array* $sa(S)$.

				1234567
sa	F	M	L	S = CATACT#
	↓		↓	
1:7	#CATACT			
2:4	ACT#CAT			BWT=
3:2	ATACT#C			(L = TTC#ACA, row 4)
4:1	CATACT#			
5:5	CT#CATA			
6:6	T#CATAC			
7:3	TACT#CA			

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

$$S^{-1} = \# \text{TCATAC}$$

stable sort



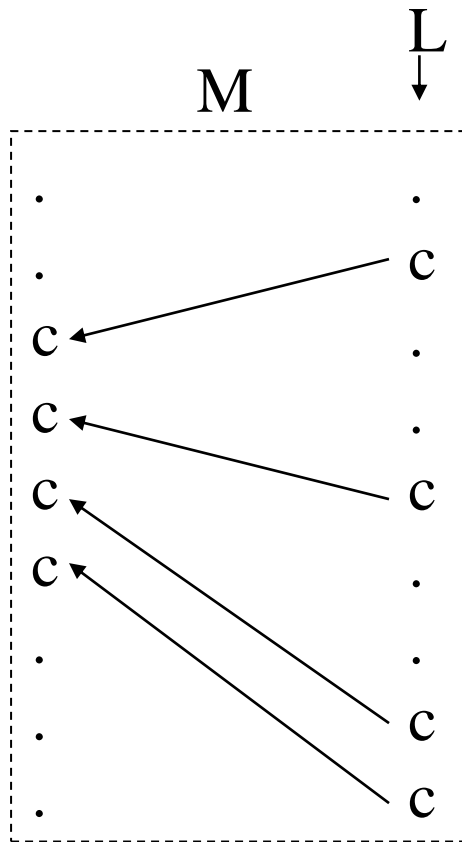
LF-mapping

- Let the i -th row of M contain cyclic shift fXI , and j -th row cyclic shift lfX .
- $LF[i] = j$.
- Hence, $L[i] L[LF[i]] L[LF[LF[i]]] \dots$ 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 , $c \in \{1, 2, \dots, \sigma\}$.
- **Lemma 1:** $LF[i] \in [C[L[i]]+1, C[L[i]+1]]$
- Let $\text{rank}_c(L, i)$ be the amount of symbols c in the prefix $L[1, i]$.
- **Lemma 2:** $LF[i] = C[L[i]] + \text{rank}_{L[i]}(L, i)$.
- **Lemma 3:** When L is stable sorted into L' , then $L[i]$ is mapped to $L'[LF[i]]$.

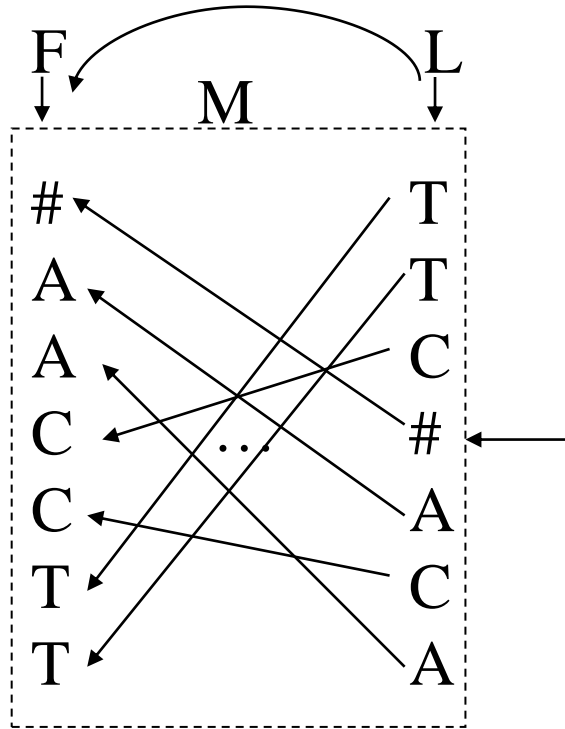
Proving Lemmas 2 and 3



- a) Let $Xc < Yc$. Then $X < Y$ and so is $cX < cY$.
- b) There must be bijection from rows ending with c to rows starting with c .
- c) Sorting $\{(L[i],i)\}_i$ gives the only bijective mapping not leading to contradiction with a).
- That is, $LF[i] = C[c] + \text{Rank}_c(L,i)$.

$$S^{-1} = \#TCATAC$$

stable sort



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

$$\begin{aligned} \text{LF}(6) &= C[C] + \text{rank}_C(L, 6) \\ &= 3 + 2 = 5 \end{aligned}$$

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 $\text{rank}_1(B,i)$ and $\text{rank}_0(B,i)=i-\text{rank}_1(B,i)$ 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 $\text{rank}_c(L,i)$ for all $c \in \Sigma$ in $O(\log \sigma)$ time.
- *Proofs.* See end of these lecture slides.

Key functionalities of BWT indexing

COMPRESSED SUFFIX ARRAY AND BACKWARD SEARCH

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	↓		↓	1:7 #CATACT		1:1 → 1:7
2:4				2:4 ACT#CAT		2:1 → 2:4
3:2				3:2 ATRACT#C		3:0 ↗ 3:1
4:1				4:1 CATACT#		4:1 ↗
5:5				5:5 CT#CATA		5:0
6:6				6:6 T#CATAC		6:0
7:3				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:
 - $LF[i]=C[L[i]]+\text{rank}_{L[i]}(L,i)$
 - 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"

Step 1: search for "a"

Step 2: search for "la"

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

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

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

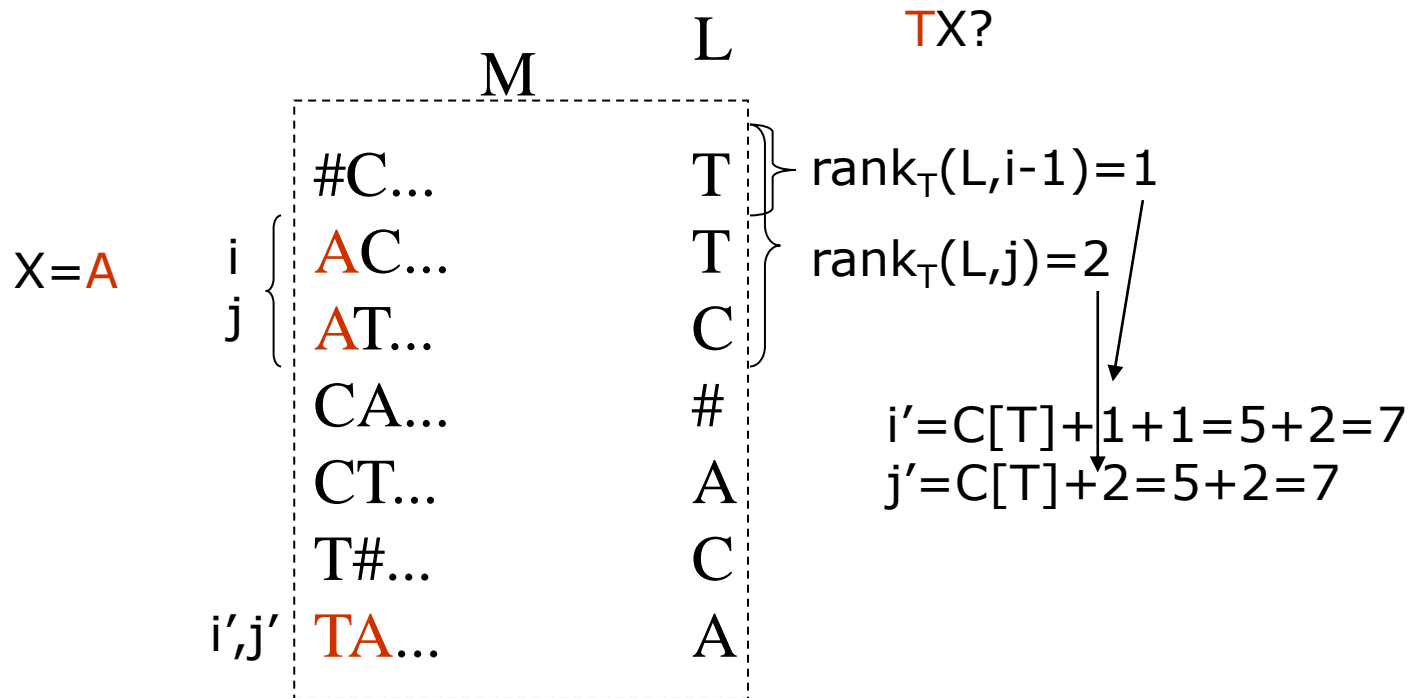
$$j' = LF[9] = C['l'] + \text{Rank}_{\eta}(L, 9) = C['l'] + \text{Rank}_{\eta}(L, j) = 16 + 3 = 19$$

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:

$$i' := C[c] + \text{Rank}_c(L, i-1) + 1,$$
$$j' := C[c] + \text{Rank}_c(L, j).$$

Backward search with rank-queries



Backward search - pseudocode

Algorithm Count($P[1,m], L[1,n], C[1,\sigma]$)

(1) $c = P[m]; k = m;$

(2) $i = C[c]+1; j = C[c+1];$

(3) while ($i \leq j$ **and** $k > 1$) **do begin**

(4) $c = P[k-1]; k = k-1;$

(5) $i = C[c]+Rank_c(L,i-1)+1;$

(6) $j = C[c]+Rank_c(L,j);$ **end;**

(7) if ($j < i$) **then return** 0 **else return** ($j-i+1$);

Backward search...

- Algorithm **Count** makes $O(m)$ queries to function $\text{Rank}_c(L,i)$.
- Depending on the underlying structure to support $\text{Rank}_c(L,i)$, different time/space tradeoffs can be obtained.

Combining sequence and its index

SELF-INDEXING WITH BWT

Compressed suffix array as a self-index

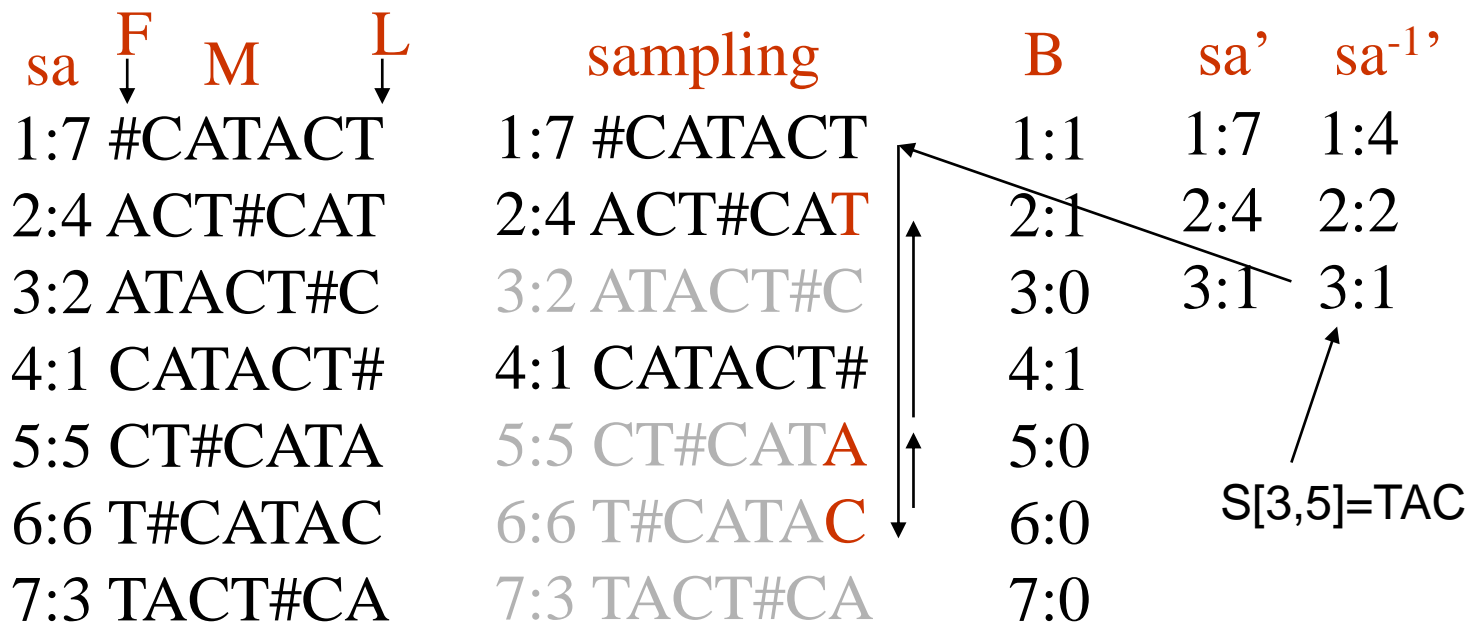
- Let us define a self-index $csa(S)$ as a structure that replaces a sequence S with a compressed representation that supports:
 - $Count(P)$: Compute the number of occurrences of a given pattern P in S .
 - $Range(P)$: Return the suffix array range $[i,j]$ containing the suffixes prefixed by the pattern.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

$\log n = 3$



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 $n(1+o(1))$ 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 $O(m \log \sigma)$.

Practical read mapping

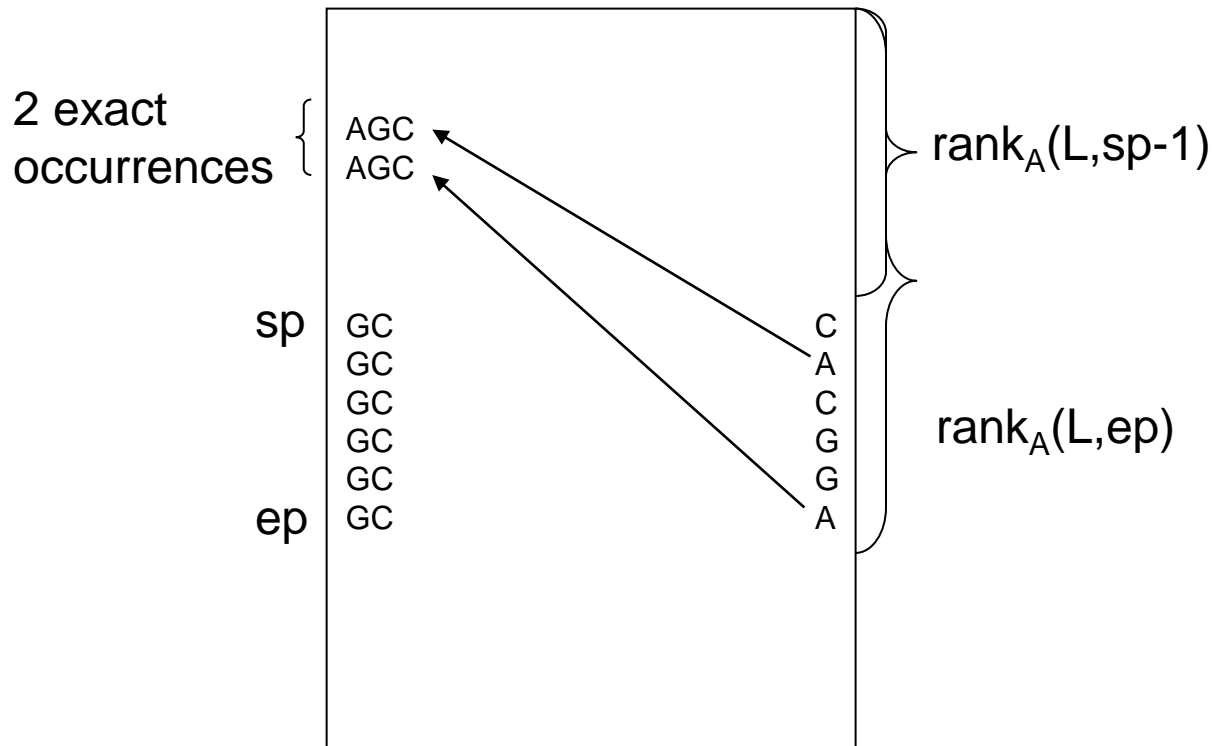
BACKTRACKING WITH BWT

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.

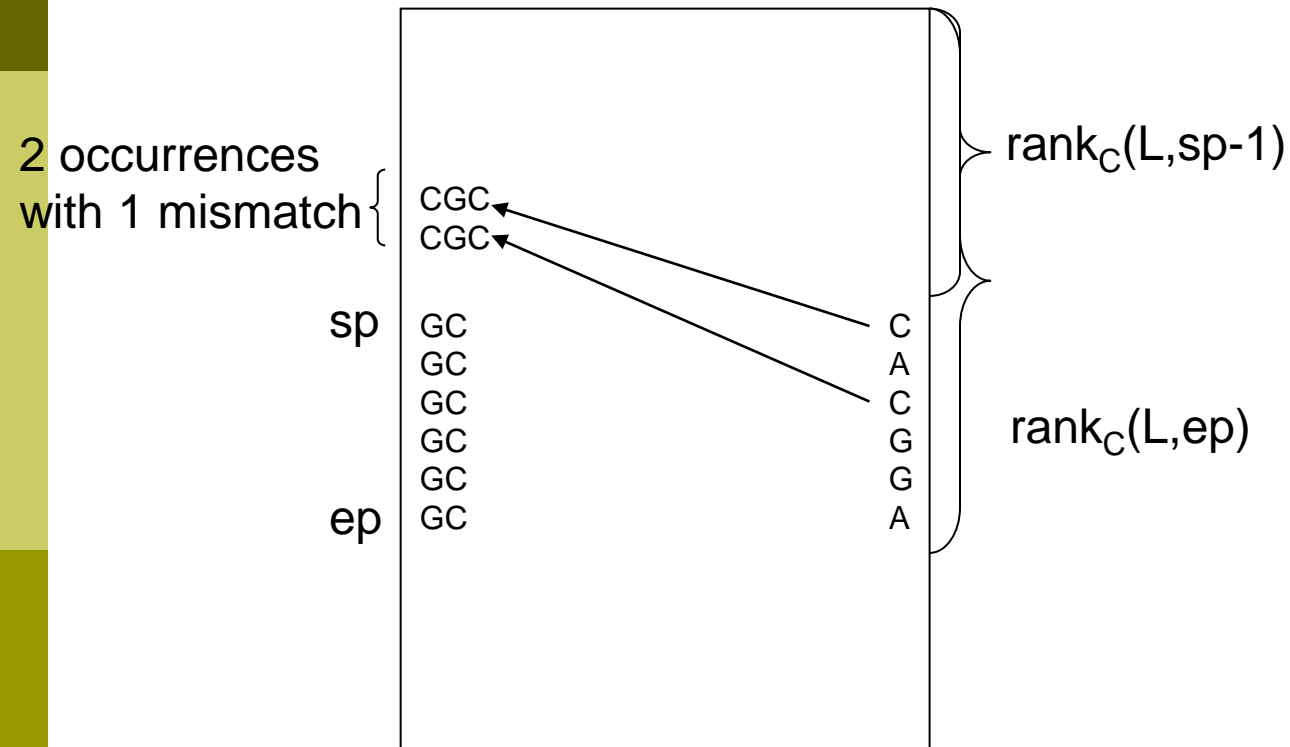
Backward backtracking, one step

P=AGC, k=1



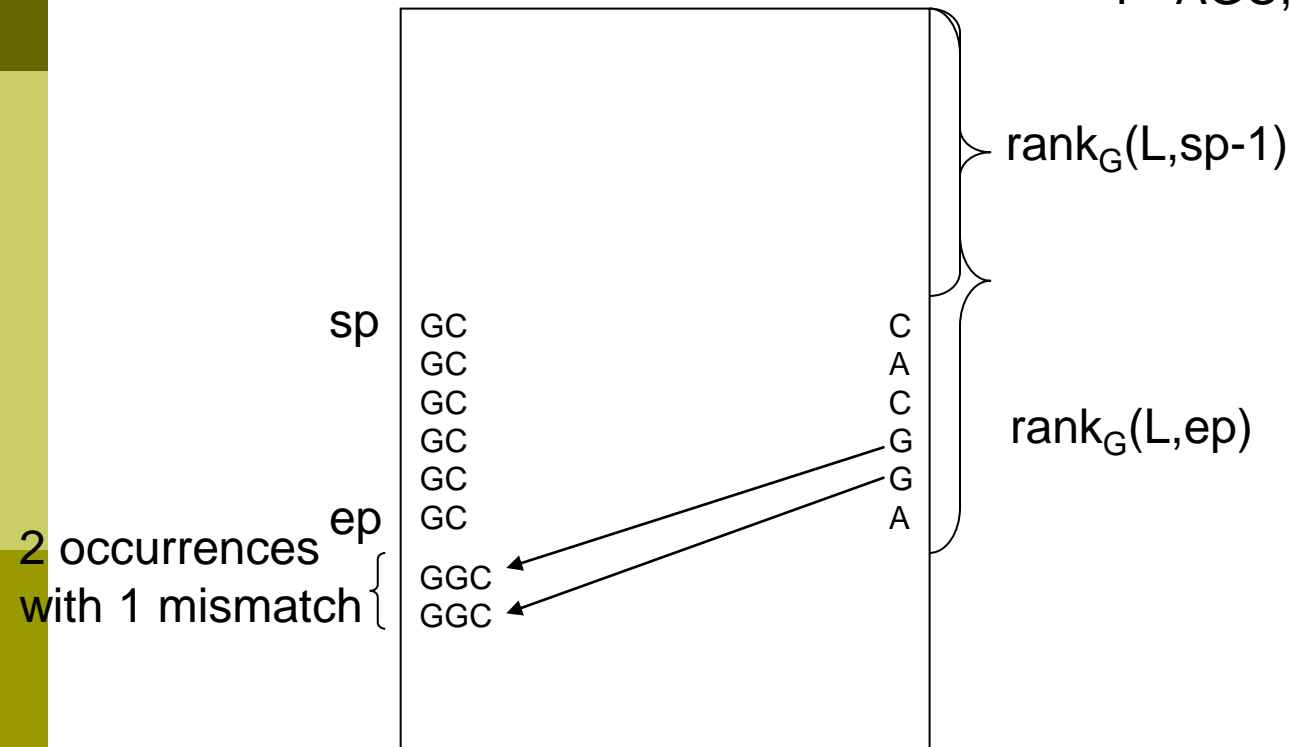
Backward backtracking, one step

$P=AGC, k=1$



Backward backtracking, one step

$P=AGC, k=1$



Backward backtracking – pseudocode

Algorithm kmismatches(P, L, k, j, sp, ep)

(1) **if** ($j = 0$) **then**

(2) Report occurrences **Pos[sp], . . . , Pos[ep]**; **return**;

(3) **for each** $s \in \Sigma$ **do**

(4) $sp' \leftarrow C[s] + \text{rank}_s(L, sp - 1) + 1$;

(5) $ep' \leftarrow C[s] + \text{rank}_s(L, ep)$;

(6) **if** ($P[j] \neq s$) $k' \leftarrow k - 1$; **else** $k' \leftarrow k$;

(7) **if** ($k' \geq 0$) **kmismatches**($P, L, k', j - 1, sp', ep'$);

First call: $\text{kmismatches}(P, L, k, m, 1, n)$

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

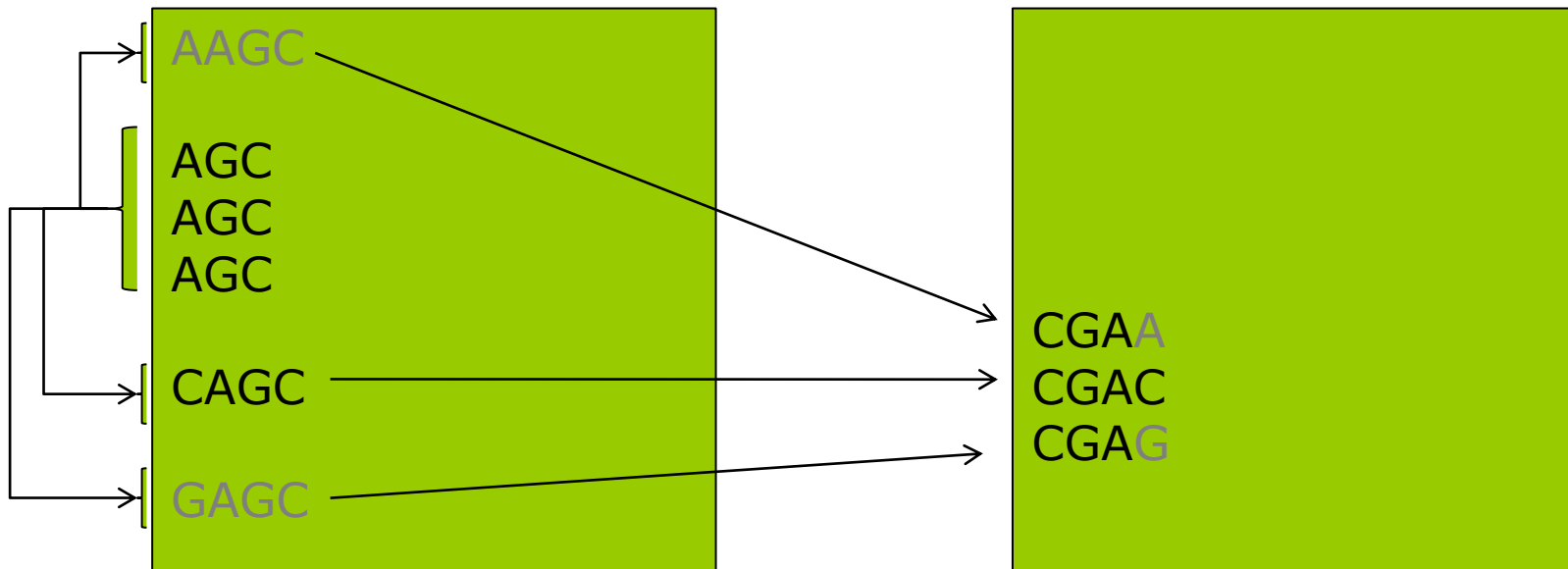
- Build compressed suffix array (aka FM-index) for S and its reverse S^r . Call these *forward FM-index* and *reverse FM-index*.
- Compute a table $\kappa[1,m]$ such that $\kappa[i]$ gives a lower bound for the minimum amount of errors needed to match $P[1,i]$ in S .
 - E.g. $\kappa[i]$ = minimum number of times $P[1,i]$ need to be split such that each piece occurs exactly in S .
 - Initialize $\kappa[0]=0$. With reverse FM-index, backward search P^r from $i=m$ to $i=1$ setting $\kappa[m-i+1]=\kappa[m-i]$ until empty interval, say at $P^r[i']$, then set $\kappa[m-i'+1]=\kappa[m-i']+1$ and continue in the same way.
- Consider search space state corresponding to suffix $P[j,m]$, interval $[sp,ep]$, and k' mismatches. If $k'+\kappa[j-1]>k$, 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 forward FM-index for the search; no branching in first $m/2$ characters. For (ii) category occurrences use reverse 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 errors 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 bidirectional BWT index in the text book).

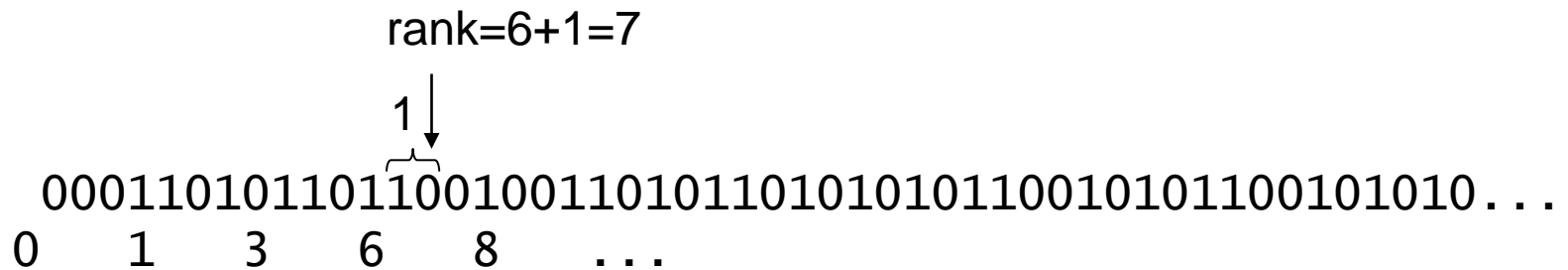


Appendix

PROVING RANK LEMMAS

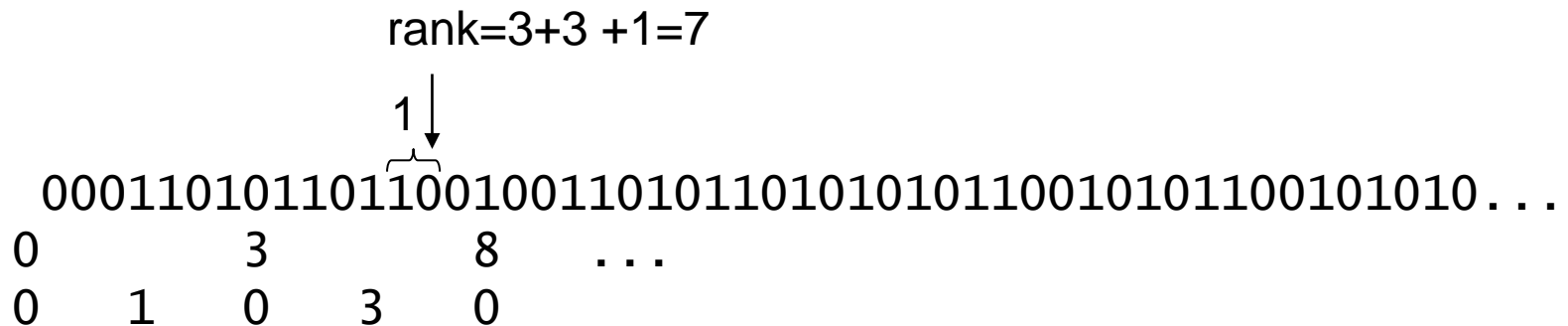
Constant time rank using $o(n)$ extra bits

- $\langle O(n) \text{ space}, O(\log n) \text{ time} \rangle$: Store answers at each $\log n$:th position.
 - Read the rest from the original bitvector.



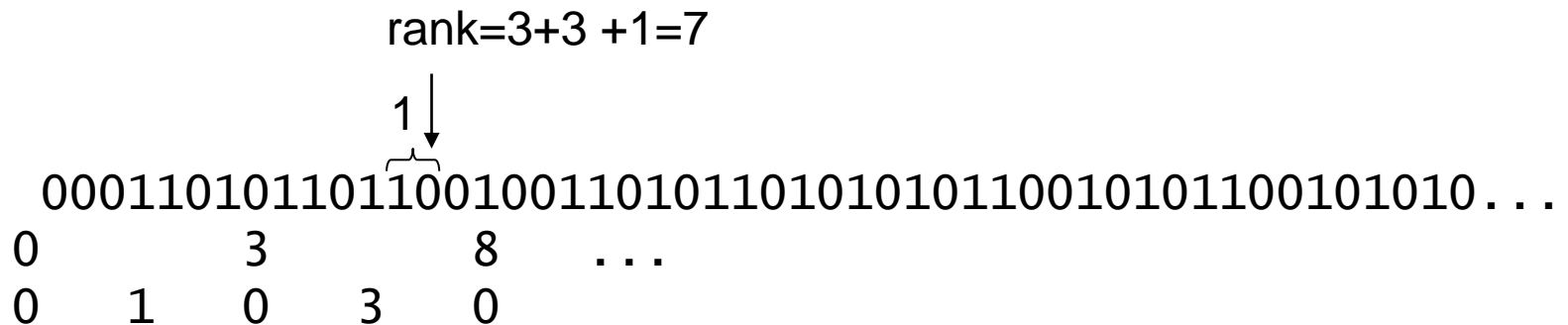
Constant time rank using $o(n)$ extra bits

- $\langle o(n) \text{ space}, O(\log n) \text{ time} \rangle$: 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.



Constant time rank using $o(n)$ extra bits

- $\langle o(n) \text{ space}, O(1) \text{ time} \rangle$: Same as before, but read the last answer from a precomputed table of size $o(n)$.



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

							smallrank	0	1				
	B	0	1	0	1	1	0	1	0	0	1	0	0
superblockrank	0									8	00	0	0
blockrank	0		2		4		7		0		01	0	1
											10	1	1
											11	1	2

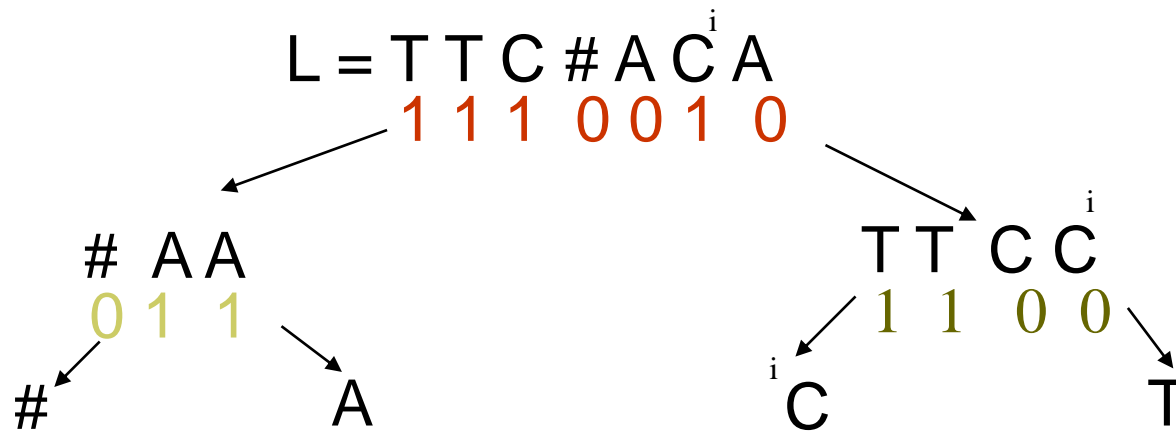
$$\begin{aligned}
 \text{rank}_1(\mathbf{B}, 11) &= \text{superblockrank}[0] + \text{blockrank}[2] + \\
 &\quad \text{smallrank}[\mathbf{01}, 1] + \text{smallrank}[\mathbf{11}, 0] \\
 &= 0 + 4 + 1 + 1 = 6
 \end{aligned}$$

Wavelet tree

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

Wavelet tree, example 1

$\Sigma = \{\#, A, C, G, T\}$



$i = 6$

$B = \color{red}{1110010}$, $B[6] = 1 \rightarrow \text{right}$
 $i = \text{rank}_1(\color{red}{1110010}, 6) = 4$

$B = \color{yellow}{1100}$, $B[4] = 0 \rightarrow \text{left}$
 $i = \text{rank}_0(\color{yellow}{1100}, 4) = 2$

$L[6] = C$

Wavelet tree, example 2

$\Sigma = \{\#, A, C, G, T\}$



$i = 6$

$A \in \{\#, A\} \rightarrow$ left
 $i = \text{rank}_0(1110010, 6) = 2$

$A \in \{A\} \rightarrow$ right
 $i = \text{rank}_1(011, 2) = 1$

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

Rank() function space/time

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