Suffix arrays, BWT and FM-index

Alan Medlar

Wednesday 16th March 2016

Outline

- **Lecture:** Technical background for read mapping tools used in this course
  - Suffix array
  - Burrows-Wheeler transform (BWT)
  - FM-index

- **Lab session:** Using BWA to map paired-end data against the human genome, SAM/BAM files, etc

Read mapping

- Sequencers can generate up to **100 million** reads per sample
- Human genome is **~3 billion** basepairs
- Need to map reads to the genome to discover variants (SNVs, indels), counts (gene expression)

Preliminaries

- String
  - sequence of characters,
    - e.g. "banana", "ATGC", "MDLISTFS"
- Alphabet { A, C, G, T, $ }, { A-Z, a-z, $ }
- Lexicographical order
  - $ < A < C < G < T

Preliminaries

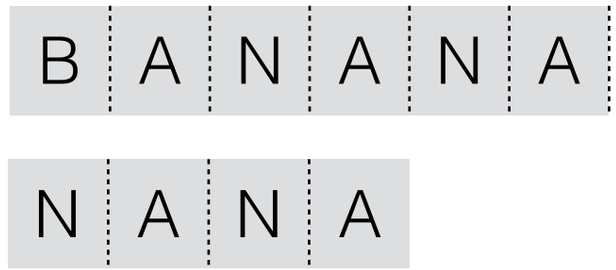
- Prefix
  - non-empty substring that is the beginning of another string (left-to-right)
    - e.g. "**bana**na", "**AT**GC", "**MDL**ISTFS"
- Suffix
  - non-empty substring that is the ending of another string (right-to-left)
    - e.g. "ba**nana**", "AT**GC**", "**MDLISTFS**"

Naïve exact search

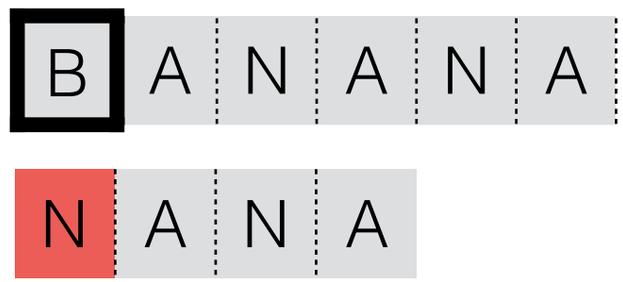
- Text = "banana"
- Query = "nana"
- Linear search

Naïve exact search

- Text = "banana"
- Query = "nana"
- Linear search



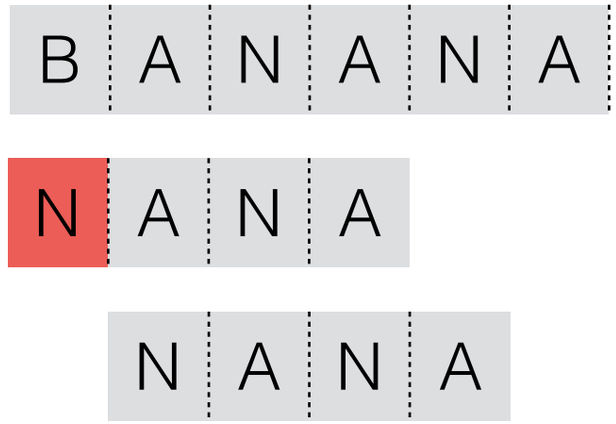
Naïve exact search



- Text = "banana"
- Query = "nana"
- Linear search

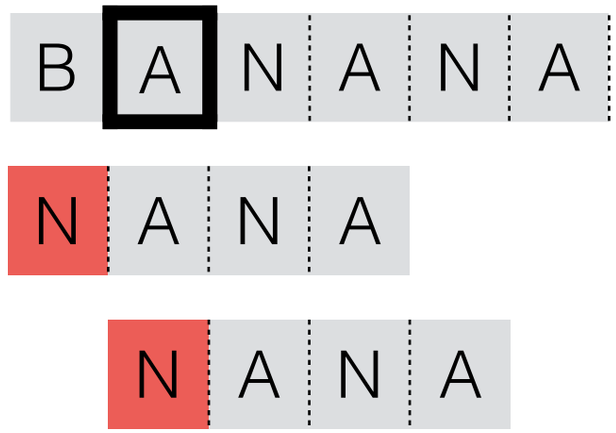
Naïve exact search

- Text = "banana"
- Query = "nana"
- Linear search



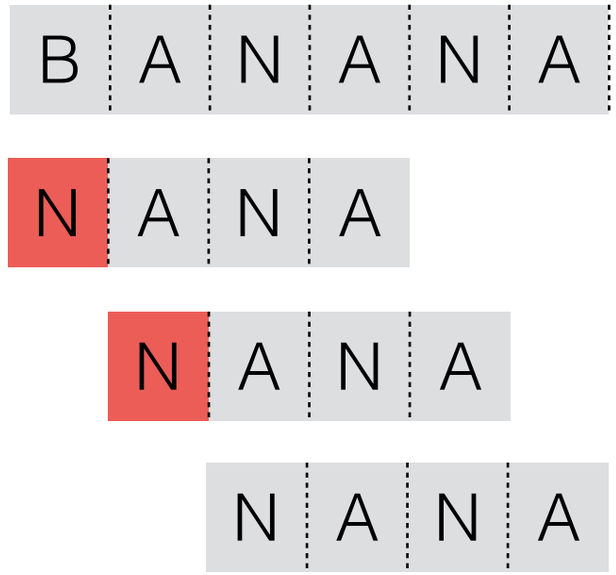
Naïve exact search

- Text = "banana"
- Query = "nana"
- Linear search



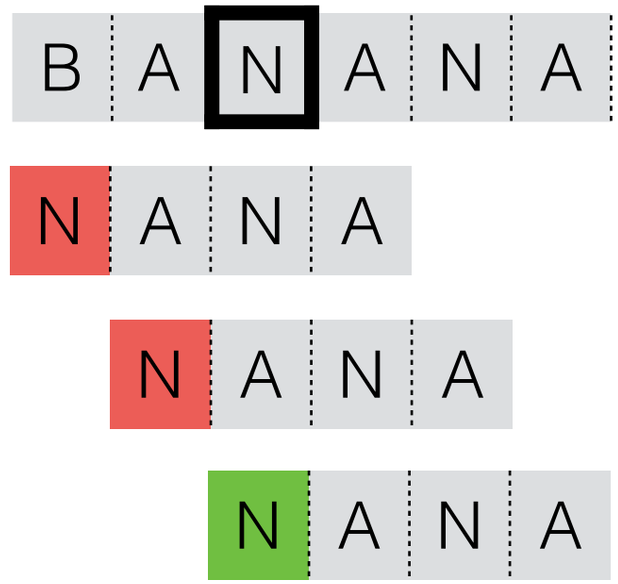
Naïve exact search

- Text = "banana"
- Query = "nana"
- Linear search



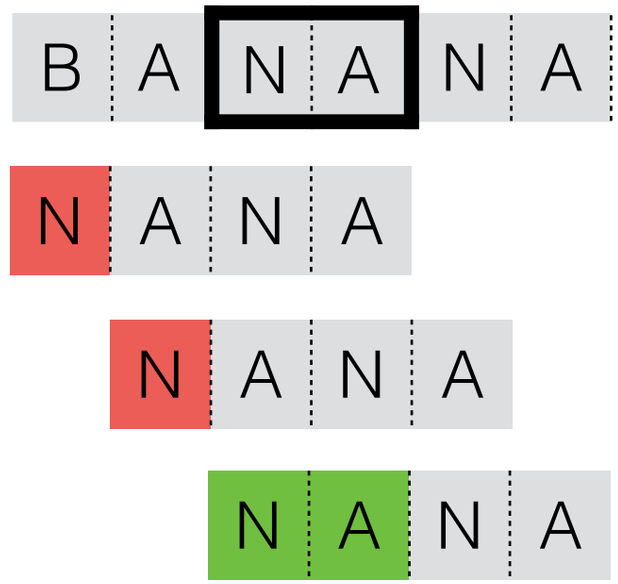
Naïve exact search

- Text = "banana"
- Query = "nana"
- Linear search



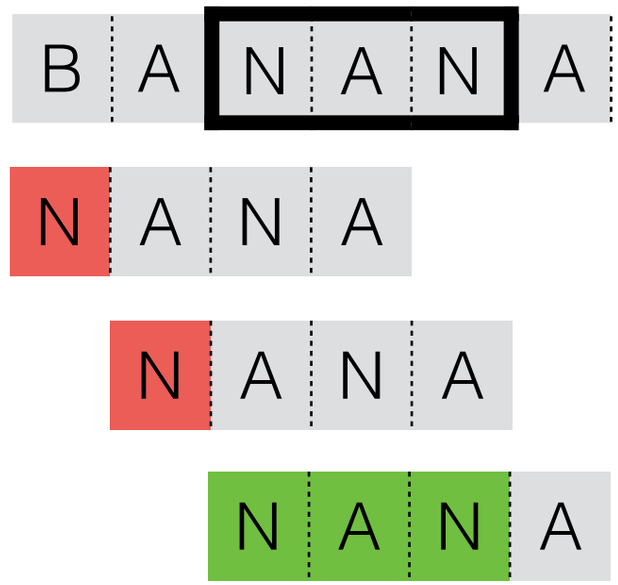
Naïve exact search

- Text = "banana"
- Query = "nana"
- Linear search



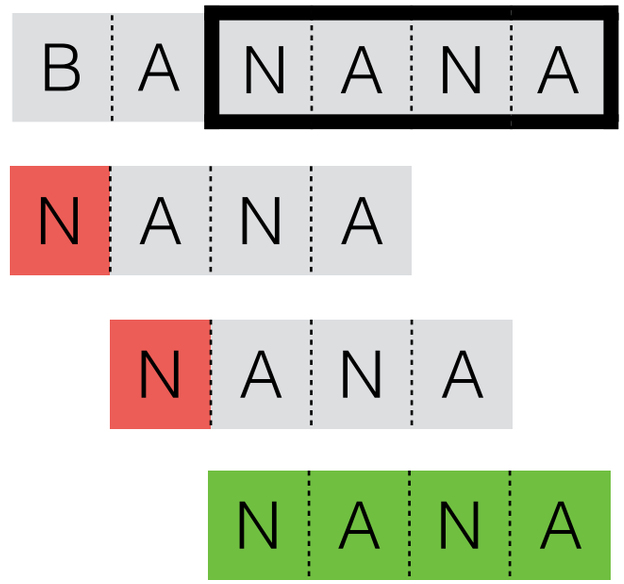
Naïve exact search

- Text = "banana"
- Query = "nana"
- Linear search



Naïve exact search

- Text = "banana"
- Query = "nana"
- Linear search



Naïve search is too slow

- Human genome ~3 billion basepairs
- Read 100 basepairs
- Complexity of search scales **linearly** with the length of the text!

Suffix array

- Introduced by Manber and Myers (1990) as a space efficient alternative to suffix tree (independently by Gonnet (1987))
- Sorted array of all suffixes of a given text
- Allows fast search of very large texts (e.g. genomes)

SA: building

B	A	N	A	N	A	$	

$ is lexicographically **lower** than all other characters in the alphabet and cannot appear in the text otherwise

SA: building

B	A	N	A	N	A	$	
A	N	A	N	A	$		

SA: building

B	A	N	A	N	A	$	
A	N	A	N	A	$		
N	A	N	A	$			
A	N	A	$				
N	A	$					
A	$						
$							

SA: building

B	A	N	A	N	A	$	0
A	N	A	N	A	$		1
N	A	N	A	$			2
A	N	A	$				3
N	A	$					4
A	$						5
$							6

SA: building

B	A	N	A	N	A	$	0
A	N	A	N	A	$		1
N	A	N	A	$			2
A	N	A	$				3
N	A	$					4
A	$						5
$							6

$							6
A	$						5
A	N	A	$				3
A	N	A	N	A	$		1
B	A	N	A	N	A	$	0
N	A	$					4
N	A	N	A	$			2

SA: building

B	A	N	A	N	A	$	0
A	N	A	N	A	$		1
N	A	N	A	$			2
A	N	A	$				3
N	A	$					4
A	$						5
$							6

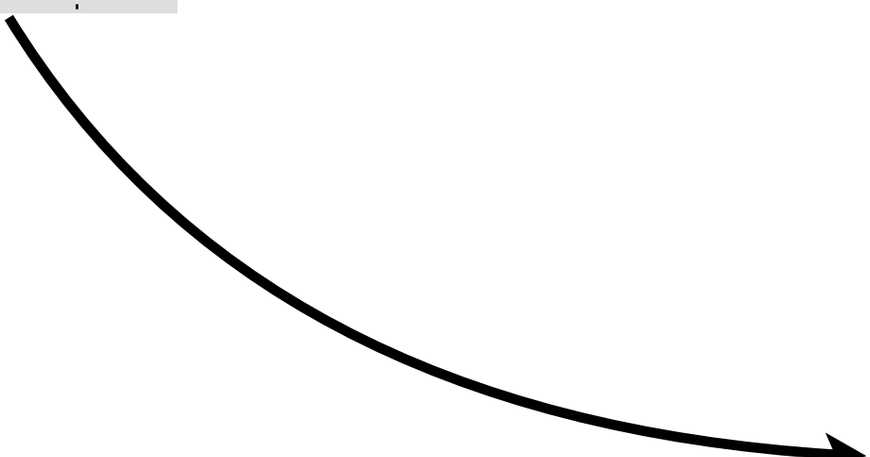
$							6
A	$						5
A	N	A	$				3
A	N	A	N	A	$		1
B	A	N	A	N	A	$	0
N	A	$					4
N	A	N	A	$			2

SA: querying

- Search for **prefixes** in the suffix array that match our query string

- SA is sorted, so we can use binary search!

N	A	N	A



$							6
A	$						5
A	N	A	$				3
A	N	A	N	A	$		1
B	A	N	A	N	A	$	0
N	A	$					4
N	A	N	A	$			2

N	A	N	A

$							6
A	$						5
A	N	A	$				3
A	N	A	N	A	$		1
B	A	N	A	N	A	$	0
N	A	$					4
N	A	N	A	$			2

$							6
A	$						5
A	N	A	$				3
A	N	A	N	A	$		1
B	A	N	A	N	A	$	0
N	A	$					4
N	A	N	A	$			2

N	A	N	A

$							6
A	$						5
A	N	A	$				3
A	N	A	N	A	$		1
B	A	N	A	N	A	$	0
N	A	$					4
N	A	N	A	$			2

N	A	N	A

$							6
A	$						5
A	N	A	$				3
A	N	A	N	A	$		1
B	A	N	A	N	A	$	0
N	A	$					4
N	A	N	A	$			2

N	A	N	A

$							6
A	$						5
A	N	A	$				3
A	N	A	N	A	$		1
B	A	N	A	N	A	$	0
N	A	$					4
N	A	N	A	$			2

N	A	N	A

$							6
A	$						5
A	N	A	$				3
A	N	A	N	A	$		1
B	A	N	A	N	A	$	0
N	A	$					4
N	A	N	A	$			2

N	A	N	A

$							6
A	$						5
A	N	A	$				3
A	N	A	N	A	$		1
B	A	N	A	N	A	$	0
N	A	$					4
N	A	N	A	$			2

N	A	N	A

$							6
A	$						5
A	N	A	$				3
A	N	A	N	A	$		1
B	A	N	A	N	A	$	0
N	A	$					4
N	A	N	A	$			2

N	A	N	A

$							6
A	$						5
A	N	A	$				3
A	N	A	N	A	$		1
B	A	N	A	N	A	$	0
N	A	$					4
N	A	N	A	$			2

N	A	N	A

$							6
A	$						5
A	N	A	$				3
A	N	A	N	A	$		1
B	A	N	A	N	A	$	0
N	A	$					4
N	A	N	A	$			2

N	A	N	A

$							6
A	$						5
A	N	A	$				3
A	N	A	N	A	$		1
B	A	N	A	N	A	$	0
N	A	$					4
N	A	N	A	$			2

N	A	N	A

$							6
A	$						5
A	N	A	$				3
A	N	A	N	A	$		1
B	A	N	A	N	A	$	0
N	A	$					4
N	A	N	A	$			2

N	A	N	A

$							6
A	$						5
A	N	A	$				3
A	N	A	N	A	$		1
B	A	N	A	N	A	$	0
N	A	$					4
N	A	N	A	$			2

N	A	N	A

SA vs. naïve search

- Searching the human genome (~3 billion basepairs, n) for a single-end read (100 basepairs, m)
- Naïve search O(mn)
- Suffix array search O(m log(n))

SA vs. naïve search

- Searching the human genome (~3 billion basepairs, n) for a single-end read (100 basepairs, m)
- Naïve search O(mn)
- Suffix array search O(m log(n))

n	O(n)	O(log(n))
8	8	3
16	16	4
32	32	5
64	64	6
128	128	7
256	256	8
512	512	9

Good enough for read mapping?

- Human genome is ~3 billion basepairs
- Assume 5 bytes per basepair (1 byte characters, 4 byte integers) = **~14 GB**
- NGS data really hit in 2009 (16 GB RAM at the time was a luxury!)

Burrows-Wheeler transform

- Invented by Burrows and Wheeler (1994) while working at DEC
- Used in compression (.bz2 files)
- Interested in three things:
  - how to perform BWT
  - why BWT is useful for compression
  - how to reverse BWT

BWT

B	A	N	A	N	A	$

BWT

B	A	N	A	N	A	$
A	N	A	N	A	$	

BWT

B	A	N	A	N	A	$
A	N	A	N	A	$	B

BWT

B	A	N	A	N	A	$
A	N	A	N	A	$	B
N	A	N	A	$		

BWT

B	A	N	A	N	A	$
A	N	A	N	A	$	B
N	A	N	A	$	B	A

BWT

B	A	N	A	N	A	$
A	N	A	N	A	$	B
N	A	N	A	$	B	A
A	N	A	$	B	A	N
N	A	$	B	A	N	A
A	$	B	A	N	A	N
$	B	A	N	A	N	A

BWT

B	A	N	A	N	A	$
A	N	A	N	A	$	B
N	A	N	A	$	B	A
A	N	A	$	B	A	N
N	A	$	B	A	N	A
A	$	B	A	N	A	N
$	B	A	N	A	N	A

$	B	A	N	A	N	A
A	$	B	A	N	A	N
A	N	A	$	B	A	N
A	N	A	N	A	$	B
B	A	N	A	N	A	$
N	A	$	B	A	N	A
N	A	N	A	$	B	A

BWT

$	B	A	N	A	N	**A**
A	$	B	A	N	A	**N**
A	N	A	$	B	A	**N**
A	N	A	N	A	$	**B**
B	A	N	A	N	A	**$**
N	A	$	B	A	N	**A**
N	A	N	A	$	B	**A**

BWT compression

- T = "banana$"
- BWT(T) = "annb$aa"

BWT compression

- T = "peter_piper_picked_a_peck_of_pickled_peppers_a_peck_of_pickled_peppers_peter_piper_picked_if_peter_piper_picked_a_peck_of_pickled_peppers_wheres_the_peck_of_pickled_peppers_peter_piper_picked"

- BWT(T) = "ddsddkkkkaeaaddddsfsrrrffffrrrrss___eeeeiiiiiiieeeeeeeehppppkkkkllllpppppppppttttthpppprpppppiooootwpppppppppp_ppppccccccccccckkkk_____iiiipppp_____eeeeeeeeeeeeeeeerrrereeee__"

Relation to suffix array

- BWT matrix truncated at "$" in each row **is** the suffix array of the same text
- BWT can be computed directly from the suffix array

$	B	A	N	A	N	A
A	**$**	B	A	N	A	N
A	**N**	**A**	**$**	B	A	N
A	**N**	**A**	**N**	**A**	**$**	B
B	**A**	**N**	**A**	**N**	**A**	**$**
N	**A**	**$**	B	A	N	A
N	**A**	**N**	**A**	**$**	B	A

Reverse BWT

- It not very useful to compress something if we cannot get the original text back!

- BWT'(BWT(T)) = T

LF mapping (T-rank)

B	A	N	A	N	A	$

LF mapping (T-rank)

	B	A	N	A	N	A	$
T-rank	B_0	A_0	N_0	A_1	N_1	A_2	$

LF mapping (T-rank)

	B	A	N	A	N	A	$
T-rank	B_0	A_0	N_0	A_1	N_1	A_2	$

F						**L**
$	B_0	A_0	N_0	A_1	N_1	A_2
A_2	$	B_0	A_0	N_0	A_1	N_1
A_1	N_1	A_2	$	B_0	A_0	N_0
A_0	N_0	A_1	N_1	A_2	$	B_0
B_0	A_0	N_0	A_1	N_1	A_2	$
N_1	A_2	$	B_0	A_0	N_0	A_1
N_0	A_1	N_1	A_2	$	B_0	A_0

LF mapping (T-rank)

	B	A	N	A	N	A	$
T-rank	B_0	A_0	N_0	A_1	N_1	A_2	$

F						**L**
$	B_0	A_0	N_0	A_1	N_1	**A_2**
A_2	$	B_0	A_0	N_0	A_1	**N_1**
A_1	N_1	A_2	$	B_0	A_0	**N_0**
A_0	N_0	A_1	N_1	A_2	$	**B_0**
B_0	A_0	N_0	A_1	N_1	A_2	**$**
N_1	A_2	$	B_0	A_0	N_0	**A_1**
N_0	A_1	N_1	A_2	$	B_0	**A_0**

LF mapping (T-rank)

	B	A	N	A	N	A	$
T-rank	B_0	A_0	N_0	A_1	N_1	A_2	$

F						**L**
$	B_0	A_0	N_0	A_1	N_1	**A_2**
A_2	$	B_0	A_0	N_0	A_1	**N_1**
A_1	N_1	A_2	$	B_0	A_0	**N_0**
A_0	N_0	A_1	N_1	A_2	$	**B_0**
B_0	A_0	N_0	A_1	N_1	A_2	**$**
N_1	A_2	$	B_0	A_0	N_0	**A_1**
N_0	A_1	N_1	A_2	$	B_0	**A_0**

LF mapping (T-rank)

	B	A	N	A	N	A	$
T-rank	B_0	A_0	N_0	A_1	N_1	A_2	$

F						**L**
$	B_0	A_0	N_0	A_1	N_1	**A_2**
A_2	$	B_0	A_0	N_0	A_1	**N_1**
A_1	N_1	A_2	$	B_0	A_0	**N_0**
A_0	N_0	A_1	N_1	A_2	$	**B_0**
B_0	A_0	N_0	A_1	N_1	A_2	**$**
N_1	A_2	$	B_0	A_0	N_0	**A_1**
N_0	A_1	N_1	A_2	$	B_0	**A_0**

LF mapping (T-rank)

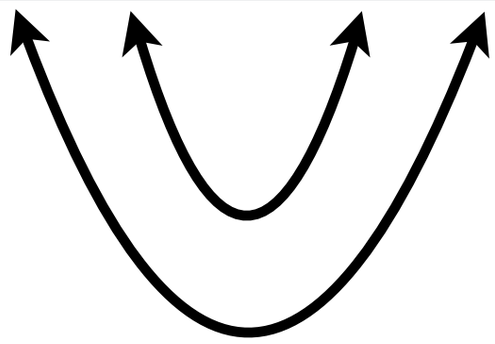
	B	A	N	A	N	A	$
T-rank	B_0	A_0	N_0	A_1	N_1	A_2	$

Ns in the L column are sorted by their "**right context**", same as Ns in F column!

F						**L**
$	B_0	A_0	N_0	A_1	N_1	**A_2**
A_2	$	B_0	A_0	N_0	A_1	**N_1**
A_1	N_1	A_2	$	B_0	A_0	**N_0**
A_0	N_0	A_1	N_1	A_2	$	**B_0**
B_0	A_0	N_0	A_1	N_1	A_2	**$**
N_1	A_2	$	B_0	A_0	N_0	**A_1**
N_0	A_1	N_1	A_2	$	B_0	**A_0**

LF mapping (B-rank)

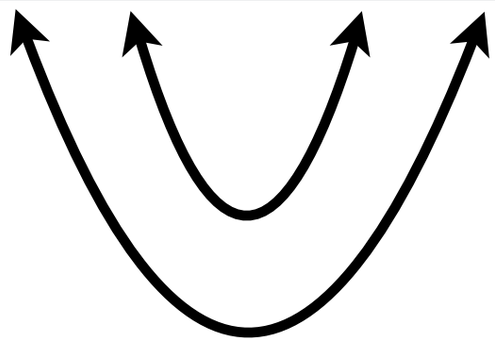
T-rank	B_0	A_0	N_0	A_1	N_1	A_2	$\$$
B-rank	B_0	A_2	N_1	A_1	N_0	A_0	$\$$



F						**L**
$\$$	B_0	A_2	N_1	A_1	N_0	A_0
A_0	$\$$	B_0	A_2	N_1	A_1	N_0
A_1	N_0	A_0	$\$$	B_0	A_2	N_1
A_2	N_1	A_1	N_0	A_0	$\$$	B_0
B_0	A_2	N_1	A_1	N_0	A_0	$\$$
N_0	A_0	$\$$	B_0	A_2	N_1	A_1
N_1	A_1	N_0	A_0	$\$$	B_0	A_2

LF mapping (B-rank)

T-rank	B_0	A_0	N_0	A_1	N_1	A_2	$\$$
B-rank	B_0	A_2	N_1	A_1	N_0	A_0	$\$$



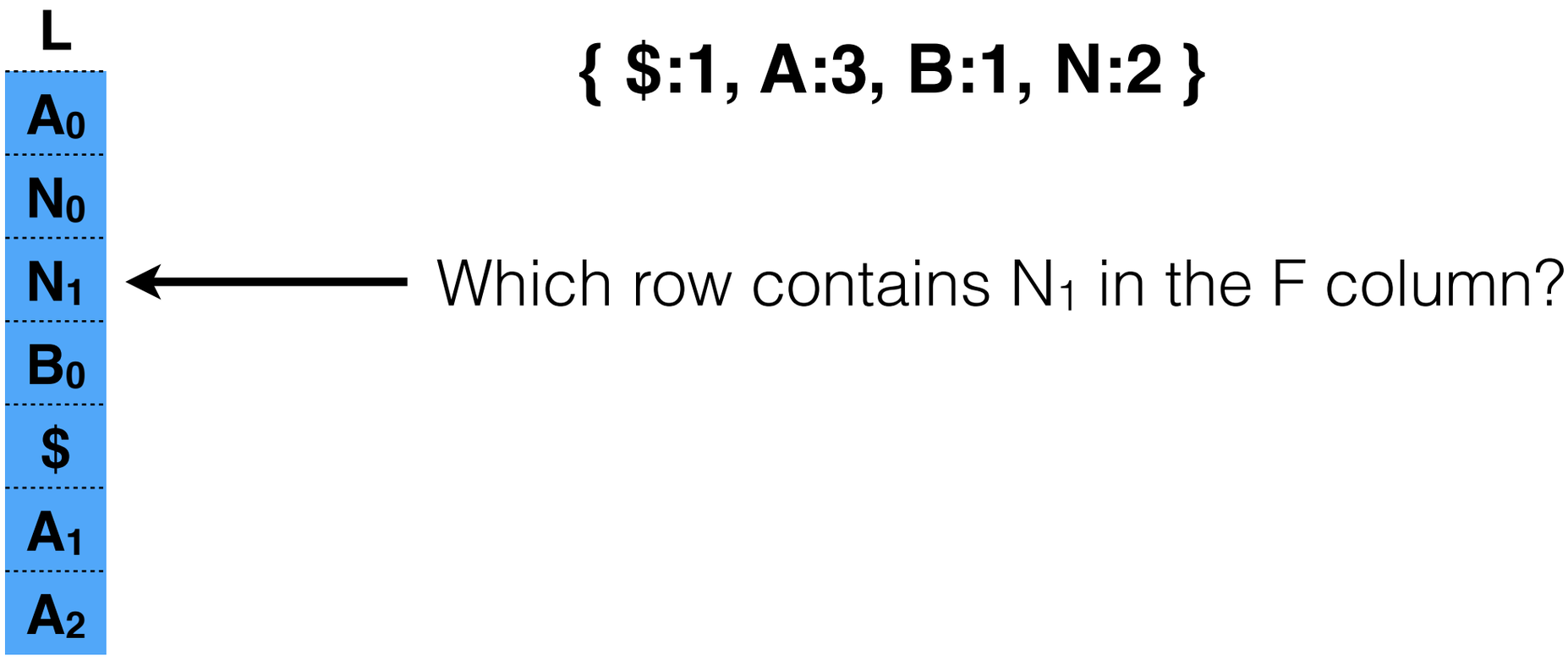
F						**L**
$	B_0	A_2	N_1	A_1	N_0	**A_0**
A_0	$\$$	B_0	A_2	N_1	A_1	**N_0**
A_1	N_0	A_0	$\$$	B_0	A_2	**N_1**
A_2	N_1	A_1	N_0	A_0	$\$$	**B_0**
B_0	A_2	N_1	A_1	N_0	A_0	**$\$$**
N_0	A_0	$\$$	B_0	A_2	N_1	**A_1**
N_1	A_1	N_0	A_0	$\$$	B_0	**A_2**

LF mapping (B-rank)

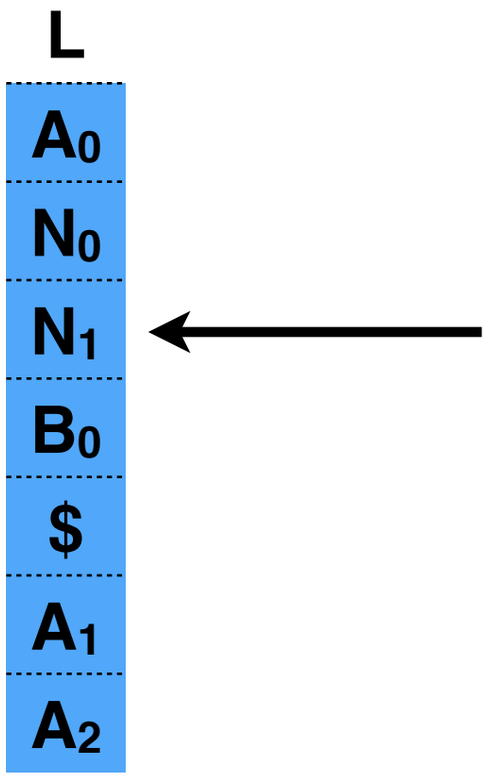
- F column contains very little information, just counts of each character

F						L
$	B_0	A_2	N_1	A_1	N_0	**A_0**
A_0	$	B_0	A_2	N_1	A_1	**N_0**
A_1	N_0	A_0	$	B_0	A_2	**N_1**
A_2	N_1	A_1	N_0	A_0	$	**B_0**
B_0	A_2	N_1	A_1	N_0	A_0	**$**
N_0	A_0	$	B_0	A_2	N_1	**A_1**
N_1	A_1	N_0	A_0	$	B_0	**A_2**

LF mapping (B-rank)



LF mapping (B-rank)



{ $:1, A:3, B:1, N:2 }

Which row contains N_1 in the F column?

- Skip $ (+1)
- Skip As (+3)
- Skip Bs (+1)
- Skip first N (+1) = 6

LF mapping (B-rank)

	F	L
0	**$**	**A_0**
1	**A_0**	**N_0**
2	**A_1**	**N_1**
3	**A_2**	**B_0**
4	**B_0**	**$**
5	**N_0**	**A_1**
6	**N_1**	**A_2**

{ $:1, A:3, B:1, N:2 }

Which row contains N_1 in the F column?

- Skip $ (+1)
- Skip As (+3)
- Skip Bs (+1)
- Skip first N (+1) = 6

Reverse BWT

- Use B-ranking to reverse BWT, recreating the text T from right-to-left

{ $:1, A:3, B:1, N:2 }

	F	L
0	**$**	**A_0**
1	**A_0**	**N_0**
2	**A_1**	**N_1**
3	**A_2**	**B_0**
4	**B_0**	**$**
5	**N_0**	**A_1**
6	**N_1**	**A_2**

B_0	A_2	N_1	A_1	N_0	A_0	$
						$

Reverse BWT

- Use B-ranking to reverse BWT, recreating the text T from right-to-left

{ $:1, A:3, B:1, N:2 }

	F	**L**
0	**$**	**A_0**
1	**A_0**	**N_0**
2	**A_1**	**N_1**
3	**A_2**	**B_0**
4	**B_0**	**$**
5	**N_0**	**A_1**
6	**N_1**	**A_2**

B_0	A_2	N_1	A_1	N_0	A_0	$

					A_0	$

Reverse BWT

- Use B-ranking to reverse BWT, recreating the text T from right-to-left

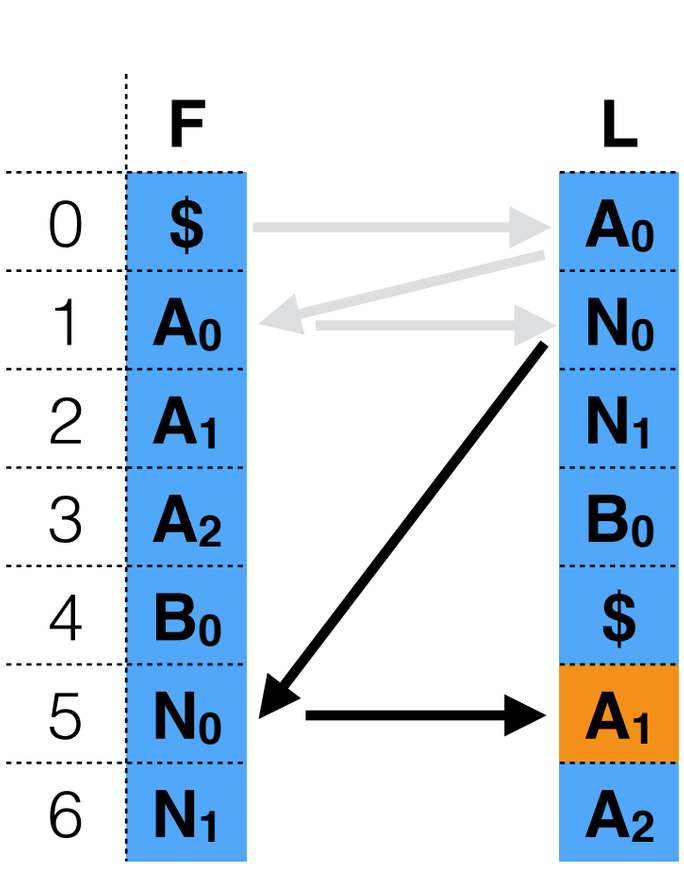
{ $:1, A:3, B:1, N:2 }

	F	L
0	$	A_0
1	A_0	N_0
2	A_1	N_1
3	A_2	B_0
4	B_0	$
5	N_0	A_1
6	N_1	A_2

B_0	A_2	N_1	A_1	N_0	A_0	$
				N_0	A_0	$

Reverse BWT

- Use B-ranking to reverse BWT, recreating the text T from right-to-left

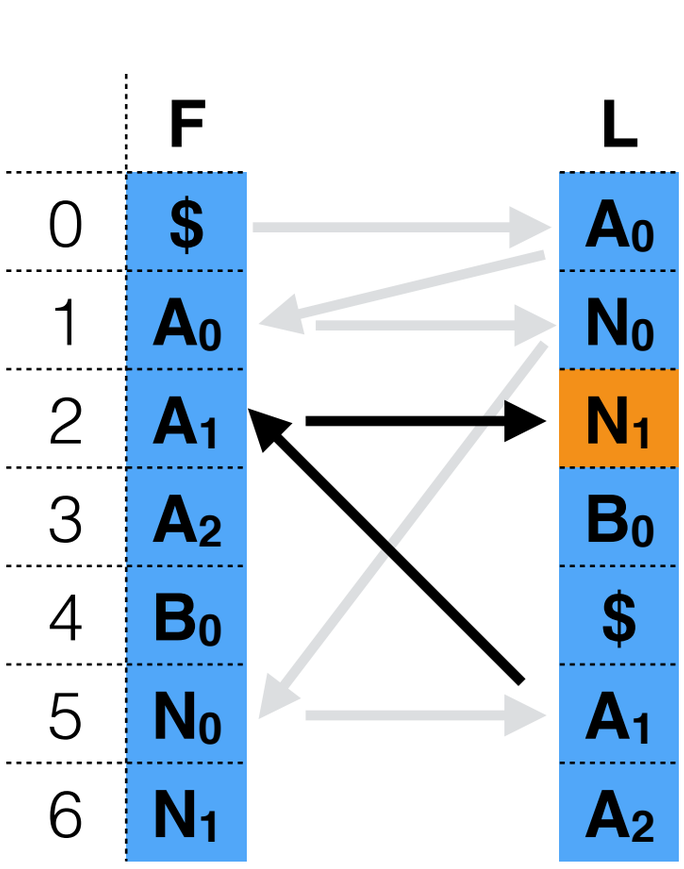


B_0	A_2	N_1	A_1	N_0	A_0	$

			A_1	N_0	A_0	$

Reverse BWT

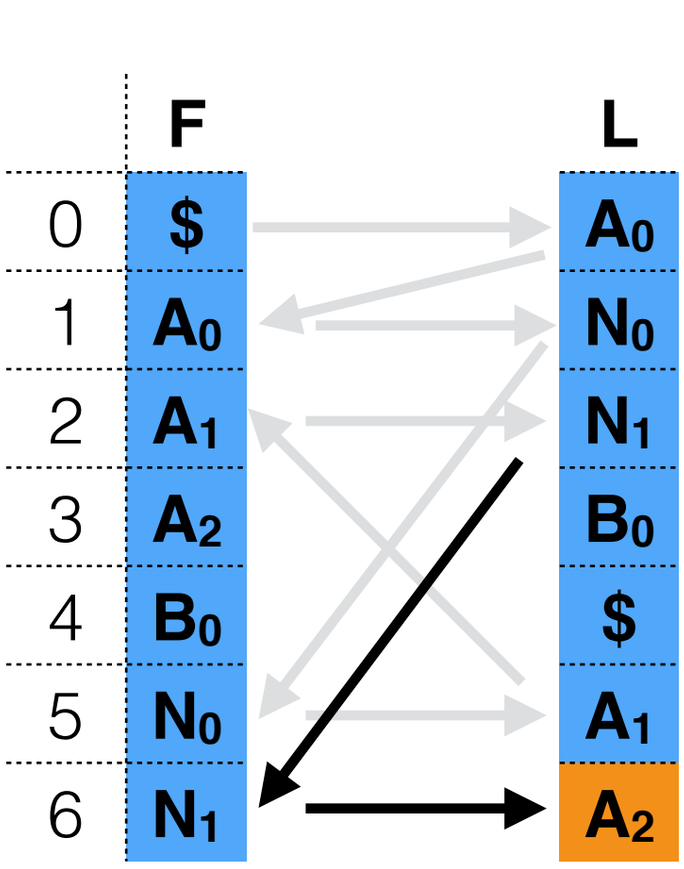
- Use B-ranking to reverse BWT, recreating the text T from right-to-left



B_0	A_2	N_1	A_1	N_0	A_0	$
		N_1	A_1	N_0	A_0	$

Reverse BWT

- Use B-ranking to reverse BWT, recreating the text T from right-to-left

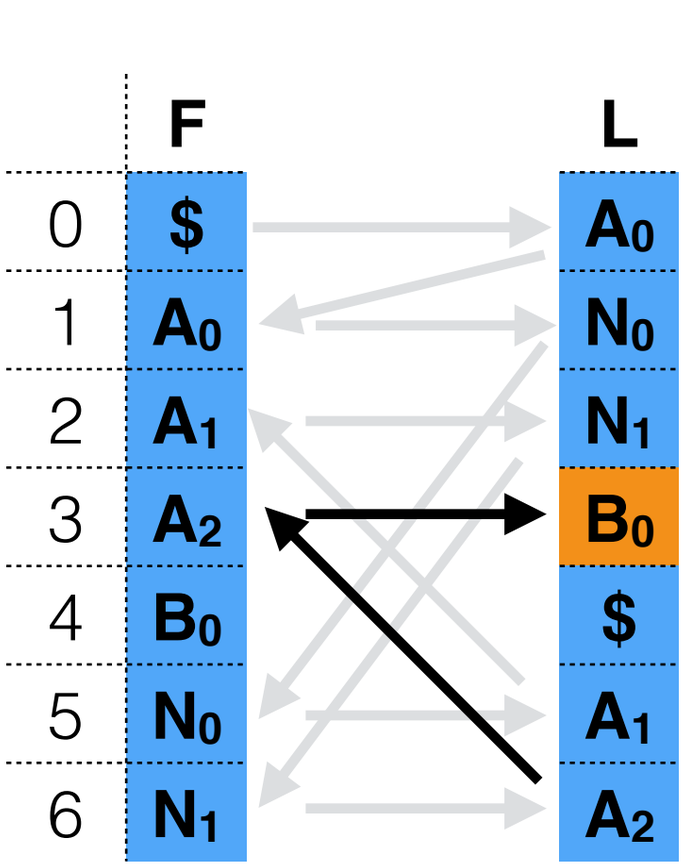


B_0	A_2	N_1	A_1	N_0	A_0	$\$$

	A_2	N_1	A_1	N_0	A_0	$\$$

Reverse BWT

- Use B-ranking to reverse BWT, recreating the text T from right-to-left

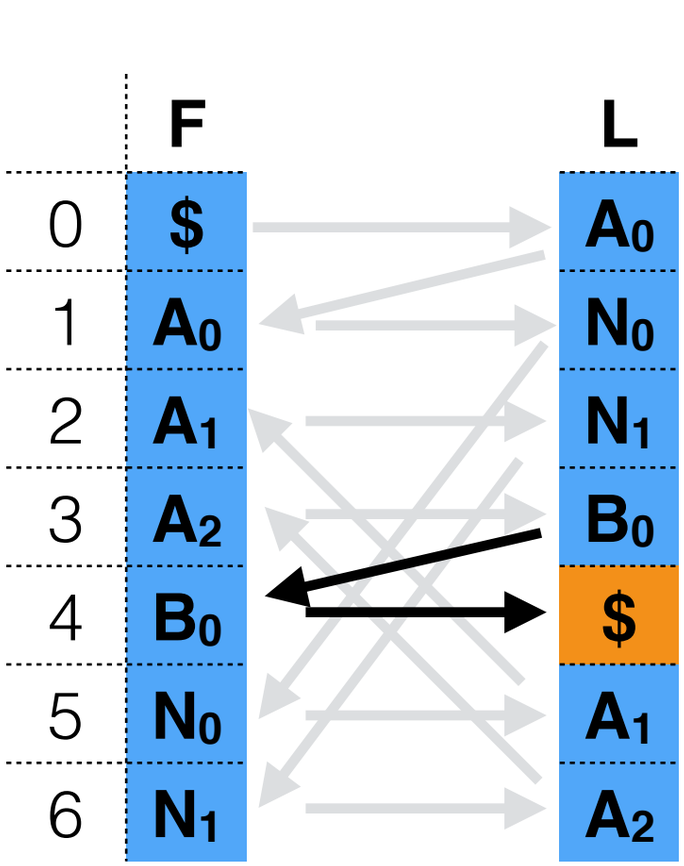


B_0	A_2	N_1	A_1	N_0	A_0	$\$$

B_0	A_2	N_1	A_1	N_0	A_0	$\$$

Reverse BWT

- Use B-ranking to reverse BWT, recreating the text T from right-to-left



B_0	A_2	N_1	A_1	N_0	A_0	$\$$

B_0	A_2	N_1	A_1	N_0	A_0	$\$$

FM-index

- All BWT allows us to do is compress text
- Ferragina and Manzini (2000)
  "**F**ull-text index in **M**inute space"
- Combine BWT with other auxiliary data structures to get an index
- Space savings: e.g. Human genome (3 billion bp)
  - SA = ~14 GB (5 bytes/bp)
  - FM = ~1.5 GB (2 bits/bp)

Cannot search BWT like SA

- Rotation matrix contains the suffix array

$	B	A	N	A	N	A
A	**$**	B	A	N	A	N
A	**N**	**A**	**$**	B	A	N
A	**N**	**A**	**N**	**A**	**$**	B
B	**A**	**N**	**A**	**N**	**A**	**$**
N	**A**	**$**	B	A	N	A
N	**A**	**N**	**A**	**$**	B	A

Cannot search BWT like SA

- Rotation matrix contains the suffix array
- But we only store F and L columns, so binary search of prefixes not possible

$	B	A	N	A	N	**A**
A	$	B	A	N	A	**N**
A	N	A	$	B	A	**N**
A	N	A	N	A	$	**B**
B	A	N	A	N	A	**$**
N	A	$	B	A	N	**A**
N	A	N	A	$	B	**A**

BWT search

- In SA, we matched successively longer **prefixes** (left-to-right) of query string (binary search)
- In BWT, we will match successively longer **suffixes** (right-to-left) of query string (reverse BWT transform)

BWT search

N	A	N	A

F						**L**
$	B_0	A_2	N_1	A_1	N_0	**A_0**
A_0	$\$$	B_0	A_2	N_1	A_1	**N_0**
A_1	N_0	A_0	$\$$	B_0	A_2	**N_1**
A_2	N_1	A_1	N_0	A_0	$\$$	**B_0**
B_0	A_2	N_1	A_1	N_0	A_0	**$\$$**
N_0	A_0	$\$$	B_0	A_2	N_1	**A_1**
N_1	A_1	N_0	A_0	$\$$	B_0	**A_2**

BWT search

N	A	N	A

F						**L**
$	B_0	A_2	N_1	A_1	N_0	**A_0**
A_0	$\$$	B_0	A_2	N_1	A_1	**N_0**
A_1	N_0	A_0	$\$$	B_0	A_2	**N_1**
A_2	N_1	A_1	N_0	A_0	$\$$	**B_0**
B_0	A_2	N_1	A_1	N_0	A_0	**$\$$**
N_0	A_0	$\$$	B_0	A_2	N_1	**A_1**
N_1	A_1	N_0	A_0	$\$$	B_0	**A_2**

BWT search

N A N A

F						L
$	B_0	A_2	N_1	A_1	N_0	**A_0**
A_0	$\$$	B_0	A_2	N_1	A_1	**N_0**
A_1	N_0	A_0	$\$$	B_0	A_2	**N_1**
A_2	N_1	A_1	N_0	A_0	$\$$	**B_0**
B_0	A_2	N_1	A_1	N_0	A_0	**$\$$**
N_0	A_0	$\$$	B_0	A_2	N_1	**A_1**
N_1	A_1	N_0	A_0	$\$$	B_0	**A_2**

BWT search

N	A	N	A

F						**L**
$	B_0	A_2	N_1	A_1	N_0	**A_0**
A_0	$\$$	B_0	A_2	N_1	A_1	**N_0**
A_1	N_0	A_0	$\$$	B_0	A_2	**N_1**
A_2	N_1	A_1	N_0	A_0	$\$$	**B_0**
B_0	A_2	N_1	A_1	N_0	A_0	**$\$$**
N_0	A_0	$\$$	B_0	A_2	N_1	**A_1**
N_1	A_1	N_0	A_0	$\$$	B_0	**A_2**

BWT search

N	A	N	A

F						**L**
$	B_0	A_2	N_1	A_1	N_0	**A_0**
A_0	$\$$	B_0	A_2	N_1	A_1	**N_0**
A_1	N_0	A_0	$\$$	B_0	A_2	**N_1**
A_2	N_1	A_1	N_0	A_0	$\$$	**B_0**
B_0	A_2	N_1	A_1	N_0	A_0	**$\$$**
N_0	A_0	$\$$	B_0	A_2	N_1	**A_1**
N_1	A_1	N_0	A_0	$\$$	B_0	**A_2**

BWT search

N	A	N	A

F						L
$	B_0	A_2	N_1	A_1	N_0	**A_0**
A_0	$	B_0	A_2	N_1	A_1	**N_0**
A_1	N_0	A_0	$	B_0	A_2	**N_1**
A_2	N_1	A_1	N_0	A_0	$	**B_0**
B_0	A_2	N_1	A_1	N_0	A_0	**$**
N_0	A_0	$	B_0	A_2	N_1	**A_1**
N_1	A_1	N_0	A_0	$	B_0	**A_2**

BWT search

N	A	N	A

F						**L**
$	B_0	A_2	N_1	A_1	N_0	**A_0**
A_0	$\$$	B_0	A_2	N_1	A_1	**N_0**
A_1	N_0	A_0	$\$$	B_0	A_2	**N_1**
A_2	N_1	A_1	N_0	A_0	$\$$	**B_0**
B_0	A_2	N_1	A_1	N_0	A_0	**$\$$**
N_0	A_0	$\$$	B_0	A_2	N_1	**A_1**
N_1	A_1	N_0	A_0	$\$$	B_0	**A_2**

BWT search

N	A	N	A

We know BWT **contains** the query, but unlike SA, we do not know the **location** of the match in T!

F						**L**
$	B_0	A_2	N_1	A_1	N_0	**A_0**
A_0	$\$$	B_0	A_2	N_1	A_1	**N_0**
A_1	N_0	A_0	$\$$	B_0	A_2	**N_1**
A_2	N_1	A_1	N_0	A_0	$\$$	**B_0**
B_0	A_2	N_1	A_1	N_0	A_0	**$\$$**
N_0	A_0	$\$$	B_0	A_2	N_1	**A_1**
N_1	A_1	N_0	A_0	$\$$	B_0	**A_2**

BWT search

F						L								
$	B_0	A_2	N_1	A_1	N_0	**A_0**	$							6
A_0	$	B_0	A_2	N_1	A_1	**N_0**	A	$						5
A_1	N_0	A_0	$	B_0	A_2	**N_1**	A	N	A	$				3
A_2	N_1	A_1	N_0	A_0	$	**B_0**	A	N	A	N	A	$		1
B_0	A_2	N_1	A_1	N_0	A_0	**$**	B	A	N	A	N	A	$	0
N_0	A_0	$	B_0	A_2	N_1	**A_1**	N	A	$					4
N_1	A_1	N_0	A_0	$	B_0	**A_2**	N	A	N	A	$			2

N	A	N	A

Idea: just store SA as well?

BWT search

F						**L**
$	B_0	A_2	N_1	A_1	N_0	**A_0**
A_0	$	B_0	A_2	N_1	A_1	**N_0**
A_1	N_0	A_0	$	B_0	A_2	**N_1**
A_2	N_1	A_1	N_0	A_0	$	**B_0**
B_0	A_2	N_1	A_1	N_0	A_0	**$**
N_0	A_0	$	B_0	A_2	N_1	**A_1**
N_1	A_1	N_0	A_0	$	B_0	**A_2**

$							6
A	$						
A	N	A	$				3
A	N	A	N	A	$		
B	A	N	A	N	A	$	0
N	A	$					
N	A	N	A	$			

N	A	N	A

Idea 2: store part of SA?

BWT search

F						L
$	B_0	A_2	N_1	A_1	N_0	**A_0**
A_0	$	B_0	A_2	N_1	A_1	**N_0**
A_1	N_0	A_0	$	B_0	A_2	**N_1**
A_2	N_1	A_1	N_0	A_0	$	**B_0**
B_0	A_2	N_1	A_1	N_0	A_0	**$**
N_0	A_0	$	B_0	A_2	N_1	**A_1**
N_1	A_1	N_0	A_0	$	B_0	**A_2**

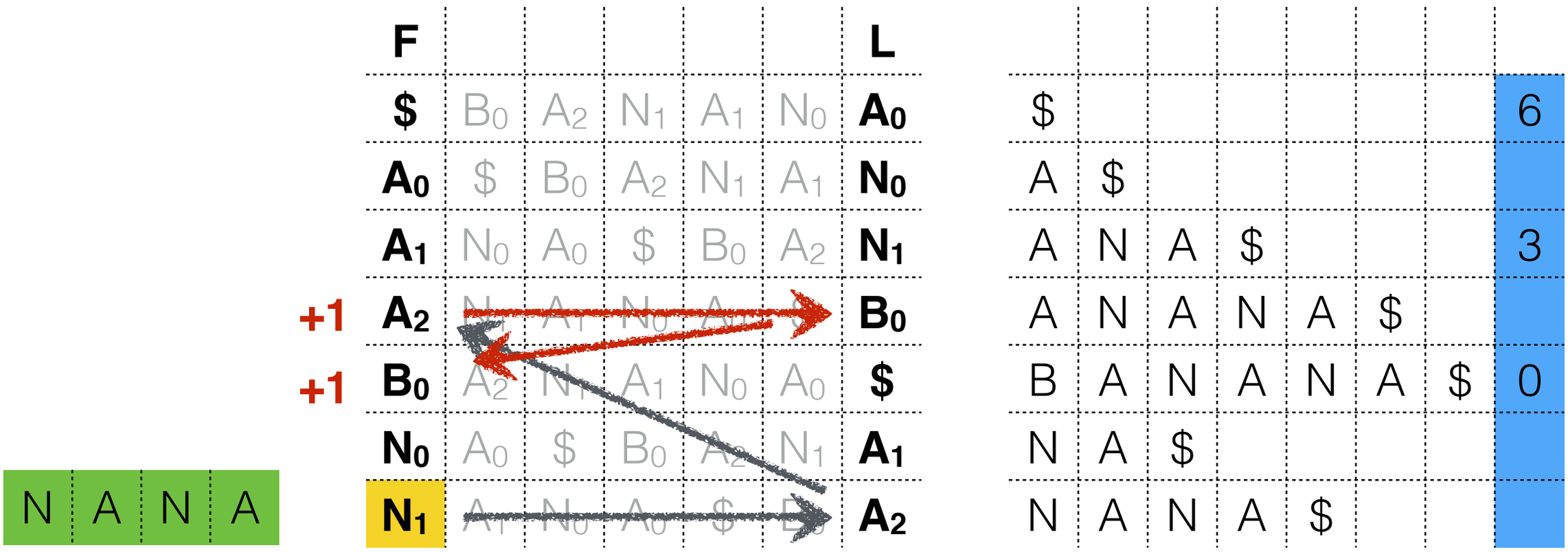
+1

$							6
A	$						
A	N	A	$				3
A	N	A	N	A	$		
B	A	N	A	N	A	$	0
N	A	$					
N	A	N	A	$			

N	A	N	A

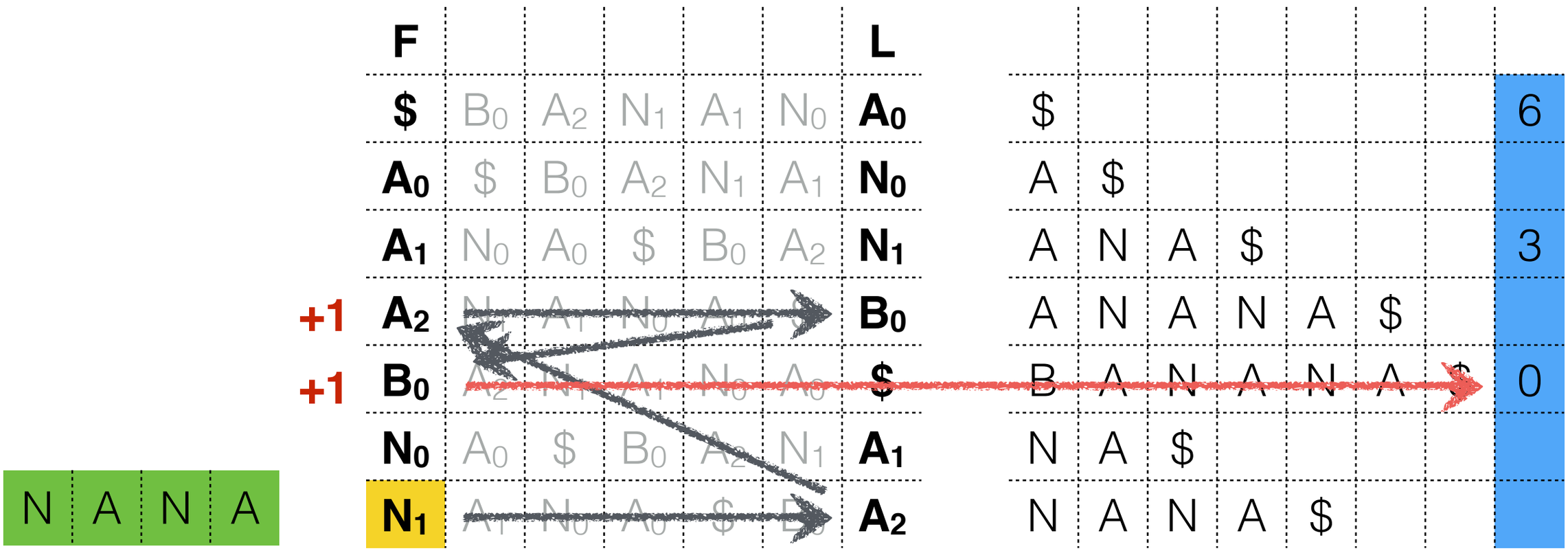
... and walk backwards through the BWT!

BWT search



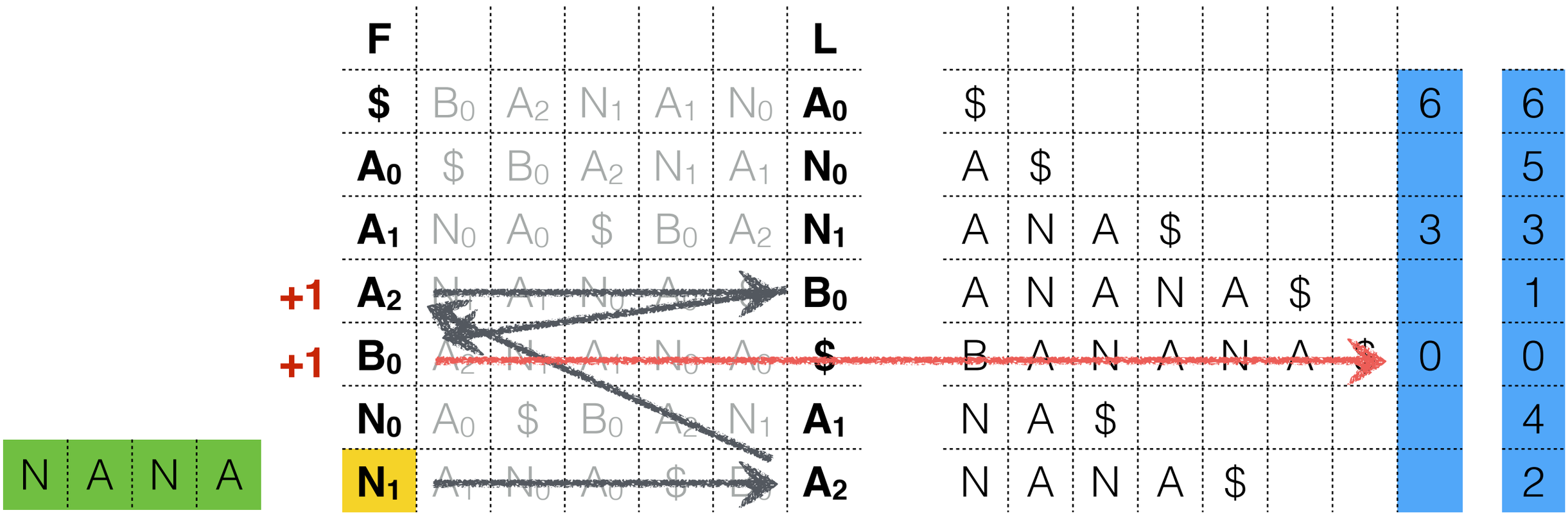
... and walk backwards through the BWT!

BWT search



... and walk backwards through the BWT!

BWT search



... and walk backwards through the BWT!

BWT search

- Finding location takes constant time if the offsets into T are **evenly spaced in T**, not in the SA!
- Make tradeoff between space (RAM) and time (how long lookups take)

Things we left out

- Rank calculations in the BWT need to be fast! Needs another auxiliary data structure
- Only covered exact matching, read alignment requires mismatches (e.g. SNP in read, not in genome)
- Other details: store forwards and backwards indices of genome due to sequencing error profile