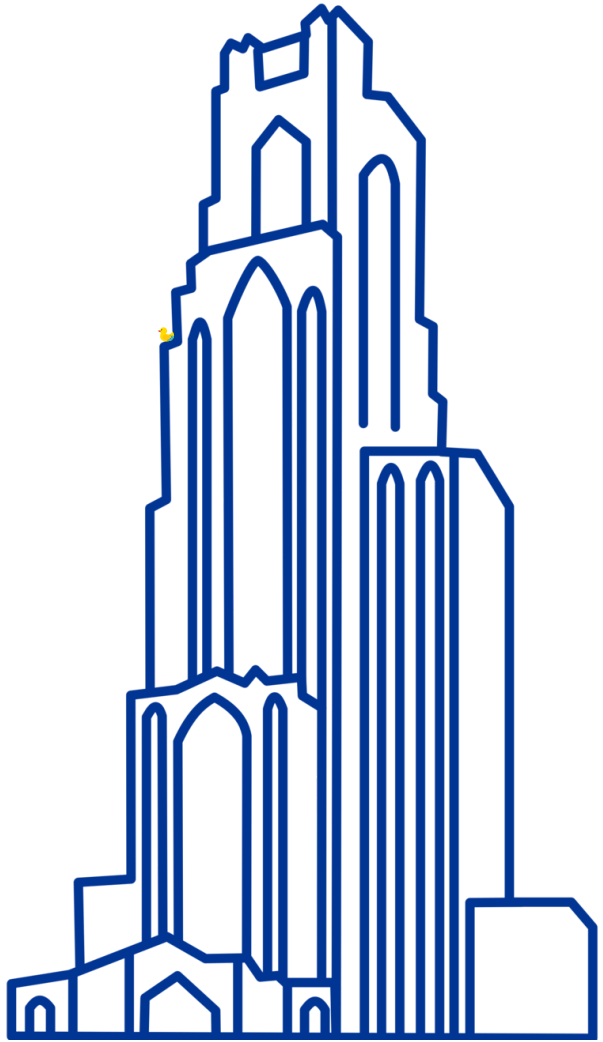


Computational Biology
(BIOSC 1540)

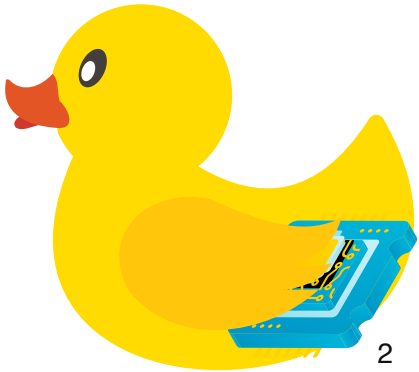
Lecture 08:
Read mapping

Sep 19, 2024

University of Pittsburgh

Announcements

- A03 is due tonight by 11:59 pm



After today, you should be able to

1. **Describe the challenges of aligning short reads to a large reference genome.**
2. Compare read alignment algorithms, including hash-based and suffix tree-based approaches.
3. Explain the basic principles of the Burrows-Wheeler Transform (BWT) for sequence alignment.

We are dealing with enormous datasets

Reference genome sizes

- *Homo sapiens*: 3,200,000,000 bp **(~3.2 GB if using u8)**
- *Mus musculus*: 2,700,000,000 bp
- *Drosophila melanogaster*: 140,000,000 bp
- *Saccharomyces cerevisiae*: 12,000,000 bp

RNA-seq data

- Illumina RNA-seq is around 120 GB

Most computers have 8 - 12 GB of RAM

Contextualization

The best movie ever is only 1.2 GB

The Trade-off: Fast vs. Precise

Performance considerations

- Balancing speed and accuracy
- Efficient alignment for downstream analyses
- Resource management (CPU, memory)

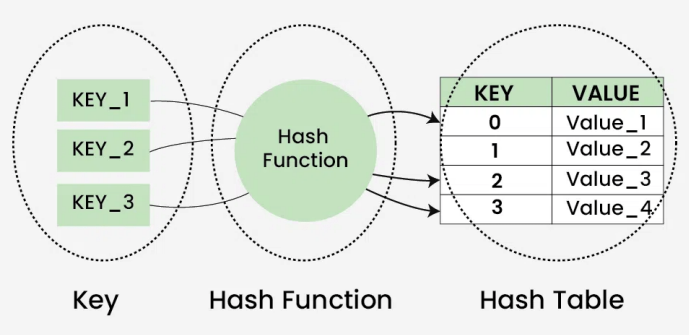
After today, you should be able to

1. Describe the challenges of aligning short reads to a large reference genome.
2. **Compare read alignment algorithms, including hash-based and suffix tree-based approaches.**
3. Explain the basic principles of the Burrows-Wheeler Transform (BWT) for sequence alignment.

A spectrum of alignment strategies

Hash tables

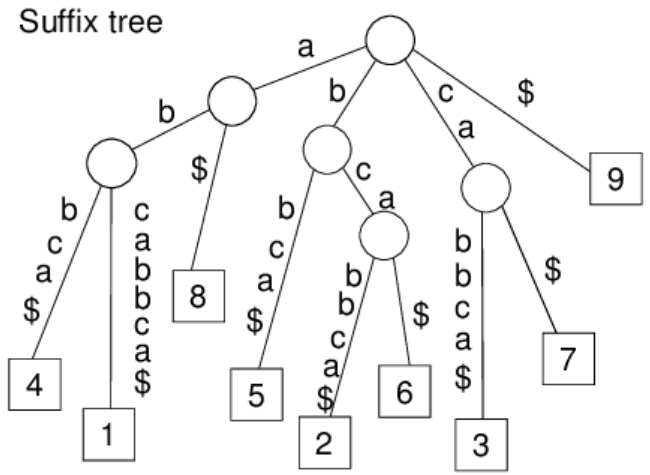
Mid 2000s



E.g., SOAP and MAQ

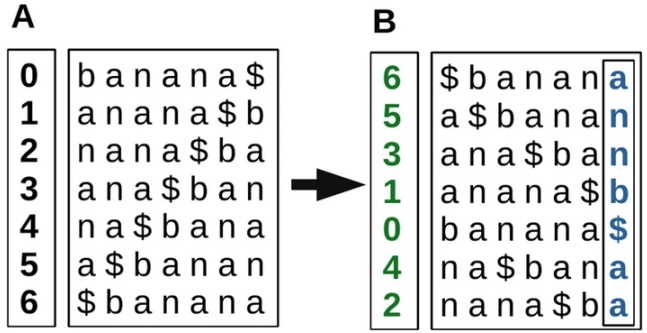
Suffix arrays/trees

Late 2000s



Burrows-Wheeler transforms

Late 2000s

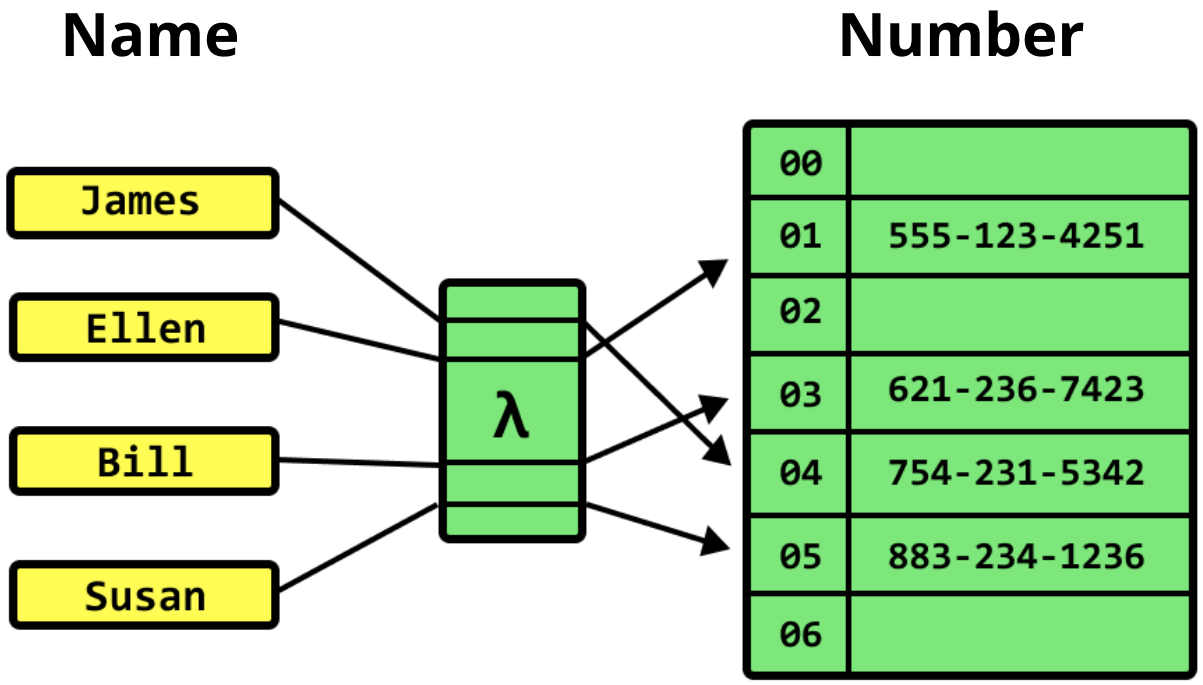


E.g., Bowtie2, BWA, STAR

Hash tables link a key to a value

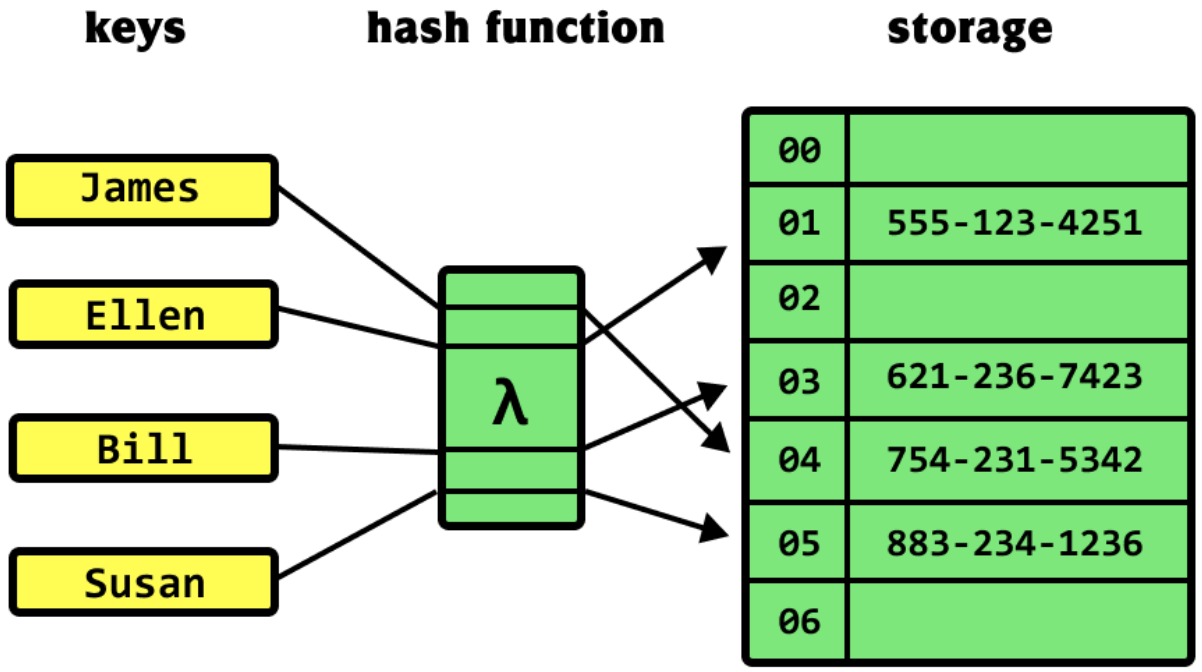
Keys represent a "label" we can use to get information

Example: Contacts



A "hash function" determines where to find their number

Hash functions convert labels to table indices



Example

$$h(k) = \text{len}(k)$$

Index: We take the key, count how many characters are in it

len("James") = 5

James ⟶ 883-234-1236

Note: This is a bad hashing function since "Alex" and "John" would result in the same index

Hashing our reference genome seeds our hash table with k-mer locations

Reference genome

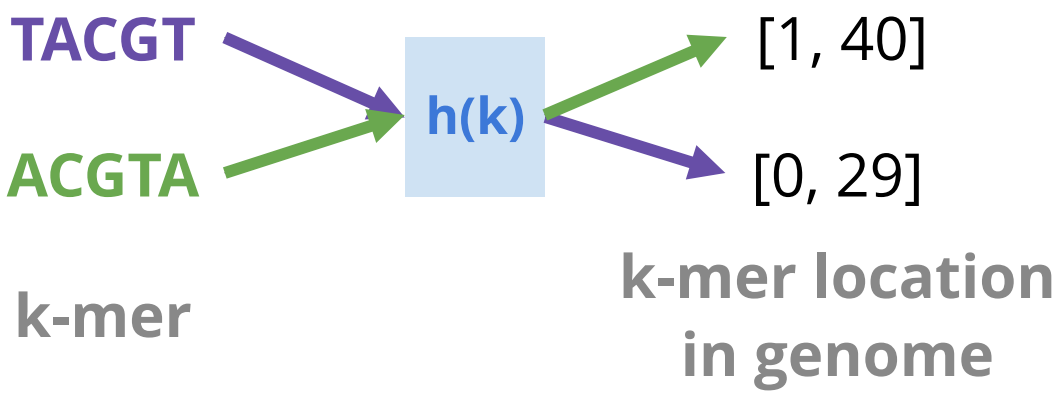
0 10 20 30 40 50 60

TACGTACGATAGTCGACTAGCATGCATGCTACGTGCTAGCACGTATGCATGCATGCATGCC

5-mers

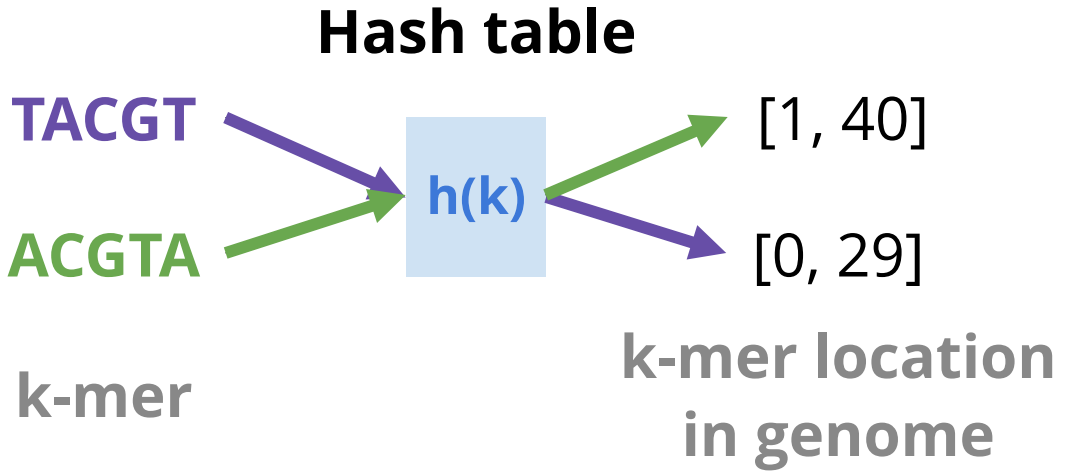
TACGT, ACGTA, CGTAC, GTACG, . . .

We hash our k-mer, and add the starting index where that k-mer occurs in our reference genome



Hashing our RNA-seq data provides quick lookups of our reference genome

Query a **k-mer read** to get indices that of possible reference genome locations



Reference genome

0 10 20 30 40 50 60

TACGTACGATAGTCGACTAGCATGCATGCTACGTGCTAGCACGTATGCATGCATGCATGCC

Seed-and-extend in hash-based alignment

Seed

Read: ATC**GAT**TGCA

k-mers (k=3)
ATC, TCG, CGA, **GAT**, ATT,
TTG, TGC, GCA

Use hash table for rapid lookup of potential matches quickly

GAT → h(k) → [7, 14]

Extend

Start from seed match and grow in both directions with reference genome

Check to see if we can align the read to reference

Hash-Based Alignment: Divide and Conquer

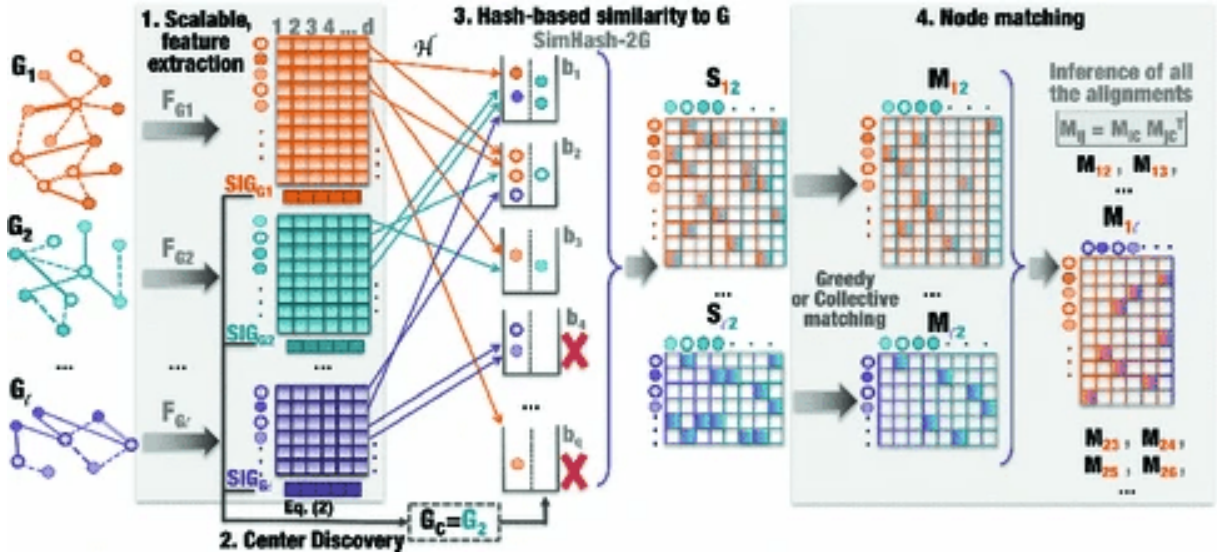
A "DNA dictionary" with quick lookup and direct access to potential matches

Pros

- Easily parallelizable
- Flexible for allowing mismatches
- Conceptually simple

Cons

- Large memory footprint for index
- Can be slower for very large genomes



Suffix trees represent all suffixes of a given string

A suffix tree is used to find starting index of suffix

Node Split point

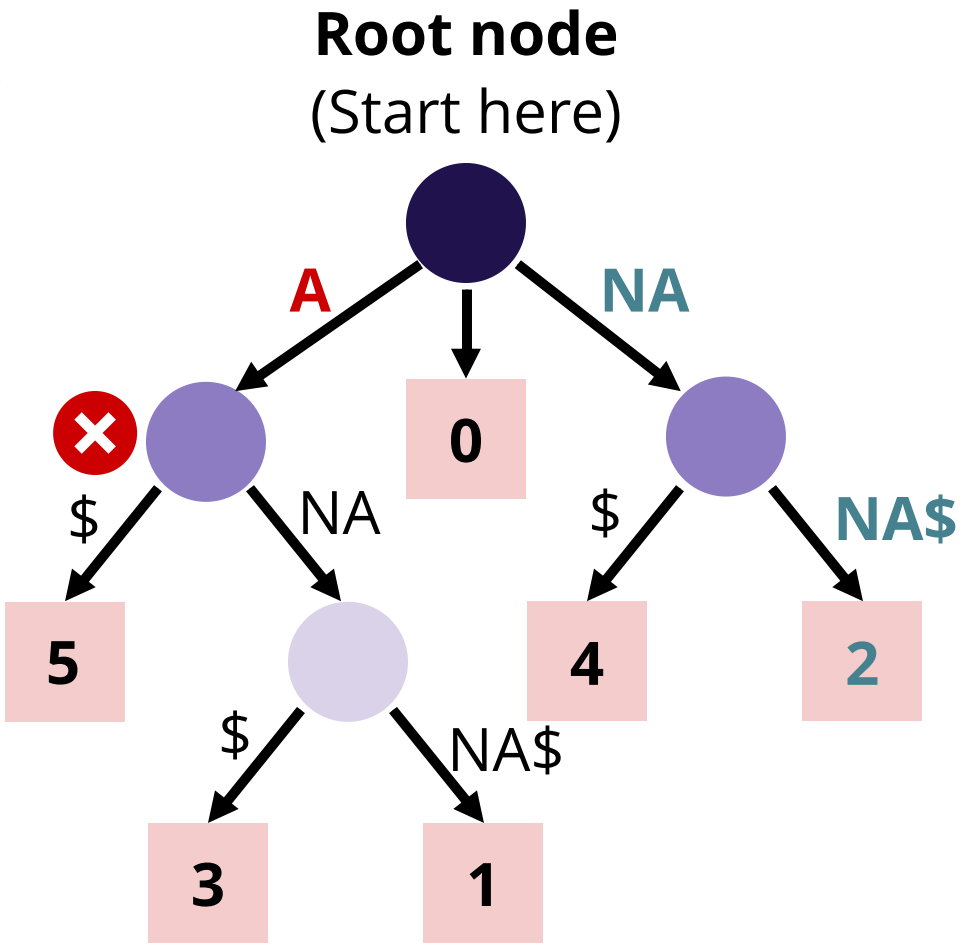
Leaf node Suffix start index

Edge Next part of suffix

Example: Where does **NANA$** start? Index 2.

Where does **AANA** start? Nowhere.

Note: We use $ to represent the end of a string



Suffix arrays are memory-efficient alternatives to trees

Requires less memory, but is also less powerful

BANANA$

1. Create all suffixes

BANANA$
ANANA$
NANA$
ANA$
NA$
A$
$

2. Sort lexicographically

3. Store starting indices in original string

6	**$**
5	**A$**
3	**ANA$**
1	**ANANA$**
0	**BANANA$**
2	**NA$**
4	**NANA$**

Symbols come before letters for sorting

After today, you should be able to

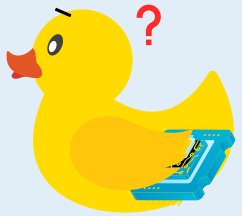
1. Describe the challenges of aligning short reads to a large reference genome.
2. Compare read alignment algorithms, including hash-based and suffix tree-based approaches.
3. **Explain the basic principles of the Burrows-Wheeler Transform (BWT) for sequence alignment.**

Compression reduces the amount of data we have to store

Suppose we need to store this string:

"Alex keeps talking and talking and talking and talking and eventually stops."

How could we store this string and save space?



Run-length encoding

"talking and talking and talking and talking and" = "talking and" 4

"Alex keeps talking and talking and talking and talking and eventually stops." → **"Alex keeps"** + **"talking and"** 4 +**"eventually stops."**

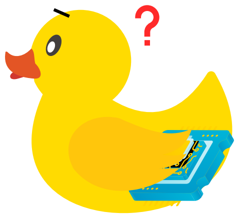
Not all strings have repeats

Can you find any repeats?

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec iaculis risus vulputate dui condimentum congue. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

How can we force repeats?

Sorting the letters does!



.aaaaaaaaaaaabbcccccccccddddddeee
eeeeeeeeeeeeeeeeeeeefggghiiiiiiiiiiiiiiiil
lllllllmmmmmmmmnnnnnnnnnnoooo
ooooppppppqqrrrrrrrsssssssssssssssssst
tttttttttttttttttttuuuuuuuuuuuuuuuv

→

Run-length encoding

a12b2c9d6e23f1g3h1i16l8m8
n10o8p5q2r7s17t19u15v1

Sorting lexicographically forces repeats, but loses original data

The **Burrows-Wheeler Transform (BWT)** is a way to sort our strings without losing the original data

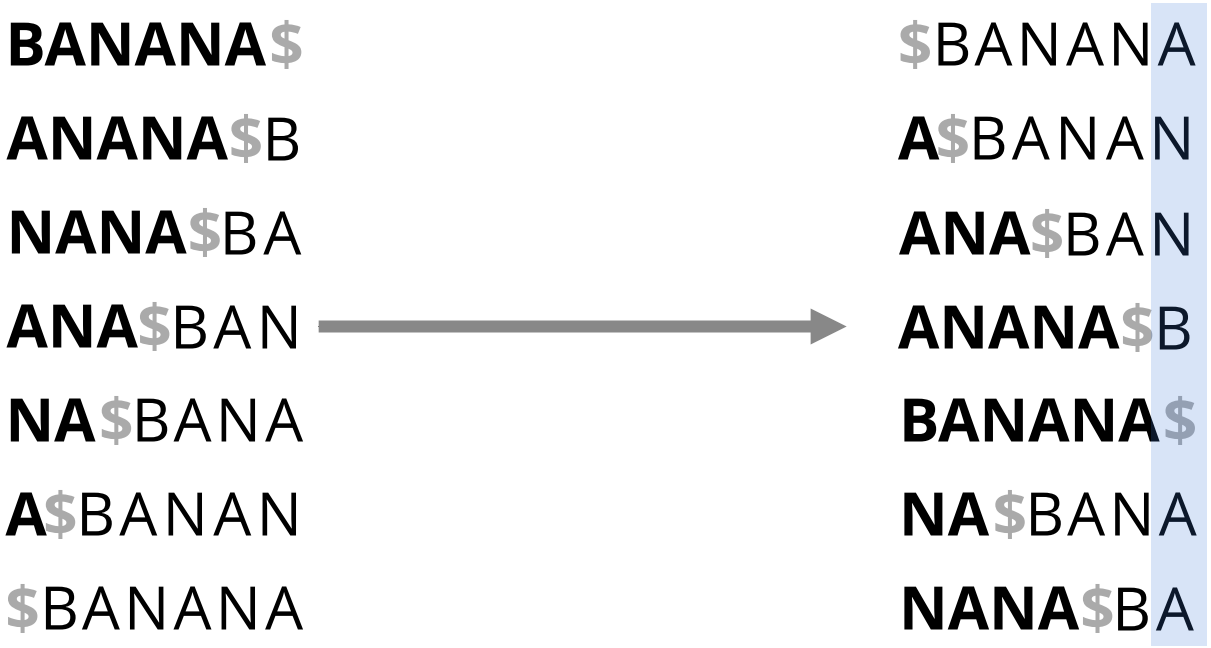
(And also search through it!)

Developed by Michael Burrows and David Wheeler in 1994

Basic concept of BWT

1. Append a unique end-of-string (EOS) marker to the input string.
2. Generate all rotations of the string.
3. Sort these rotations lexicographically.
4. Extract the last column of the sorted matrix as the BWT output.

BANANA



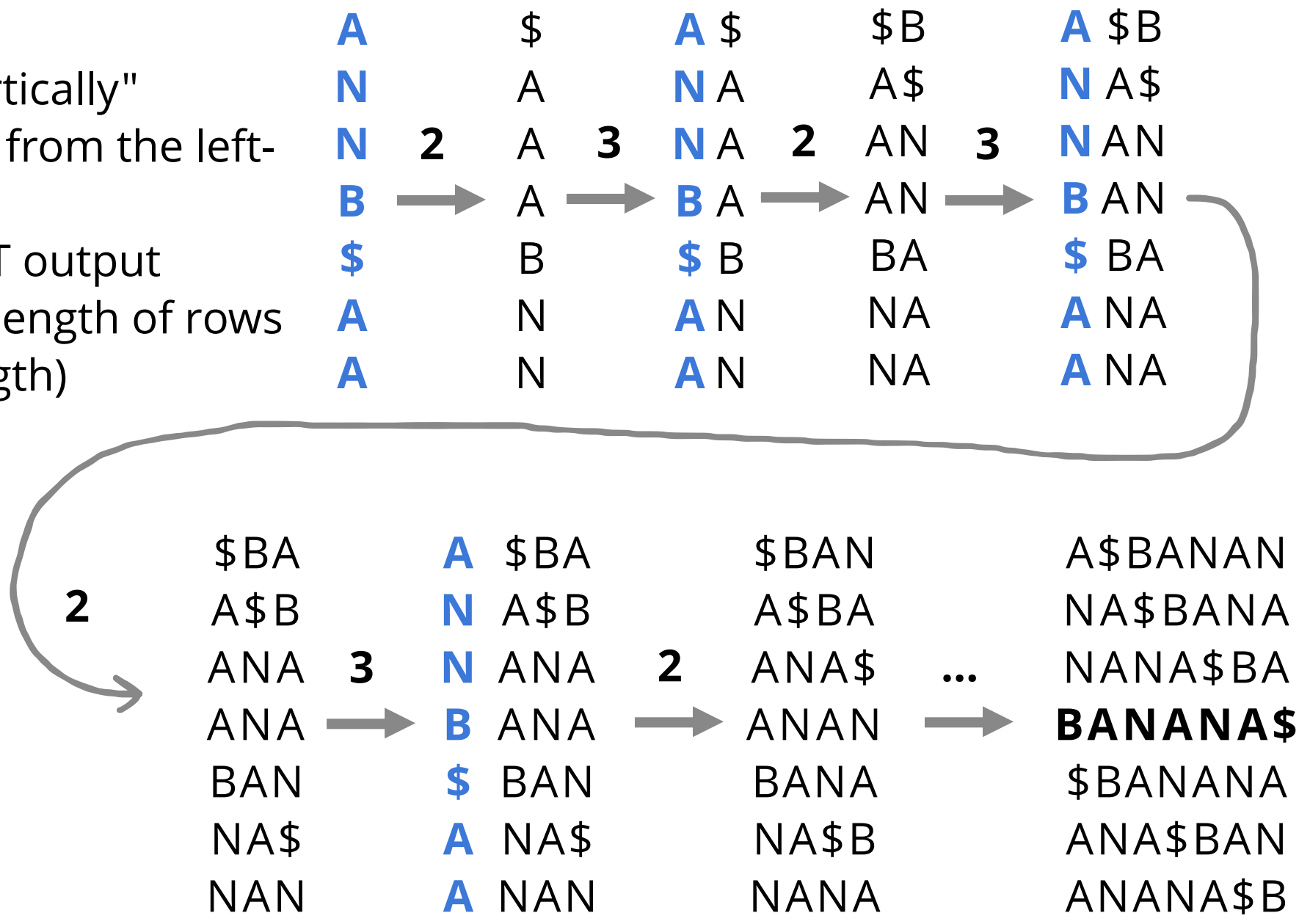
First column is more compressible, but we lose context and reversibility

ANNB$AA

(We can also get first column by sorting the output)

BWT output is reversible!

1. Write BWT output "vertically"
2. Sort each row starting from the left-most character
3. Append the same BWT output
4. Repeat until finished (length of rows equal BWT output length)



Enhancing BWT for Rapid Searching

The backward search algorithm efficiently finds occurrences of a pattern in a text using the LF-mapping

ABRACADABRA$ → **BWT matrix**

F-column		L-column
$	ABRACADABR	A
A	$ABRACADAB	R
A	BRA$ABRACA	D
A	BRACADABRA	$
A	CADABRA$AB	R
A	DABRA$ABRA	C
B	RA$ABRACAD	A
B	RACADABRA$	A
C	ADABRA$ABR	A
D	ABRA$ABRAC	A
R	A$ABRACADA	B
R	ACADABRA$A	B

Number

$	ABRACADABR	A_0
A_0	$ABRACADAB	R_0
A_1	BRA$ABRACA	D_0
A_2	BRACADABRA	$
A_3	CADABRA$AB	R_1
A_4	DABRA$ABRA	C_0
B_0	RA$ABRACAD	A_1
B_1	RACADABRA$	A_2
C_0	ADABRA$ABR	A_3
D_0	ABRA$ABRAC	A_4
R_0	A$ABRACADA	B_0
R_1	ACADABRA$A	B_1

Number characters with the number of times they have appeard

Suppose I want to find where **ABRA** is located

ABRACADABRA$

1. Find rows with last character in search string (e.g., A) in F-column
2. Note which rows has the next letter (e.g., R) in L-column
3. Work backwards in search string until the first letter

A		R
$	ABRACADABR	A_0
A_0	$ABRACADAB	**R_0**
A_1	BRA$ABRACA	D_0
A_2	BRACADABRA	$
A_3	CADABRA$AB	**R_1**
A_4	DABRA$ABRA	C_0
B_0	RA$ABRACAD	A_1
B_1	RACADABRA$	A_2
C_0	ADABRA$ABR	A_3
D_0	ABRA$ABRAC	A_4
R_0	A$ABRACADA	B_0
R_1	ACADABRA$A	B_1

R		B
$	ABRACADABR	A_0
A_0	$ABRACADAB	R_0
A_1	BRA$ABRACA	D_0
A_2	BRACADABRA	$
A_3	CADABRA$AB	R_1
A_4	DABRA$ABRA	C_0
B_0	RA$ABRACAD	A_1
B_1	RACADABRA$	A_2
C_0	ADABRA$ABR	A_3
D_0	ABRA$ABRAC	A_4
R_0	A$ABRACADA	**B_0**
R_1	ACADABRA$A	**B_1**

B		A
$	ABRACADABR	A_0
A_0	$ABRACADAB	R_0
A_1	**BRA$ABRACA**	**D_0**
A_2	**BRACADABRA**	**$**
A_3	CADABRA$AB	R_1
A_4	DABRA$ABRA	C_0
B_0	RA$ABRACAD	**A_1**
B_1	RACADABRA$	**A_2**
C_0	ADABRA$ABR	A_3
D_0	ABRA$ABRAC	A_4
R_0	A$ABRACADA	B_0
R_1	ACADABRA$A	B_1

Backward search enables efficient searching using only first and last columns of BWT

BWT practice

Given the string "mississippi$", complete the following tasks:

- Construct the Burrows-Wheeler Transform (BWT) of the string.
- Use the LF-mapping to find the number and positions of occurrences of the following patterns in the original string:
  - a) "si"
  - b) "iss"
  - c) "pp"

Before the next class, you should

Lecture 08:
Read mapping

Lecture 09:
Quantification

Today

Tuesday

- Submit A03
- Start A04