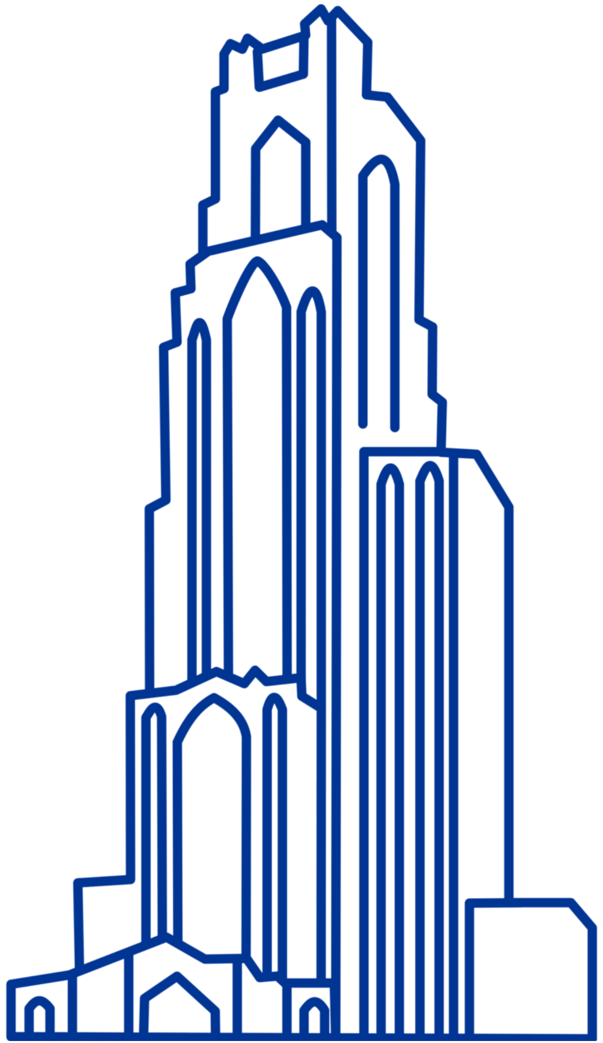
Computational Biology (BIOSC 1540)



Lecture 06B

Read mapping

Methodology

Feb 13, 2025

Announcements

Assignments

- Assignment P01D is due Friday (Feb 14)

Quizzes

- Quiz 02 is on Feb 18 and will cover lectures 04A to 06A

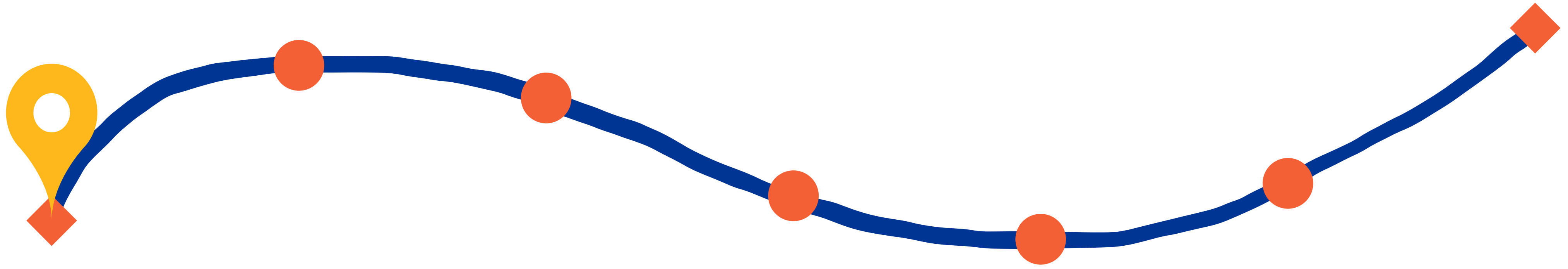
CBytes

- CByte 03 **expires** on Feb 15
- CByte 04 **expires** on Feb 28

Next reward: Checkpoint Submission Feedback

ATP until the next reward: 653

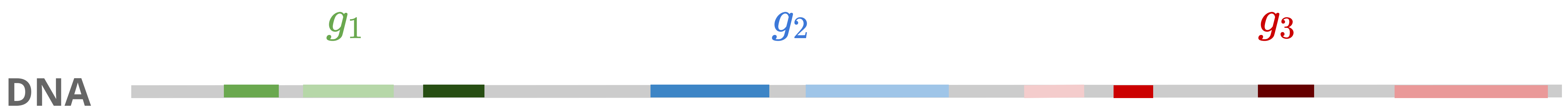
After today, you should have a better understanding of



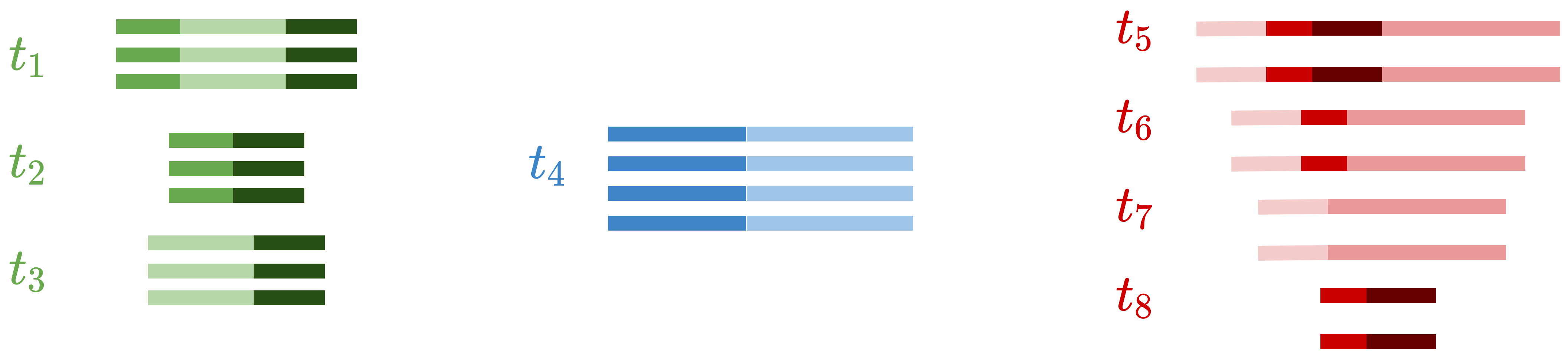
The purpose of reference-based mapping

Understanding how we get our reads

Suppose we have the following three **coding** (i.e., genes) and **non-coding** regions with introns and exons

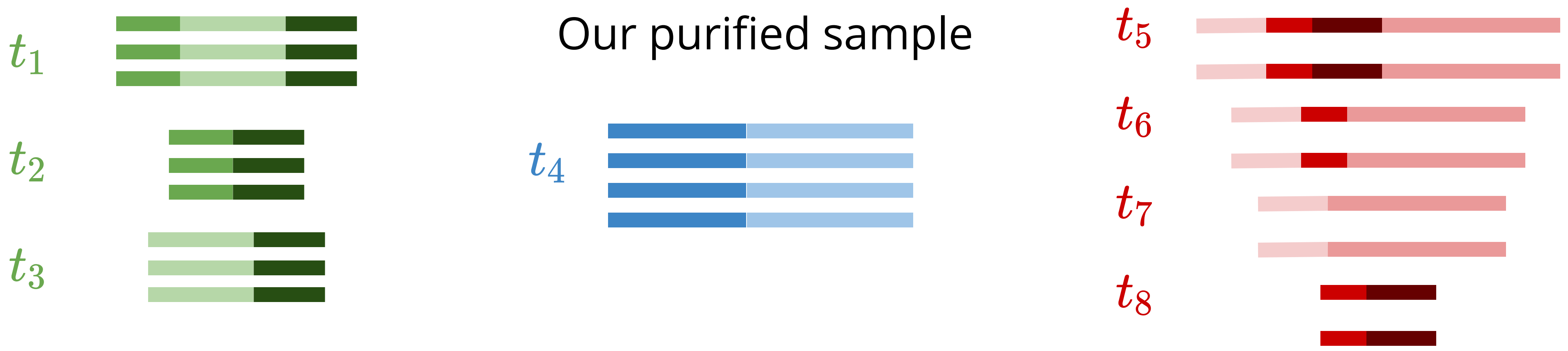


Cells will transcribe these genes into mRNA (i.e., transcripts)



We collect, convert to complementary DNA, and then amplify

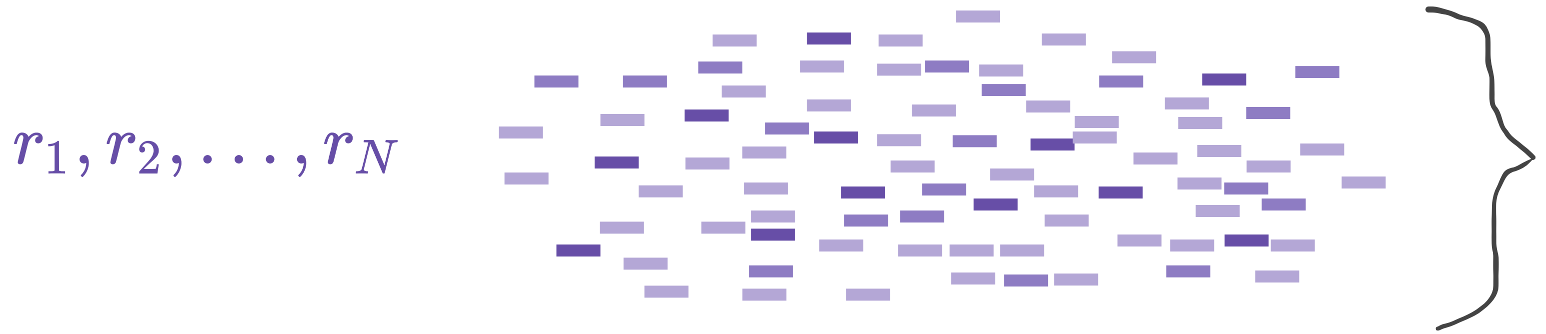
Understanding how we get our reads



Our purified sample

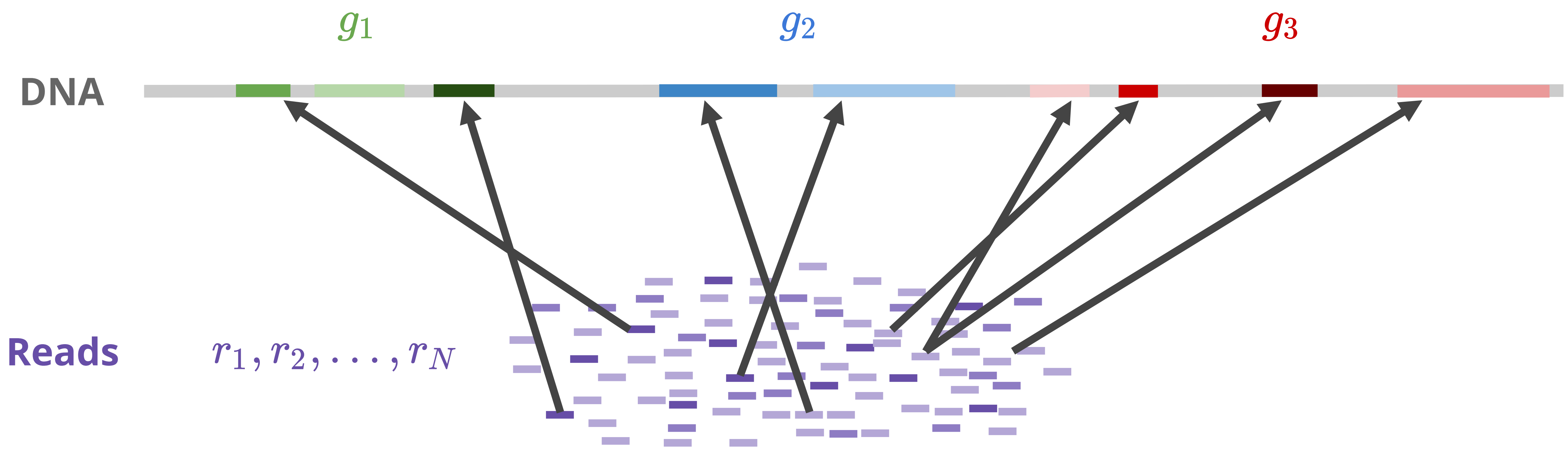
Fragmentation and sequencing

Each RNA-seq read represents a **small fragment** of a transcript



FASTQ reads

The Goal of Read Alignment is to Reconstruct Gene Expression Patterns



Read mapping determines where in the genome did these reads originate from.

By mapping reads to a **reference genome or transcriptome**, we can:

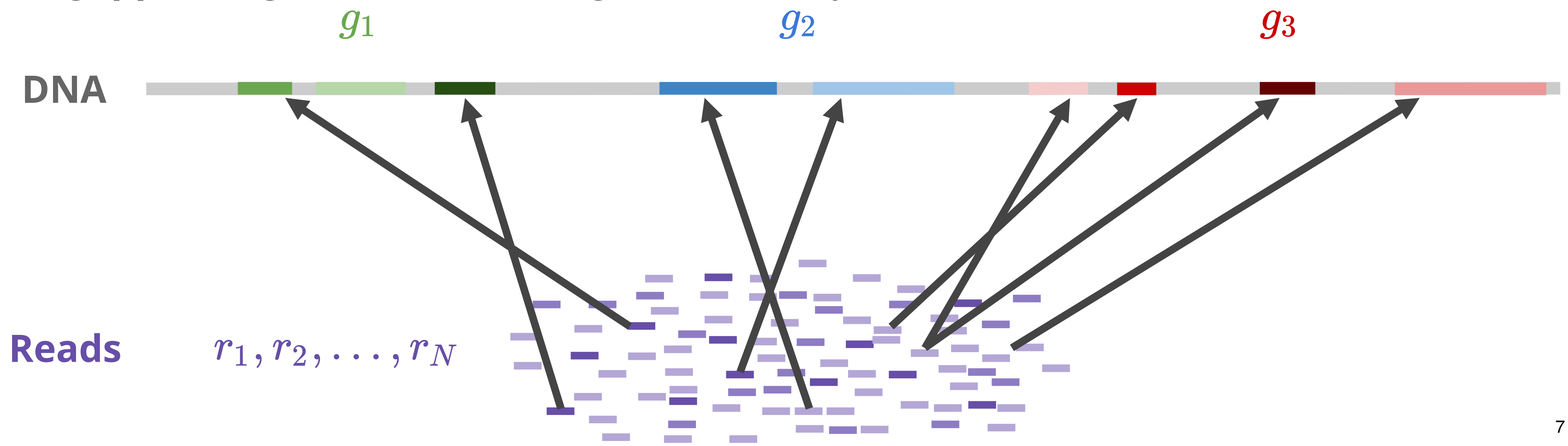
- Identify **which genes are active** in a sample.
- Measure the **relative abundance** of different transcripts.
- Detect **novel isoforms and alternative splicing events**.

RNA-seq must account for alternative splicing

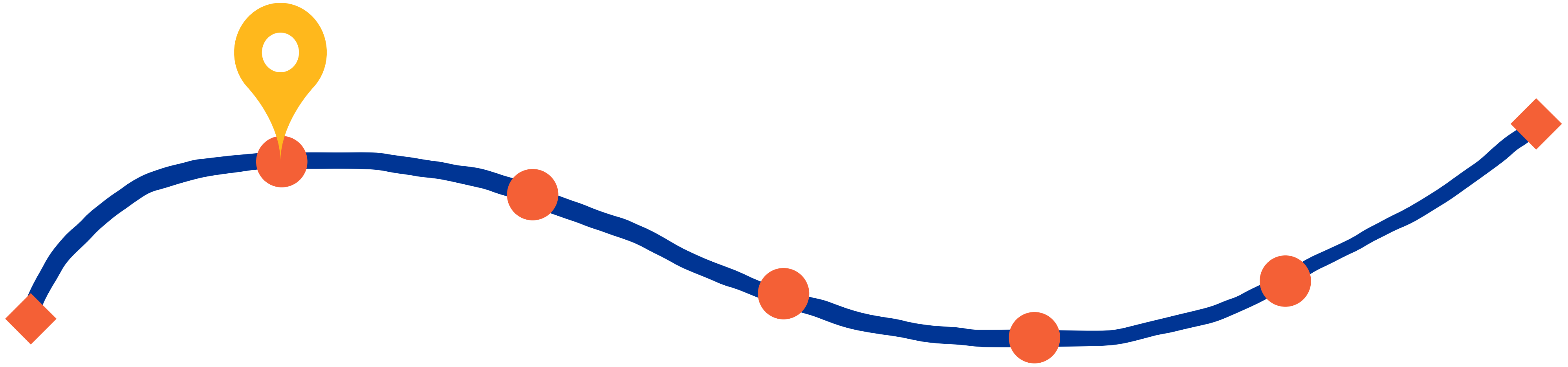
Unlike DNA sequencing, **RNA sequencing includes spliced transcripts.**

Key problem: Reads from mRNA span exon-exon junctions, but the genome contains introns.

Solution: Transcriptomic aligners must allow for **gapped alignments** that bridge exon-exon junctions.



After today, you should have a better understanding of



Hash-based methods for handling introns

Activity

Read mapping exercise

Let's consider this short story as our genome containing coding and non-coding regions

Danny loved spotting shapes in the clouds and had an entire journal filled with sketches of dragons, castles, and sailing ships. One day, he noticed a small cloud following him, shifting to match whatever he imagined. He tested it by thinking of a giant ice cream cone; sure enough, it transformed before his eyes. Delighted, he ran home, wondering how much fun he could have with a personal cloud. His only concern was making sure it didn't rain inside his room.

The reads below were built by taking random words, slicing three letters, and then concatenating without spaces (all lowercase)

0. danclodrashi	"Danny clouds dragons ships"

In groups, please work together to determine which words were used for your read

1. entmagcretra	?
2. spomatwoncon	?
3. notgiasaibed	?
4. lovdrathirai	?

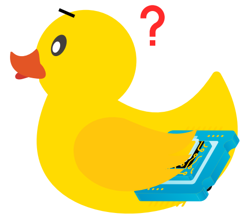
Be prepared to explain how you approached the problem

We could do maybe one or two in a few minutes

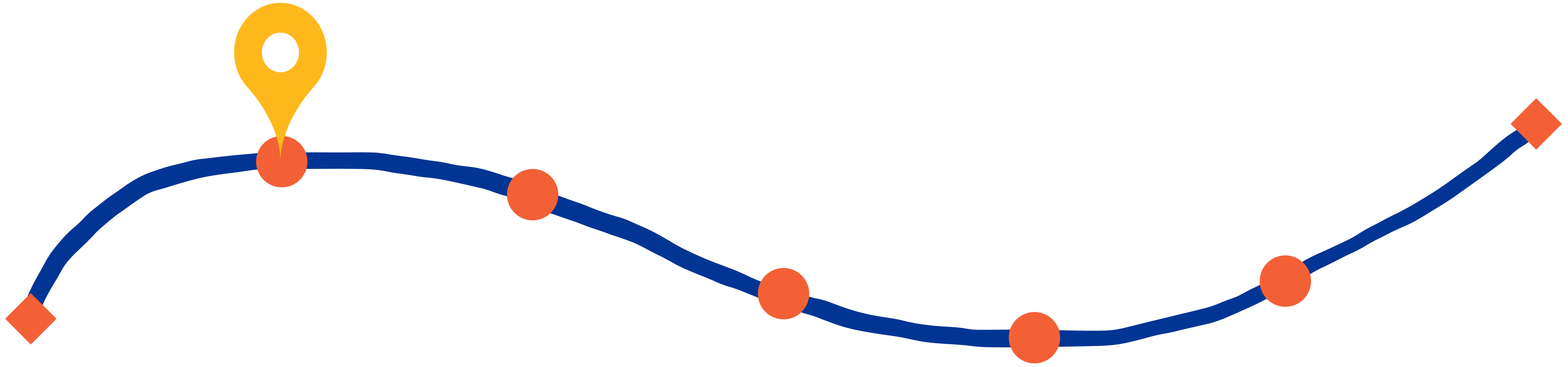
What about doing 30?

danclodrashi	consurenotra	rigbefeyedel
lovshaentjou	cloonlconmak	ranhomwonmuc
spofilskecas	surdidraiins	funcouhavwit
saionenotsma	danlovsposke	percloonlcon
clofolshimat	shajoufildra	maksurdidrai
whaimatestthi	cassaishiday	insbeddanlov
giaicecrecon	notclofolshi	sposhacloent
surenotratrig	matwhaimates	joufilskedra
befeyedelran	thigiaicecre	casnotsmaclo
homwonmucfun	couhavwitper	folshimathwa

We need to map millions of reads to our genome, so how could we approach this computationally?



After today, you should have a better understanding of



Hash-based methods for handling introns

K-mer indexing

We can pre-compute k-mer locations of our story

Danny loved spotting shapes in the clouds and had an entire journal filled with sketches of dragons, castles, and sailing ships [...]

We can chunk our story into k-mers and store where in the story they occur

danny

```
"dan": [0]
"ann": [1]
"nny": [2]
```

clouds, cloud, cloud

```
"clo": [35, 153, 382]
"lou": [36, 154, 383]
"oud": [37, 155, 384]
```

spotting, sailing, following, shifting, thinking, wondering, making

```
"ing": [17, 116, 165, 178, 233, 335, 412]
```

We can store these *k*-mers and indices and then use these to find potential sources

Mapping a read to our genome involves checking where *k*-mers exist

Genome

Danny loved spotting shapes in the clouds and had an entire journal filled with sketches of dragons, castles, and sailing ships [...]

Read → **Query *k*-mers**

danclodrashi → dan, clo, dra, shi

`"dan": [0]`	danny
`"clo": [35, 153, 382]`	clouds, cloud, cloud
`"dra": [92]`	dragon
`"shi": [120, 173]`	ships, shifting

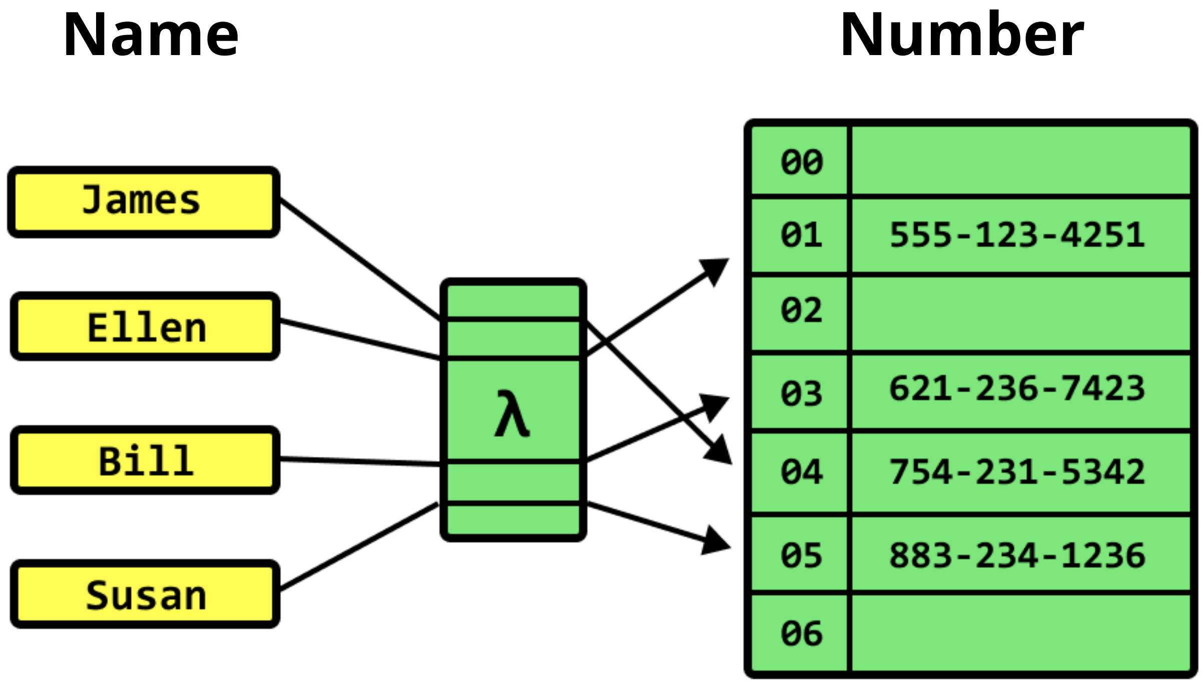
Check if *k*-mers are in our genome and the starting index of that *k*-mer

This data structure is called a **hash table** (i.e., dictionary in Python)

Hash tables link a key to a value

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

Example: Phone book / Contacts list



A "hash function" determines where to find their number in our computer's memory

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

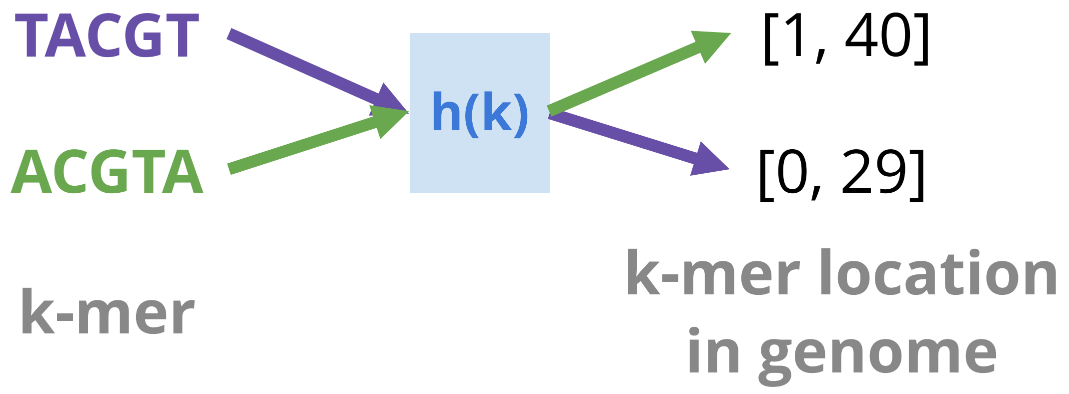
Reference genome

```
0         10        20        30        40        50        60
TACGTACGATAGTCGACTAGCATGCATGCTACGTGCTAGCACGTATGCATGCATGCATGCC
```

5-mers

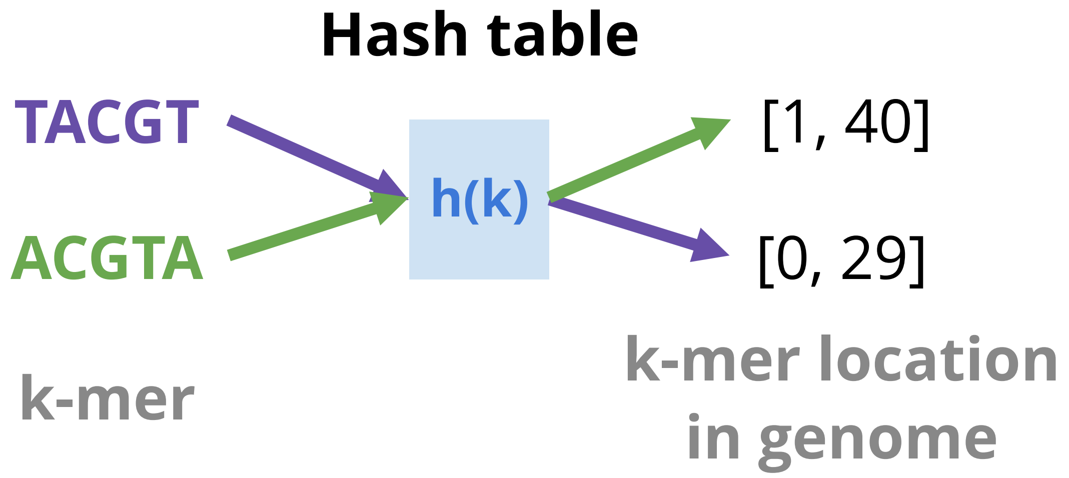
TACGT, ACGTA, CGTAC, GTACG, . . .

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

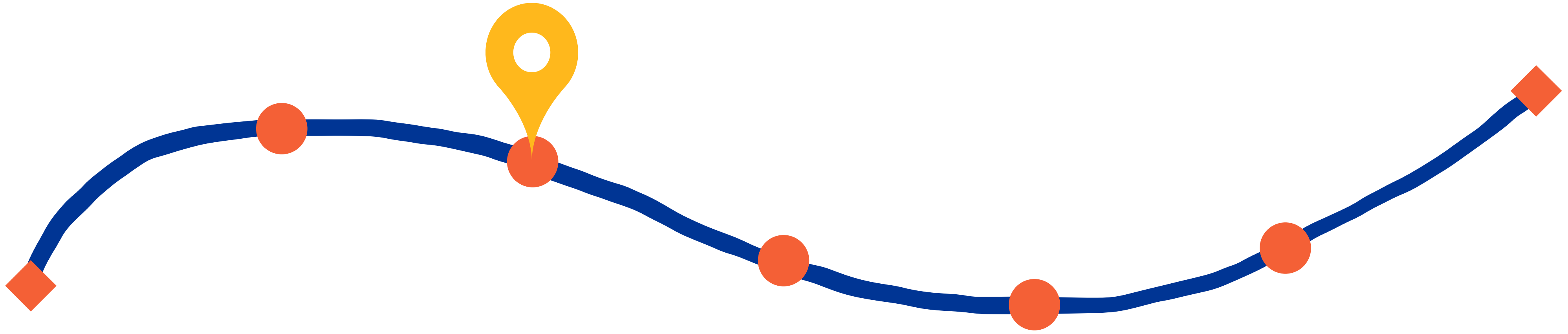
Query a **k-mer read** to get indices that of possible reference genome locations



Reference genome

```
0         10        20        30        40        50
TACGTACGATAGTCGACTAGCATGCATGCTACGTGCTAGCACGTATGCATGCATGCA
```

After today, you should have a better understanding of



Suffix arrays for efficient substring searches

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

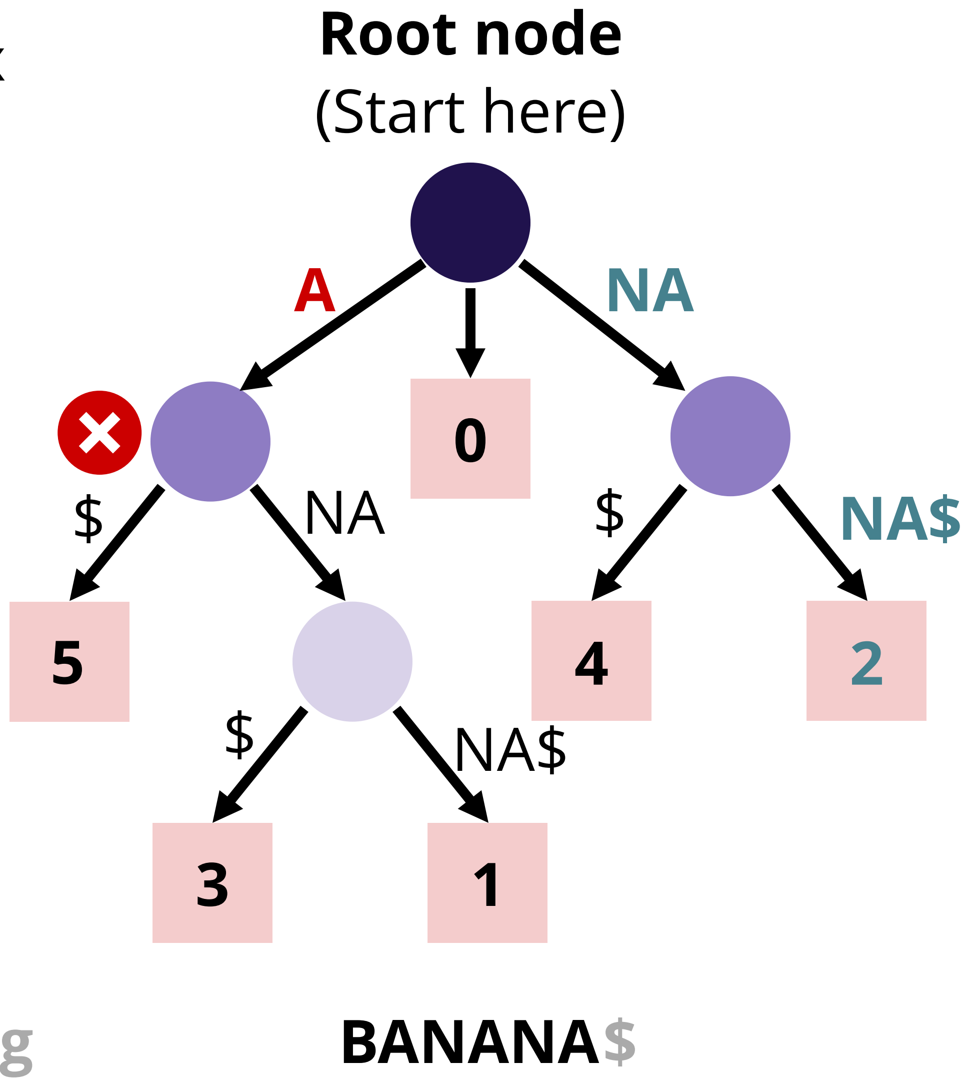
- Large memory footprint for index
- Can be slower for very large genomes

```
"dan": [0]

"clo": [35, 153, 382]

"dra": [92]

"shi": [120, 173]
```

Suffix trees compress all k-mers into a single data structure

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

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

$ comes before letters for sorting

After today, you should have a better understanding of

Burrows-Wheeler Transform (BWT) string compression

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

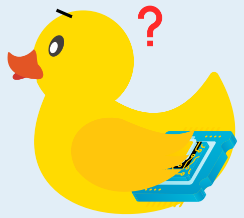
Contextualization

The best movie ever is only 1.2 GB

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." $\longrightarrow$ **"Alex keeps"** + **"talking and"** 4 +**"eventually stops."**

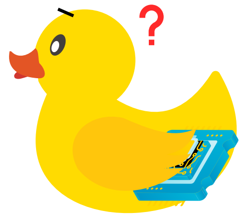
Not all strings have repeats

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!



.aaaaaaaaaaaabbcccccccccdddddeee
eeeeeeeeeeeeeeeeeeeefggghiiiiiiiiiiiiiiiil
lllllllmmmmmmmmnnnnnnnnnnoooo
oooopppppqqrrrrrrrssssssssssssssssst
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

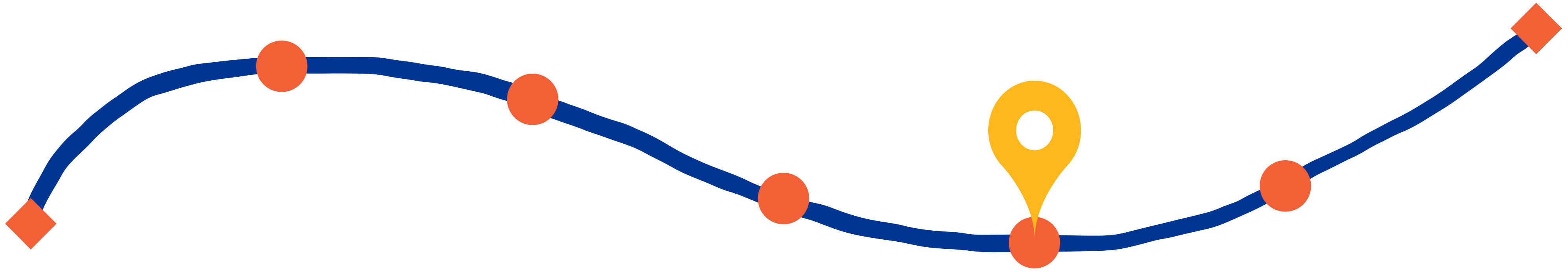
BANANA$		$BANANA
ANANA$B		**A**$BANAN
NANA$BA		**ANA**$BAN
ANA$BAN	→	**ANANA**$B
NA$BANA		**BANANA**$
A$BANAN		**NA**$BANA
$BANANA		**NANA**$BA

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

ANNB$AA

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

After today, you should have a better understanding of



FM-index for efficient substring searches

Enhancing BWT for Rapid Searching

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

ABRACADABRA$ $\xrightarrow{\textbf{BWT matrix}}$

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

$\xrightarrow{\textbf{Number}}$

$	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

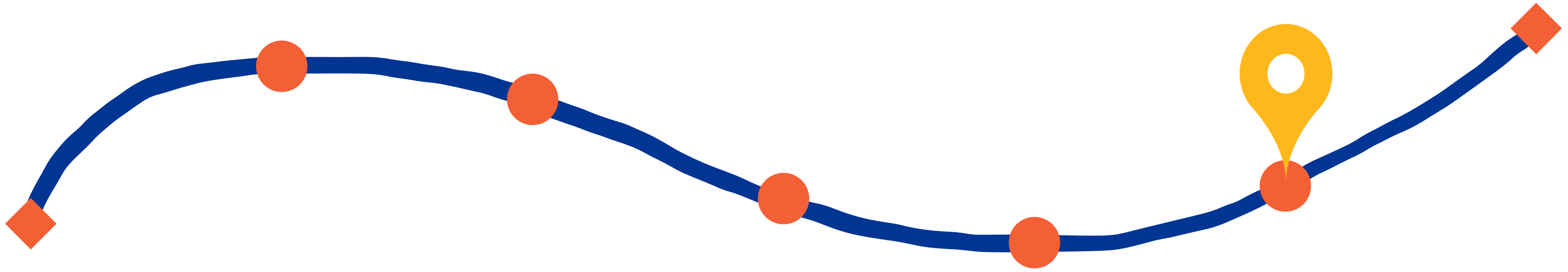
Backward search enables efficient searching using only first and last columns of BWT

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"

After today, you should have a better understanding of



Splice-aware mapping with seed-chain-extend strategy

Seed-and-extend in hash-based alignment

Seed

Read: ATC**GAT**TGCA

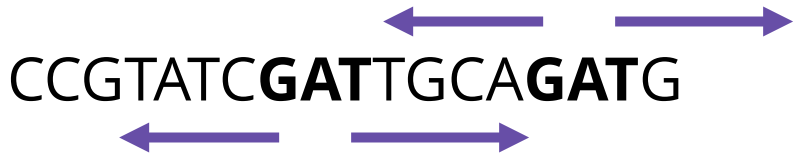
k-mers (k=3)
ATC, TCG, CGA, **GAT**, ATT,
TTG, TGC, GCA

Use hash table for rapid lookup of potential matches quickly

$$\begin{tikzcd}\textbf{GAT} \arrow[r] & \boxed{h(k)} \arrow[r] & {[7, 14]}\end{tikzcd}$$

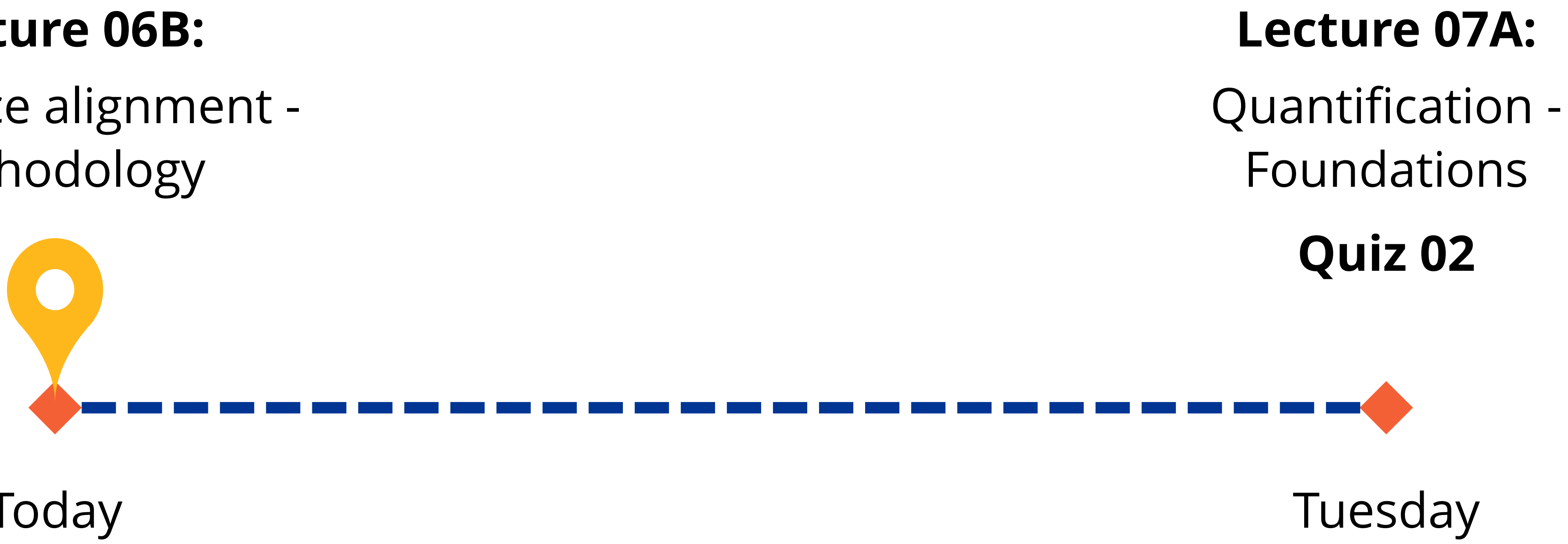
Extend

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



Check to see if we can align the read to reference

Before the next class, you should


Lecture 06B:
Sequence alignment - Methodology
Today
Lecture 07A:
Quantification - Foundations
Quiz 02
Tuesday

- Work on P01D (due Feb 14)
- Study for Quiz 02 (on Feb 18)