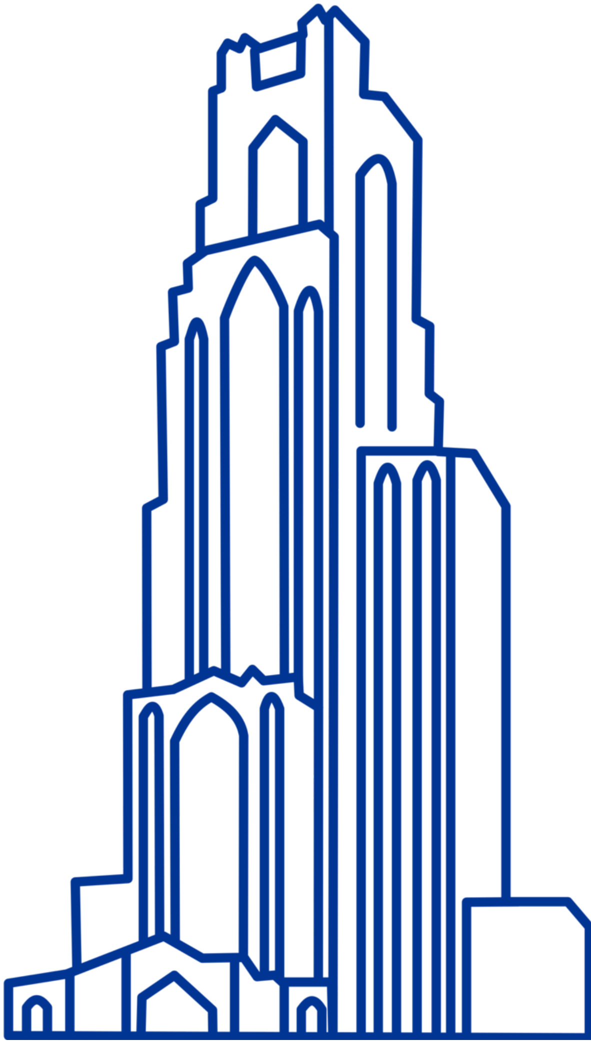


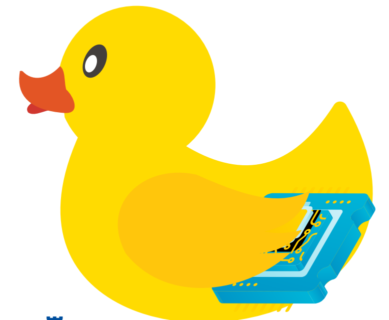
Computational Biology
(BIOSC 1540)

Lecture 03B

Genome assembly

Methodology

Jan 23, 2025



University of Pittsburgh

Announcements

Assignments

- Assignment P01B is due Friday (Jan 24)
- Assignment P01C is due next Friday (Jan 31)

Quizzes

- Quiz 01 is next week (Jan 28) and will cover lectures 02A to 03B

CBytes

- CByte 01 is live and will expire on Feb 1
- CByte 02 will be released Friday (Jan 24) and expire on Feb 7

Next reward: Checkpoint Submission Feedback

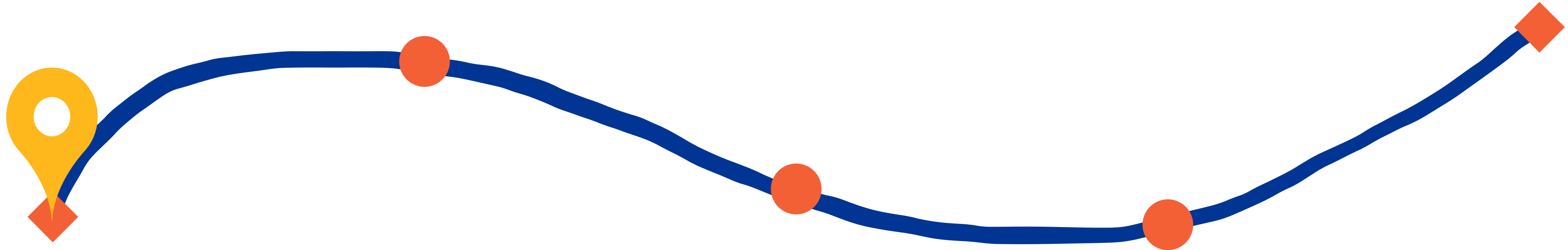
ATP until the next reward: 1,903

Quick homework tip

When asking for **five FASTQ entries**, here is what it should look like

```
1  fastq_five = """
2  @synthetic_read_1/f
3  TACGGCTAGGCATCTCGAGATCTGTGACGTTTCAGATCCCCTGCTGCGTGCGTTTGATGTCCAACTGTCGTACTCACGCCGGACGGGGAGTAACTTCTTTTCGAGCCGTAGTT
4  +
5  46:47287653825380557902185865586;11784536:8>:7946436;67:04>8671293:53991474581727927476120866:4;;441889567264523
6  @synthetic_read_2/f
7  GACGATCGTAGCTCAGTCGGACCAACGACTCGCTGCTTACTGGAAGATCCTCGTAGACGGTTTTTTTGCGAAAGTACAGGCGACCCAGTACAAATCGGATAGTGGTCACTTA
8  +
9  GGDIHIFGEHGGIGGIHGFGIIFIHFDEFEFCCFFIIHIGIEEFIEFFICDGFHICFEICGGFFEIEEFGIFGFIHIIBDGHIGHIIGGGHFGIEHIIIDIIECAIHDHCEDE
10 @synthetic_read_3/f
11 CGTAGCTGACGTAGATTCGATTTAAGAAACGCAGATATGGACATTGTCGCCGTGCCTTTATATTCCACATATCGTGGTAATCATACCGGCATAGGGTCATGTCCGCAGCTGTC
12 +
13 :=9<<:7<9::=<?<<6;;=;?;<7;9=9?6:8;8A9:=>=<:A79;=>=;:==:<4::7<9?E4<9;;:97=<7@9;8?@<7999:A9:=;6:?>:@988A?97=A>=@:;9
14 @synthetic_read_4/f
15 TACGGCTAGGCACGTTTTCAGCAATCACGCGTGAGAATGCAATACAGCTGAGTATAGGTGGCCGGGCGTACGTTTCTACGTGAGCATGTTTTTTTATTACAGAGTACCGGTAC
16 +
17 >:A@=@=<ABB><=:==?>@=><<<9=?3:>@CHD;?=7:@?6G<8<@?AEE<=?;<;C<66B3>>>>=8488<8>?@9>43>?A?A61:@8;:6@97;825=>7>8><1<85
18 @synthetic_read_5/f
19 GTACGATCGTACCTGCGTACAAAACAGTTTCGGGGTCCAAACCACGCCTCAACTGTTCTCGGTTAGTACCGTAGCTACACTCGGTCTATCTGTCAGCTGCCGTTCATTCGAGC
20 +
21 78<8675<68;9;9<72;4==:689<;95=5;?76:57<16;:4@;9.=:1:;?<49;89;0<>?6327778:8:518?7=79:6:<7><A@16:65<98:6<7446<;@9=9
22 """
```

After today, you should have a better understanding of



Problem formulation of genome assembly

Why we need genome assembly

Genome assembly **reconstructs a long DNA sequence** from **short, error-prone reads**, ensuring as many reads fit into the final sequence

Overlapping reads

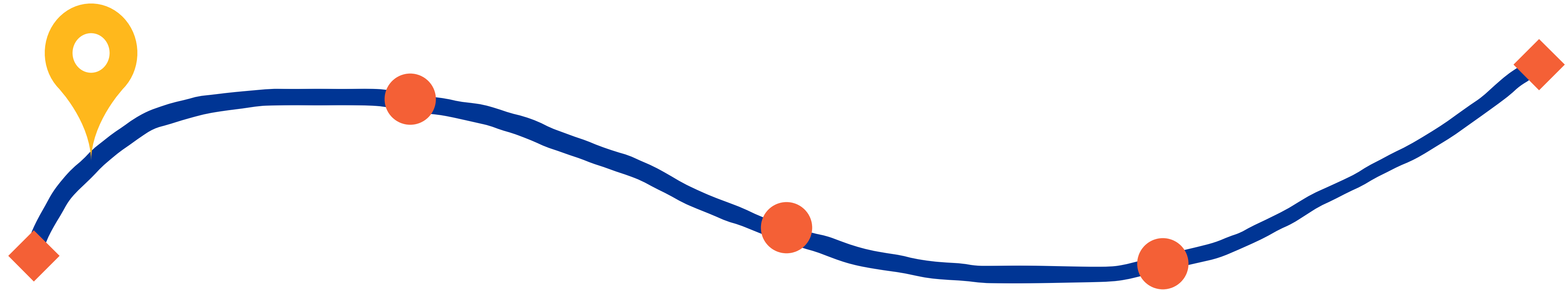
Assembly algorithms

TACGATCGGATTACGCGTAGGCTAGCTTACGGACTCGATGTACGATCGGATTACG

DNA sequence (i.e., contig)

Recap from L03A

After today, you should have a better understanding of



Problem formulation of genome assembly

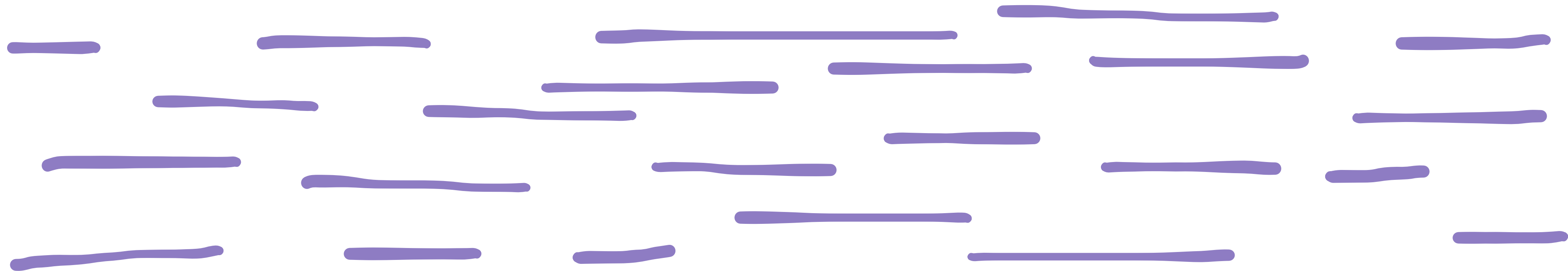
Assumptions

We make simplifying assumptions to address challenges and make assembly tractable

Reads originate from a single, contiguous genome

If we had **two sources of DNA**

Chance of overlap is likely, and it would be **challenging to differentiate the origin of each read**

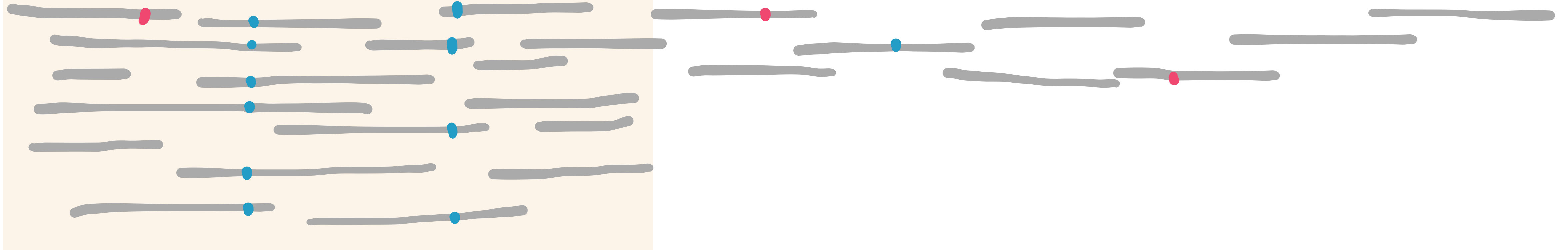


It dramatically simplifies our problem if we assume only a single source of reads

Sequencing coverage is sufficient for redundancy and error correction

Assume that we have **high coverage**

TACGATCGGATTACGCGTAGGCTAGCTTACGGACTCGATGTACGATCGGATTACGCGTAGG



Real sequencing errors can be fixed in high-coverage areas

Real SNPs can be confidently detected when all reads have the same base

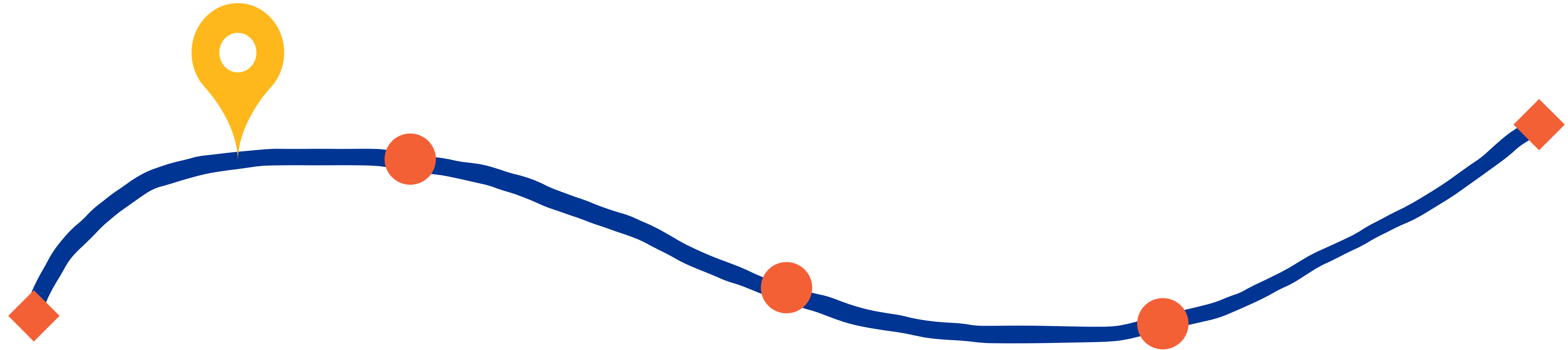
What if your sequencing data does not meet these assumptions?

This happens all the time in science!

If you more robust options are available, using those may be required

If there is no other option, use the best approach and disclose how this could impact your results and interpretation

After today, you should have a better understanding of



Problem formulation of genome assembly

String manipulation in Python

Review: DNA sequences are represented as strings in Python

A DNA sequence is simply a sequence of letters: A, T, C, and G. In Python, we can represent this using quotation marks (" " or ' ').

```
1  read1 = "ATCG"
2  read2 = "TCGA"
```

```
1  read_long = """
2  CGTAGCTGACGTAGATTCGATTTAAGAAACGCAGATATGGACATTGTCGCCGTGCCTTTATAT
3  TCCACATATCGTGGTAATCATACCGGCATAGGGTCATGTCCGCAGCTGTCCAACTATCGGTTA
4  ACGTTCCCCCTACTATCTCTGCGCGAGCCTAGAGTAAATCGATGAGTCTGAAGAACGCCTCAT
5  ATCTGCTGTATGCCCGCCGCGTGAACTCTCAGTATTCGCGAACACATTGGTCTTGCTATCCTC
6  GGTAAGGAAC
7  """
```

Comparing strings allows us to detect similarities or differences between DNA reads

To compare strings, we can use the equality operator ==

```
1 read1 = "ATCG"
2 read2 = "TCGA"
3
4 # Compare two strings
5 print(read1 == read2)  # Output: False
```

```
1 read1 = "ATCG"
2 read2 = "ATCG"
3
4 # Compare two strings
5 print(read1 == read2)  # Output: True
```

We can extract parts of a string using indices in Python

Each **character** in a string has an **index**

Example: "C" has index of 2

```
0123
ATCG
```

Use square brackets `[ ]` to get a character by its index

```
1 read = "ATCG"
2 print(read[0])  # Output: A
3 print(read[2])  # Output: C
```

Use slicing with `start:stop` to get part of a string

```
1 print(read[0:2])  # Output: AT
2 print(read[1:3])  # Output: TC
```

Python does not include the `stop` index

We can use loops to check every position in a string

Use a `for` loop to go through each character one by one

```
1 read = "ATCG"
2
3 for char in read:
4     print(char)
5 # Output:
6 # A
7 # T
8 # C
9 # G
```

You can also slice inside of a `for` loop with an index

```
1  read = "ATCG"
2
3  for i in range(len(read)):
4      # Print substrings starting at index i
5      print(read[i:])
6  # Output:
7  # ATCG
8  # TCG
9  # CG
10 # G
```

`range(len(read))` generates integers from 0 until the length of the read (in this case 4)

Comparing parts of strings allows us to find overlaps between DNA reads

Let's find where `read1` overlaps with `read2`

```
1  read1 = "ATCG"
2  read2 = "TCGA"
3
4  for i in range(len(read1)):
5      if read1[i:] == read2[:len(read1) - i]:
6          print(f"Overlap found: {read1[i:]}")
7          break
8  # Output: Overlap found: TCG
```

When `i = 0`:

- `read1[0:]` gives us `"ATCG"` (the full string)
- `read2[:4]` gives us `"TCGA"` (first 4 characters)
- Comparison: `"ATCG" == "TCGA"`
- **Result:** No match

Comparing parts of strings allows us to find overlaps between DNA reads

```python
read1 = "ATCG"
read2 = "TCGA"

for i in range(len(read1)):
    if read1[i:] == read2[:len(read1) - i]:
        print(f"Overlap found: {read1[i:]}")
        break
# Output: Overlap found: TCG
```

Next is `i = 1`:

- `read1[1:]` gives us `"TCG"` (excluding `'A'`)
- `read2[:3]` gives us `"TCG"` (first 3 characters)
- Comparison: `"TCG" == "TCG"`
- **Result:** Match found! 🎉

Comparing parts of strings allows us to find overlaps between DNA reads

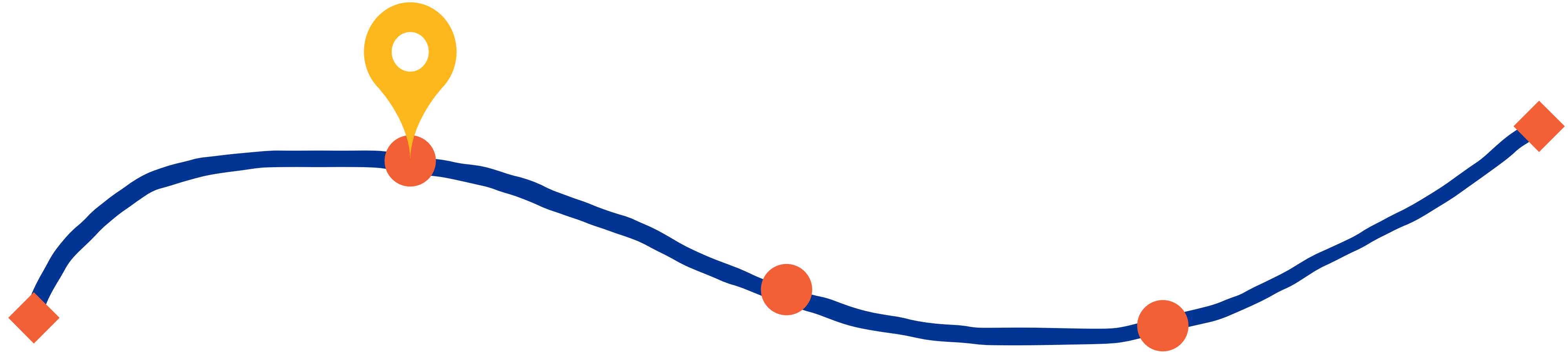
Once we find the overlap, we can merge the reads

We can use this approach of **finding overlaps** and **merging reads** to form a contig

```
1 read1 = "ATCG"
2 read2 = "TCGA"
3
4 i = 1
5
6 merged = read1[:i] + read2
7 print(merged)
8 # Output: ATCGA
```

This idea of **finding overlaps and merging** motivates our first assembly approach: the **greedy algorithm**

After today, you should have a better understanding of



The greedy algorithm for genome assembly

Overlaps and merges

The greedy algorithm builds genome assemblies by iteratively merging the best overlaps

Algorithm

1. Check every possible read for the largest overlap.

2. Merge the two reads with largest overlap.

3. Repeat until no further merges are possible.

At the end, we have a set of contigs that represent our original DNA sequence

The greedy algorithm minimizes repeats by maximizing overlap

A **superstring** is a single string that contains all reads as substrings

Example: AC**GT**AC is a superstring of

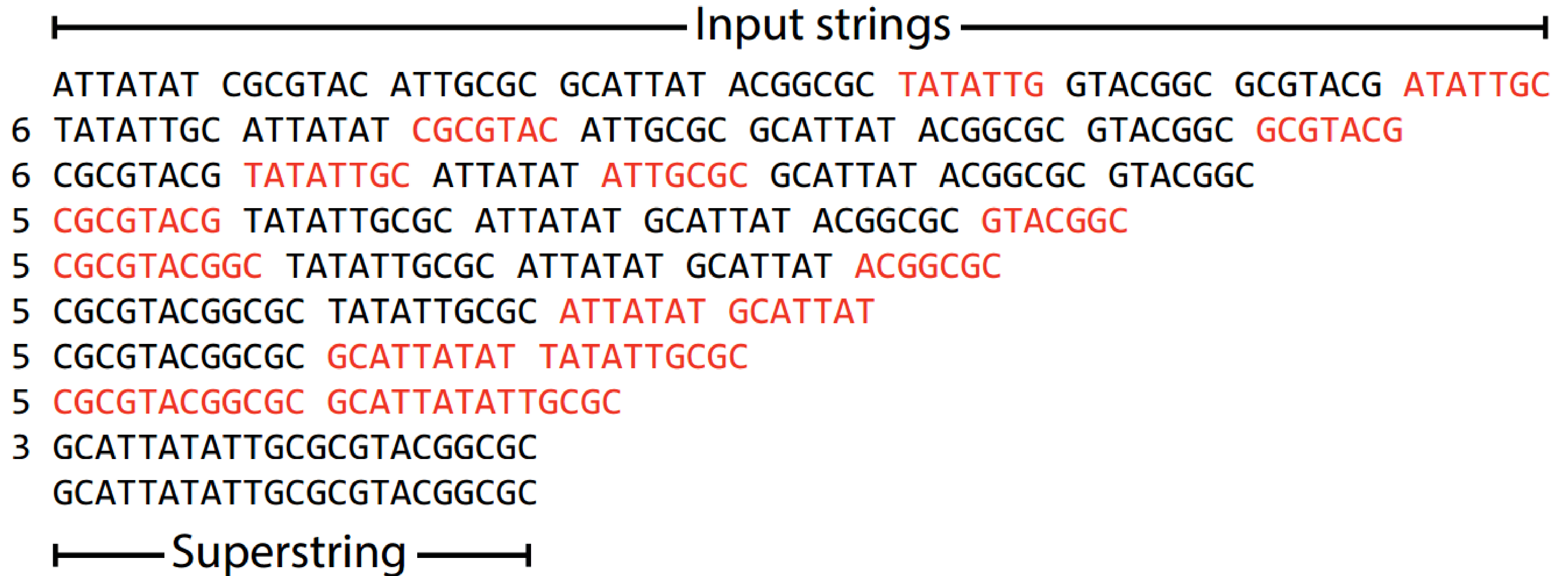
AC**GT**, C**GT**A, **GT**AC

The greedy algorithm aims to find the *shortest superstring*, which minimizes unnecessary duplication.

The greedy algorithm focuses on **selecting the best immediate option** (i.e., local optimal) at each step **without full consideration of the overall global solution**

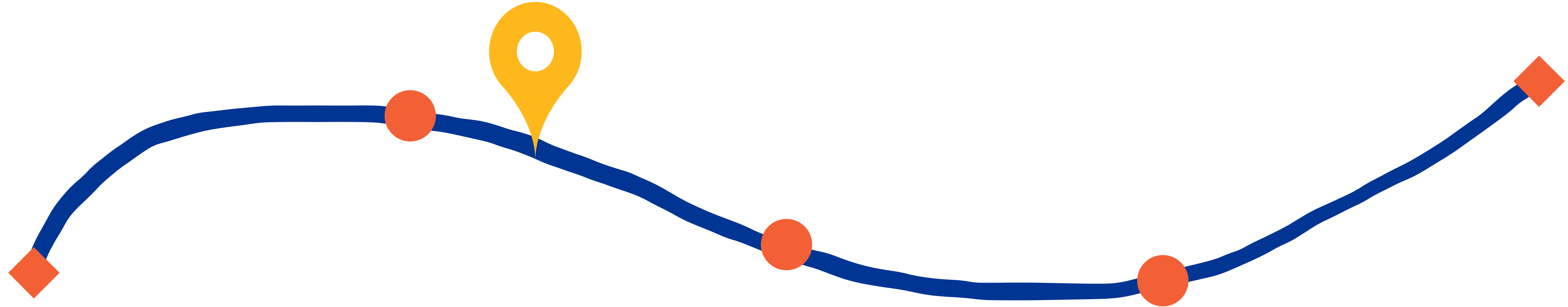
This means the greedy algorithm will always make the best move in the moment even if it gives the wrong final answer

Being greedy makes genome assembly tractable



Rounds of merging, one merge per line.
Number in first column = length of overlap merged before that round.

After today, you should have a better understanding of



The greedy algorithm for genome assembly

Breaking ties

Tie-breaking rules are necessary when overlaps are identical

Suppose we have these three reads with a highest **overlap of five**

R1	R2	R3
TAACGT	ACGTAA	CGTAAC

We merge reads R2 and R3: (and keep R1)

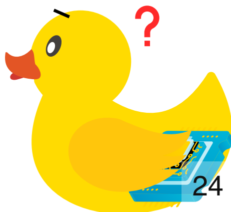
TAACGT ACGTAAC

However, now we have a problem

TAACGT ACGTAAC **Overlap of 4** **ACGTAACGT**	TAACGT ACGTAAC **Overlap of 4** **TAACGTAAC**

Both have a length of 9, which one is the correct move?

Talk with your neighbors



Tie breakers are a personal preference

First encountered, first merged	The one you found first
Highest quality base calls	Use sequence with highest quality
Highest coverage	Whichever results in more coverage
Look ahead	Do both and evaluate consequences
Exclude	Be petty and don't merge them (separate contigs)

After today, you should have a better understanding of

The greedy algorithm for genome assembly

Trouble with repeats

Greedy assembly will incorrectly collapse repeats if possible

Let's take a string and cyclically permute it with k = 6

a_long_long_long_time

```
ng_lon _long_ a_long long_l ong_ti ong_lo long_t g_long g_time ng_tim
ng_time ng_lon _long_ a_long long_l ong_ti ong_lo long_t g_long
ng_time g_long_ ng_lon a_long long_l ong_ti ong_lo long_t
ng_time long_ti g_long_ ng_lon a_long long_l ong_lo
ng_time ong_lon long_ti g_long_ a_long long_l
ong_lon long_time g_long_ a_long long_l
long_lon long_time g_long_ a_long
long_lon g_long_time a_long
long_long_time a_long
a_long_long_time
a_long_long_time
```

Then perform the greedy algorithm

We are missing a "_long". **Why?**

Longer reads and genome assembly

k = 8

a_long_long_long_time

```
long_lon ng_long_ _long_lo g_long_t ong_long g_long_l ong_time a_long_l _long_ti long_tim
long_time long_lon ng_long_ _long_lo g_long_t ong_long g_long_l a_long_l _long_ti
_long_time long_lon ng_long_ _long_lo g_long_t ong_long g_long_l a_long_l
_long_time a_long_lo long_lon ng_long_ g_long_t ong_long g_long_l
_long_time ong_long_ a_long_lo long_lon g_long_t g_long_l
g_long_time ong_long_ a_long_lo long_lon g_long_l
g_long_time ong_long_ a_long_lon g_long_l
g_long_time ong_long_l a_long_lon
g_long_time a_long_long_l
a_long_long_long_time
a_long_long_long_time
```

We get the correct string back, but how did increasing our k fix this?

By having one read span all three "long"s, (i.e., the repeating region) we prevented a collapse

a_long_long_long_time

g_long_l

Remember: This is why long sequencing reads are very helpful in resolving repeats!

After today, you should have a better understanding of

De Bruijn graphs and their role in assembly

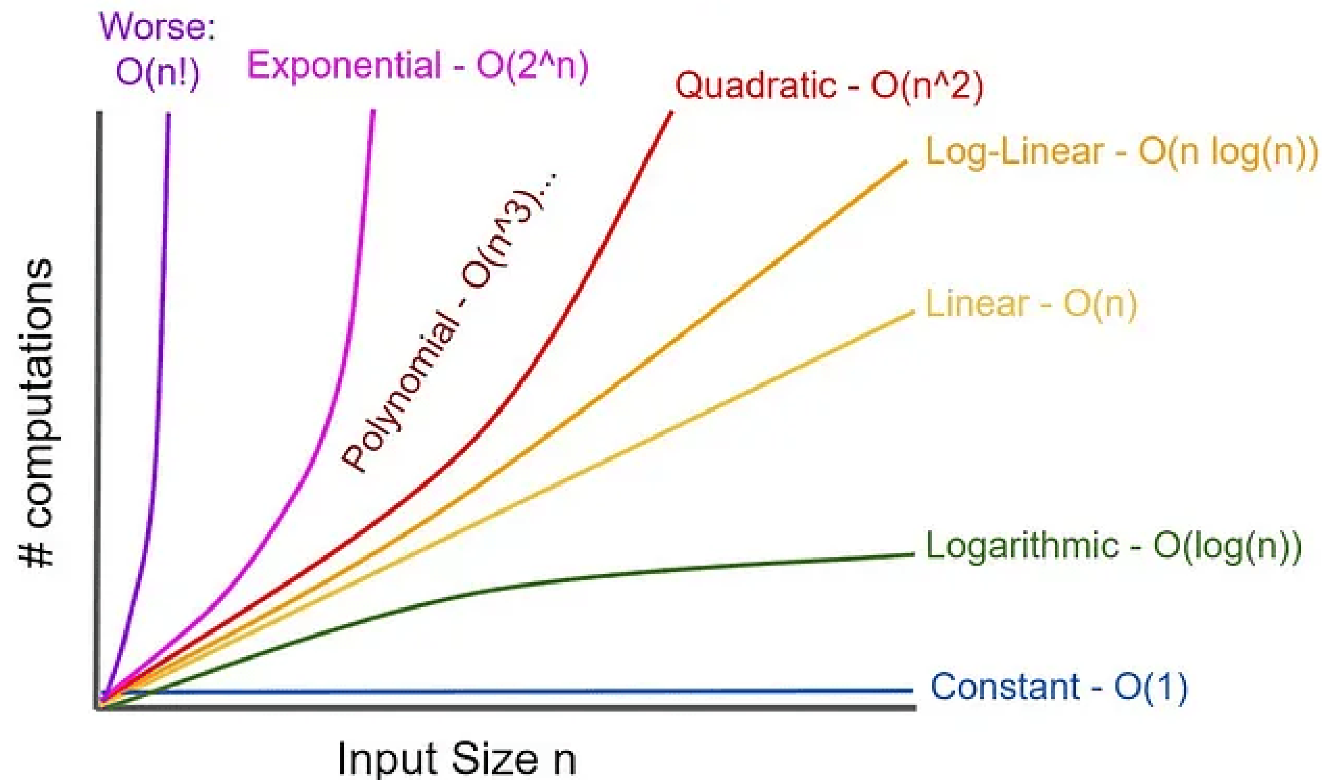
K-mers

The greedy algorithm provides insights but is rarely used in modern genome assembly

The greedy approach is computationally efficient but fails for large, complex genomes.

Finding overlaps between all reads scales poorly with genome size

Full pairwise comparisons between reads require $O(n^2)$ operations where n is the number of reads



As our number of reads increases, our time to find overlaps dramatically increases

However, the number of reads also improves our assembly

k-mers break reads into manageable, fixed-length pieces

Instead of comparing whole sequences, we can compare k-mers!

A k-mer is a substring of length k extracted from a sequence

Example: For the sequence **ATCGT**, the 3-mers are **ATC, TCG, CGT**.

By decomposing reads into k-mers, we can:

- Represent sequences as collections of overlapping k-mers.
- Avoid comparing entire reads by focusing on k-mer matches.
- Use fixed-length k-mers to tolerate sequencing errors in overlaps.
- Number of reads does not change number of k-mers

Building k-mers from a string

Spectrum with k = 3

GGCGATTCATCG

1. Slice first k characters
2. Shift right one character
3. Repeat

GGC
GCG
CGA
GAT
ATT
TTC
TCA
CAT
ATC
TCG

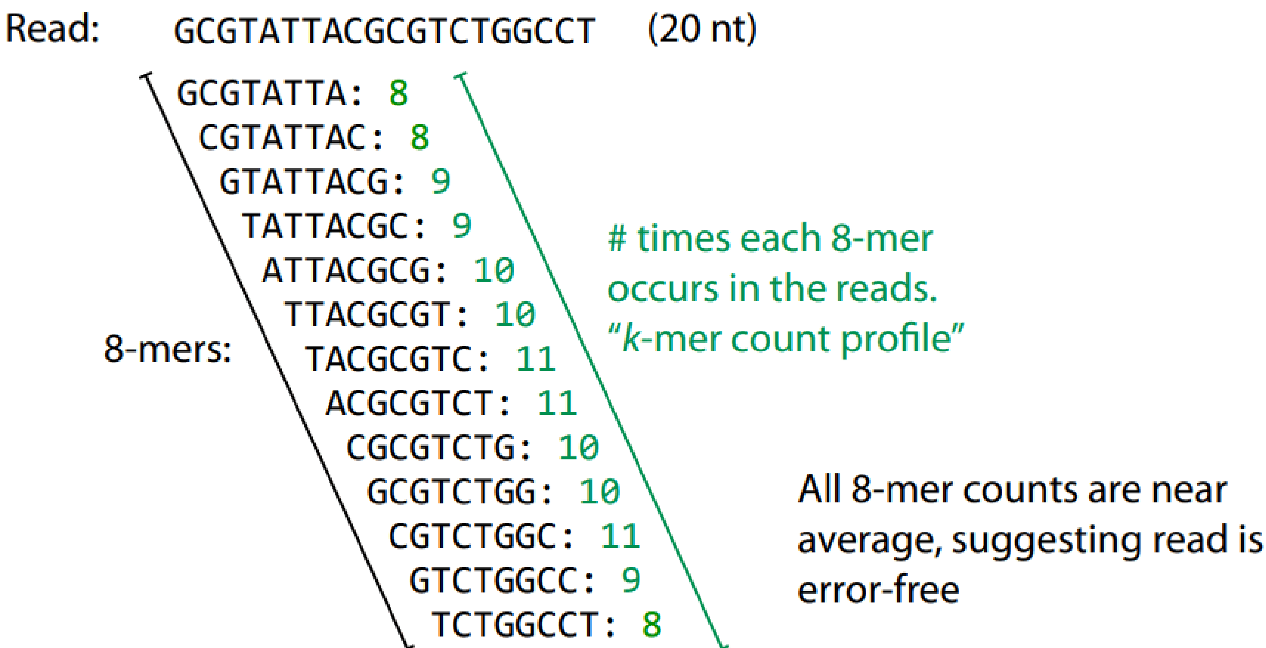
All 3-mers

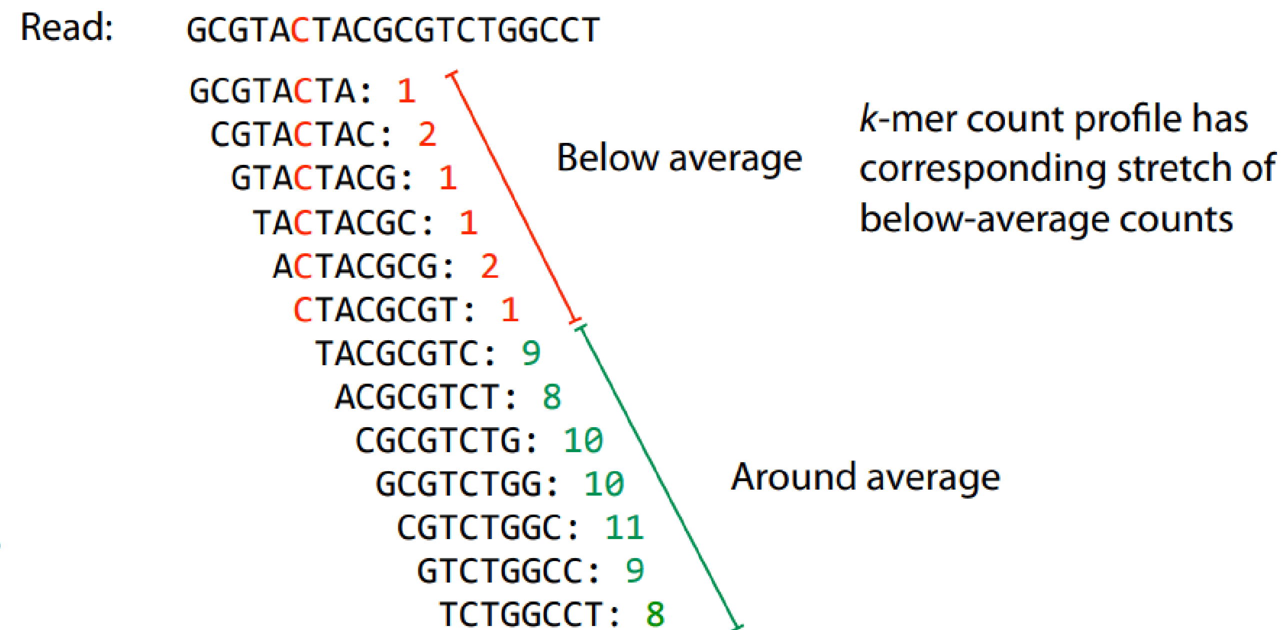
k-mers are robust to sequencing errors

Sequencing errors affect only a few k-mers in a read, not the entire sequence.

Even if a single read has errors, most k-mers will match correctly to others.

Longer k-mers provide specificity, while shorter k-mers ensure sensitivity.



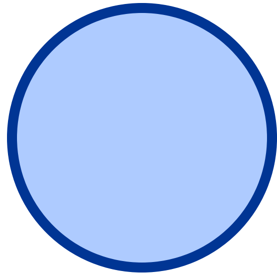


After today, you should have a better understanding of

De Bruijn graphs and their role in assembly

Building graphs

Graphs is a data structure for drawing relationships between items



Node

Represents a single entity

- Person
- Location
- Protein
- Sequencing read

Edge

Represents a connection (possibly with a direction)

- Instagram follower
- Flights
- Protein-protein interaction
- Sequence overlap

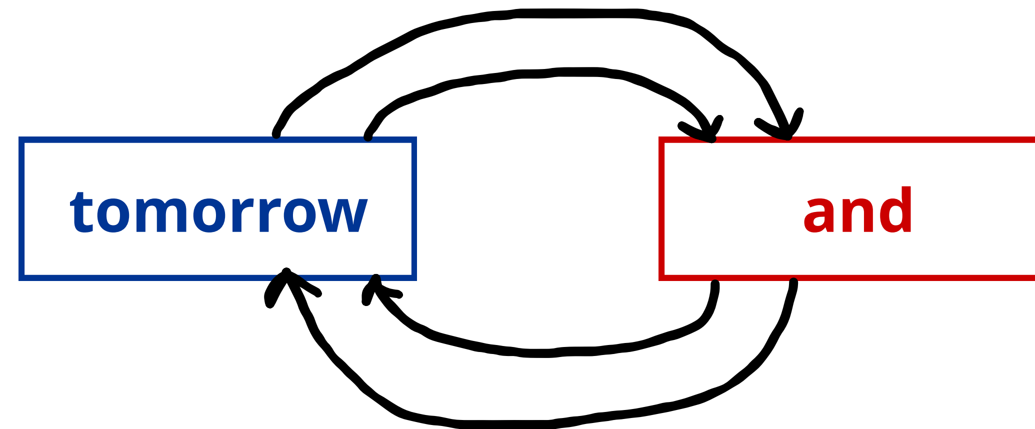
Genome assembly uses direct edges to specify overlap and concatenation

Let's build a **directed multigraph:**

"tomorrow and tomorrow and tomorrow"

1. Each unique k-mer is a node
2. Add directed edges for each overlap and concatenation

K-mer is a substring of length k



(We will cheat here and write down just unique words)

Build a De Bruijn graph with k-1 nodes

5' AATGGCGTA 3'

Step 1: Build k-mers

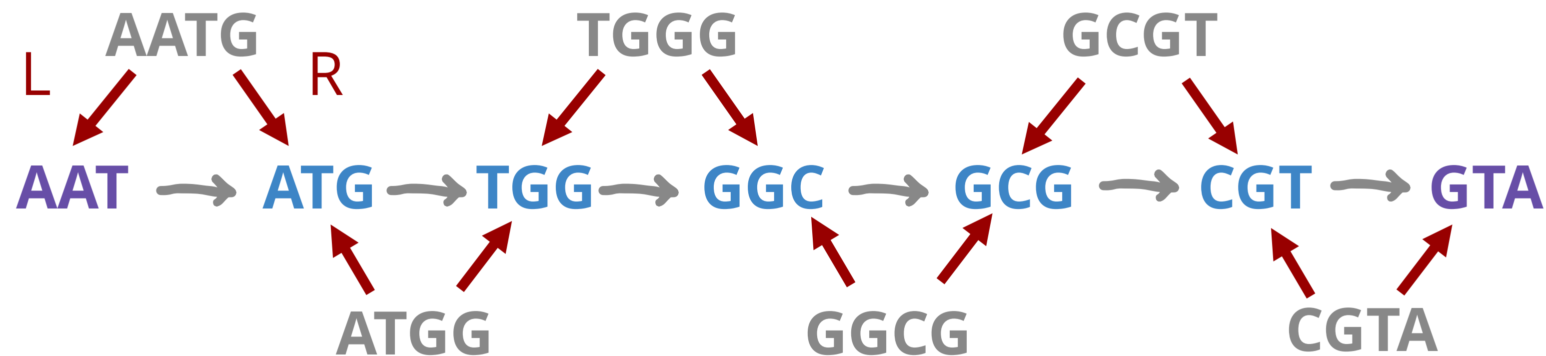
Let's use k = 4

AATG **ATGG** **TGGG**

GGCG **GCGT** **CGTA**

Step 2: Take left and right k-1 mer and make two connected nodes

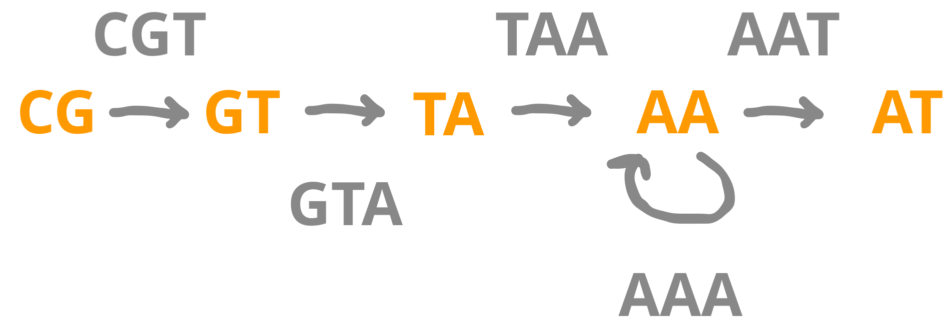
Step 3: Repeat

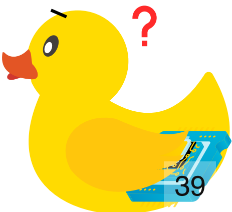


Building De Bruijn graphs with a read

Build a De Bruijn graph with k = 3

CGTAAAT





De Bruijn graphs with multiple reads

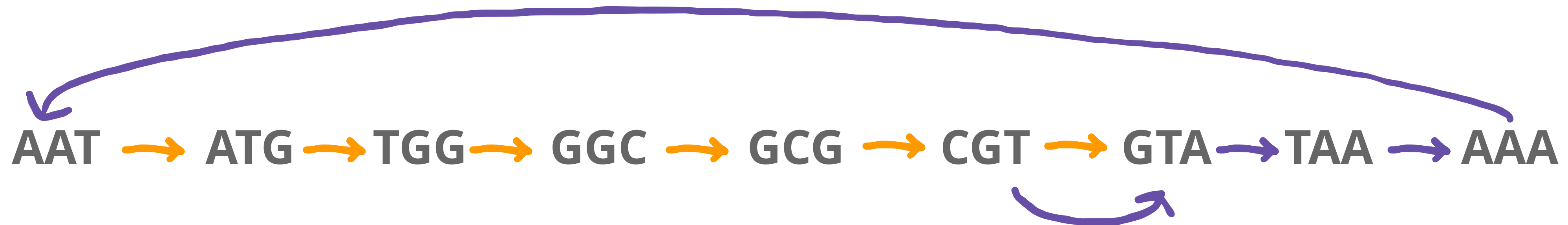
Read 1
5' AATGGCGTA 3'

Let's use nodes of length 4

Read 2
5' CGTAAAT 3'

Frist, build the De Bruijn graph for **Read 1**

Add edges and any new k-mers from **Read 2**



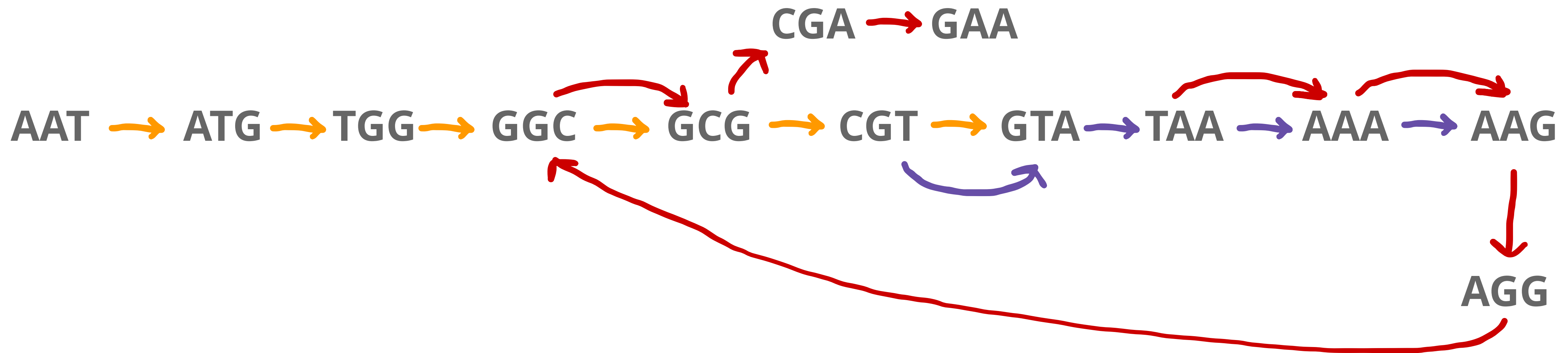
Note: This is a circular genome

Another example, but not circular

Read 1
5' AATGGCGTA 3'

Read 2
5' CGTAAAG 3'

Read 3
5' TAAAGGCGAA3'

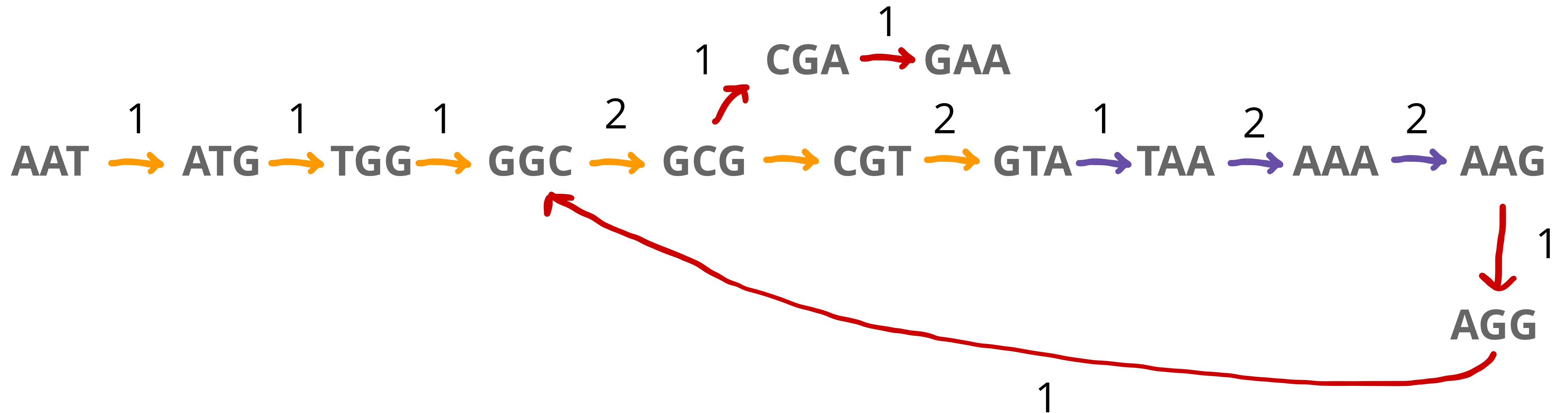


We can add weights to edges instead of drawing multiple edges

Read 1
5' AATGGCGTA 3'

Read 2
5' CGTAAAG 3'

Read 3
5' TAAAGGCGAA3'

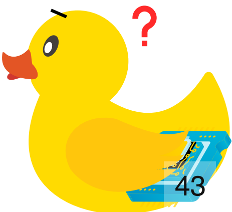


Another (another) example, but not circular

GATTAC TACAGATT AGATTAC TACCGG GGATTA

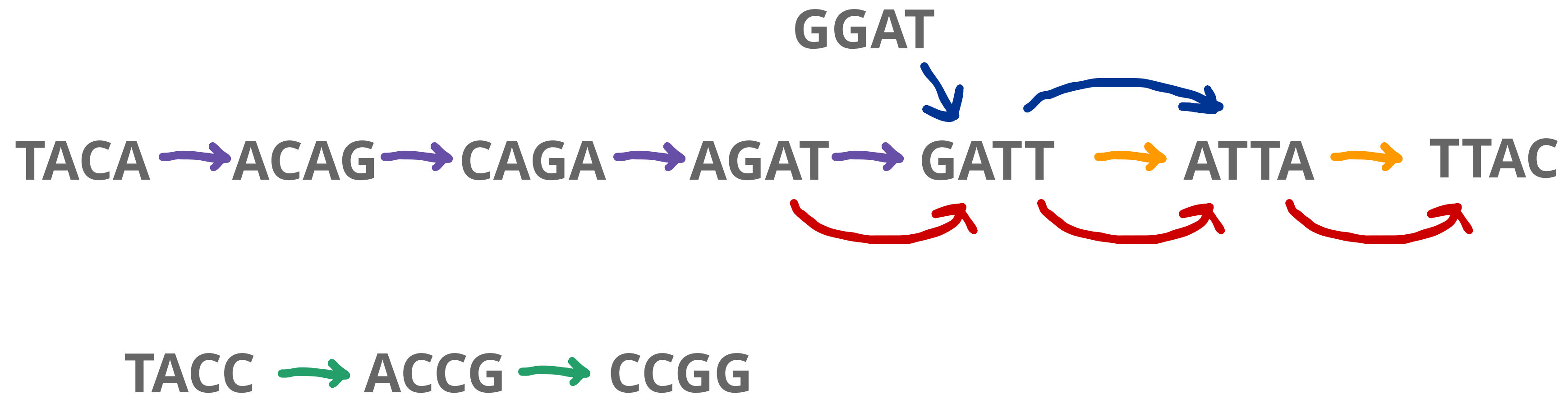
De Bruijn graphs is one of the most missed questions on assessments, let's get some practice

The solution is on the next slide (no peeking!)



Another example, but not circular

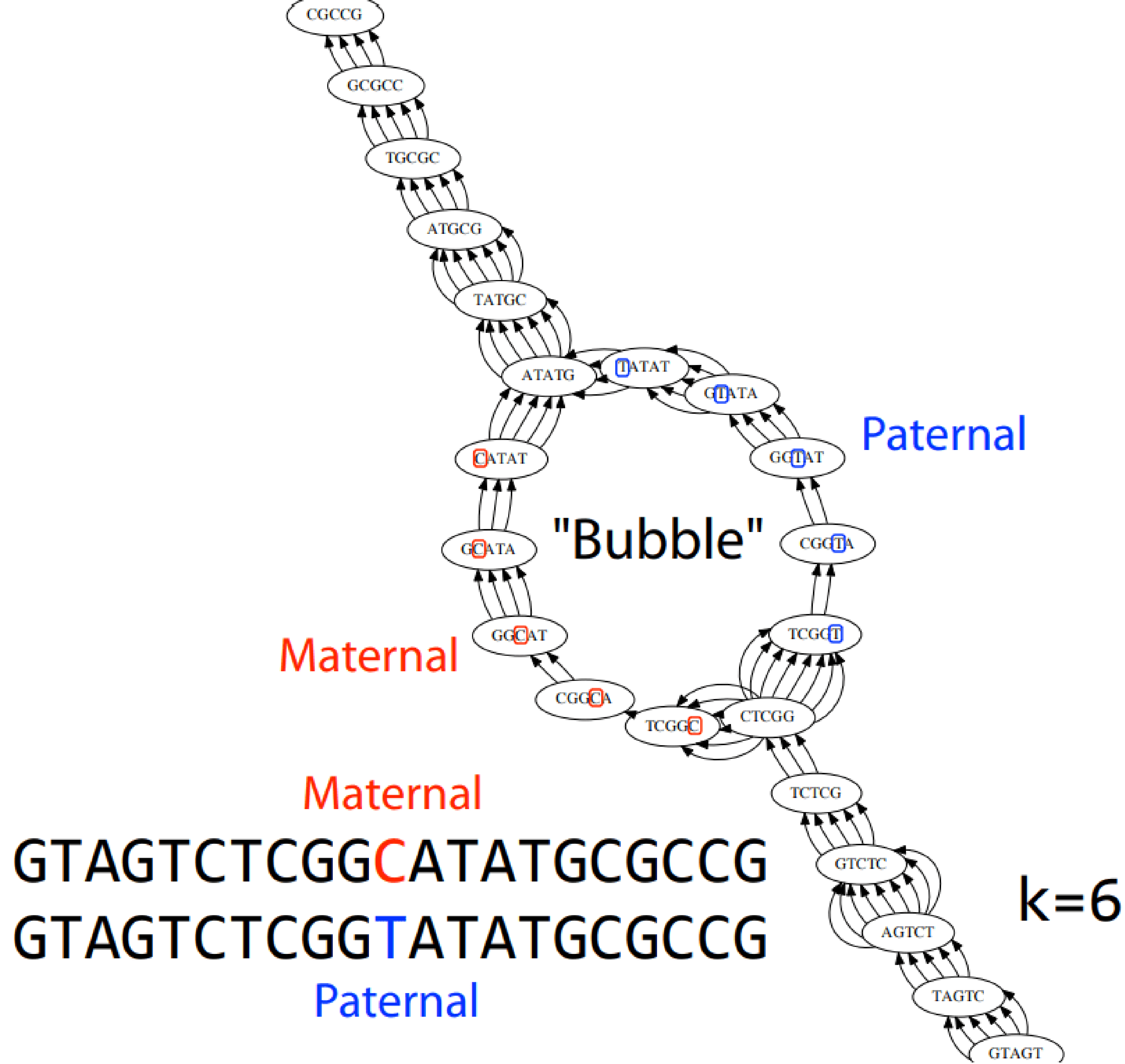
GATTAC TACAGATT AGATTAC TACCGG GGATTA



After today, you should have a better understanding of

De Bruijn graphs and their role in assembly

Characteristics


CGCCG
GCGCC
TGCGC
ATGCG
TATGC
ATATG
TATAT
GTATA
GGTAT
CGGTA
TCGGT
CATAT
GCATA
GGCAT
CGGCA
TCGGC
CTCGG
TCTCG
GTCTC
AGTCT
TAGTC
GTAGT
Paternal
"Bubble"
Maternal
Maternal
GTAGTCTCGGCATATGCGCCG
GTAGTCTCGGTATATGCGCCG
Paternal
k=6

Tips are good starting points of contigs if they have high coverage

Islands are reads we couldn't merge

"Tip"

"Island"

After today, you should have a better understanding of

De Bruijn graphs and their role in assembly

Graph data structures in Python

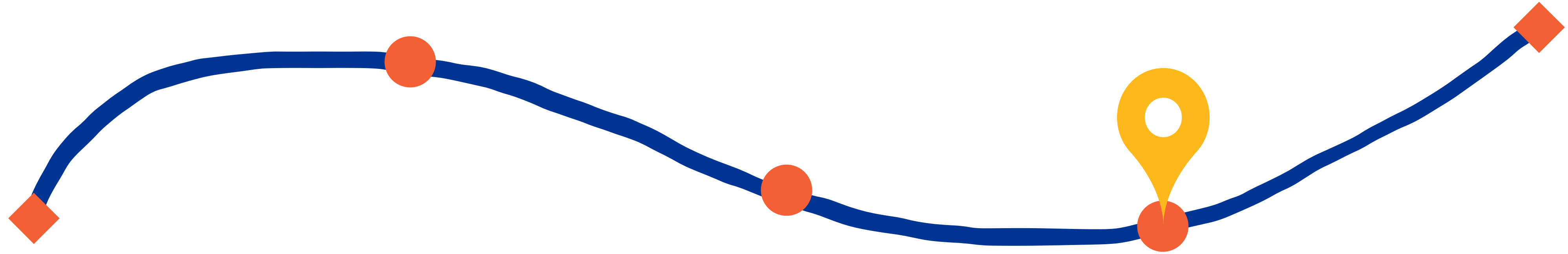
Graph representation in Python

Adjacency lists can be used to computationally represent graphs

```python
# Example of a weighted directed graph
graph = {
    'A': {'B': 5, 'C': 10},  # A -> B (weight 5), A -> C (weight 10)
    'B': {'D': 15},          # B -> D (weight 15)
    'C': {'D': 20},          # C -> D (weight 20)
    'D': {}                  # D has no outgoing edges
}
```

Perhaps conceptually helpful for CByte 02!

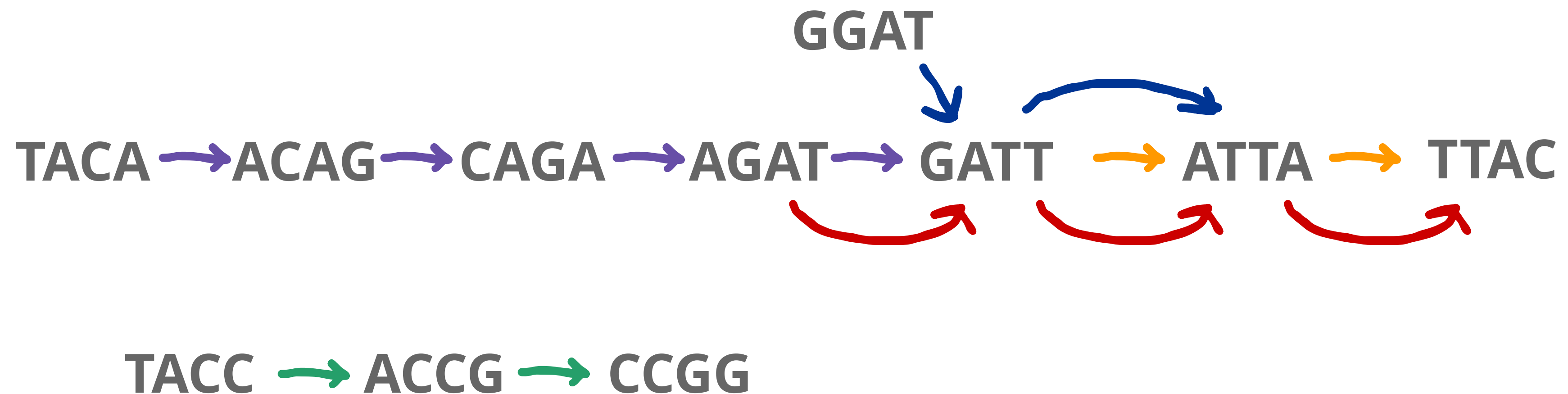
After today, you should have a better understanding of



Graph traversal methods for extracting contigs

De Bruijn graphs are traversed to extract contiguous genome sequences

Traversal is the process of finding contigs (continuous DNA sequences) by walking through the De Bruijn graph



Nodes: Represent k-mers derived from sequencing reads.

Edges: Represent k-mer overlaps between nodes.

Standard traversal methods, such as breadth-first search (BFS) and depth-first search (DFS), are building blocks for more advanced assembly techniques.

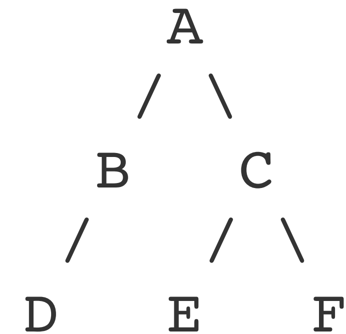
DFS explores as far as possible along each branch before backtracking

Imagine exploring a maze with this strategy:

- Keep walking forward until you hit a dead end
- Backtrack only when necessary
- Take the first unexplored path you see

DFS Traversal from A (one possible order):

1. A → B → D (follow first path to end)
2. Backtrack to A
3. A → C → E
4. Backtrack to C
5. C → F



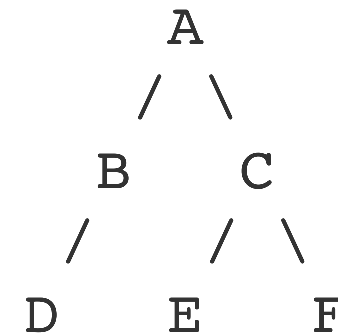
BFS explores all neighbors of a node before moving deeper into the graph

Imagine you're dropping a pebble in a pond:

- First, you see ripples reach nearby points
- Then, they spread outward in circles
- Each "wave" represents a level of exploration

DFS Traversal from A (one possible order):

1. A → B → D (follow first path to end)
2. Backtrack to A
3. A → C → E
4. Backtrack to C
5. C → F



Standard traversal methods struggle with genome assembly challenges

- Repeats, cycles, and ambiguous paths in De Bruijn graphs complicate DFS and BFS.
- Genome assembly requires visiting all overlaps (edges) or all reads (nodes) systematically.
- Specialized traversal methods, like Eulerian and Hamiltonian paths, address these challenges.

Before the next class, you should

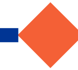
Lecture 03B:

Genome assembly - Methodology

Lecture 04A:

Genome annotation - Foundations

Quiz 01



Today

Tuesday

- Finish and submit P01B (due Jan 24)
- Start P01C (due Jan 31)
- Work on CByte 01 and CByte 02
- Review Lectures 02A, 02B, 03A, and 03B for quiz (Jan 28)