CS 364 COMPUTATIONAL BIOLOGY

Sara Mathieson
Haverford College

Outline

- Finish BWT (runtime, inexact matching)
- Begin: genome assembly
- Overlap graphs for genome assembly

Finish BWT: runtime and inexact matching

Pattern matching with BWT

- Setup time *O(N)*
- Search time *O(M)*
- Storage space *O(N)*
  - *O(1)* to store *F* (i.e. *M*)
  - *O(N)* to store *L* (i.e. *BWT(S))*
  - *O(N)* to store *A*
  - *O(N|Σ|) to store OCC* ("check-pointing" extension allows you to store only part of *OCC*, without increasing complexity).
- Inexact matching can be implemented in a similar way to inexact matching with little extra cost (as long as few mismatches)

Summary

Algorithm	Setup time	Lookup time	Storage space
Boyer-Moore	O(M)	O(N)	O(M)
k-mer hash table	O(N)	O(M)	O(N)
BWT/FM-index	O(N)	O(M)	O(N)

But, in practice, for the read mapping problem, BWT approaches have turned out to be the most efficient. Almost all sequence data is processed with a program called *bwa* which uses BWT to map.

Brief history of BWT and read mapping application

- 1994, BWT introduced (as a compression algorithm)
  - Burrows, M. and Wheeler, D.J. (1994) A block-sorting lossless data compression algorithm. *Technical report 124*, Palo Alto, CA, Digital Equipment Corporation.

- 2000, FM-index for fast searching
  - Ferragina,P. and Manzini,G. (2000) Opportunistic data structures with applications. In *Proceedings of the 41st Symposium on Foundations of Computer Science (FOCS 2000)*, IEEE Computer Society, pp. 390–398.

- 2008, *BWT-SW* for sequence alignment
  - Tam, C. K. Wong, S. M. Yiu (2008) Compressed indexing and local alignment of DNA, Bioinformatics 24

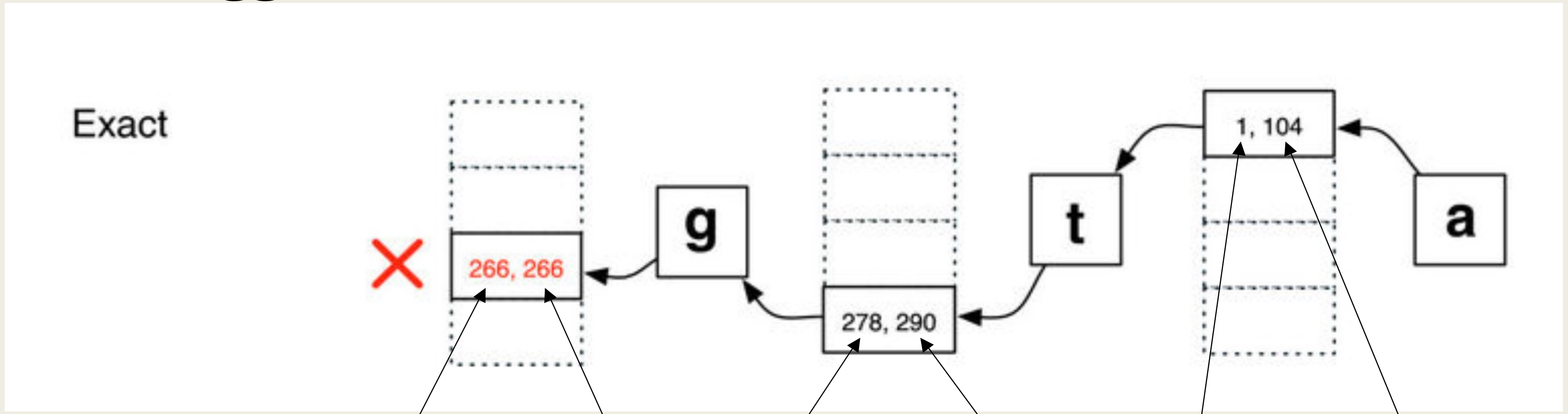
- 2009, *Bowtie* for short read alignment (~23,000 citations to date)
  - Langmead,B. Trapnell,C. Pop, M. Salzberg, S. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biology 10:R25

- 2009, *bwa* (~46,000 citations in 2024)
  - Li, H. and Durbin, R. Fast and accurate short read alignment with Burrows–Wheeler transform Bioinformatics 25: *1754–1760*

Bowtie: exact matching

- Figure 2 from the Bowtie paper: exactly what we have done in class, except exclusive of end-point

Read: 'ggta'

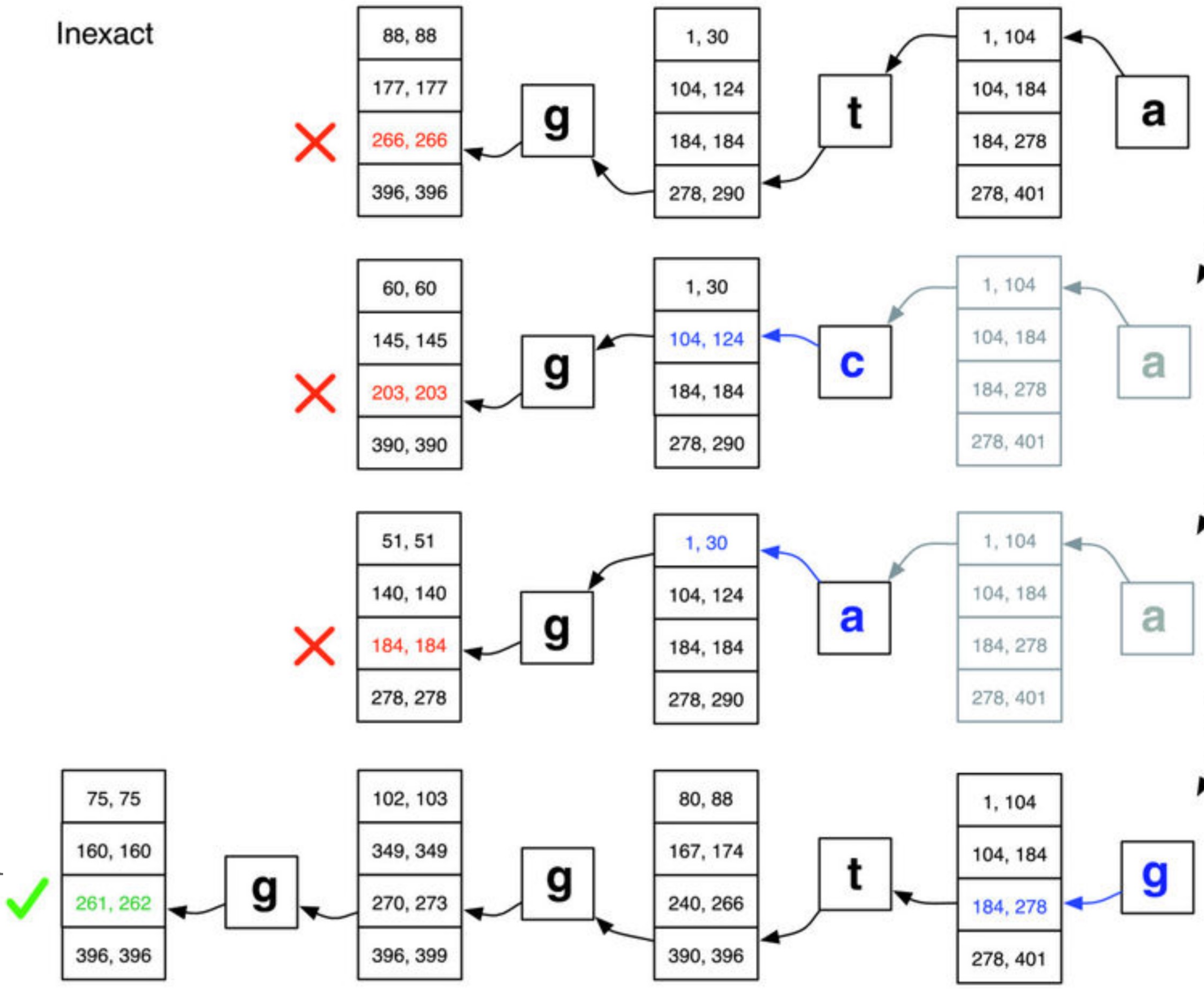


sp(*gta*) = ep(*gta*) sp(*ta*) ep(*ta*) sp(*a*) ep(*a*)

Equal (in this case), means no occurrences!
(for us, start > end means no occurrences)

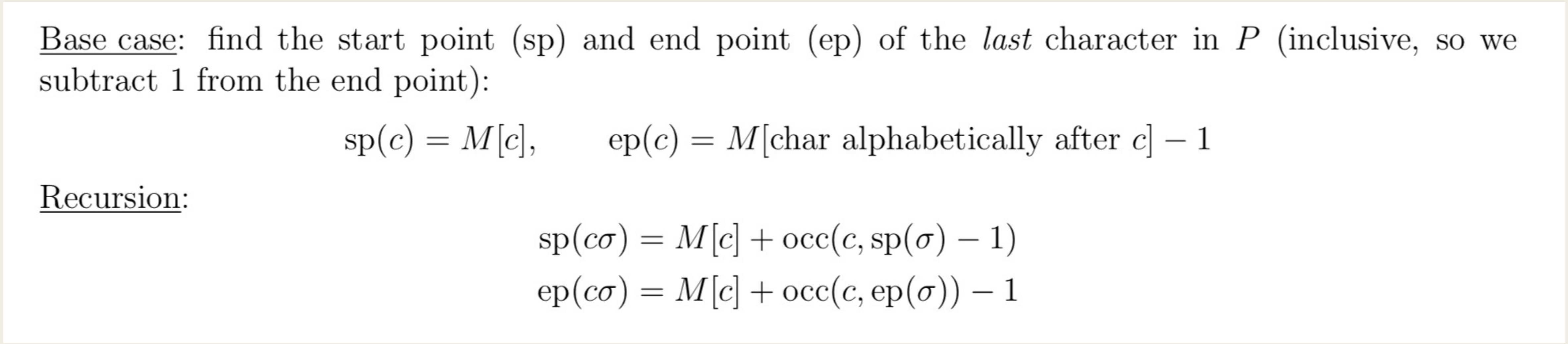
Bowtie: inexact matching

Read: 'ggta'



Did not find the read, but we did find: 'ggtg'

BWT pattern matching algorithm



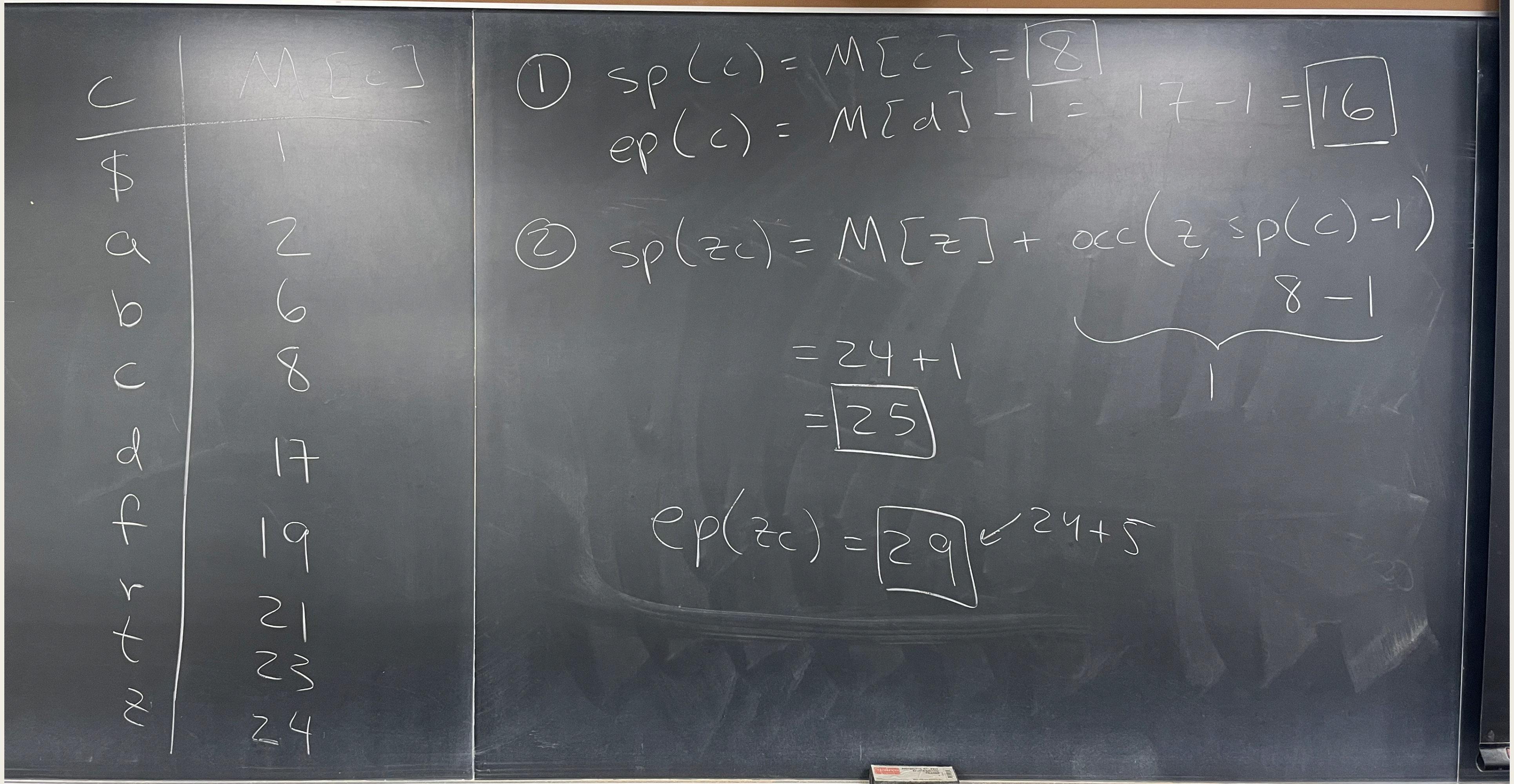
Base case: find the start point (sp) and end point (ep) of the *last* character in P (inclusive, so we subtract 1 from the end point):

$$\text{sp}(c) = M[c], \qquad \text{ep}(c) = M[\text{char alphabetically after } c] - 1$$

Recursion:

$$\text{sp}(c\sigma) = M[c] + \text{occ}(c, \text{sp}(\sigma) - 1)$$
$$\text{ep}(c\sigma) = M[c] + \text{occ}(c, \text{ep}(\sigma)) - 1$$

Handout 4



c	M[c]
$	1
a	2
b	6
c	8
d	17
f	19
r	21
t	23
z	24

① $sp(c) = M[c] = \boxed{8}$

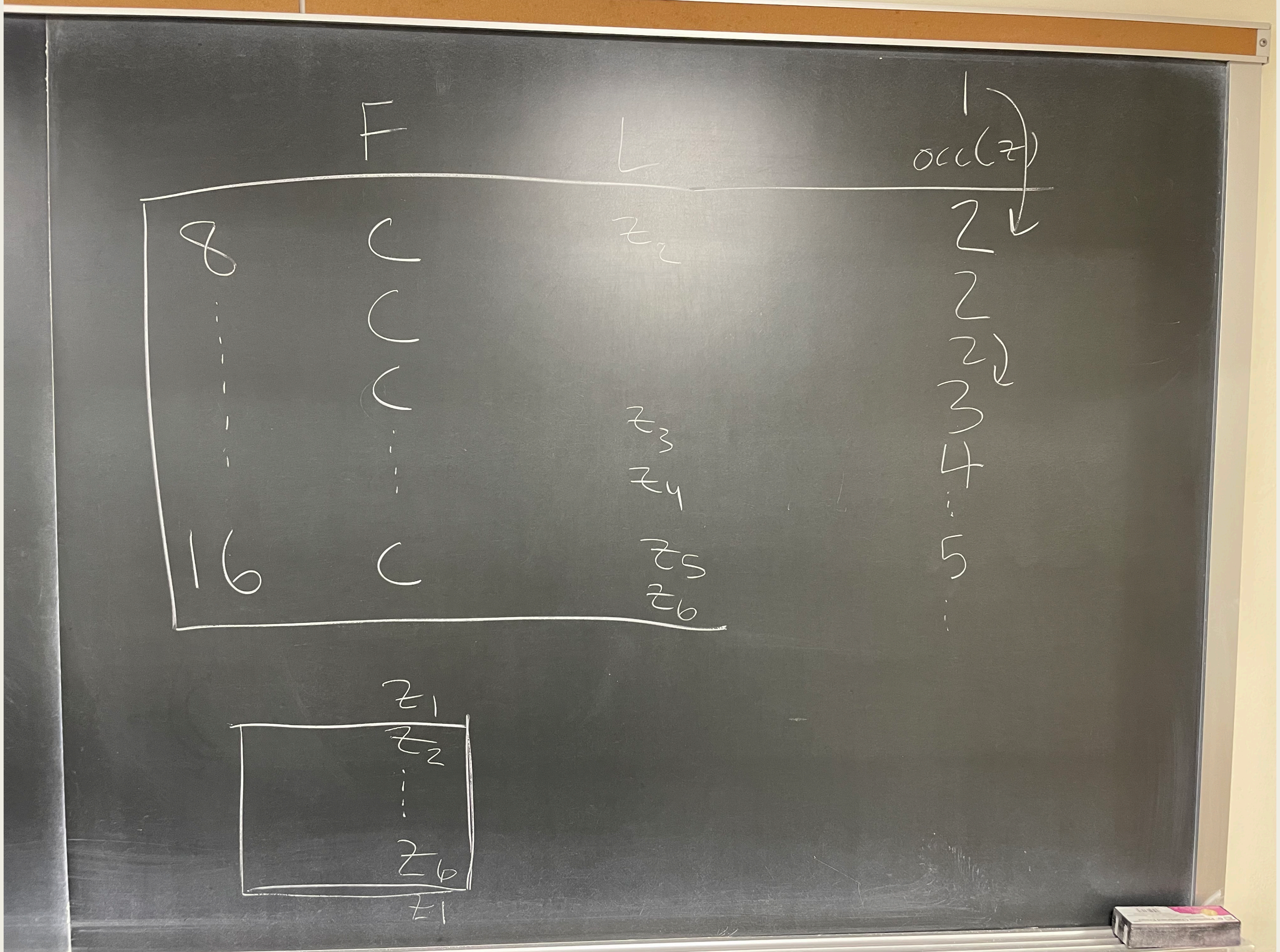
$ep(c) = M[d] - 1 = 17 - 1 = \boxed{16}$

② $sp(zc) = M[z] + \underbrace{occ(z, \overset{8-1}{sp(c)-1})}_{1}$

$= 24 + 1$

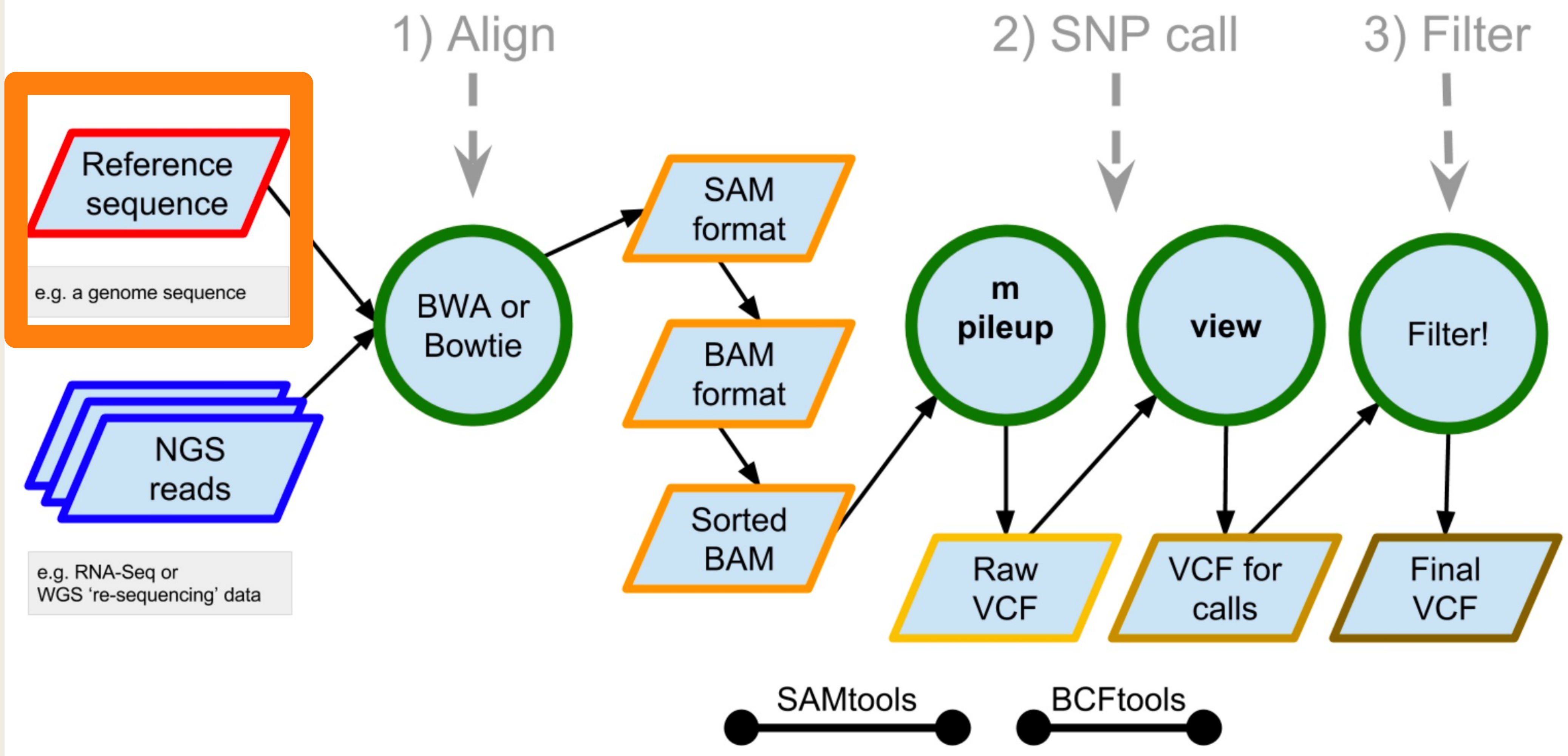
$= \boxed{25}$

$ep(zc) = \boxed{29} \leftarrow 24+5$


F
L
1
occ(z)
8
C
C
C
16
C
z_2
z_3
z_4
z_5
z_6
2
2
2
3
4
5
z_1
z_2
z_6
z_1

Begin: genome assembly
(i.e. how do we get the reference?)

Pipeline overview



Credit: Dan Bolster (EBI)

How do we get the reference genome in the first place?

CTCACCAGACCTCCTAGGCGACCCAGACAATTATACCCTAGCCAACCCCTTAAACACCCCTCCCCAC

↓ Shotgun sequencing

AGACCTCCTAGGCGA

CAGACAATTATACCCTAG

CCTCCTAGGCGACCC

ACCCTAGCCAACCCCTT

AGGCGACCCAGACAATTAT

ACACCCCTCCCCA

CCCTAGCCAACCCCTT

↓ ???

Can we put them back together again?

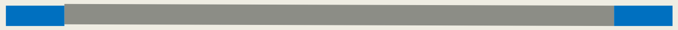
Goal of *de novo* genome assembly

- Input: millions of “reads” (patterns) from next-generation sequencing

- Output: (ideally) entire “consensus” sequence of the original DNA (creating a reference)

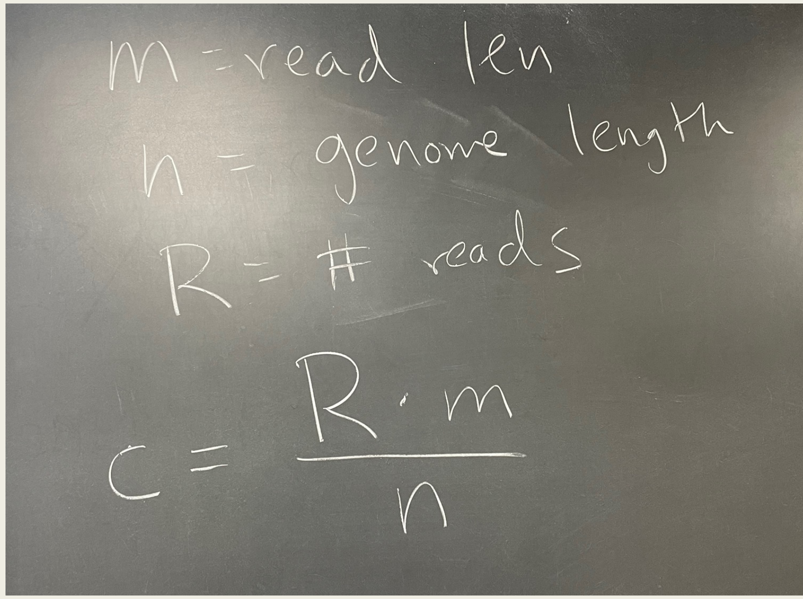
Assembly vocabulary

- **Long read**: a fragment that has been “read” from a genomic sequence (DNA for us), usually > 1000 bp
- **Short read**: same as a long read but < 1000 bp (usually 100-150 bp)
- **Paired-end read**: both ends of a fragment are “read”, but the portion between them is unknown



- **bp**: base pair
- **kb, Mb, Gb**: kilo bases 10^3, mega bases 10^6, giga bases 10^9
- **Coverage**: number of times (on average) any given base is sequenced. Total number of bases in all reads (R reads × m bases/read), divided by the length of the genome n.

Coverage



m = read len

n = genome length

R = # reads

$$c = \frac{R \cdot m}{n}$$

Genome, length n=30

```
CTCACCAGACCTCCTAGGCGACCCAGACAA

CTCACCAGACCT
   ACCAGACCTCCT
      AGACCTCCTAGG
         CCTAGGCGACCC
          CTCCTAGGCGAC
          CTCCTAGGCGAC
            CCTAGGCGACCC
               AGGCGACCCAGA
                  CGACCCAGACA
                   GACCCAGACAA
```

R=10 reads

each of length m=12

Coverage C=12*10/30=4 (mean number of times each base in the genome is seen)

For humans $n=3\times10^9$, you might have $m=100$ and $R=1\times10^9$

Could you design an algorithm for genome assembly?

1) With a partner, analyze these given reads. What is ***m* (length of each read)**? What is ***R* (number of reads)**?

2) Try to assemble these given reads into **one continuous string**. For these small numbers we can often do this “by eye”, but what if *R* = millions and *m* = 100? How would you tell a computer to assemble them?

3) What is ***n* (length of the resulting genome)**? From all the numbers, compute the **coverage**.

Overlap graph assembly

Overlap graph assembly

```
read 1:   GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
read 2:   AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

```
GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

Overlap graph assembly

```
read 1:   GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
read 2:   AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

```
 GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

Overlap graph assembly

```
read 1:   GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
read 2:   AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

```
  GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

Overlap graph assembly

```
read 1:   GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
read 2:   AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

```
   GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

Overlap graph assembly

```
read 1:   GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
read 2:   AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

```
    GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

Overlap graph assembly

```
read 1:   GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
read 2:   AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

```
                GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

Overlap graph assembly

```
read 1:   GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
read 2:   AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

```
      GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

Overlap graph assembly

```
read 1:   GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
read 2:   AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

```
              GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

Overlap graph assembly

```
read 1:   GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
read 2:   AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

```
        GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

Overlap graph assembly

```
read 1:   GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
read 2:   AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

```
         GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

Overlap graph assembly

```
read 1:   GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
read 2:   AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

```
          GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

Overlap graph assembly

```
read 1:   GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
read 2:   AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

```
               GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

Overlap graph assembly

```
read 1:   GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
read 2:   AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

```
            GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

Overlap graph assembly

```
read 1:   GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
read 2:   AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

```
             GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

Overlap graph assembly

```
read 1:   GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
read 2:   AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

```
              GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

Overlap graph assembly

```
read 1:   GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
read 2:   AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

```
               GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

Overlap graph assembly

```
read 1:   GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
read 2:   AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

```
                GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

Overlap graph assembly

```
read 1:   GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
read 2:   AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

```
                GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

overlap = 17

Overlap graph assembly

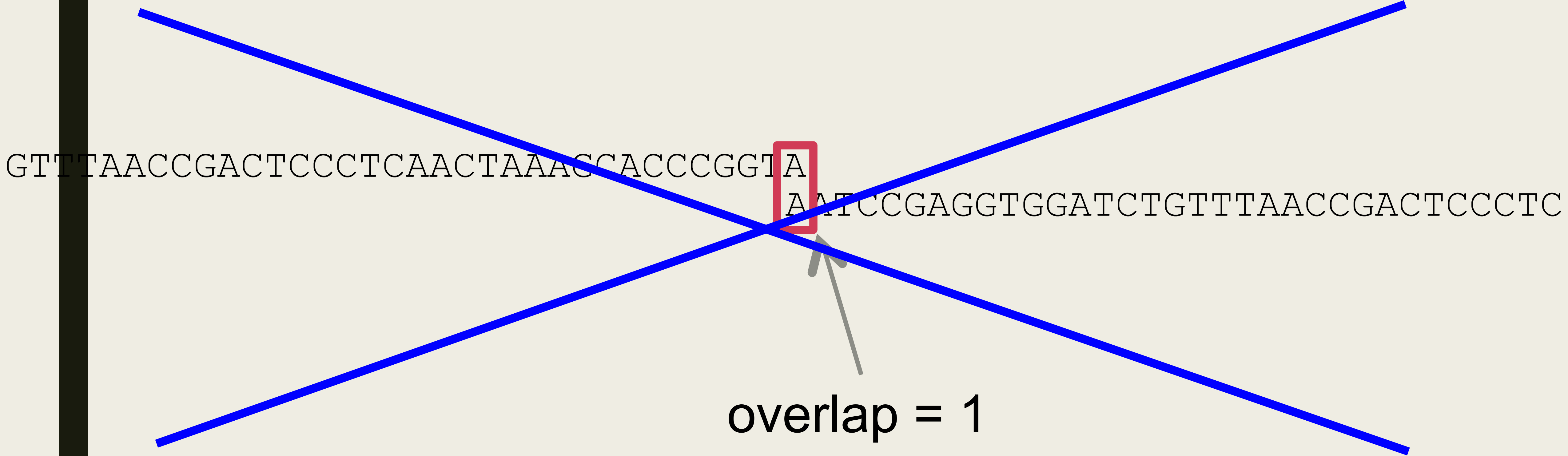
```
read 1:    GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
read 2:    AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

```
GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
                                 AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

overlap = 1

Overlap graph assembly

```
read 1:   GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
read 2:   AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```



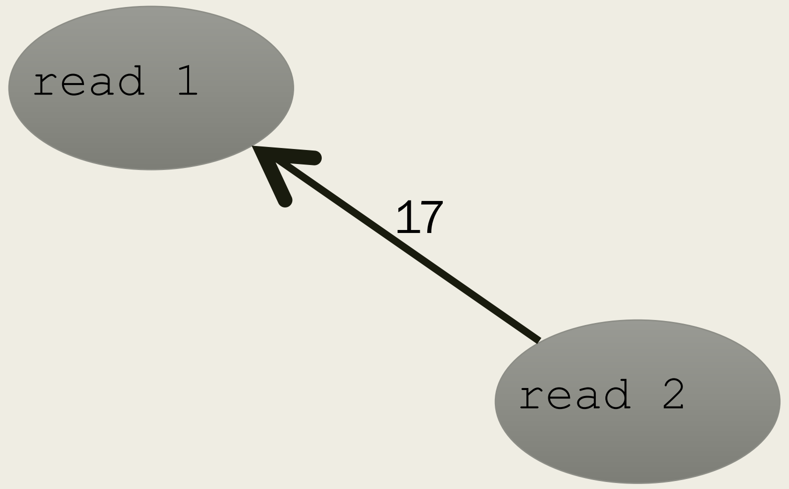
```
GTTTAACCGACTCCCTCAACTAAAGCACCCGGTA
                                 AATCCGAGGTGGATCTGTTTAACCGACTCCCTC
```

Takeaways

- Overlaps should meet some minimum threshold *T* (often 1/3 of the read length)

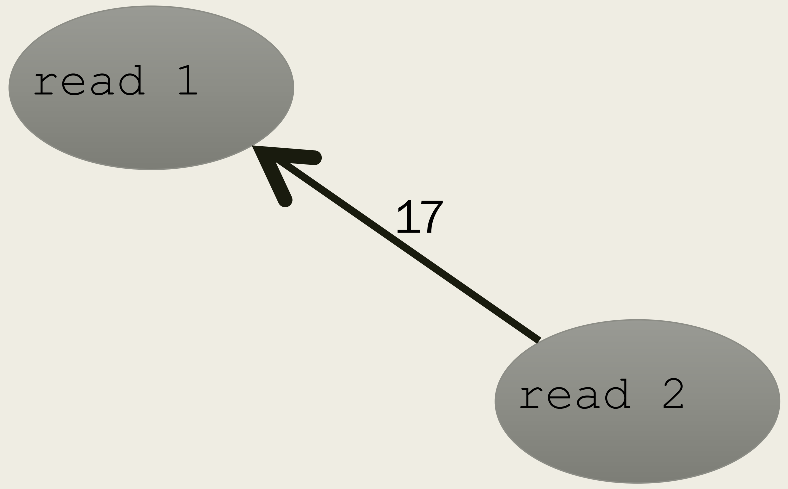
- Overlaps should have a maximum number of errors allows (roughly 2-3 depending on the error rate and overlap threshold)

Overlap graph (directed, why?)



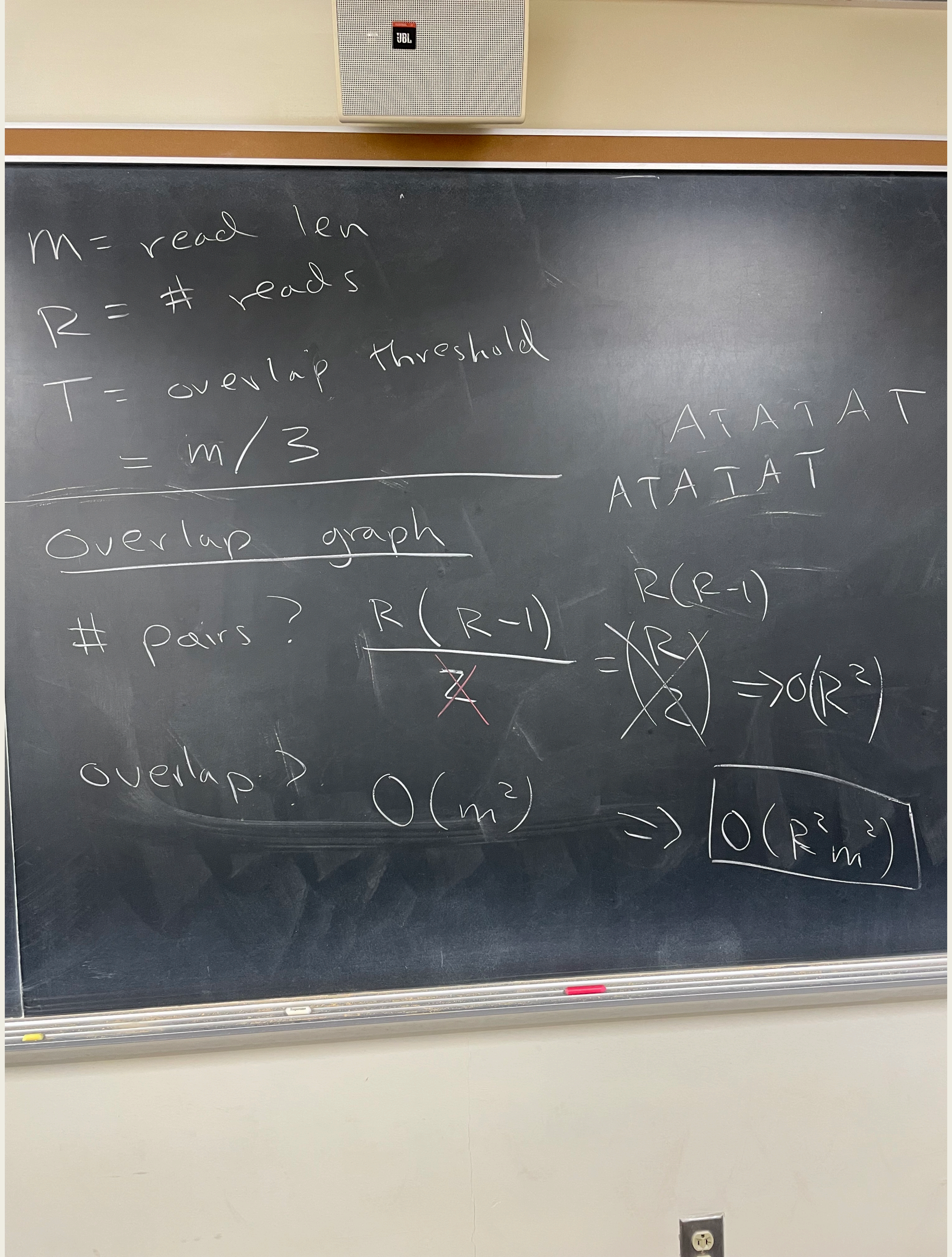
What is the runtime for creating the overlap graph?

Overlap graph (directed, why?)



What is the runtime for creating the overlap graph?

O(R^2) pairs, O(m^2) for each pair, => really slow



m = read len

R = # reads

T = overlap threshold

= m/3

ATATAT

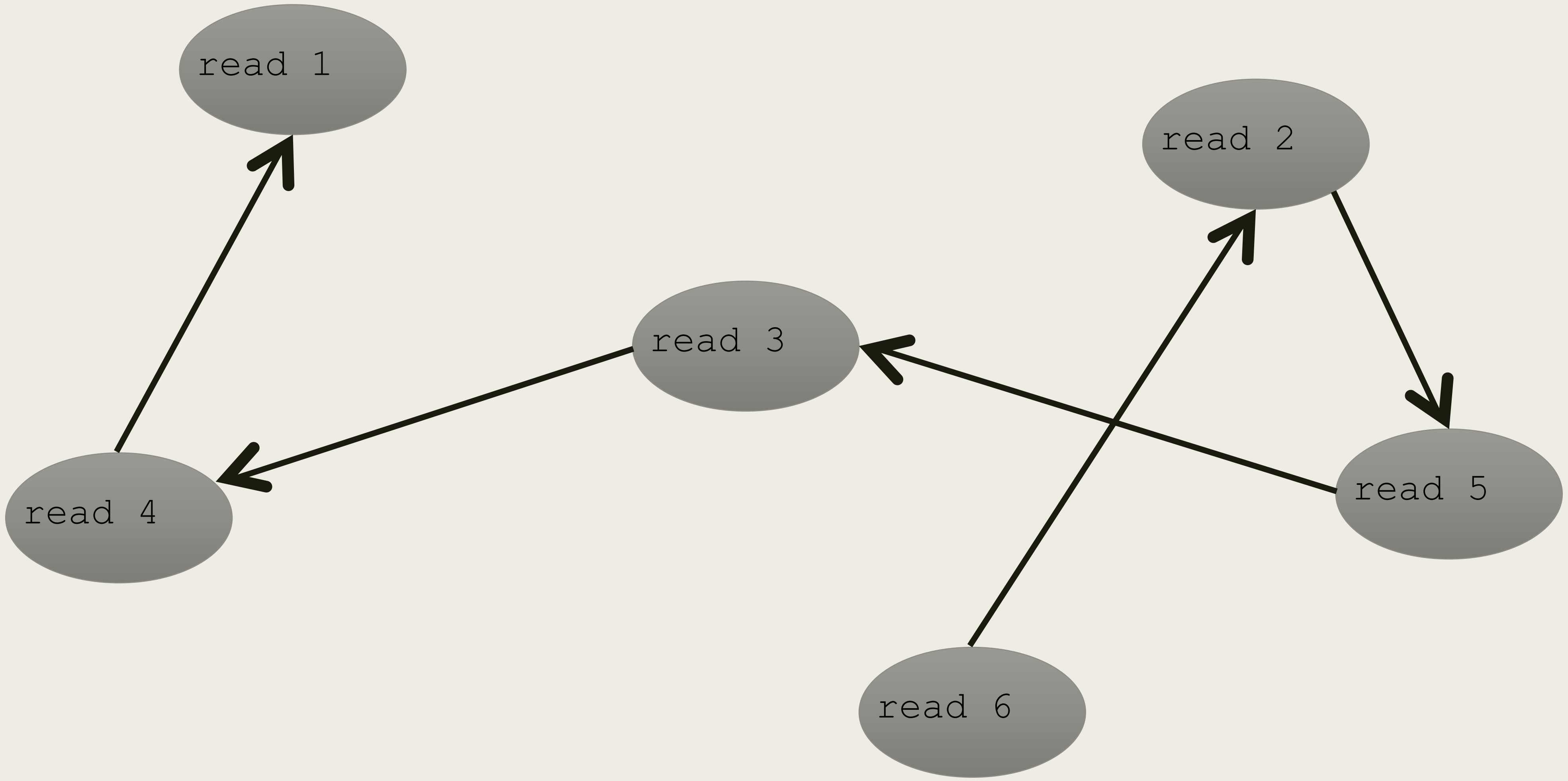
ATATAT

Overlap graph

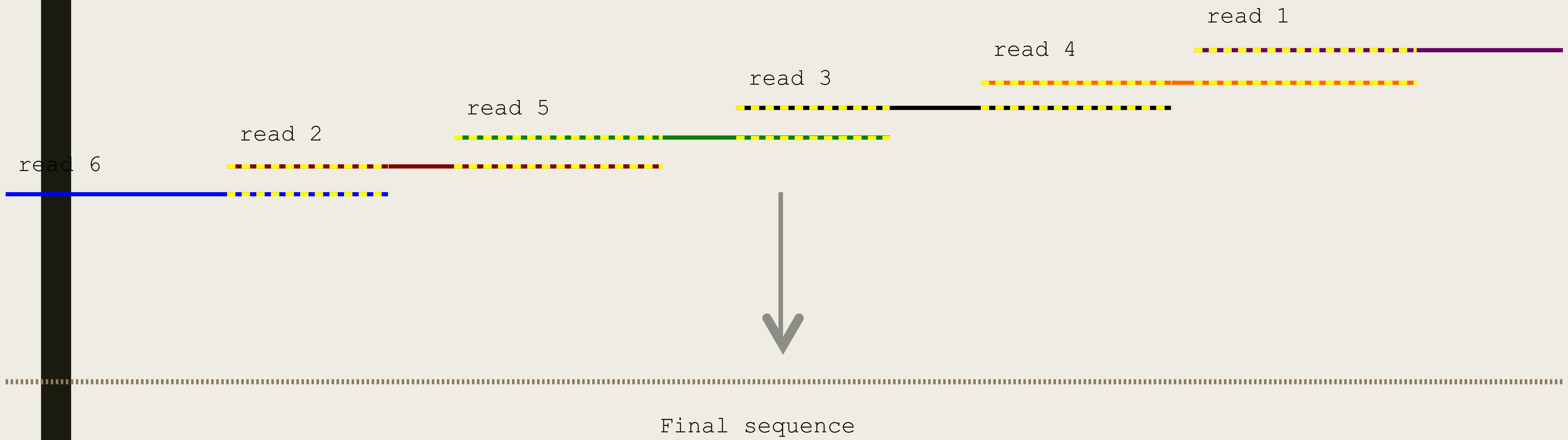
pairs? $\frac{R(R-1)}{\cancel{2}} = \cancel{\binom{R}{2}}$ $R(R-1)$ $\Rightarrow O(R^2)$

overlap? $O(m^2)$ $\Rightarrow \boxed{O(R^2 m^2)}$

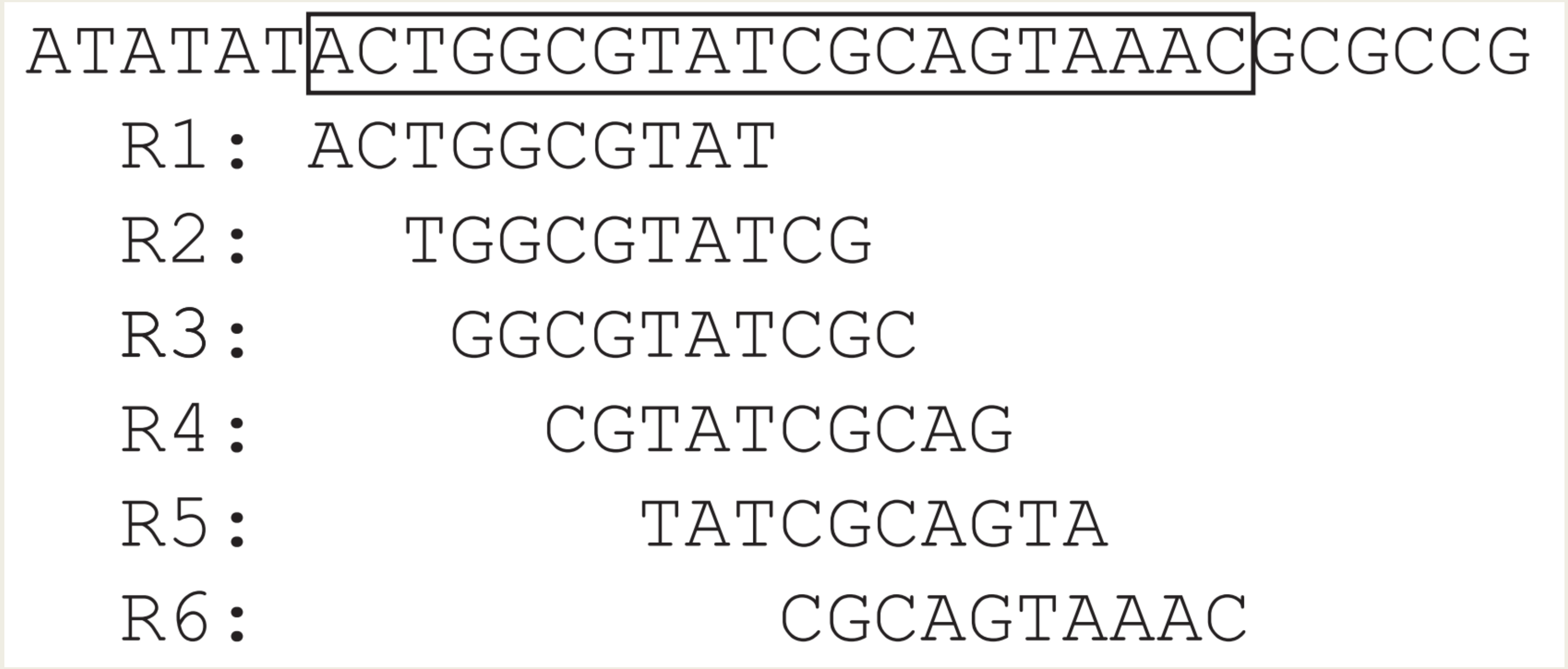
Overlap graph



Perfect graph traversal

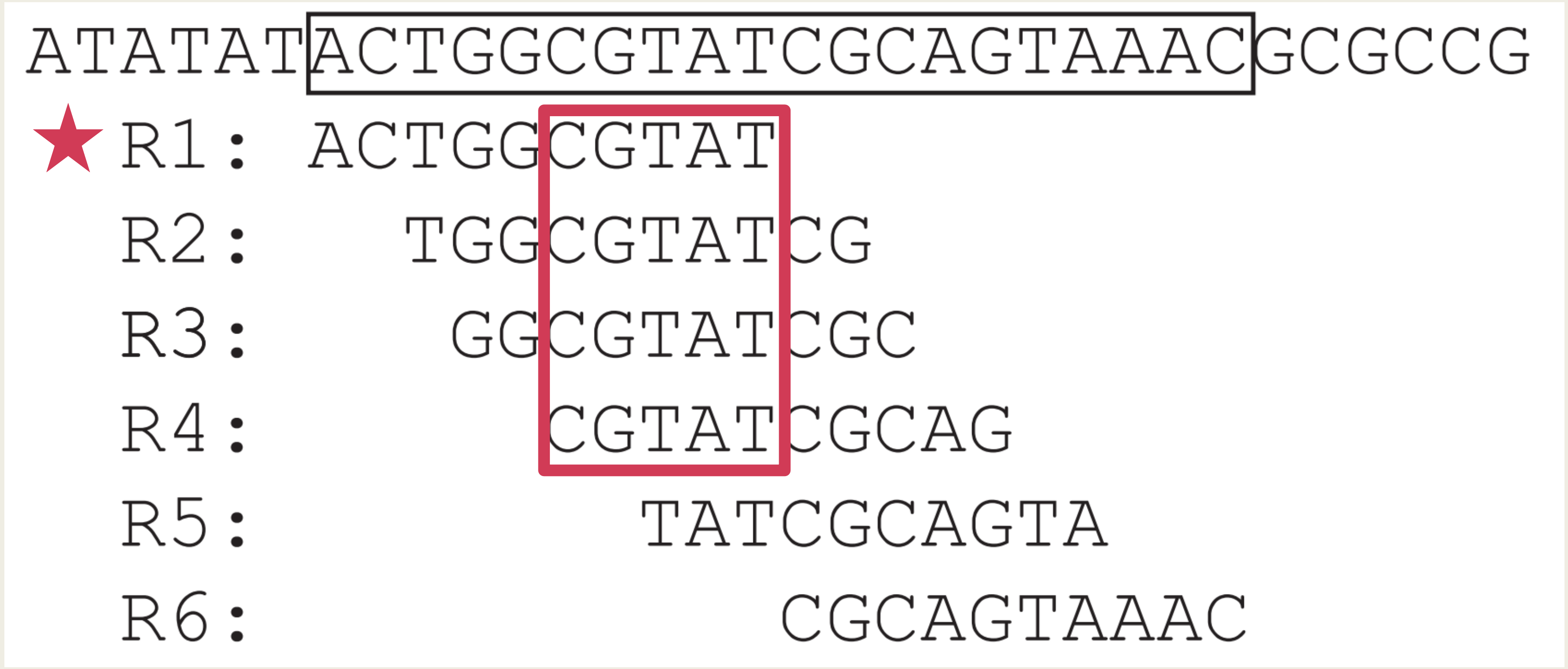


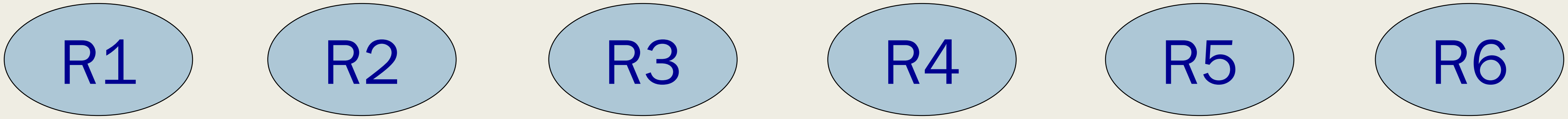
Activity example: *m* = 10, *T* = 5



```
ATATATACTGGCGTATCGCAGTAAACGCGCCG
  R1: ACTGGCGTAT
  R2:   TGGCGTATCG
  R3:    GGCGTATCGC
  R4:      CGTATCGCAG
  R5:        TATCGCAGTA
  R6:           CGCAGTAAAC
```

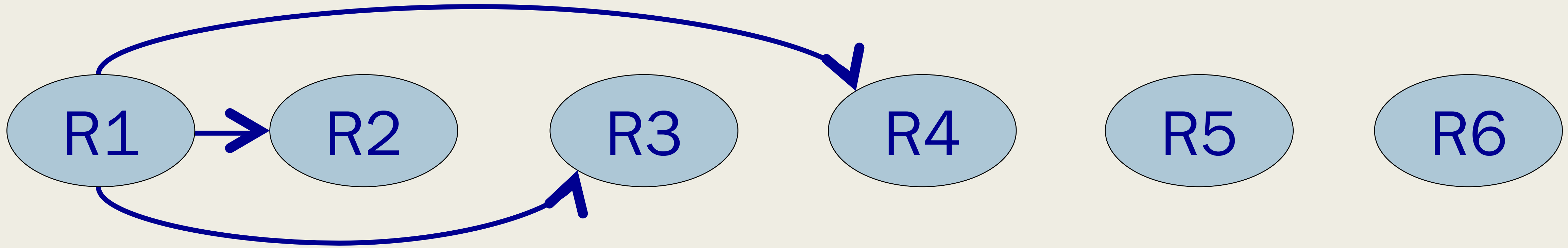
Activity example: m = 10, T = 5





Activity example: m = 10, T = 5

```
ATATATACTGGCGTATCGCAGTAAACGCGCCG
★ R1: ACTGGCGTAT
  R2:   TGGCGTATCG
  R3:    GGCGTATCGC
  R4:      CGTATCGCAG
  R5:        TATCGCAGTA
  R6:           CGCAGTAAAC
```


$$\begin{tikzcd}
R1 \arrow[r] \arrow[rr, bend right] \arrow[rrr, bend left] & R2 & R3 & R4 & R5 & R6
\end{tikzcd}$$

Activity example: m = 10, T = 5

```
ATATATACTGGCGTATCGCAGTAAACGCGCCG
  R1: ACTGGCGTAT
★ R2:   TGGCGTATCG
  R3:    GGCGTATCGC
  R4:      CGTATCGCAG
  R5:        TATCGCAGTA
  R6:           CGCAGTAAAC
```

$$\begin{tikzcd}
R1 \arrow[r] \arrow[rr, bend right] \arrow[rrr, bend left] & R2 & R3 & R4 & R5 & R6
\end{tikzcd}$$

Activity example: m = 10, T = 5

```
ATATATACTGGCGTATCGCAGTAAACGCGCCG
  R1: ACTGGCGTAT
★ R2:   TGGCGTATCG
  R3:    GGCGTATCGC
  R4:      CGTATCGCAG
  R5:        TATCGCAGTA
  R6:           CGCAGTAAAC
```

$$\begin{tikzcd}
R1 \arrow[r] \arrow[rrr, bend left] \arrow[rr, bend right] & R2 \arrow[r] \arrow[rrr, bend left] \arrow[rr, bend right] & R3 & R4 & R5 & R6
\end{tikzcd}$$

Activity example: *m* = 10, *T* = 5

```
ATATATACTGGCGTATCGCAGTAAACGCGCCG
  R1: ACTGGCGTAT
  R2:   TGGCGTATCG
★ R3:    GGCGTATCGC
  R4:      CGTATCGCAG
  R5:        TATCGCAGTA
  R6:           CGCAGTAAAC
```

$$\begin{tikzcd}
R1 \arrow[r] \arrow[rrr, bend left] \arrow[rr, bend right] & R2 \arrow[r] \arrow[rrr, bend left] \arrow[rr, bend right] & R3 & R4 & R5 & R6
\end{tikzcd}$$

Li et al, *Briefings in Functional Genomics* (2012)

Activity example: m = 10, T = 5

```
ATATATACTGGCGTATCGCAGTAAACGCGCCG
  R1: ACTGGCGTAT
  R2:   TGGCGTATCG
★ R3:    GGCGTATCGC
  R4:      CGTATCGCAG
  R5:        TATCGCAGTA
  R6:           CGCAGTAAAC
```

$$\begin{tikzcd}
R1 \arrow[r] \arrow[rr, bend right] \arrow[rrr, bend left] & R2 \arrow[r] \arrow[rr, bend right] \arrow[rrr, bend left] & R3 \arrow[r] \arrow[rr, bend right] & R4 & R5 & R6
\end{tikzcd}$$

Activity example: m = 10, T = 5

```
ATATATACTGGCGTATCGCAGTAAACGCGCCG
  R1: ACTGGCGTAT
  R2:   TGGCGTATCG
  R3:    GGCGTATCGC
★ R4:      CGTATCGCAG
  R5:        TATCGCAGTA
  R6:           CGCAGTAAAC
```

$$\begin{tikzcd}
R1 \arrow[r] \arrow[rrr, bend left] \arrow[rr, bend right] & R2 \arrow[r] \arrow[rrr, bend left] \arrow[rr, bend right] & R3 \arrow[r] \arrow[rr, bend right] & R4 & R5 & R6
\end{tikzcd}$$

Activity example: m = 10, T = 5

```
ATATATACTGGCGTATCGCAGTAAACGCGCCG
  R1: ACTGGCGTAT
  R2:   TGGCGTATCG
  R3:    GGCGTATCGC
 ★R4:      CGTATCGCAG
  R5:        TATCGCAGTA
  R6:           CGCAGTAAAC
```

$$\begin{tikzcd}
R1 \arrow[r] \arrow[rrr, bend left] \arrow[rr, bend right] & R2 \arrow[r] \arrow[rrr, bend left] \arrow[rr, bend right] & R3 \arrow[r] \arrow[rr, bend right] & R4 \arrow[r] \arrow[rr, bend right] & R5 & R6
\end{tikzcd}$$

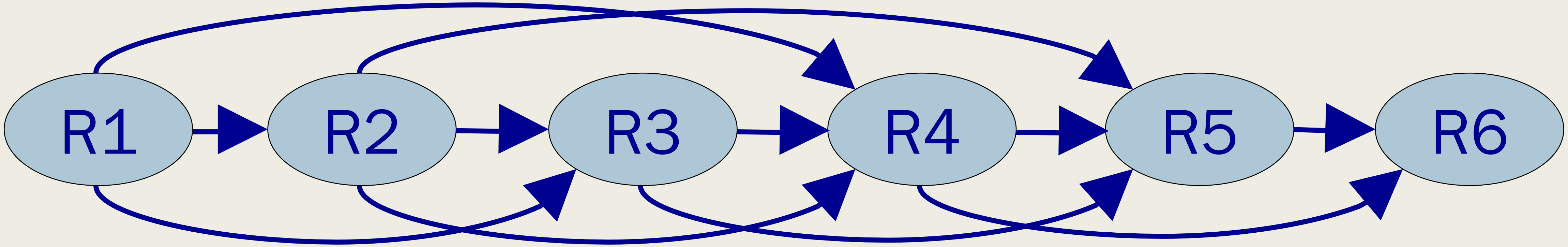
Activity example: m = 10, T = 5

```
ATATATACTGGCGTATCGCAGTAAACGCGCCG
  R1: ACTGGCGTAT
  R2:   TGGCGTATCG
  R3:    GGCGTATCGC
  R4:      CGTATCGCAG
★ R5:        TATCGCAGTA
  R6:           CGCAGTAAAC
```

$$\begin{tikzcd}
R1 \arrow[r] \arrow[rr, bend right] \arrow[rrr, bend left] & R2 \arrow[r] \arrow[rr, bend right] \arrow[rrr, bend left] & R3 \arrow[r] \arrow[rr, bend right] & R4 \arrow[r] \arrow[rr, bend right] & R5 & R6
\end{tikzcd}$$

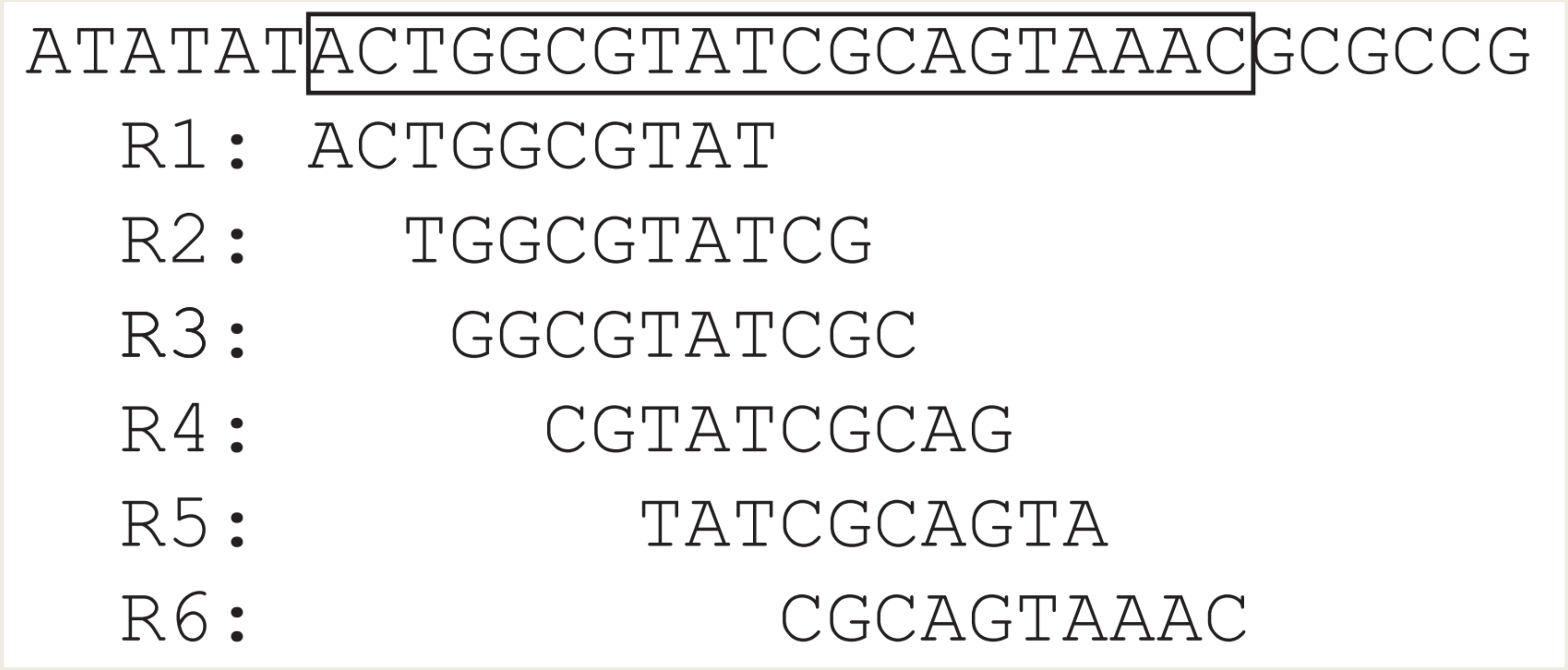
Activity example: m = 10, T = 5

```
ATATATACTGGCGTATCGCAGTAAACGCGCCG
  R1: ACTGGCGTAT
  R2:   TGGCGTATCG
  R3:    GGCGTATCGC
  R4:      CGTATCGCAG
★ R5:        TATCGCAGTA
  R6:           CGCAGTAAAC
```

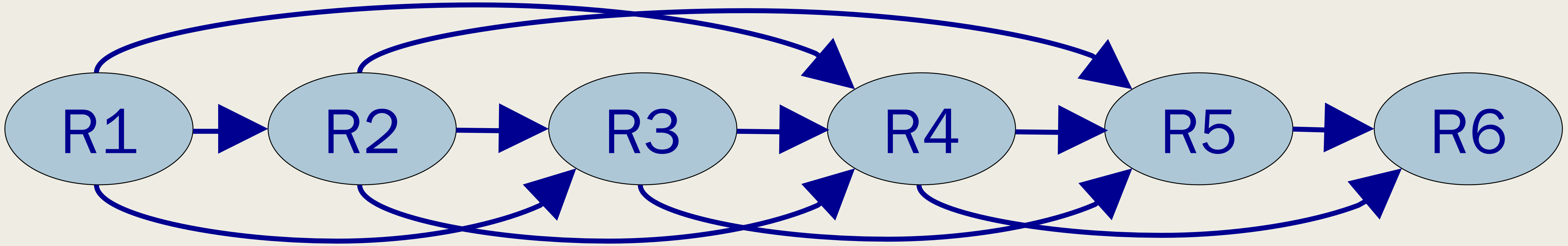


$$\begin{tikzcd}
R1 \arrow[r] \arrow[rr, bend right] \arrow[rrr, bend left] & R2 \arrow[r] \arrow[rr, bend right] \arrow[rrr, bend left] & R3 \arrow[r] \arrow[rr, bend right] & R4 \arrow[r] \arrow[rr, bend right] & R5 \arrow[r] & R6
\end{tikzcd}$$

Activity example: *m* = 10, *T* = 5



```
ATATATACTGGCGTATCGCAGTAAACGCGCCG
  R1: ACTGGCGTAT
  R2:   TGGCGTATCG
  R3:    GGCGTATCGC
  R4:      CGTATCGCAG
  R5:        TATCGCAGTA
  R6:           CGCAGTAAAC
```



$$\begin{tikzcd}
R1 \arrow[r] \arrow[rr, bend right] \arrow[rrr, bend left] & R2 \arrow[r] \arrow[rr, bend right] \arrow[rrr, bend left] & R3 \arrow[r] \arrow[rr, bend right] & R4 \arrow[r] \arrow[rr, bend right] & R5 \arrow[r] & R6
\end{tikzcd}$$

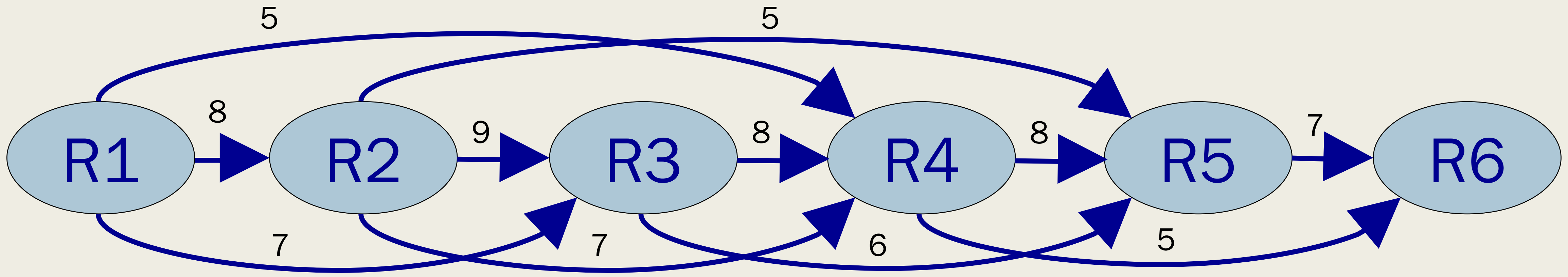
Steps of Overlap Graph Assembly (also called "overlap-layout-consensus")

1) Compute overlaps between all pairs of reads. With R = number of reads and m = length of reads, this is naively $O(R^2m^2)$. We will learn better ways of "aligning" sequences next week.

2) Construct a graph with reads as the nodes and directed, weighted edges between reads with >= T overlap.

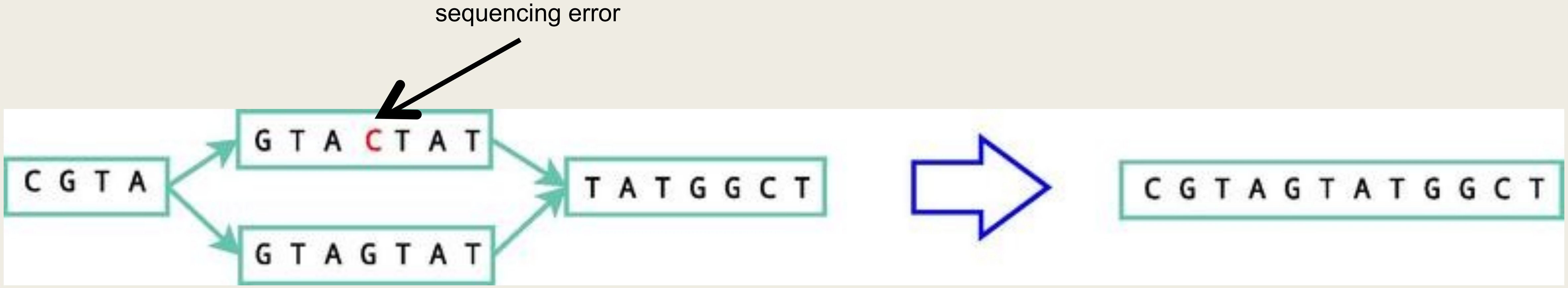
Activity example: m = 10, T = 5

```
ATATATACTGGCGTATCGCAGTAAACGCGCCG
  R1: ACTGGCGTAT
  R2:   TGGCGTATCG
  R3:    GGCGTATCGC
  R4:      CGTATCGCAG
  R5:        TATCGCAGTA
  R6:           CGCAGTAAAC
```



Issues with overlap graphs

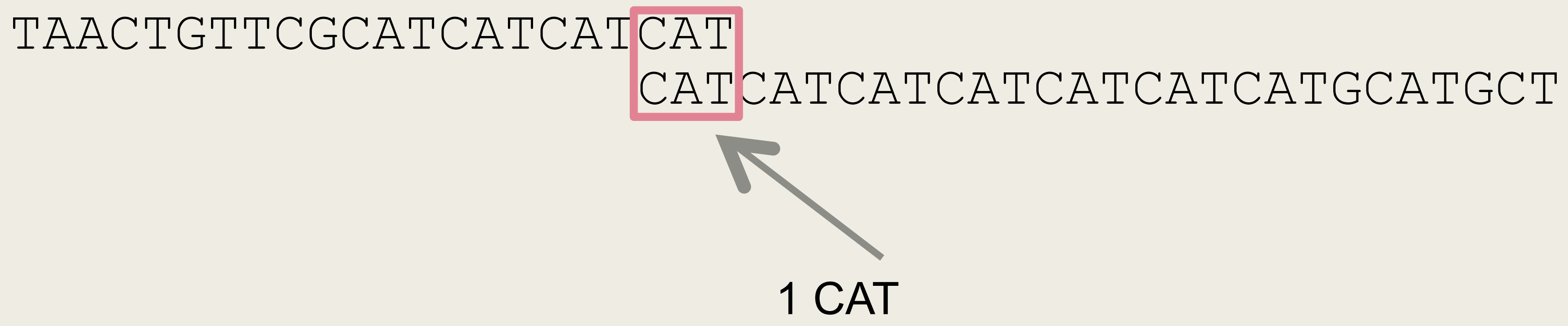
Bubbles



Velvet Assembler, Wikipedia

Repeats

```
read 1:   TAACTGTTCGCATCATCATCAT
read 2:   CATCATCATCATCATCATCATGCATGCT
```



```
TAACTGTTCGCATCATCATCAT
                   CATCATCATCATCATCATCATGCATGCT
```

Repeats

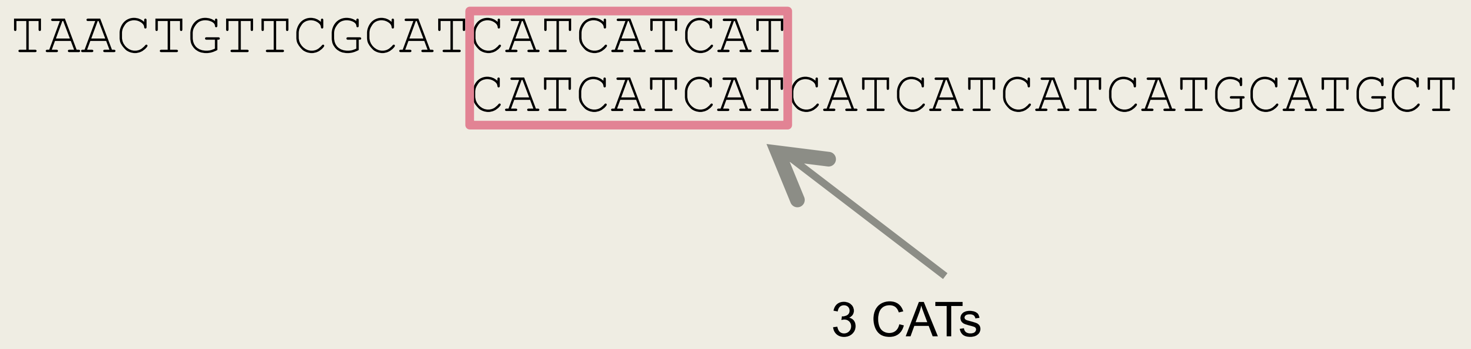
```
read 1:   TAACTGTTCGCATCATCATCAT
read 2:   CATCATCATCATCATCATCATGCATGCT
```

```
TAACTGTTCGCATCATCATCAT
                CATCATCATCATCATCATCATGCATGCT
```

2 CATs

Repeats

```
read 1:    TAACTGTTCGCATCATCATCAT
read 2:    CATCATCATCATCATCATCATGCATGCT
```



```
TAACTGTTCGCATCATCATCAT
             CATCATCATCATCATCATCATGCATGCT
```

Repeats

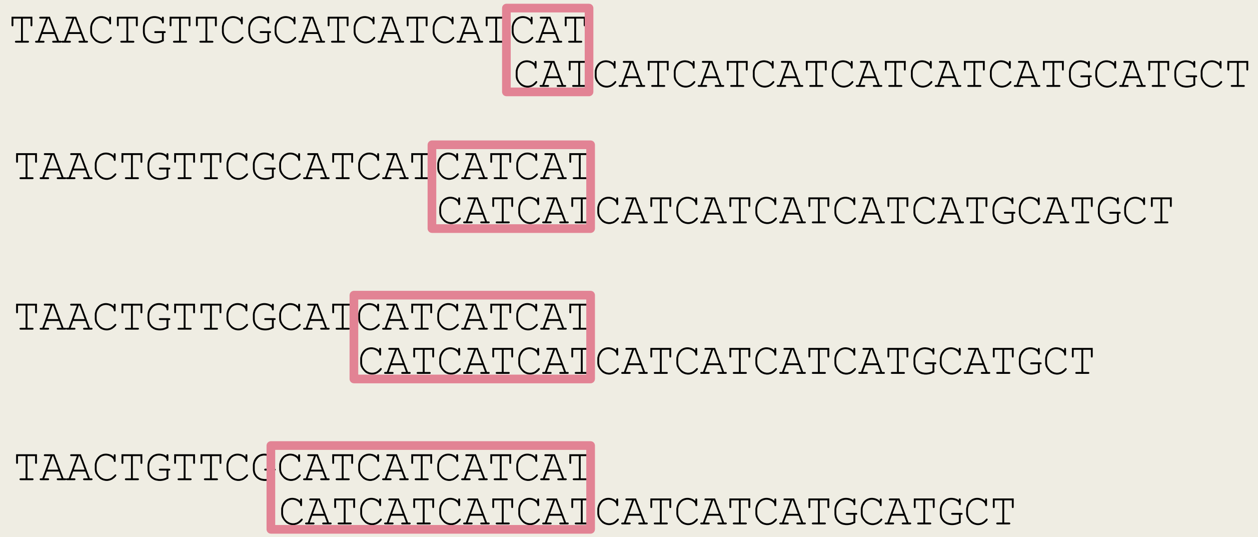
```
read 1:   TAACTGTTCGCATCATCATCAT
read 2:   CATCATCATCATCATCATCATGCATGCT
```

```
TAACTGTTCGCATCATCATCAT
          CATCATCATCATCATCATCATGCATGCT
```

4 CATs

Repeats

```
read 1:   TAACTGTTCGCATCATCATCAT
read 2:   CATCATCATCATCATCATCATGCATGCT
```

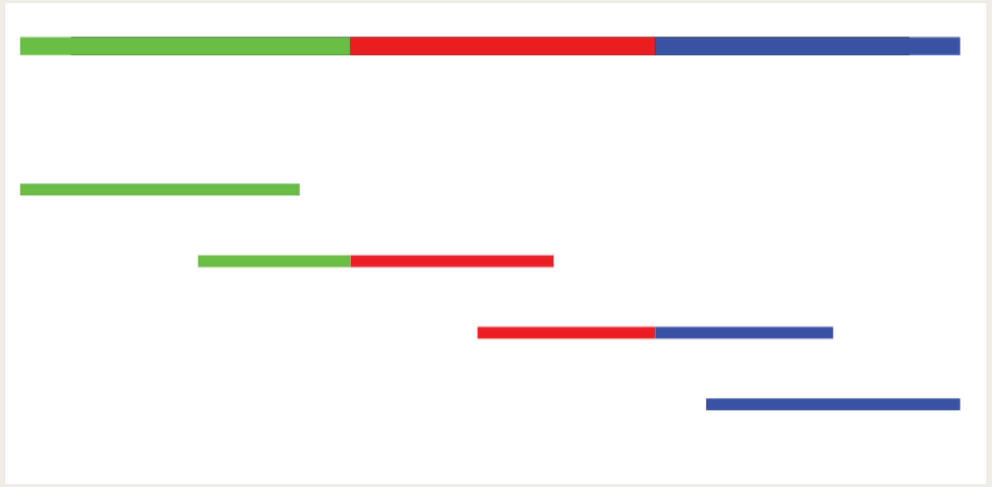


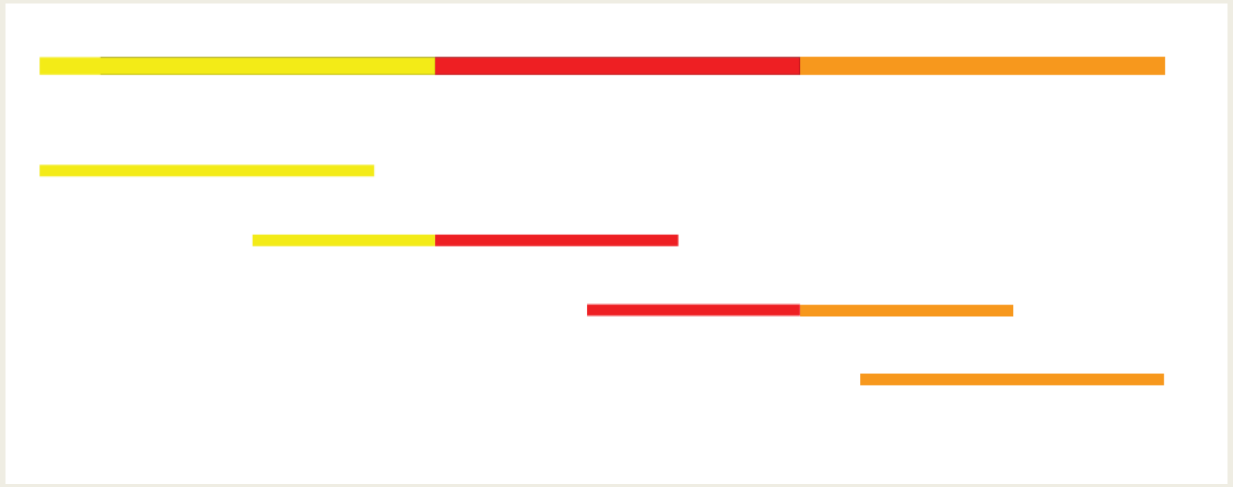
```
TAACTGTTCGCATCATCATCAT
                   CATCATCATCATCATCATCATGCATGCT

TAACTGTTCGCATCATCATCAT
                CATCATCATCATCATCATCATGCATGCT

TAACTGTTCGCATCATCATCAT
             CATCATCATCATCATCATCATGCATGCT

TAACTGTTCGCATCATCATCAT
          CATCATCATCATCATCATCATGCATGCT
```

Repeats are a major issue for assembly

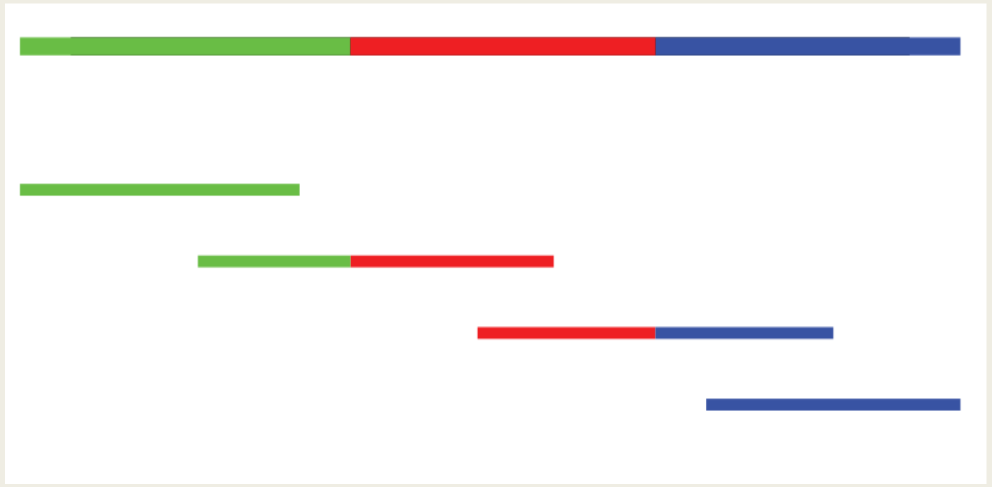
- This is the major limitation to assembling genomes.
- 40-60% of the human genome is repetitive sequence of one kind or another
- Some genomes are much higher – e.g. some pine trees >80-90%
- Some important sequences, e.g. telomeres/centromeres are almost entirely repetitive
- Long reads help to some extent and much of the work in this area is based around new technologies for sequencing longer and longer reads (e.g. 10's or 100's of kb).

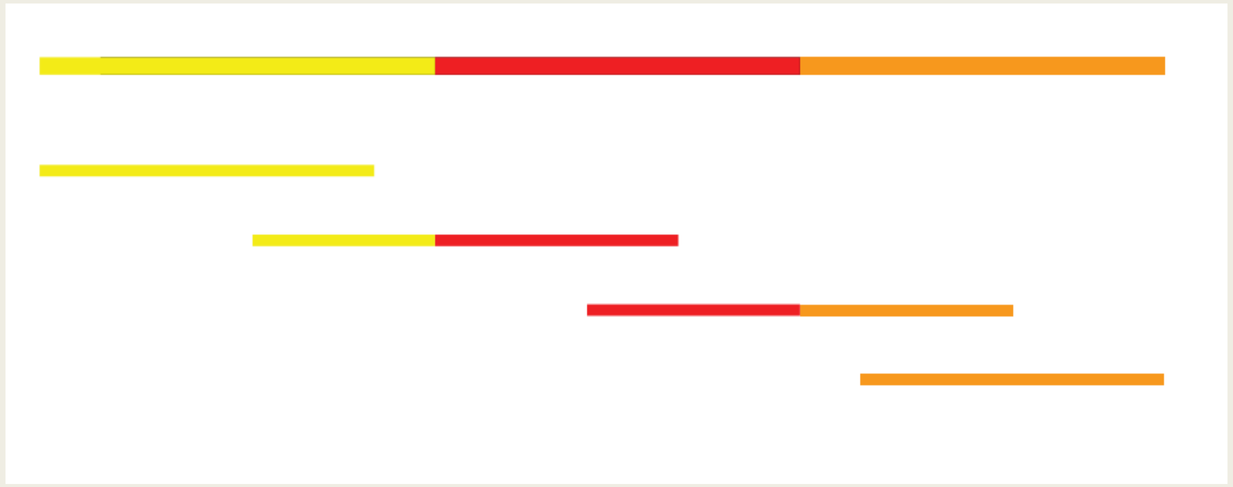
What would the graph look like for these reads?

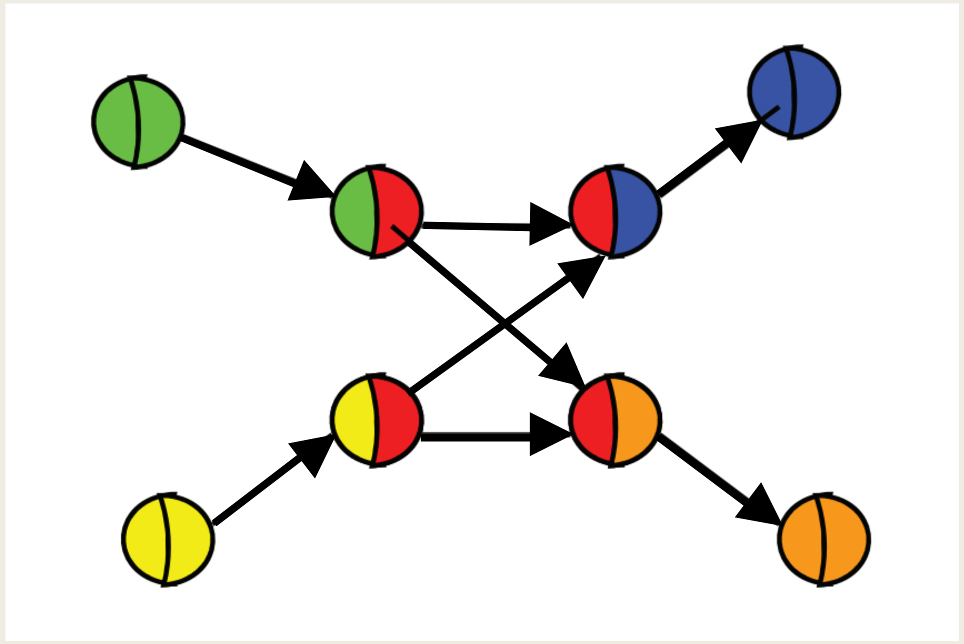




What would the graph look like for these reads?







Li et al, *Briefings in Functional Genomics* (2012)

Back to overlap graph algorithm

Steps of Overlap Graph Assembly (also called "overlap-layout-consensus")

1) Compute overlaps between all pairs of reads. With R = number of reads and m = length of reads, this is naively $O(R^2m^2)$. We will learn better ways of "aligning" sequences next week.

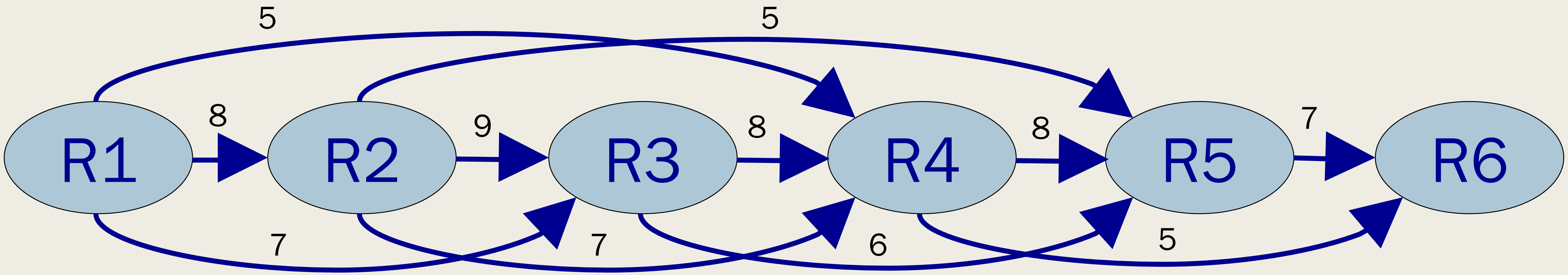
2) Construct a graph with reads as the nodes and directed, weighted edges between reads with >= T overlap.

3) "Layout" the graph and try to "group" stretches of the graph into "contigs" (short for contiguous), these are (hopefully) long portions of the original genome

4) Find a "consensus" *sequence* for each contig

Activity example: m = 10, T = 5

```
ATATATACTGGCGTATCGCAGTAAACGCGCCG
  R1: ACTGGCGTAT
  R2:   TGGCGTATCG
  R3:    GGCGTATCGC
  R4:      CGTATCGCAG
  R5:        TATCGCAGTA
  R6:           CGCAGTAAAC
```



$$\begin{tikzcd}
R1 \arrow[r, "8"] \arrow[rrr, bend left, "5"] \arrow[rr, bend right, "7"'] & R2 \arrow[r, "9"] \arrow[rrr, bend left, "5"] \arrow[rr, bend right, "7"'] & R3 \arrow[r, "8"] \arrow[rr, bend right, "6"'] & R4 \arrow[r, "8"] \arrow[rr, bend right, "5"'] & R5 \arrow[r, "7"] & R6
\end{tikzcd}$$

Activity example: m = 10, T = 5

```
ATATATACTGGCGTATCGCAGTAAACGCGCCG
  R1: ACTGGCGTAT
  R2:   TGGCGTATCG
  R3:    GGCGTATCGC
  R4:      CGTATCGCAG
  R5:        TATCGCAGTA
  R6:           CGCAGTAAAC
```

First simplification: remove edges that can be (transitively) inferred from other edges

$$\begin{tikzcd} R1 \arrow[r, "8"] & R2 \arrow[r, "9"] & R3 \arrow[r, "8"] & R4 \arrow[r, "8"] & R5 \arrow[r, "7"] & R6 \end{tikzcd}$$

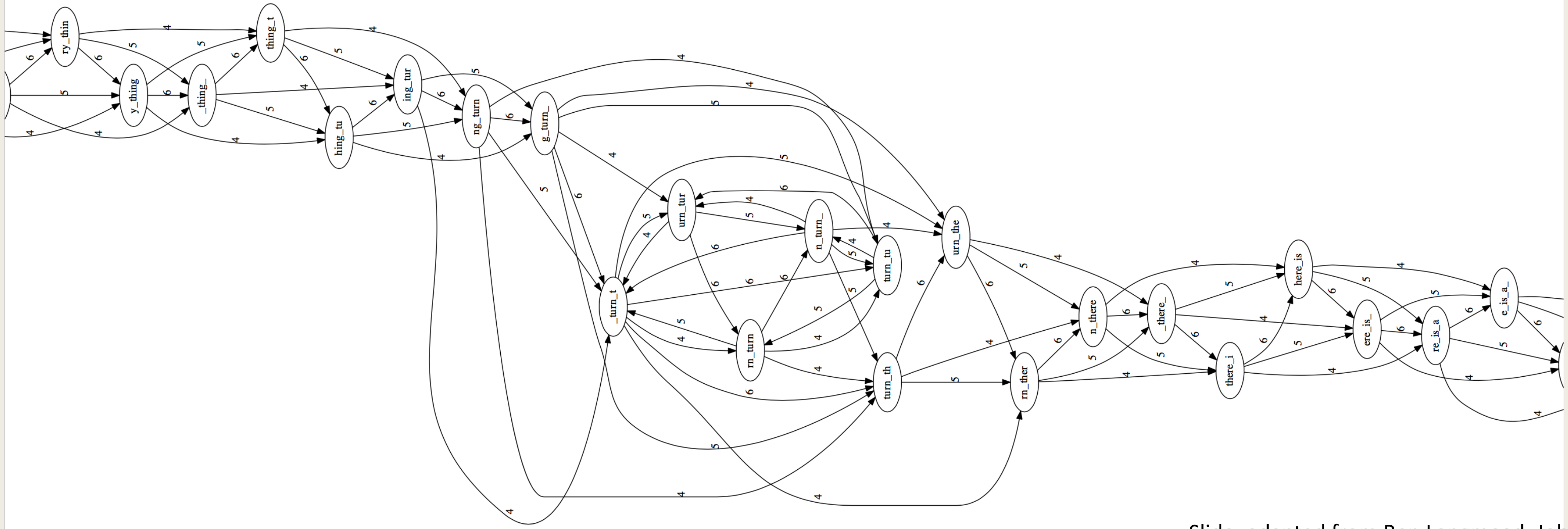
Layout

Overlap graph is big and messy. Contigs don't "pop out" at us.

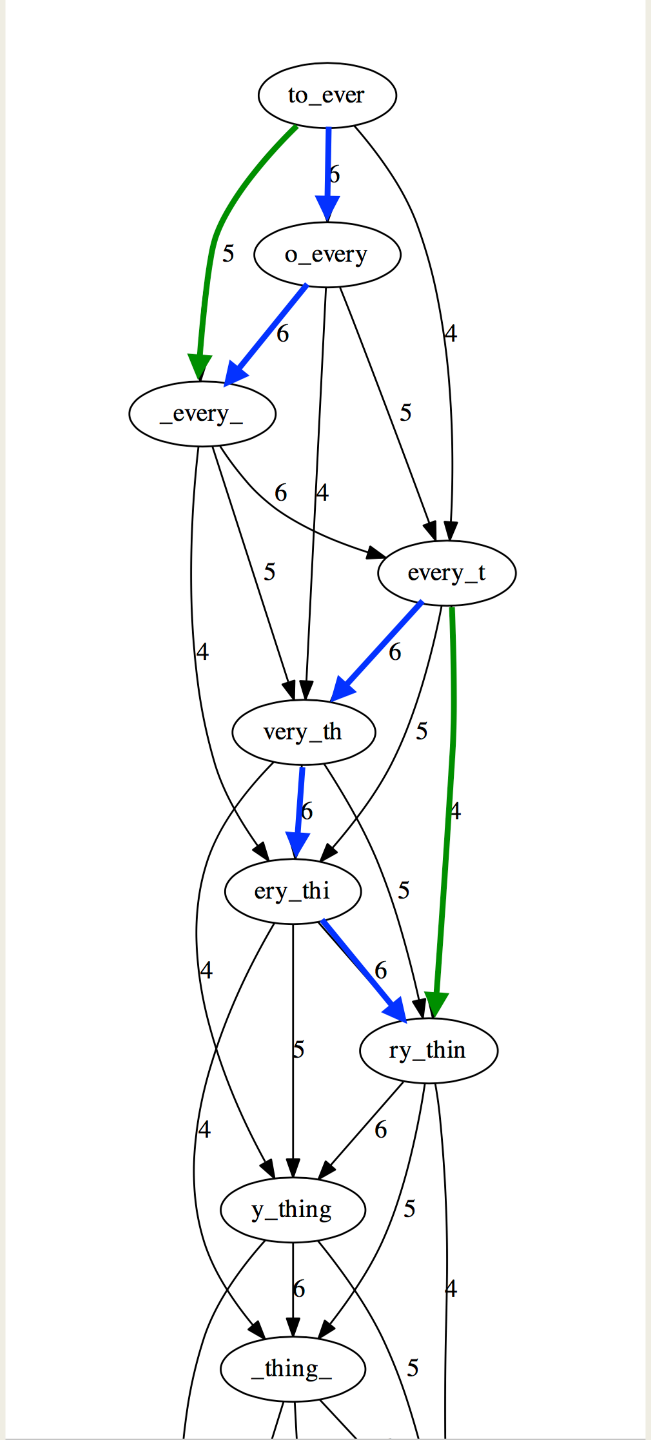
Below: part of the overlap graph for

`to_every_thing_turn_turn_turn_there_is_a_season`

$m = 7, T = 4$



Slide: adapted from Ben Langmead, John Hopkins

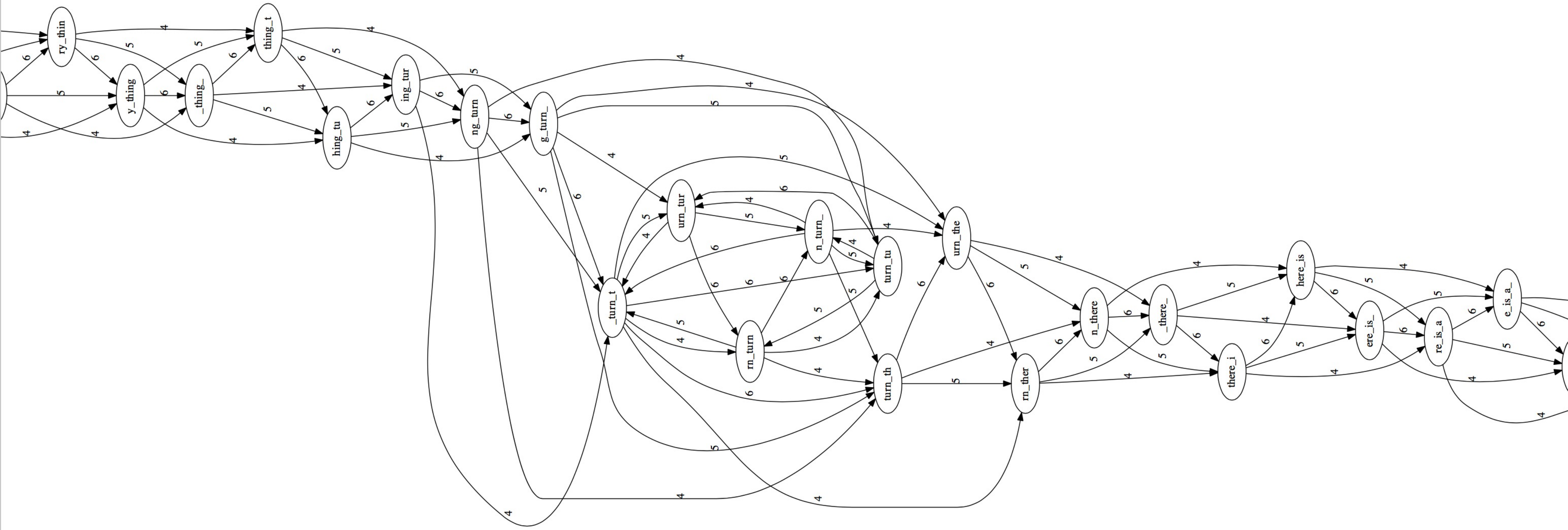


In this example: green edges can be inferred from blue

Slide: adapted from Ben Langmead, John Hopkins

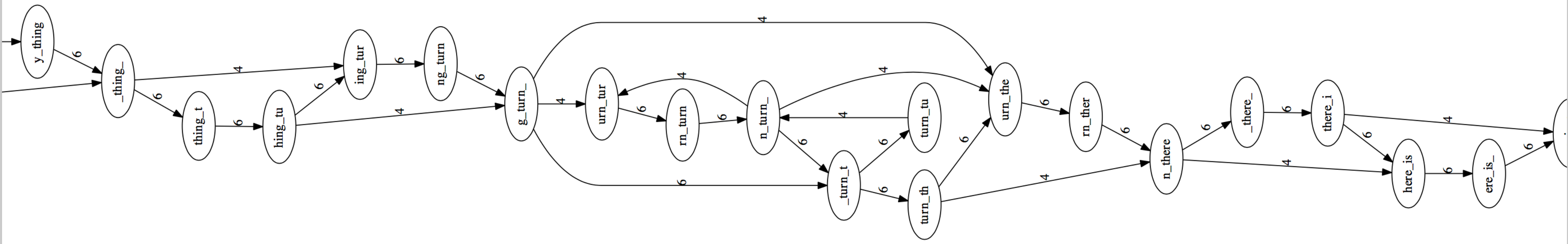
Layout: remove transitively-inferable edges

Before:



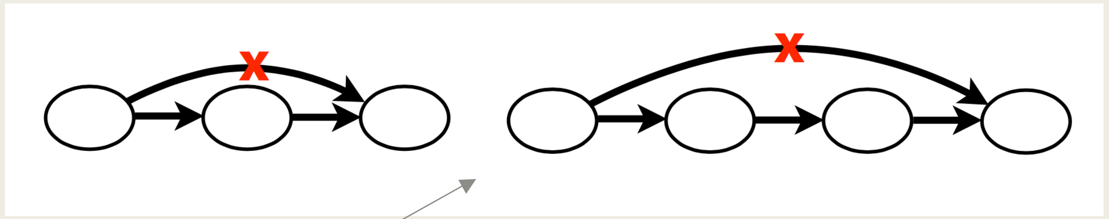
Layout: remove transitively-inferable edges

After removing edges that skip one node

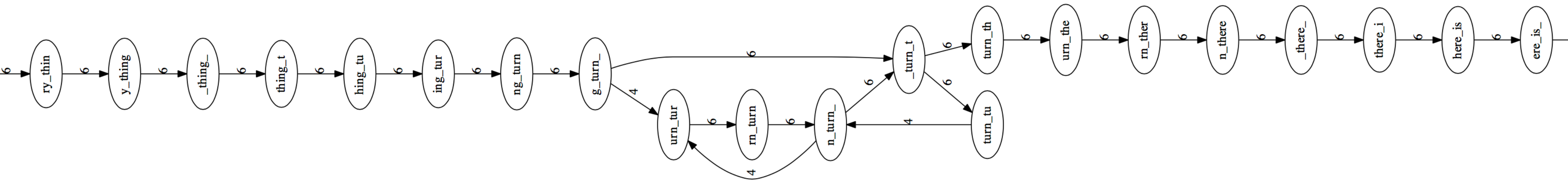


Slide: adapted from Ben Langmead, John Hopkins

Layout: remove transitively-inferable edges



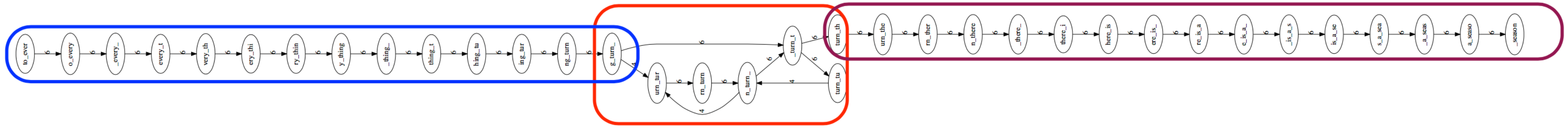
After removing edges that skip one or two nodes



Slide: adapted from Ben Langmead, John Hopkins

Layout

Emit *contigs* corresponding to the non-branching stretches



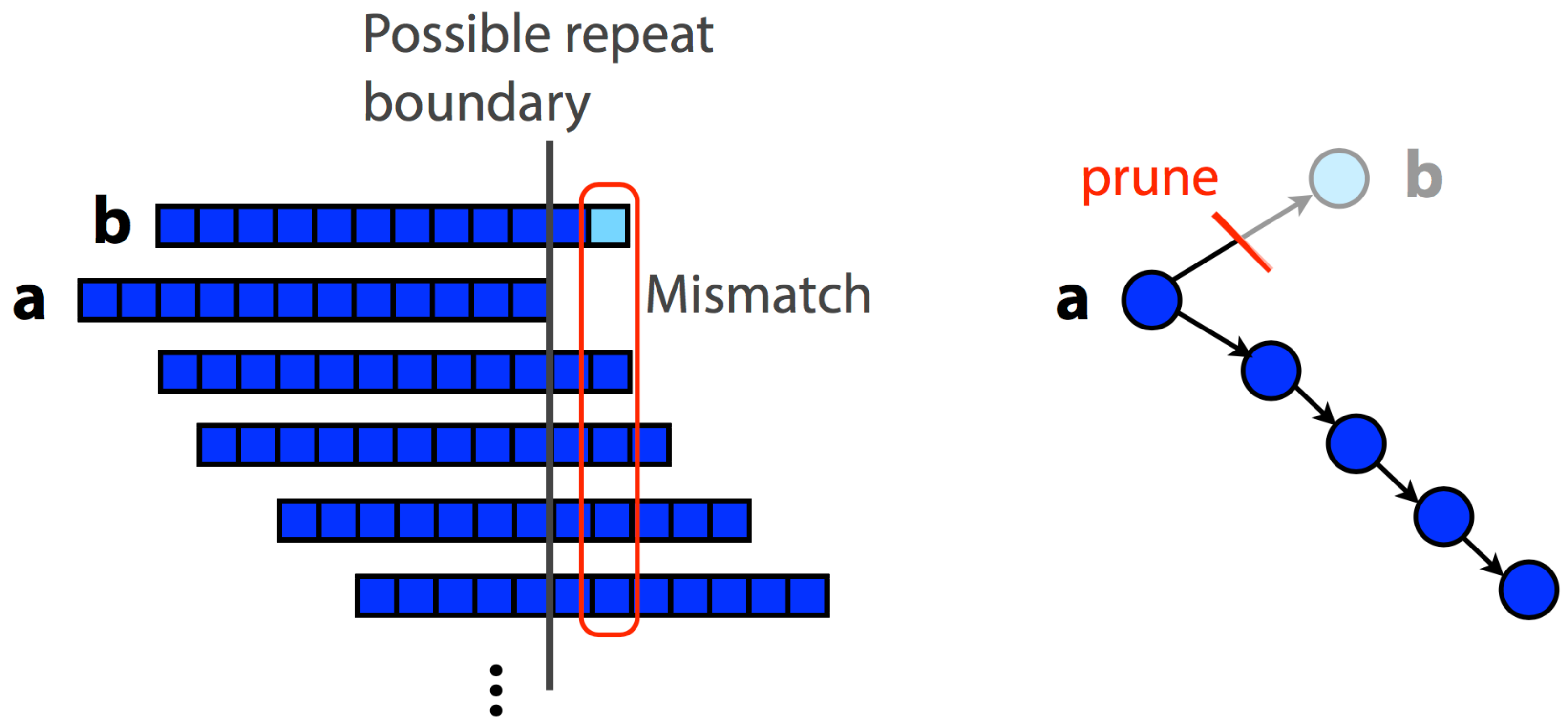
Contig 1
`to_every_thing_turn_`

Contig 2
`turn_there_is_a_season`

Unresolvable repeat

Original string: "`to_every_thing_turn_turn_turn_there_is_a_season`"

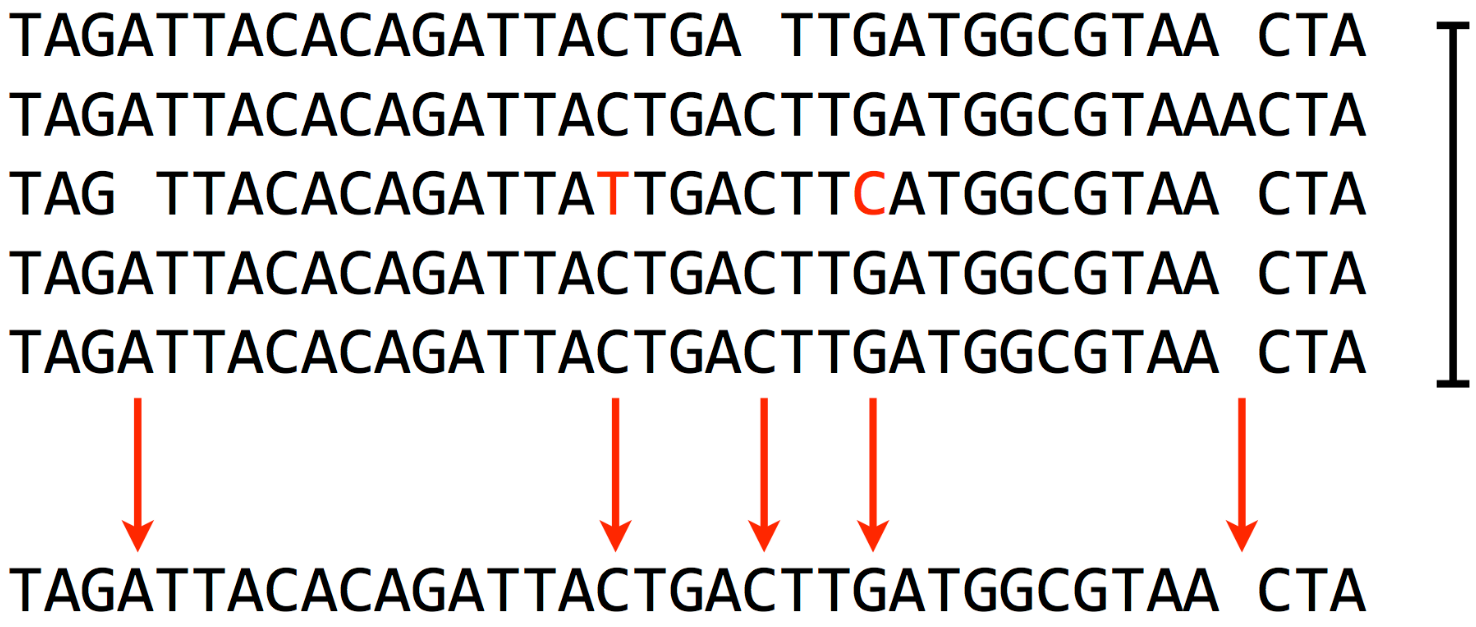
In practice, layout step also has to deal with spurious subgraphs, e.g. because of sequencing error



Mismatch could be due to sequencing error or repeat. Since the path through **b** ends abruptly we might conclude it's an error and prune **b**.

Slide: adapted from Ben Langmead, John Hopkins

Consensus



```
TAGATTACACAGATTACTGA TTGATGGCGTAA CTA
TAGATTACACAGATTACTGACTTGATGGCGTAACTA
TAG TTACACAGATTATTGACTTCATGGCGTAA CTA
TAGATTACACAGATTACTGACTTGATGGCGTAA CTA
TAGATTACACAGATTACTGACTTGATGGCGTAA CTA

TAGATTACACAGATTACTGACTTGATGGCGTAA CTA
```

Take reads that make up a contig and line them up

Take *consensus*, i.e. majority vote

Issues with overlap graph assembly

- Next-generation sequencing produces 100’s of millions (or even billions) of reads
- With one node per read this is computationally intractable for large genomes
- What if the nodes in our graph were not reads?