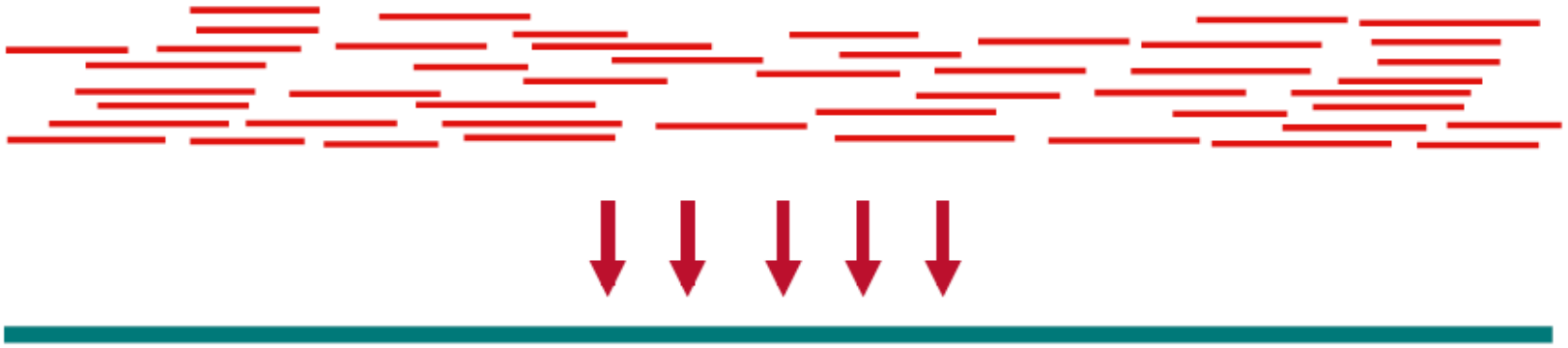
Read Mapping and Assembly

Gail Rosen

Read Mapping Leads to Assembly



```
                    CATCGACCGAGCGCGATGCTAGCTAGGTGATCGT......
                 TGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATCGT...
              GCATGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATCGT
           GTGCATGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATC
......AGGTGCATGCCGCATCGATCGAGCGCGATGCTAGCTAGCTGATCGT......
```

- Modern fast read aligners: BWT, Bowtie, SOAP
  - Based on ***Burrows-Wheeler transform***

Outline

- Traditional Alignment approaches
- Read Mapping
  - Hashing
  - BWT
- Assembly
  - Difficult for metagenomics
- Binning Contigs

Alignment

- Needleman-Wunsch
- Smith-Waterman

Uses Dynamic Programming Approaches

Needleman Wunsch

		G	C	C	C	T	A	G	C	G
	0	-2	-4	-6	-8	-10	-12	-14	-16	-18
G	-2	1	-1	-3	-5	-7	-9	-11	-13	-15
C	-4	-1	2	0	-2	-4	-6	-8	-10	-12
G	-6	-3	0	1	-1	-3	-5	-5	-7	-9
C	-8	-5	-2	1	2	0	-2	-4	-4	-6
A	-10	-7	-4	-1	0	1	1	-1	-3	-5
A	-12	-9	-6	-3	-2	-1	2	0	-2	-4
T	-14	-11	-8	-5	-4	-1	0	1	-1	-3
G	-16	-13	-10	-7	-6	-3	-2	1	0	0

Smith Waterman Alignment

		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0	1	0	0	0	0	0	1	0	1
C	0	0	2	1	1	0	0	0	2	0
G	0	1	0	1	0	0	0	1	0	3
C	0	0	2	1	2	0	0	0	2	1
A	0	0	0	1	0	1	1	0	0	1
A	0	0	0	0	0	0	2	0	0	0
T	0	0	0	0	0	1	0	1	0	0
G	0	1	0	0	0	0	0	1	0	1

Approaches to Short Read Alignment

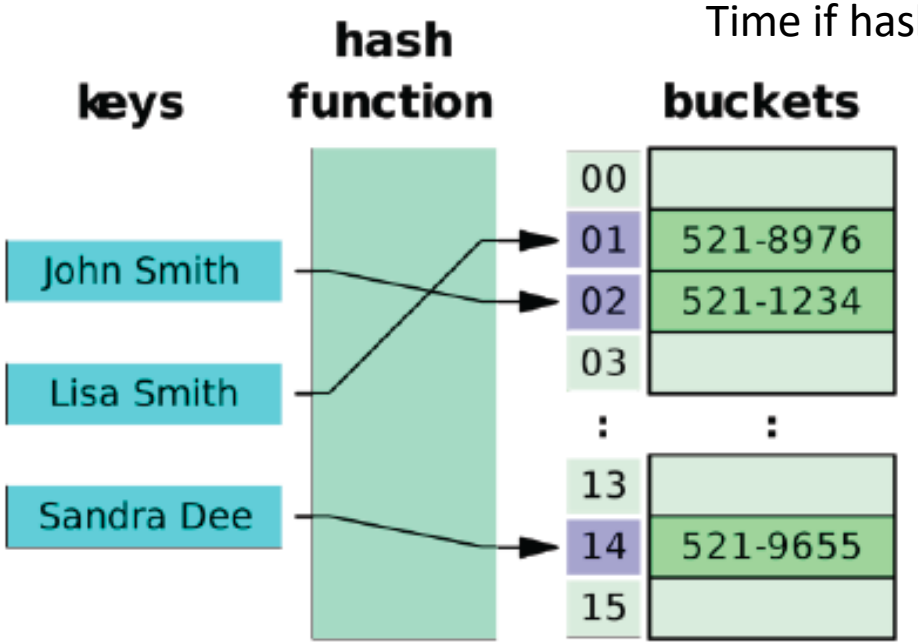
Two main approaches

- Hash-Based mapping:
  - Hashing of reads (E.g. Maq, Eland, SHRiMP)
  - Hashing of genome (E.g. novoalign, SHRiMP2)
- Indexing of tree-like structures:
  - Bowtie (ungapped)
  - Bowtie2 (gapped)
  - Bwa (gapped)
  - these all use Suffix Arrays/Burrows-Wheeler Transform (BWT), coupled with FM index

Hashing

- A hash function simply converts a string (“key”) to an integer (“value”).
- The integer is then used as an index in an array, for fast look up.

Search can
Still be O(nm)
Time if hashing not good



MAQ

- First widely used open source short read aligner
- Very fast, but at the cost of accuracy (ungapped)
- Uses hashing to index sequence reads, then scans with reference sequence
- Guaranteed to find alignments with up to 2 mismatches
- Can take advantage of paired end reads
- Uses quality scores to determine best alignments
- Generally no longer used, as has been superceded by newer aligners

http://sourceforge.net/projects/maq/

Bowtie

- Similar to MAQ, in that it uses quality scores to find best alignments.
- Uses "Burrows-Wheeler index" to keep its memory footprint small.
- Can find alignments with up to 3 mismatches in the first L bases of the read.
- Only ungapped alignments
- Also supports paired end reads.

  http://bowtie-bio.sourceforge.net/
- Bowtie2 supports gapped alignments too.

Burrows Wheeler Transform (BWT)

The BWT of a string S is a reversible permutation of S that enables the search for a pattern P in S to take linear time with respect to the length of P (O(|P|) time, independent of the length of S)

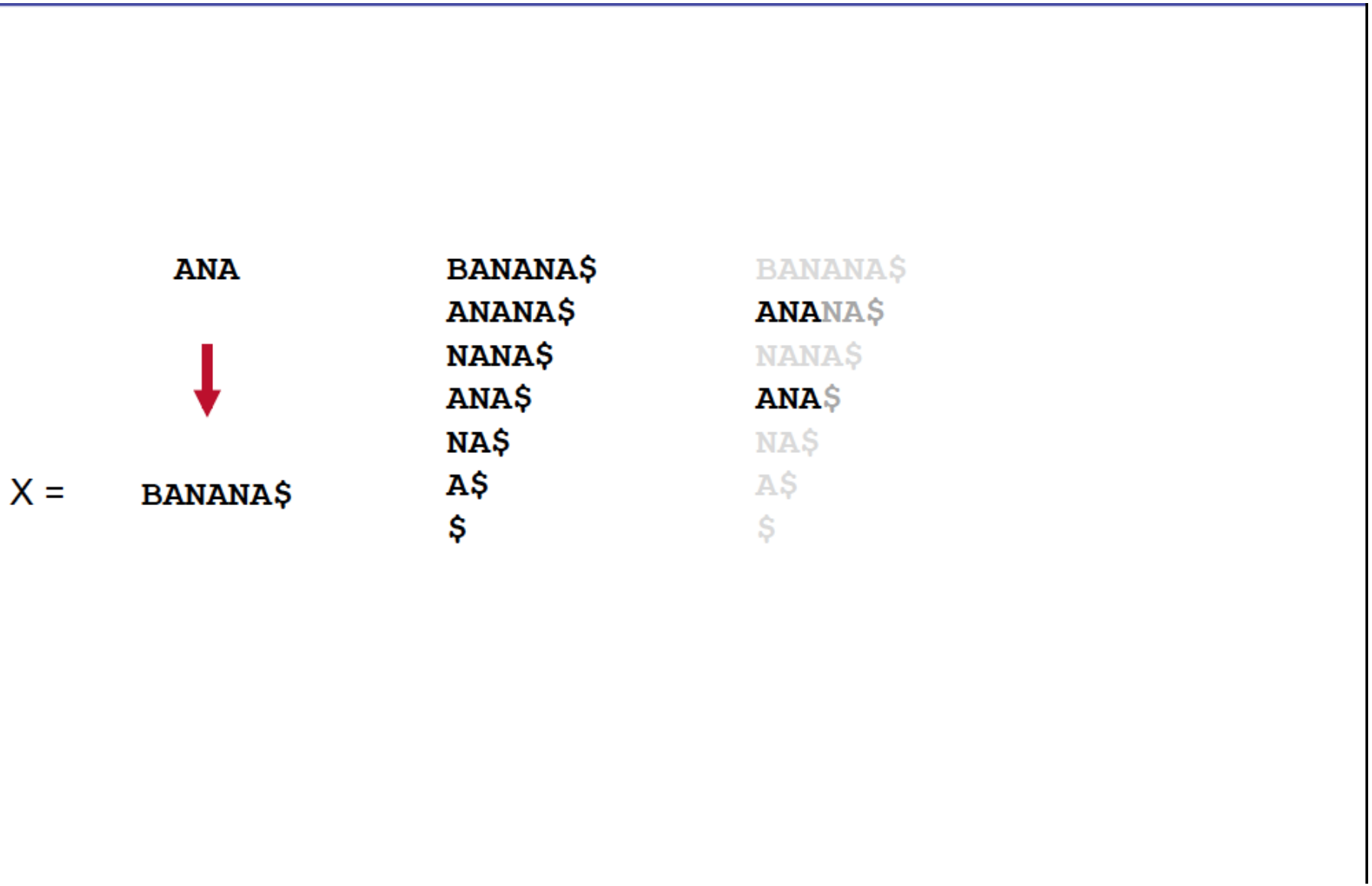
ANA

X = **BANANA**

BANANA
ANANA
NANA
ANA
NA
A

suffixes of
BANANA

BANANA
ANANA
NANA
ANA
NA
A

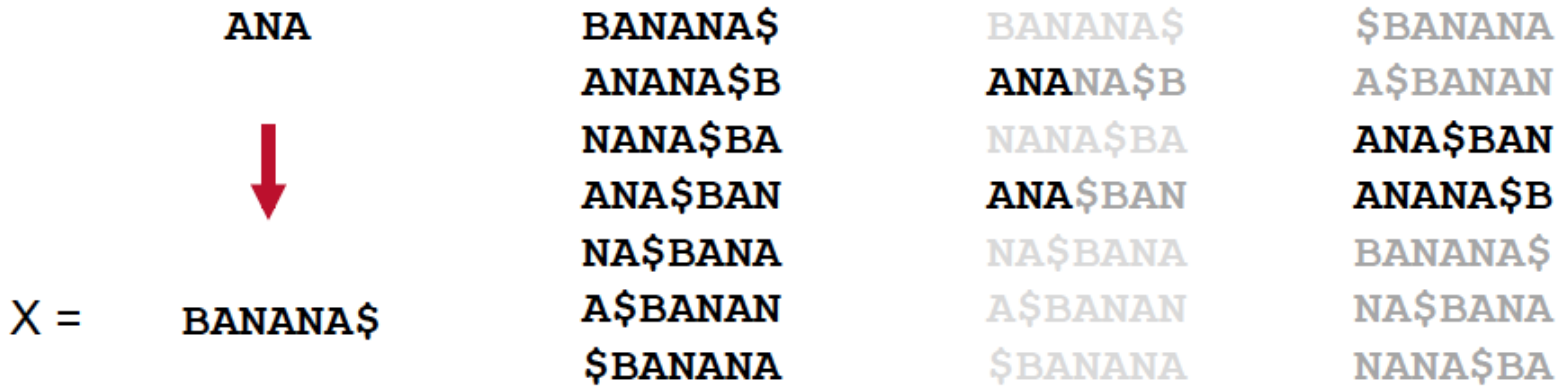

ANA
X = BANANA$
BANANA$
ANANA$
NANA$
ANA$
NA$
A$
$
BANANA$
ANANA$
NANA$
ANA$
NA$
A$
$

ANA

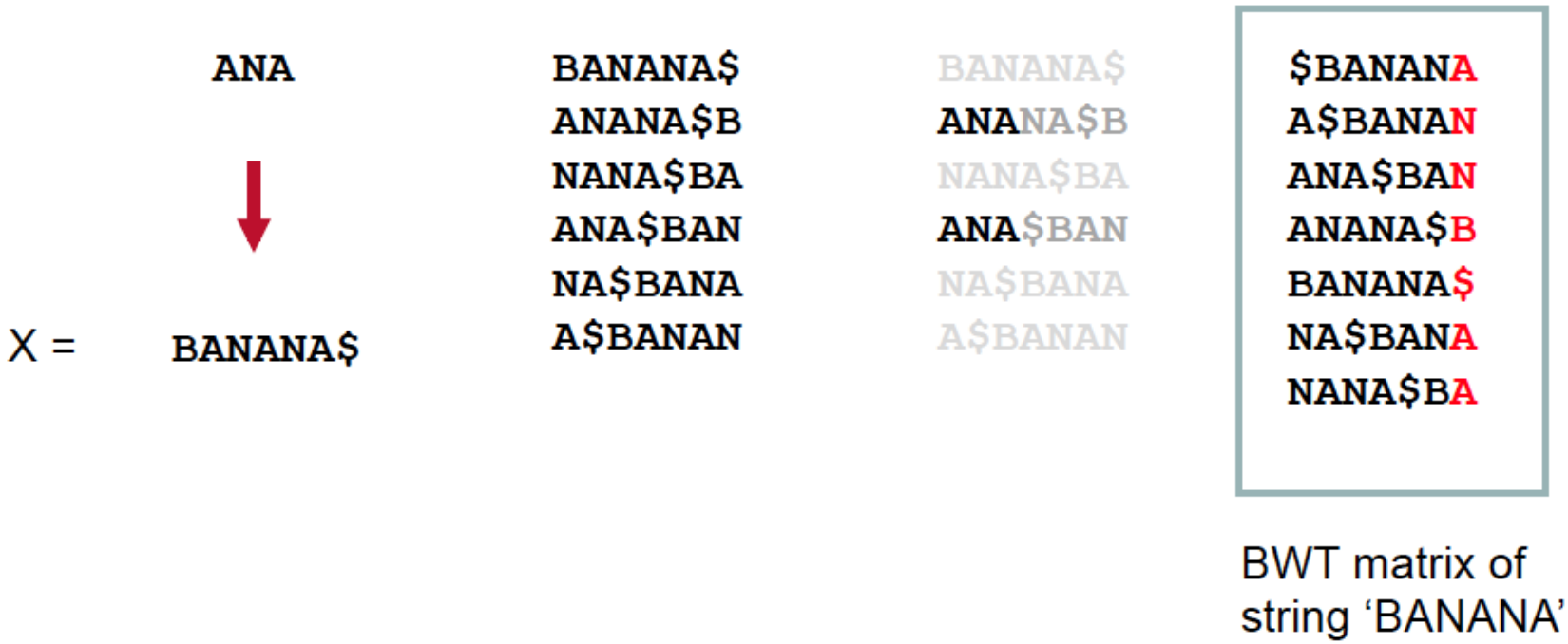
X = BANANA$

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

Suffixes sharing a common prefix are grouped together with a BWT



Tends to put runs of same character together (good for compression)



BWT matrix of string 'BANANA'

BWT(BANANA) = ANNB$AA

Suffix Array

$BANANA	1	$BANANA
A$BANAN	2	A$BANAN
ANA$BAN	3	ANA$BAN
ANANA$B	4	ANANA$B
BANANA$	5	BANANA$
NA$BANA	6	NA$BANA
NANA$BA	7	NANA$BA

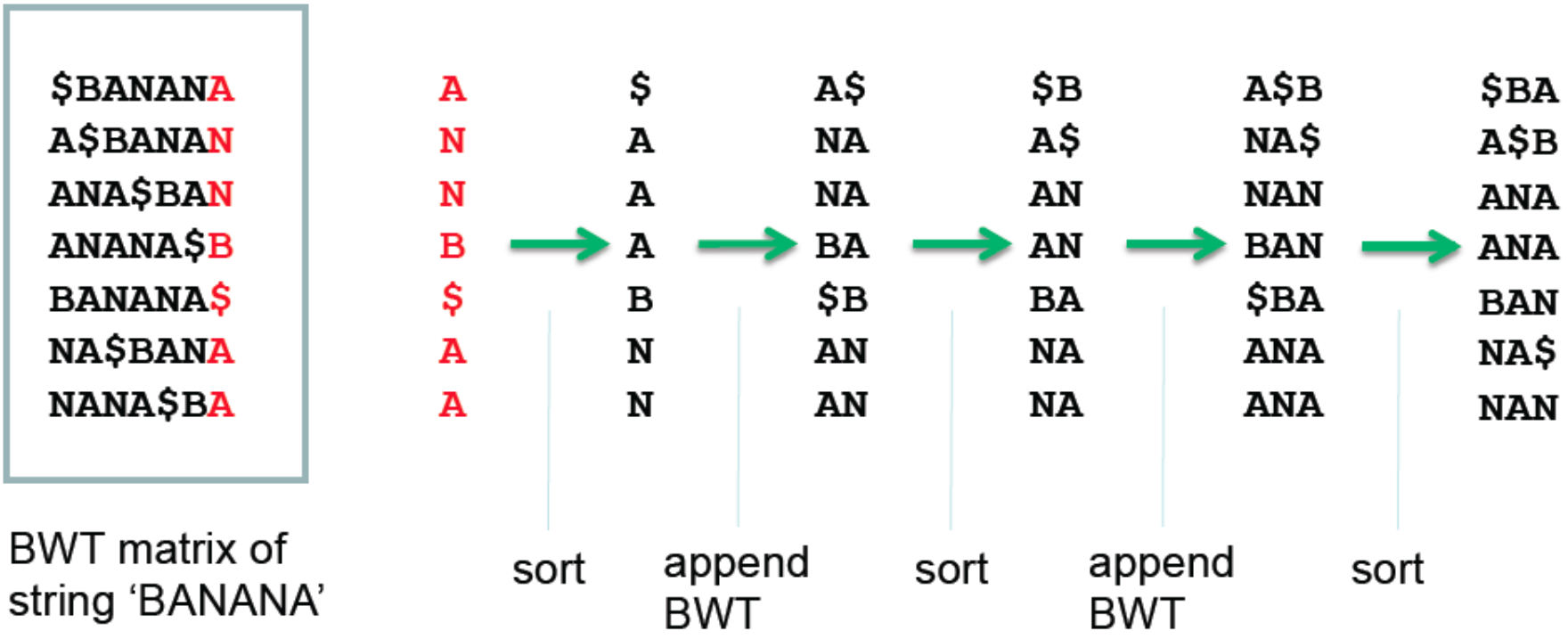
Suffixes are sorted in the BWT matrix

S(i) = j, where $X_j \ldots X_n$ is the i-th suffix lexicographically

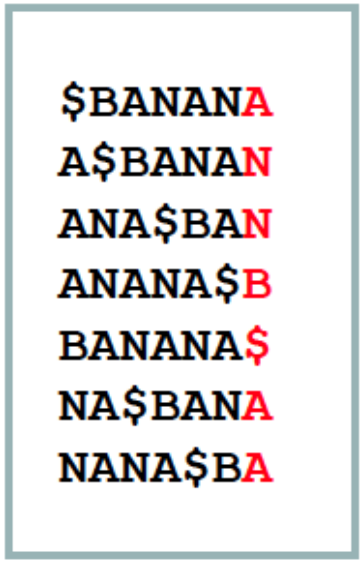
	1	2	3	4	5	6	7
X	B	A	N	A	N	A	$
S	7	6	4	2	1	5	3
BWT(X)	A	N	N	B	$	A	A

BWT(X) constructed from S:
At each position, take the letter to the left of the one pointed by S

Reconstructing BANANA



Reconstructing BANANA - faster



BWT matrix of string ‘BANANA’

Lemma. The i-th occurrence of character c in last column is the same text character as the i-th occurrence of c in the first column

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

Reconstructing BANANA - faster

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

BWT matrix of string ‘BANANA’

Lemma. The i-th occurrence of character c in last column is the same text character as the i-th occurrence of c in the first column

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

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

BWT matrix of string ‘BANANA’

Lemma. The i-th occurrence of character c in last column is the same text character as the i-th occurrence of c in the first column

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

A$BANAN
ANA$BAN
ANANA$B

Same words, same sorted order

BWT matrix of string 'BANANA'

Lemma. The i-th occurrence of character 'a' in last column is the same text character as the i-th occurrence of 'a' in the first column

LF(): Map the i-th occurrence of character 'a' in last column to the first column

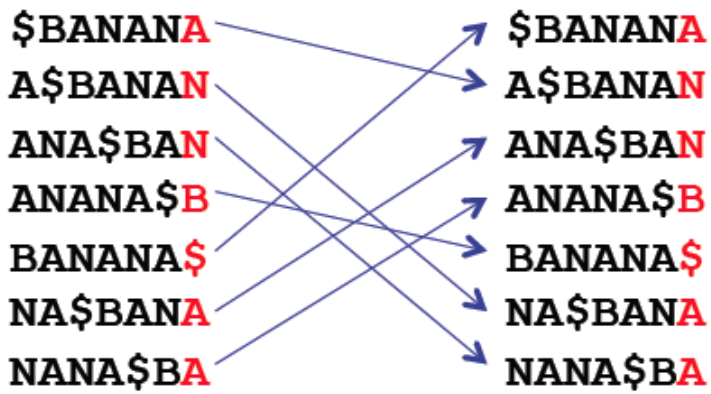
LF(r): Let row r contain the i-th occurrence of 'a' in last column

Then, LF(r) = r'; r': i-th row starting with 'a'

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

BWT matrix of string ‘BANANA’

LF(r): Let row r be the i-th occurrence of ‘a’ in last column
Then, LF(r) = r’; r’: i-th row starting with ‘a’

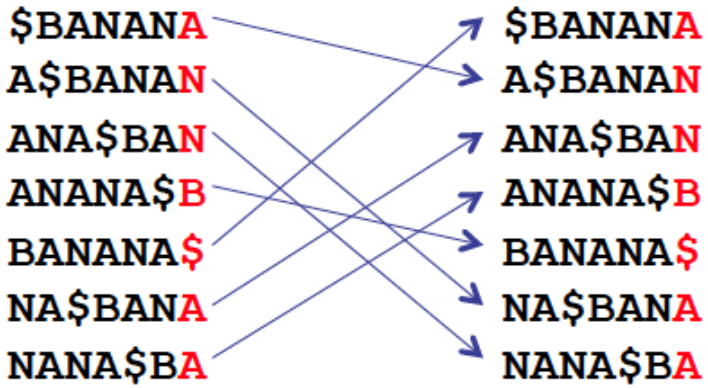


LF[] = [2, 6, 7, 5, 1, 3, 4]

Row LF(r) is obtained by rotating row r one position to the right

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

BWT matrix of string ‘BANANA’



LF[] = [2, 6, 7, 5, 1, 3, 4]

Computing LF() is easy:

Let C(a): # of characters smaller than ‘a’

Example: C($) = 0; C(A) = 1; C(B) = 4; C(N) = 5

Let row r end with the i-th occurrence of ‘a’ in last column

Then, LF(r) = C(a) + i

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

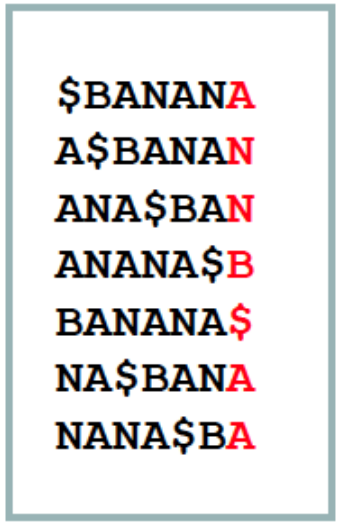
BWT matrix of string ‘BANANA’

	A	N	N	B	$	A	A	
C()	1	5	5	4	0	1	1	C() copied for convenience
index i	1	1	2	1	1	2	3	indicating this is i-th occurrence of ‘c’
LF()	2	6	7	5	1	3	4	LF() = C() + i

Reconstruct BANANA:

```
S := “”; r := 1; c := BWT[r];
UNTIL c = ‘$’ {
        S := cS;
        r := LF(r);
        c := BWT(r); }
```

Credit: Ben Langmead thesis



BWT matrix of string 'BANANA'

L(W): lowest index in BWT matrix where W is prefix
U(W): highest index in BWT matrix where W is prefix

Example:
L("NA") = 6
U("NA") = 7

Lemma (prove as exercise)
L(aW) = C(a) + i +1,
where i = # 'a's up to L(W) – 1 in BWT(X)
U(aW) = C(a) + j,
where j = # 'a's up to U(W) in BWT(X)

Example:
L("ANA") = C('A') + # 'A's up to (L("NA") – 1) + 1
= 1 + (# 'A's up to 5) + 1
= 1 + 1 + 1 = 3
U("ANA") = 1 + # 'A's up to U("NA") = 1 + 3 = 4

Summary of BWT algorithm

Suffix array of string X:

S(i) = j, where $X_j \ldots X_n$ is the j-th suffix lexicographically

- BWT follows immediately from suffix array
  - Suffix array construction possible in O(n), many good O(n log n) algorithms
- Reconstruct X from BWT(X) in time O(n)
- Search for all exact occurrences of W in time O(|W|)
- BWT(X) is easier to compress than X

BWT-based Aligners In Practice

- Inexact matching: allow mismatches and gaps during alignment

  => keep track of a set of SA intervals

- Heuristics: seeds, bounds on number of allowed differences, scoring (gap open/extend, mismatches)
- Memory considerations: sampling the suffix array and the occurrence array, compression
- Typical aligner phases

  - stage 1: BWT index construction

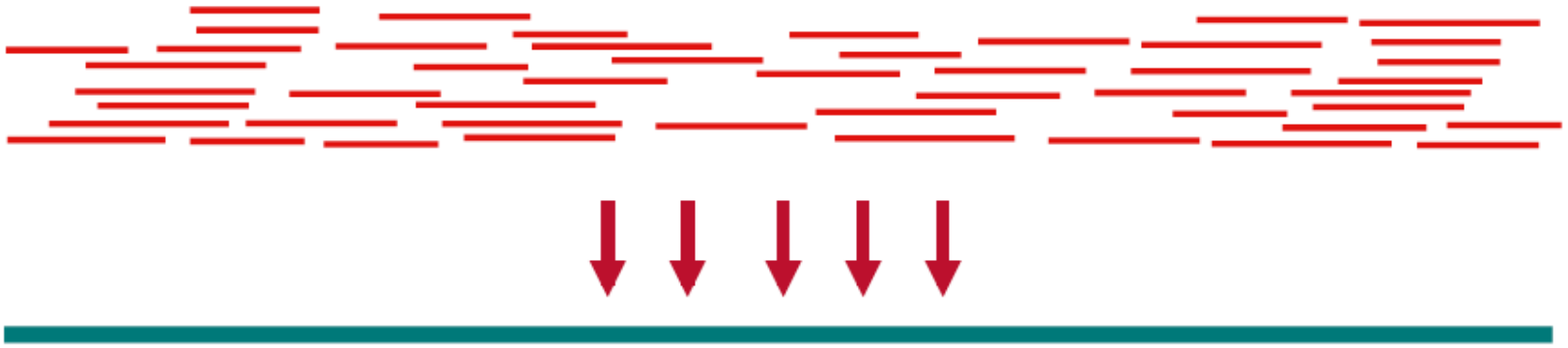
  - stage 2: short-Read Mapping

  - stage 3: alignment results reporting/evaluation

BWA

- From the author of Maq, but now also uses Burrows-Wheeler transform to significantly speed it up, and use less memory.
- Can also find small indels, in contrast to both Maq and Bowtie.
- Is slightly slower than bowtie, but ability to find indels make it more useful if SNVs are important to you.

Back to Assembly



```
                    CATCGACCGAGCGCGATGCTAGCTAGGTGATCGT......
                 TGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATCGT...
              GCATGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATCGT
           GTGCATGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATC
......AGGTGCATGCCGCATCGATCGAGCGCGATGCTAGCTAGCTGATCGT......
```

k-mer

"*k*-mer" is a substring of length *k*

S: GGCGATTCATCG

A 4-mer of *S*: ATTC

All 3-mers of *S*:

GGC
GCG
CGA
GAT
ATT
TTC
TCA
CAT
ATC
TCG

mer: from Greek meaning "part"

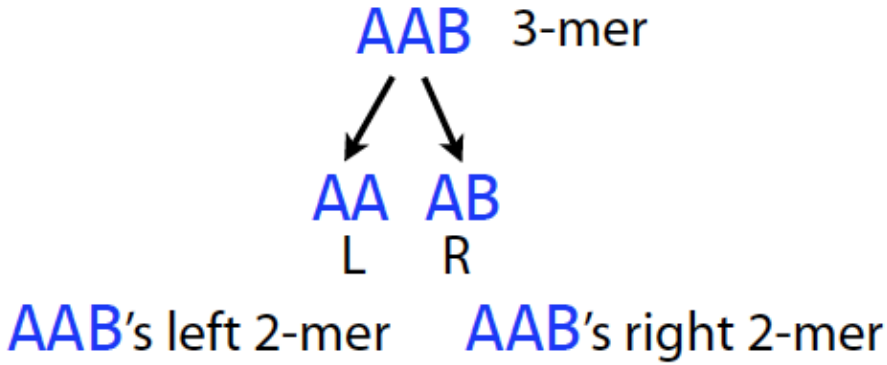
I'll use "*k*-1-mer" to refer to a substring of length *k* - 1

De Bruijn graph

As usual, we start with a collection of reads, which are substrings of the reference genome.

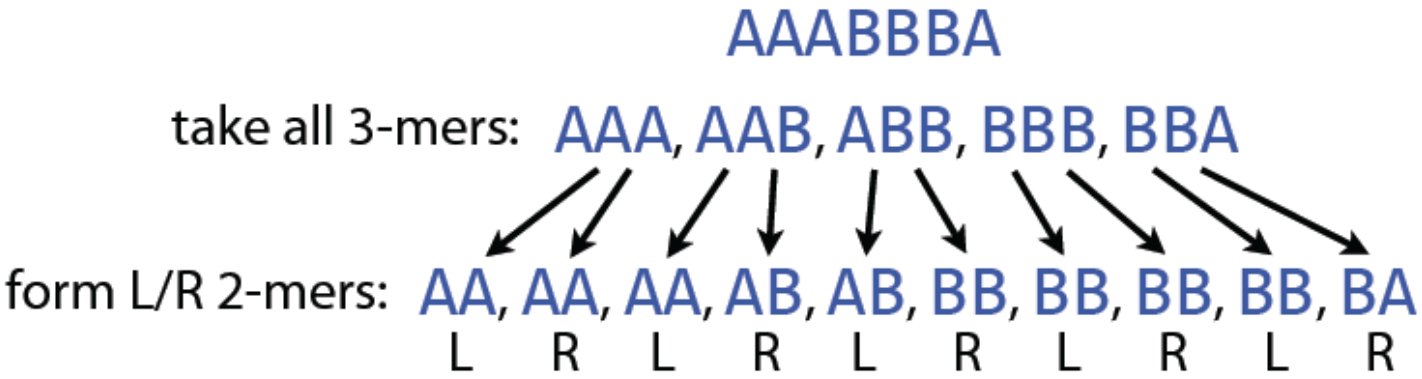
AAA, AAB, ABB, BBB, BBA

AAB is a *k*-mer (*k* = 3). AA is its *left* *k*-1-mer, and AB is its right *k*-1-mer.

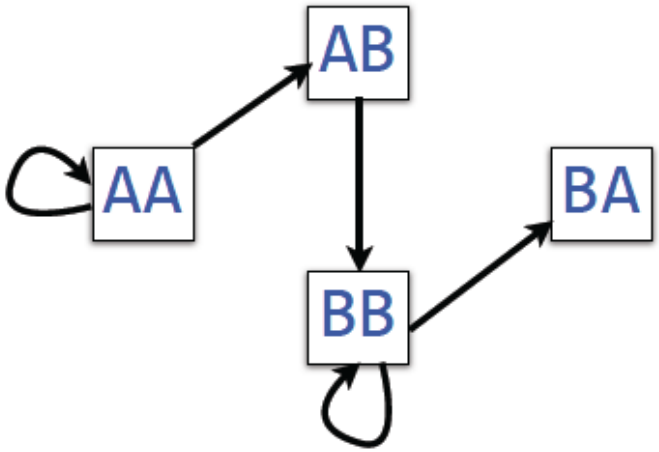


De Bruijn graph

Take each length-3 input string and split it into two overlapping substrings of length 2. Call these the *left* and *right 2-mers*.

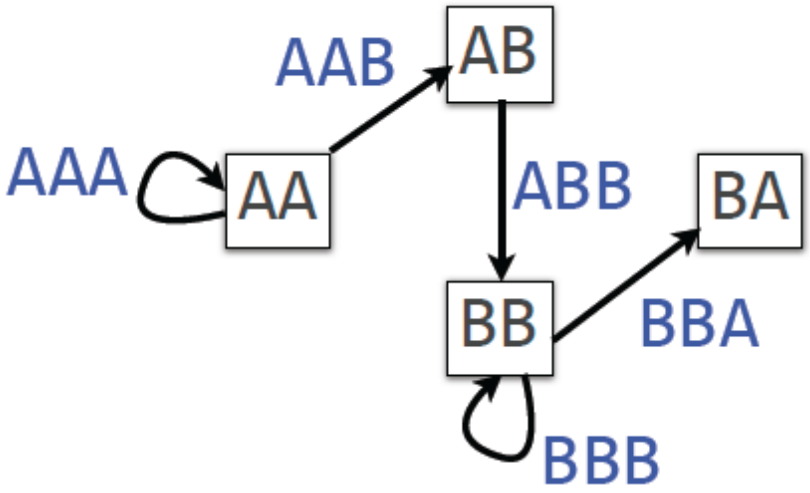


Let 2-mers be nodes in a new graph. Draw a directed edge from each left 2-mer to corresponding right 2-mer:



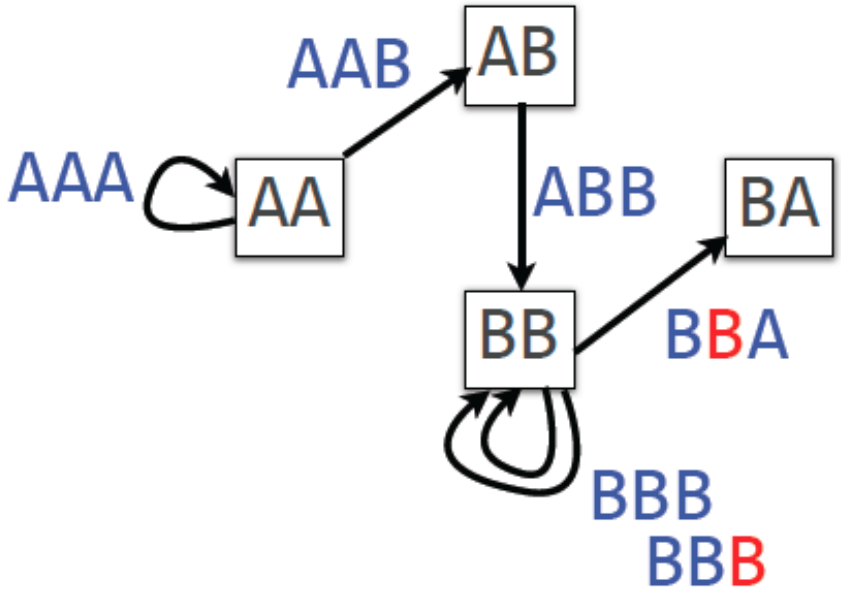
Each *edge* in this graph corresponds to a length-3 input string

De Bruijn graph



An edge corresponds to an overlap (of length *k*-2) between two *k*-1 mers.
More precisely, it corresponds to a *k*-mer from the input.

De Bruijn graph



If we add one more B to our input string: AAABBBBA, and rebuild the De Bruijn graph accordingly, we get a *multiedge*.

De Bruijn graph

A procedure for making a De Bruijn graph for a genome

Assume *perfect sequencing* where each length-*k* substring is sequenced exactly once with no errors

Pick a substring length *k*: 5

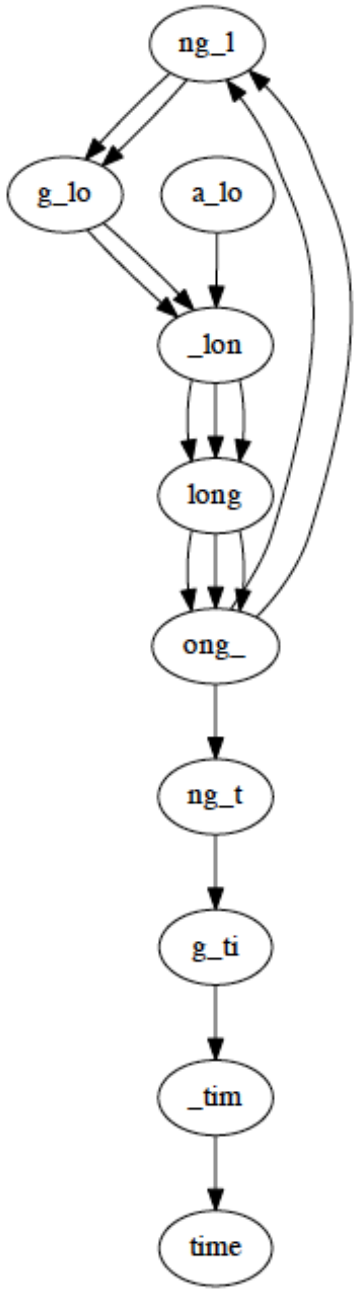
Start with an input string: `a_long_long_long_time`

Take each *k* mer and split into left and right *k*-1 mers

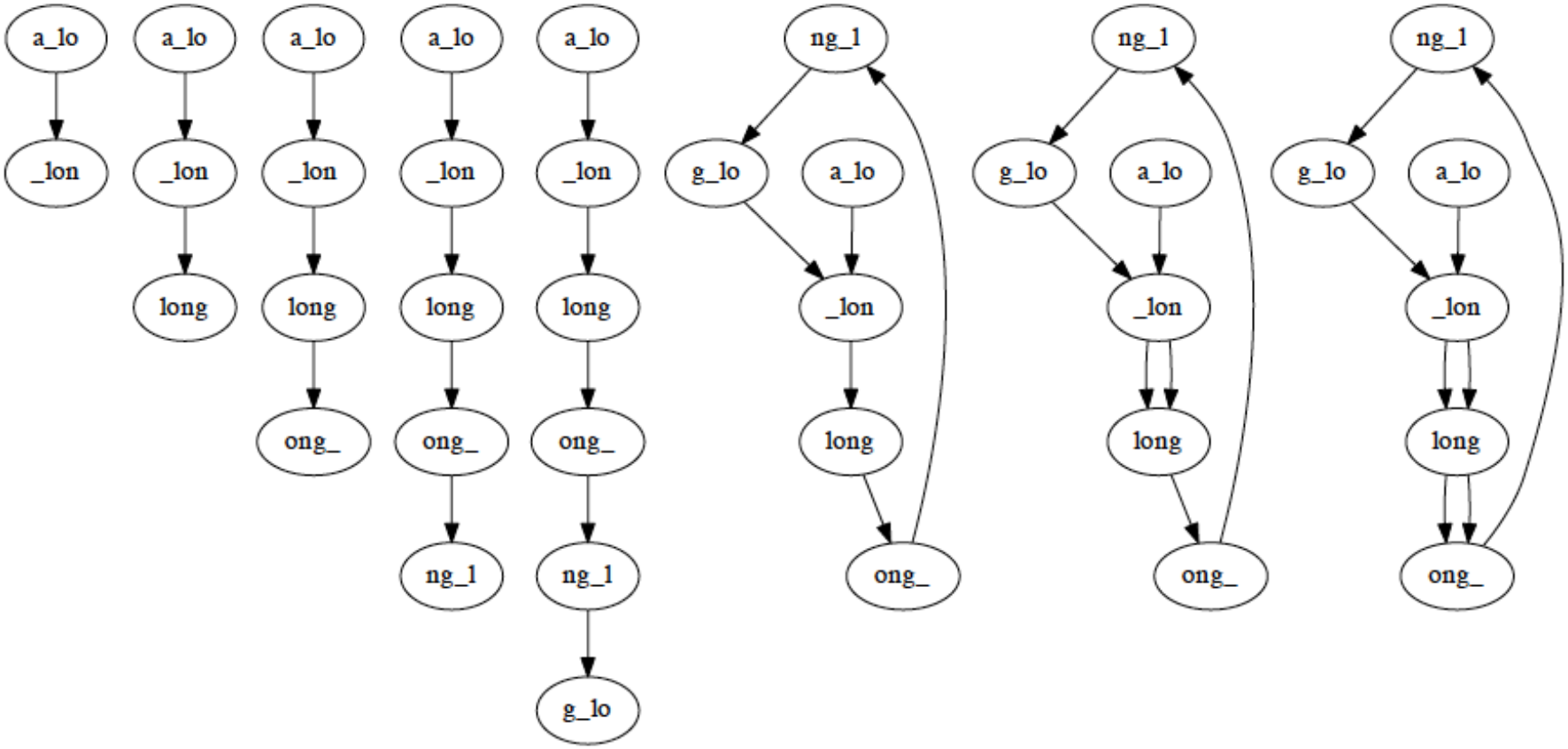
`long_`

`long ong_`

Add k-1 mers as nodes to De Bruijn graph (if not already there), add edge from left *k*-1 mer to right *k*-1 mer



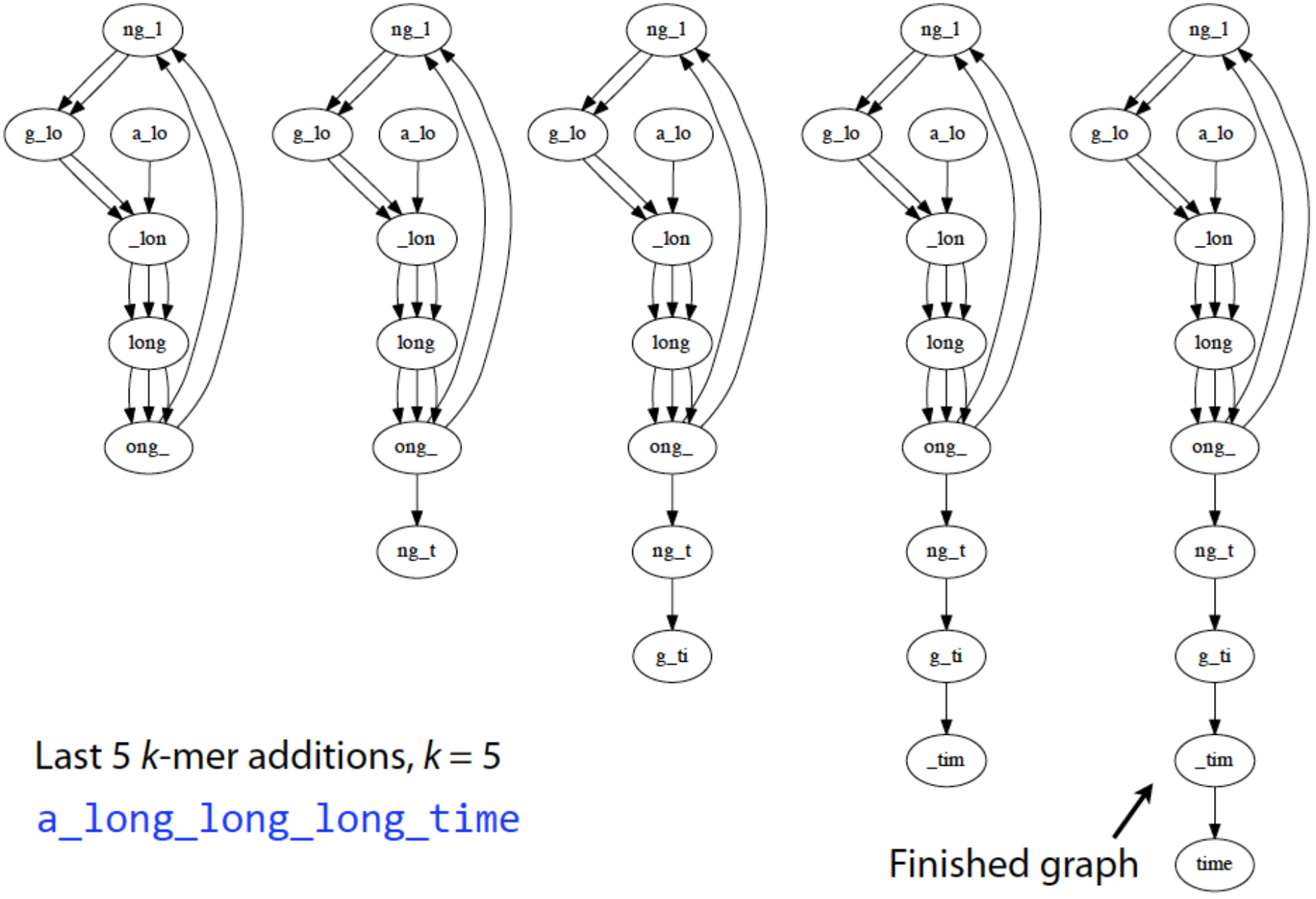
De Bruijn graph



First 8 *k*-mer additions, $k = 5$

`a_long_long_long_time`

De Bruijn graph



Last 5 *k*-mer additions, $k = 5$

`a_long_long_long_time`

Finished graph

De Bruijn graph

In practice, De Bruijn graph-based tools give up on unresolvable repeats and yield fragmented assemblies, just like OLC tools.

But first we note that using the De Bruijn graph representation has **other advantages**...

De Bruijn graph

Say a sequencer produces **d** reads of length **n** from a genome of length **m**

d = 6 x 10^9 reads
n = 100 nt
m = 3 x 10^9 nt ≈ human

} ≈ 1 sequencing run

To build a De Bruijn graph in practice:

Pick *k*. Assume *k* ≤ shortest read length (*k* = 30 to 50 is common).

For each read:

For each *k*-mer:

Add *k*-mer's left and right *k*-1-mers to graph if not there already. Draw an edge from left to right *k*-1-mer.

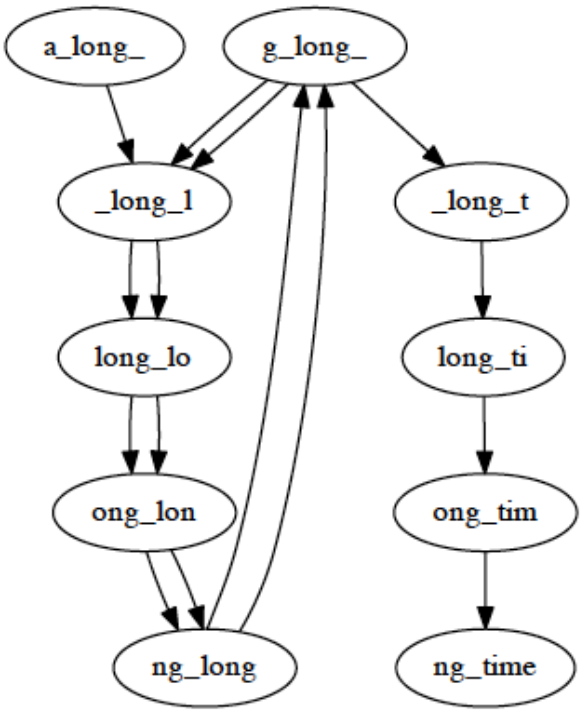
De Bruijn graph

Pick $k = 8$ Genome: a_long_long_long_time

Reads: a_long_long_long, ng_long_l, g_long_time

k-mers:

```
a_long_l          ng_long_   g_long_t
 _long_lo          g_long_l   _long_ti
  long_lon                    long_tim
   ong_long                    ong_time
    ng_long_
     g_long_l
      _long_lo
       long_lon
        ong_long
```



Given n (# reads), N (total length of all reads) and k, and assuming $k <$ length of shortest read:

Exact number of k-mers: $N - n(k-1)$ $O(N)$

This is also the number of edges, $|E|$

Number of nodes $|V|$ is at most $2 \cdot |E|$, but typically much smaller due to repeated k-1-mers

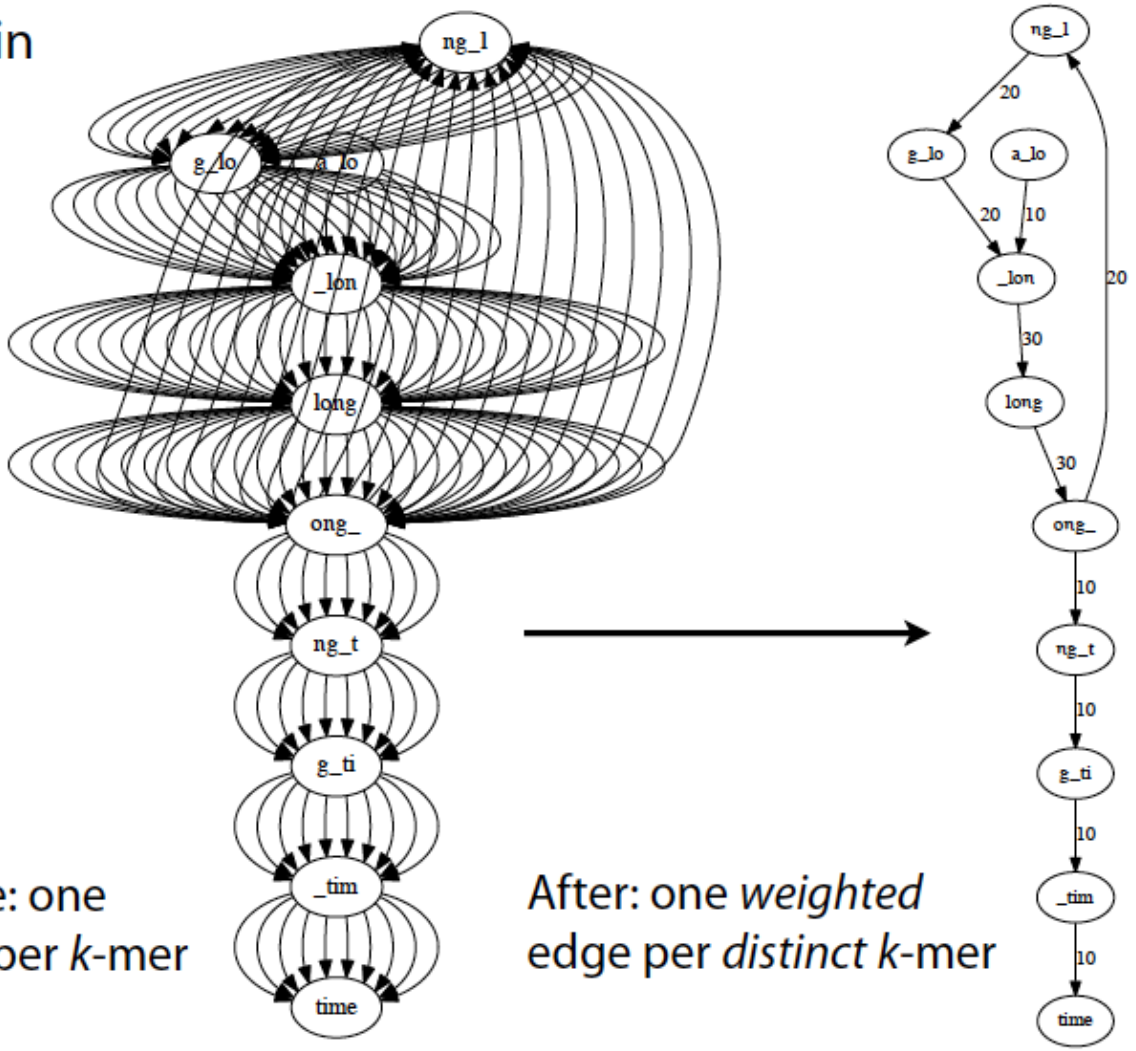
De Bruijn graph

In typical assembly projects, average coverage is ~ 30 - 50

Same edge might appear in dozens of copies; let's use edge *weights* instead

Weight = # times *k*-mer occurs

Using weights, there's one *weighted* edge for each *distinct k*-mer



Before: one edge per *k*-mer

After: one *weighted* edge per *distinct k*-mer

De Bruijn graph

What did we give up?

Reads are immediately split into shorter *k*-mers; can't resolve repeats as well as overlap graph

Only a very specific type of "overlap" is considered, which makes dealing with errors more complicated, as we'll see

Read coherence is lost. Some paths through De Bruijn graph are inconsistent with respect to input reads.

This is the OLC ↔ DBG tradeoff

Single most important benefit of De Bruijn graph is the O(min(*G*, *N*)) space bound, though we'll see this comes with large caveats

Binning

Covered last week