NUS
National University of Singapore
National University of Singapore

CS4330: Combinatorial Methods in Bioinformatics

K-mers counting on disk

Wong Limsoon

Acknowledgement: This set of slides were adapted from Ken Sung's

Disk-based techniques

Memory-based K-mer counting methods cannot handle big datasets

Disk-based approaches

Split and merge – KAnalyze

Split by hashing – DSK

Split by prefix – KMC

Check out this one on your own

Split by super K-mer – MSPKmerCounter, KMC2, KMC3

Split and merge - KAnalyze

Split all K-mers into subsets such that each subset can be stored in memory

For each subset,

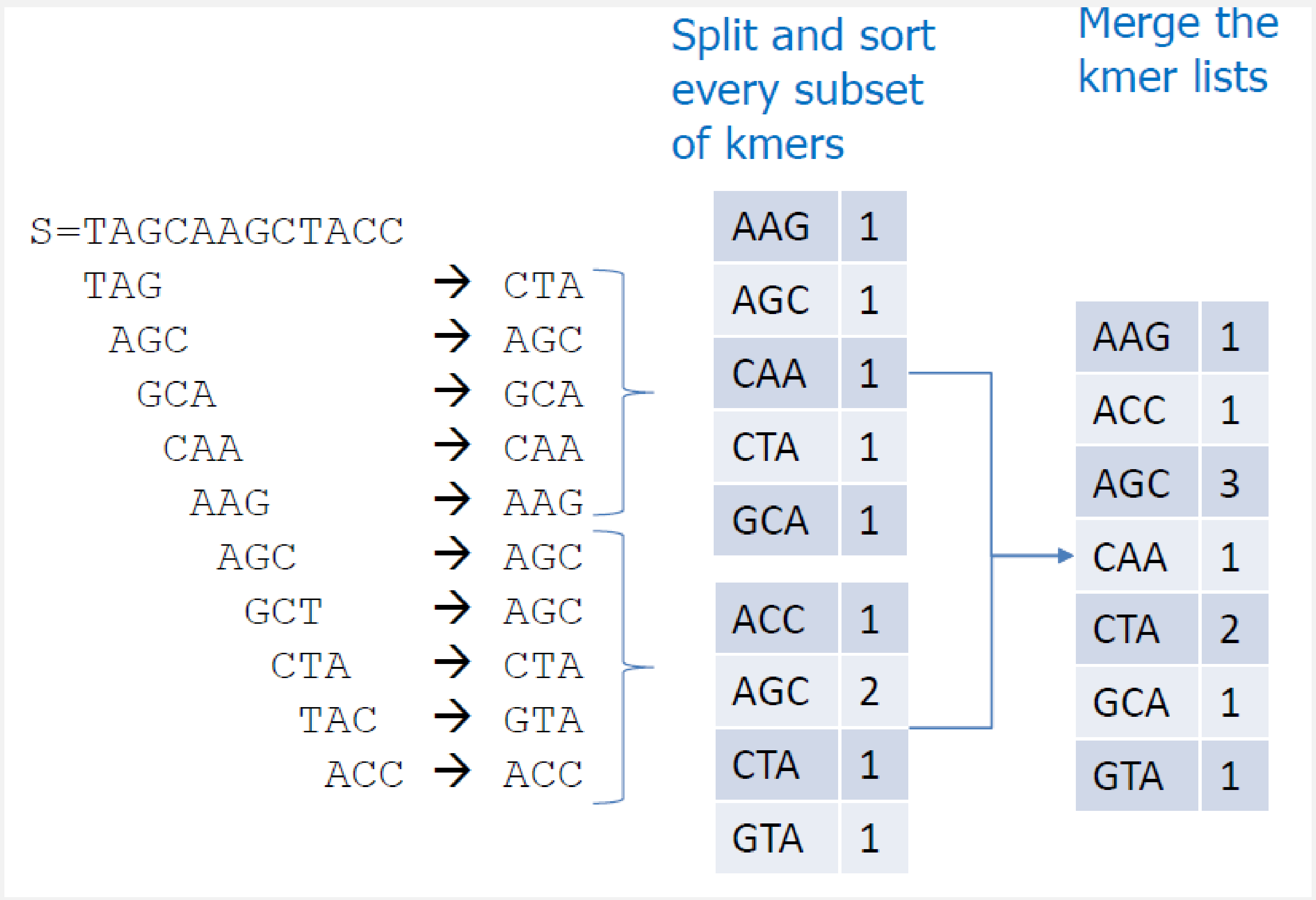
Sort K-mers in memory and obtain counts

Store sorted K-mers and counts to a disk file

Merge the files

Audano & Vannberg, “KAnalyze: A fast versatile pipelined K-mer toolkit”, *Bioinformatics* 30(14):2070-2072, 2014

Example



```
S=TAGCAAGCTACC
  TAG           → CTA
   AGC          → AGC
    GCA         → GCA
     CAA        → CAA
      AAG       → AAG
       AGC      → AGC
        GCT     → AGC
         CTA    → CTA
          TAC   → GTA
           ACC  → ACC
```

AAG	1
AGC	1
CAA	1
CTA	1
GCA	1

ACC	1
AGC	2
CTA	1
GTA	1

AAG	1
ACC	1
AGC	3
CAA	1
CTA	2
GCA	1
GTA	1

Issue with split and merge

K-mers are grouped into subsets in the order they appear in the reads

Different occurrences of the same K-mer can get into different files

Can we avoid wasting time in merging them?

One solution is hashing

Split by hashing – the DSK way

For each K-mer w, hash it to the file h(w) mod n

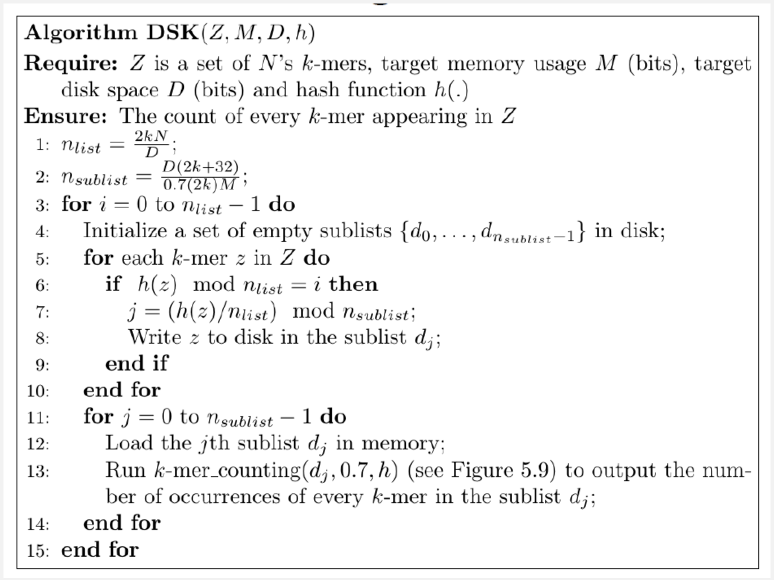
For each file:

Use Jellyfish to count its K-mers in memory

Write the K-mers and their counts to an output file

Merge the output files

The actual DSK is a bit more intricate, to ensure the files are balanced in size and can fit into memory



```
**Algorithm** DSK$(Z, M, D, h)$
**Require:** $Z$ is a set of $N$'s $k$-mers, target memory usage $M$ (bits), target
    disk space $D$ (bits) and hash function $h(.)$
**Ensure:** The count of every $k$-mer appearing in $Z$
$n_{list} = \frac{2kN}{D}$;
$n_{sublist} = \frac{D(2k+32)}{0.7(2k)M}$;
**for** $i = 0$ to $n_{list} - 1$ **do**
    Initialize a set of empty sublists $\{d_0, \ldots, d_{n_{sublist}-1}\}$ in disk;
    **for** each $k$-mer $z$ in $Z$ **do**
        **if** $h(z) \mod n_{list} = i$ **then**
            $j = (h(z)/n_{list}) \mod n_{sublist}$;
            Write $z$ to disk in the sublist $d_j$;
        **end if**
    **end for**
    **for** $j = 0$ to $n_{sublist} - 1$ **do**
        Load the $j$th sublist $d_j$ in memory;
        Run $k$-mer_counting$(d_j, 0.7, h)$ (see Figure 5.9) to output the num-
        ber of occurrences of every $k$-mer in the sublist $d_j$;
    **end for**
**end for**
```

Rizk et al., "DSK-k-mer counting with very low memory usage", *Bioinformatics* 29(5):652-653, 2013

Issue with DSK

I/O is slow

DSK writes a tmp file of 2k I/O bits per K-mer

This can be expensive

Can we reduce I/O cost per K-mer?

“Super K-mers”

Split by super K-mers - MSPKmerCounter

Group all K-mers with same minimizer in same file

For each file:

Use Jellyfish to count its K-mers in memory

Write the K-mers and their counts to an output file

Merge the output files

Use "minimum substring partitioning (MSP)" to distributes K-mers to files based on minimizers

Li & Yan, "MSPKmerCounter: A fast and memory efficient approach for K-mer counting", 2015, https://doi.org/10.48550/arXiv.1505.06550

Minimizer

The length-p minimizer, $\min_p(S)$, of a string S[1..n] is the lexicographically smallest p-mer in both S and its reverse complement

Examples

$\min_4$(GCC**AAGC**GCCAGGCAGCCG) = AAGC @ position 4

$\min_4$(GCCAGGCAGCCG**CAGT**GGG) = ACTG @ position 13

Obviously, two identical K-mers have the same minimizer

Example

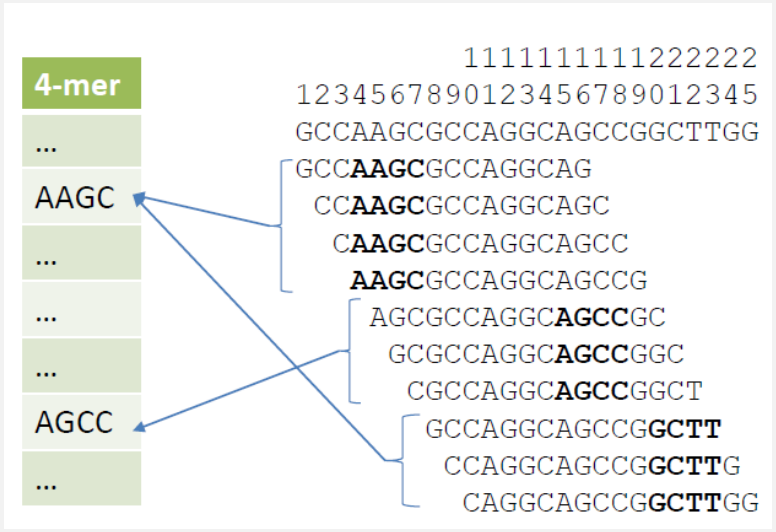
Let K = 16, p = 4

Consider a read, GCCAAGCGCCAGGCAGCCGGCTTGG

The K-mers are grouped into:

File AAGC has 7 K-mers
I/O = 7 * 16 nt = 112 nt

File AGCC has 3 K-mers
I/O = 3 * 16 nt = 48 nt



Observation

Many K-mers in the same file are consecutive

In our example,

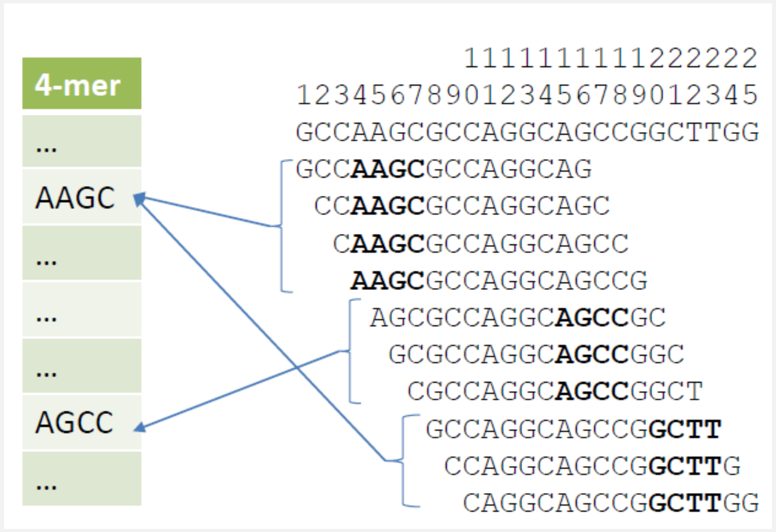
File AAGC has 7 K-mers:

1st – 4th K-mers

8th – 10th K-mers

File AGCC has 3 K-mers:

5th – 7th K-mers



Group consecutive K-mers into super K-mer

For consecutive K-mers in the same file, compress them into a “super K-mer” to minimize I/O

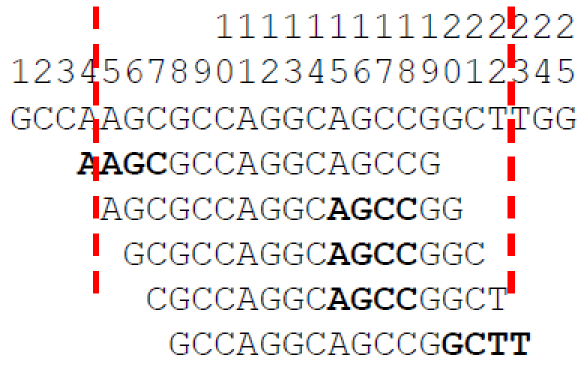
S[i..j] is a **super K-mer** if all K-mers in S[i..j] share *same* length-p minimizer but not those in S[i..j+1] and S[i-1..j]

Example: K = 16, p = 4

S = GCC**AAGC**GCCAGGCAGCCGGCTTGG

The 16-mers S[5..20], S[6..21], S[7..22] share the *same* length-4 minimizer AAGC

S[5..22] = AGCGCCAGGCAGCCGGCT is super 16-mer



```
         1111111111222222
1234567890123456789012345
GCCAAGCGCCAGGCAGCCGGCTTGG
   AAGCGCCAGGCAGCCG
    AGCGCCAGGCAGCCGG
     GCGCCAGGCAGCCGGC
      CGCCAGGCAGCCGGCT
       GCCAGGCAGCCGGCTT
```

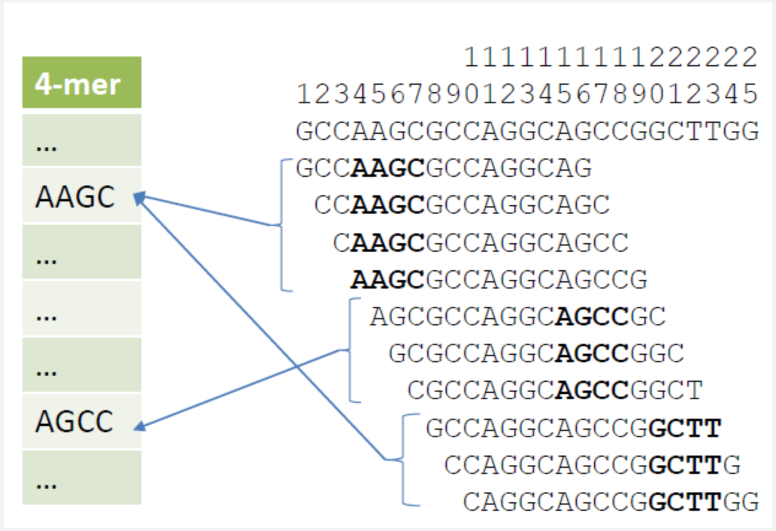
Example

Consider a read, GCCAAGCGCCAGGCAGCCGGCTTGG

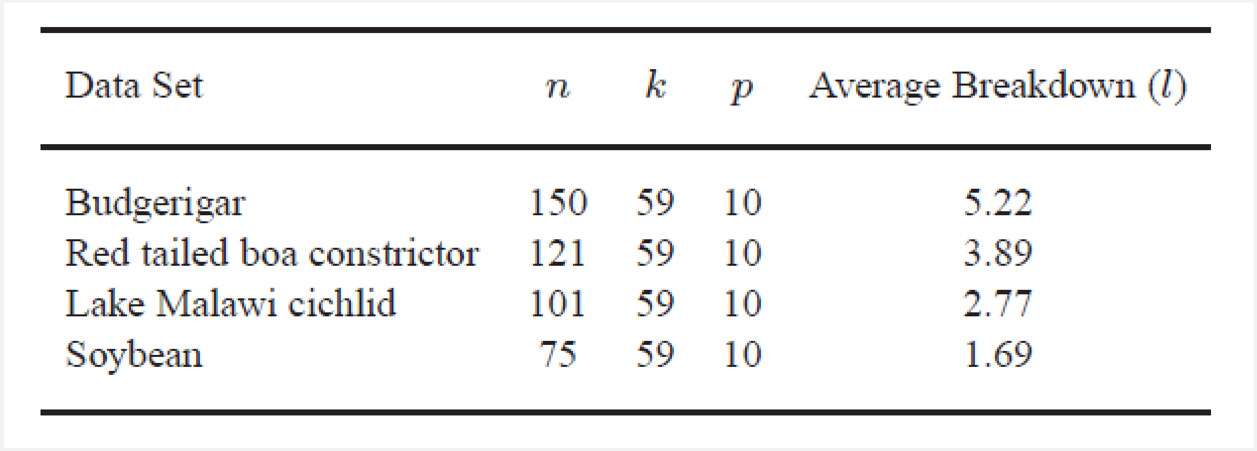
Let K = 16, p = 4. It has two files:

File AAGC has 7 K-mers,
Rep by 2 super K-mers
1..4: GCC**AAGC**GCCAGGCAGCCG
8..10: GCCAGGCAGCCG**GCTT**GG
I/O: 19 + 18 = 37 nt (vs 112 nt)

File AGCC has 3 K-mers,
Rep by 1 super K-mer
5..7: AGCGCCAGGC**AGCC**GGCT
I/O: 18 nt (vs 48 nt)



For real short read datasets, average # of super K-mers per read is usually small



Data Set	n	k	p	Average Breakdown (l)
Budgerigar	150	59	10	5.22
Red tailed boa constrictor	121	59	10	3.89
Lake Malawi cichlid	101	59	10	2.77
Soybean	75	59	10	1.69

n = read length

l = mean # of super K-mer per read

Li & Yan, "MSPKmerCounter: A fast and memory efficient approach for K-mer counting", 2015, https://doi.org/10.48550/arXiv.1505.06550

The MSP algorithm for partitioning a read into super K-mers

```
Input: S[1 .. n], K, p
$min_s$ = length-p minimizer of S[1..K]
$min_p$ = position of $min_s$ in S
st = 1
for j = 2 to n – K + 1:
        if j > $min_p$ or the last p-substring of S[j .. j + K – 1] < $min_s$ then
                Output S[st .. j – 1] as super K-mer
                st = j
                $min_s$ = length-p minimizer of S[j .. j + K – 1]
                $min_p$ = position of $min_s$ in S
```

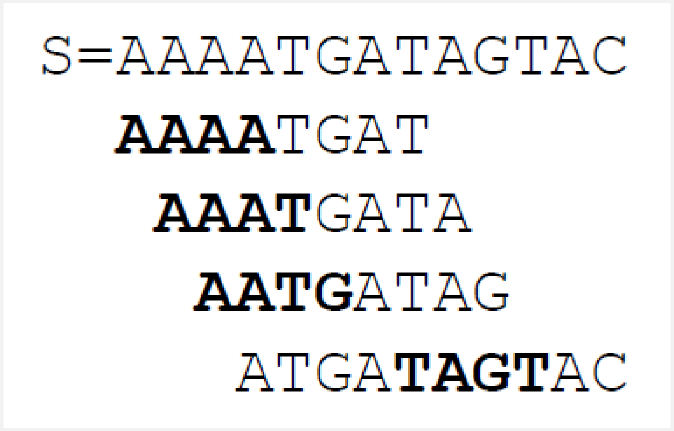
Li & Yan, "MSPKmerCounter: A fast and memory efficient approach for K-mer counting", 2015, https://doi.org/10.48550/arXiv.1505.06550

Issue with MSPKmerCounter

When a minizer starts with a few A’s, it often implies several new super K-mers spanning a single K-mer

Example: K = 8, p = 4

Due to AAAA, the first 3 super K-mers span single K-mer only



```
S=AAAATGATAGTAC
  AAAATGAT
   AAATGATA
    AATGATAG
     ATGATAGTAC
```

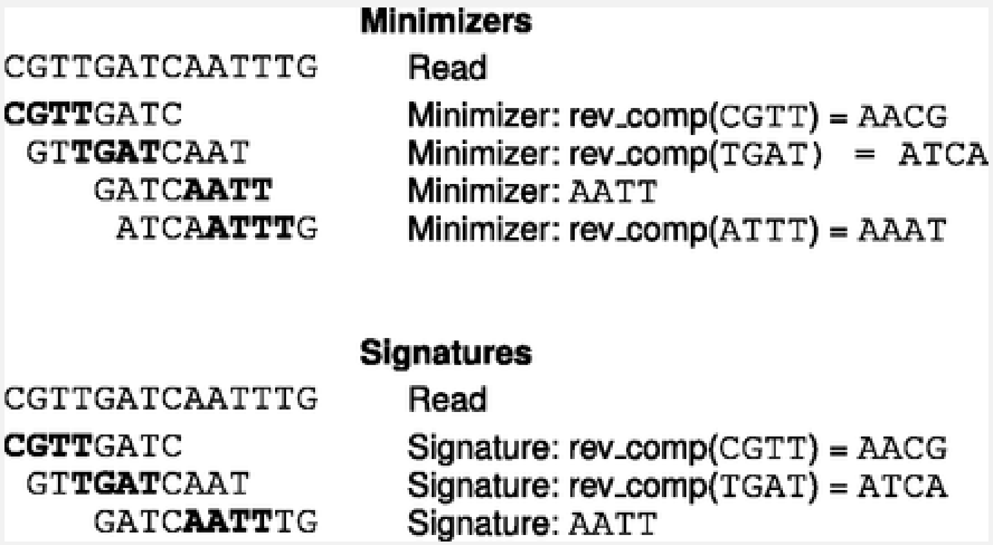
Use signature instead of minimizer

KMC2 uses canonical minimizers as **signatures** but exclude those:

Starting with AAA

Starting with ACA, or

Contain AA anywhere except at the start



Minimizers

```
CGTTGATCAATTTG      Read
CGTTGATC            Minimizer: rev_comp(CGTT) = AACG
 GTTGATCAAT         Minimizer: rev_comp(TGAT)  =  ATCA
    GATCAATT        Minimizer: AATT
     ATCAATTTG      Minimizer: rev_comp(ATTT) = AAAT
```

Signatures

```
CGTTGATCAATTTG      Read
CGTTGATC            Signature: rev_comp(CGTT) = AACG
 GTTGATCAAT         Signature: rev_comp(TGAT) = ATCA
    GATCAATTTG      Signature: AATT
```

Deorowics et al., “KMC2: Fast and resource-frugal k-mer counting”, *Bioinformatics* 31(10):1569-1576, 2015

Exercise

Given AACCACAGCTTGTTTGTTCTTG

Let K = 10, p = 4

Show super K-mers defined based on minimizer

Show super K-mers defined based on signature

KMC2

Break reads into super K-mers using signatures

Distribute super K-mers into files according to signatures

For each file:

Sort K-mers using LSD radix sort

Output K-mers and their counts

Merge the output files

Deorowics et al., “KMC2: Fast and resource-frugal k-mer counting”, *Bioinformatics* 31(10):1569-1576, 2015

Issue with KMC2

When K is large, LSD radix sort is slow

In fact, KMC2 is slower than DSK when K is large

Solution: Use MSD radix sort

KMC3

Break reads into super K-mers using signatures

Distribute super K-mers into files according to signatures

For each file:

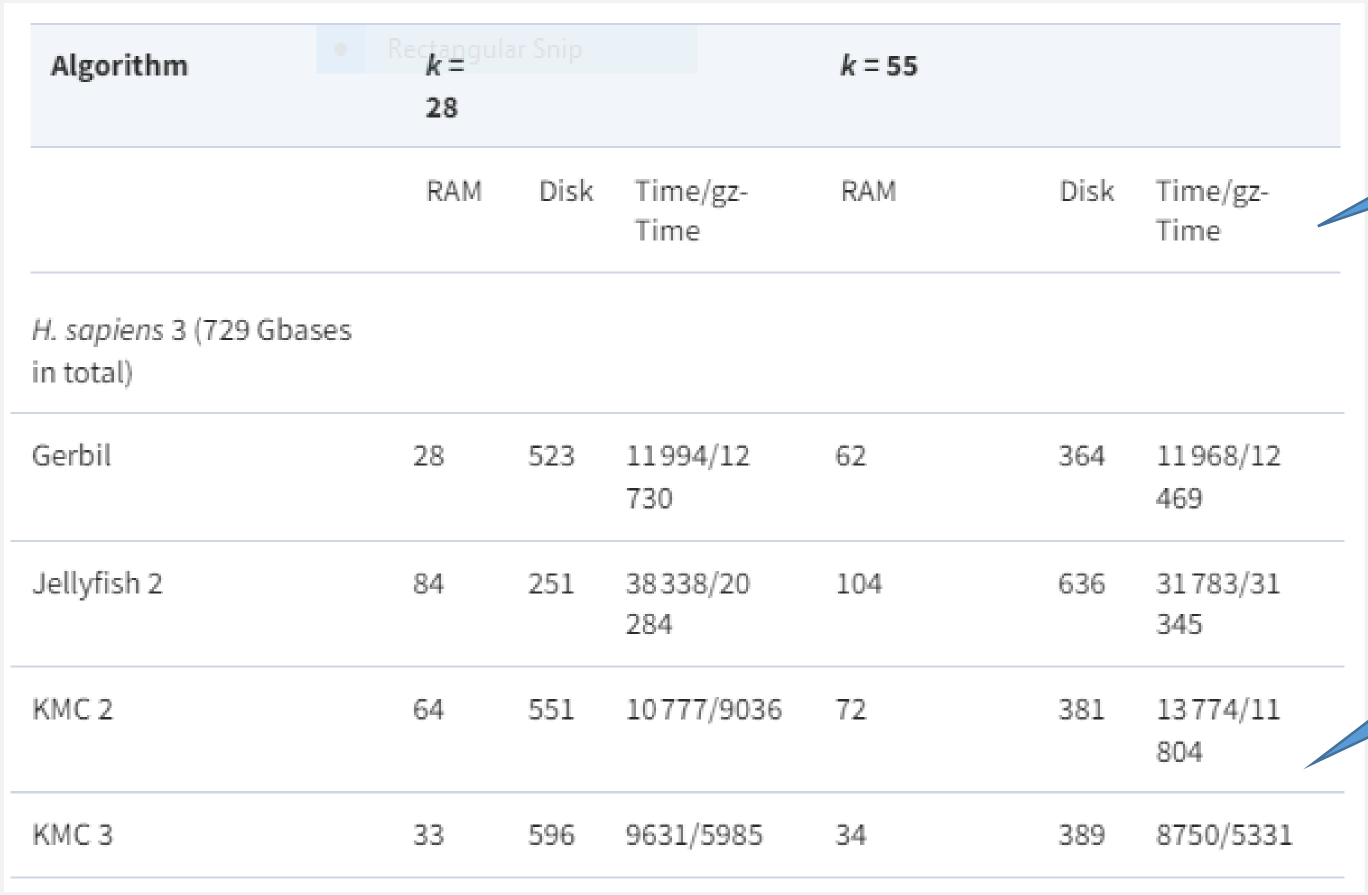
Sort K-mers using MSD radix sort

Output K-mers and their counts

Merge the output files

Kokot et al., "KMC3: Counting and manipulating k-mer statistics", *Bioinformatics* 33(17):2759-2761, 2017

Performance of KMC3



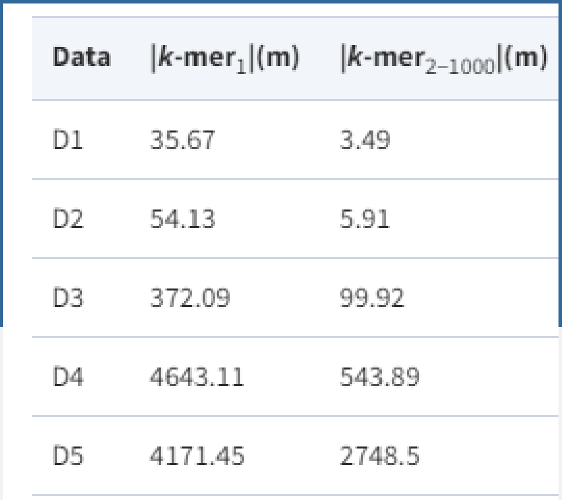
Algorithm	*k* = 28			*k* = 55		
	RAM	Disk	Time/gz-Time	RAM	Disk	Time/gz-Time
H. sapiens 3 (729 Gbases in total)						
Gerbil	28	523	11 994/12 730	62	364	11 968/12 469
Jellyfish 2	84	251	38 338/20 284	104	636	31 783/31 345
KMC 2	64	551	10 777/9036	72	381	13 774/11 804
KMC 3	33	596	9631/5985	34	389	8750/5331

Uncompressed / compressed FASTA as input

KMC2 & 3 also use other tricks to get good performance

Kokot et al., "KMC3: Counting and manipulating k-mer statistics", *Bioinformatics* 33(17):2759-2761, 2017

Observations from typical sequencing projects

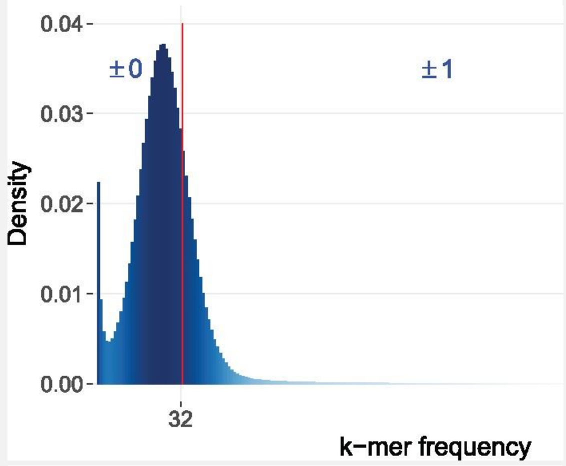


Data	$\lvert k\text{-mer}_1 \rvert$(m)	$\lvert k\text{-mer}_{2-1000} \rvert$(m)
D1	35.67	3.49
D2	54.13	5.91
D3	372.09	99.92
D4	4643.11	543.89
D5	4171.45	2748.5

50-90% of unique K-mers occur once in the read set

Call these "error K-mers"

Average # of occurrences of the other unique K-mers is close to sequencing coverage



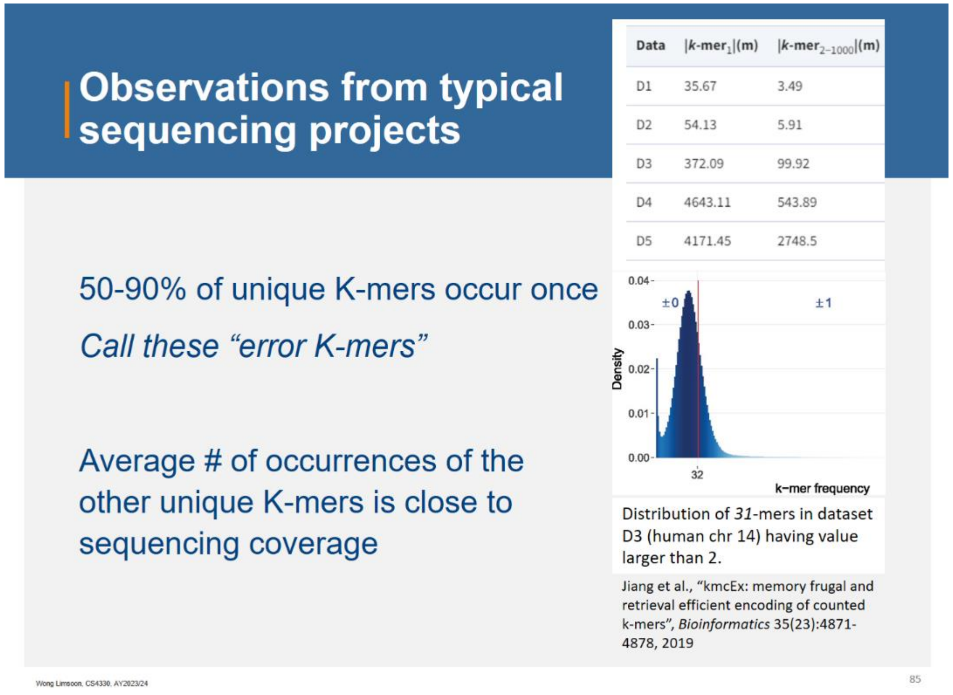
Distribution of *31*-mers in dataset D3 (human chr 14) having value larger than 2.

Jiang et al., "kmcEx: memory frugal and retrieval efficient encoding of counted k-mers", *Bioinformatics* 35(23):4871-4878, 2019

Exercise

Suppose coverage is 30 and 60% of unique K-mers are error K-mers

What is the ratio of error : non-error K-mer occurrences?



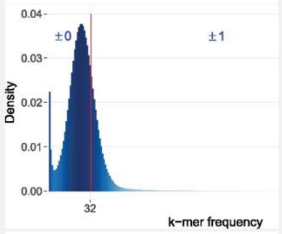
Observations from typical sequencing projects

50-90% of unique K-mers occur once

Call these "error K-mers"

Average # of occurrences of the other unique K-mers is close to sequencing coverage

Data	$\lvert k\text{-mer}_1\rvert$(m)	$\lvert k\text{-mer}_{2-1000}\rvert$(m)
D1	35.67	3.49
D2	54.13	5.91
D3	372.09	99.92
D4	4643.11	543.89
D5	4171.45	2748.5



Distribution of *31*-mers in dataset D3 (human chr 14) having value larger than 2.

Jiang et al., "kmcEx: memory frugal and retrieval efficient encoding of counted k-mers", *Bioinformatics* 35(23):4871-4878, 2019

Wong Limsoon, CS4330, AY2023/24 85

Split by hashing – "database hashjoin" style

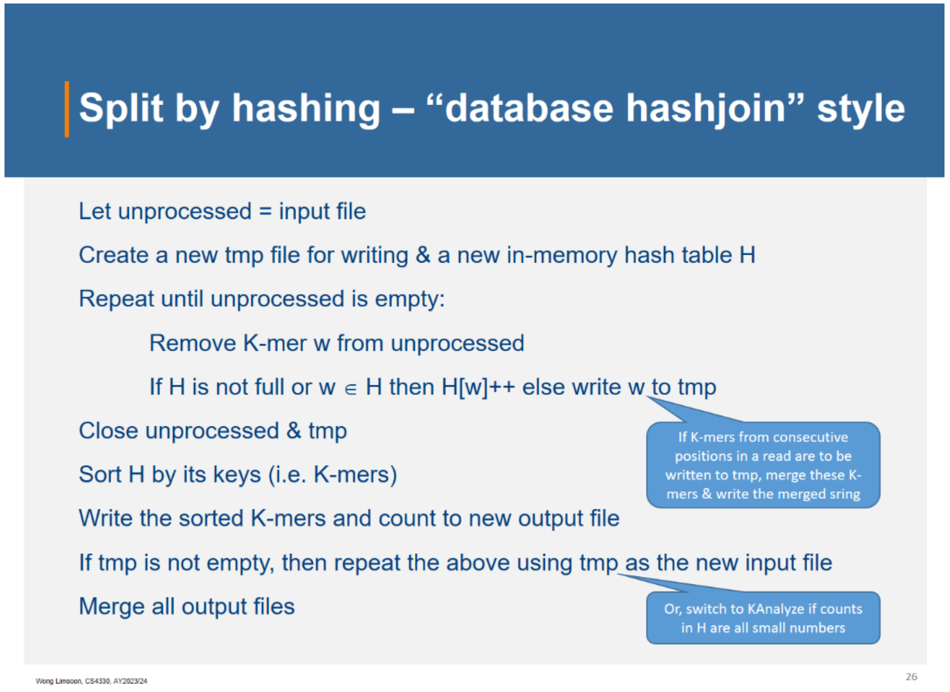
```
Let unprocessed = input file
Create a new tmp file for writing & a new in-memory hash table H
Repeat until unprocessed is empty:
        Remove K-mer w from unprocessed
        If H is not full or w ∈ H then H[w]++ else write w to tmp
Close unprocessed & tmp
Sort H by its keys (i.e. K-mers)
Write the sorted K-mers and count to new output file
If tmp is not empty, then repeat the above using tmp as the new input file
Merge all output files
```

If K-mers from consecutive positions in a read are to be written to tmp, merge these K-mers & write the merged string

Or, switch to KAnalyze if counts in H are all small numbers

Exercise

Do you think the “database hashjoin” idea is reasonable?



Split by hashing – “database hashjoin” style

```
Let unprocessed = input file
Create a new tmp file for writing & a new in-memory hash table H
Repeat until unprocessed is empty:
        Remove K-mer w from unprocessed
        If H is not full or w ∈ H then H[w]++ else write w to tmp
Close unprocessed & tmp
Sort H by its keys (i.e. K-mers)
Write the sorted K-mers and count to new output file
If tmp is not empty, then repeat the above using tmp as the new input file
Merge all output files
```

If K-mers from consecutive positions in a read are to be written to tmp, merge these K-mers & write the merged sring

Or, switch to KAnalyze if counts in H are all small numbers

Wong Limsoon, CS4330, AY2023/24

26

Summary for K-mer counting

Counting techniques

Hashing,

Sorting,

Counting Bloom filter,

Burst ties, Suffix array

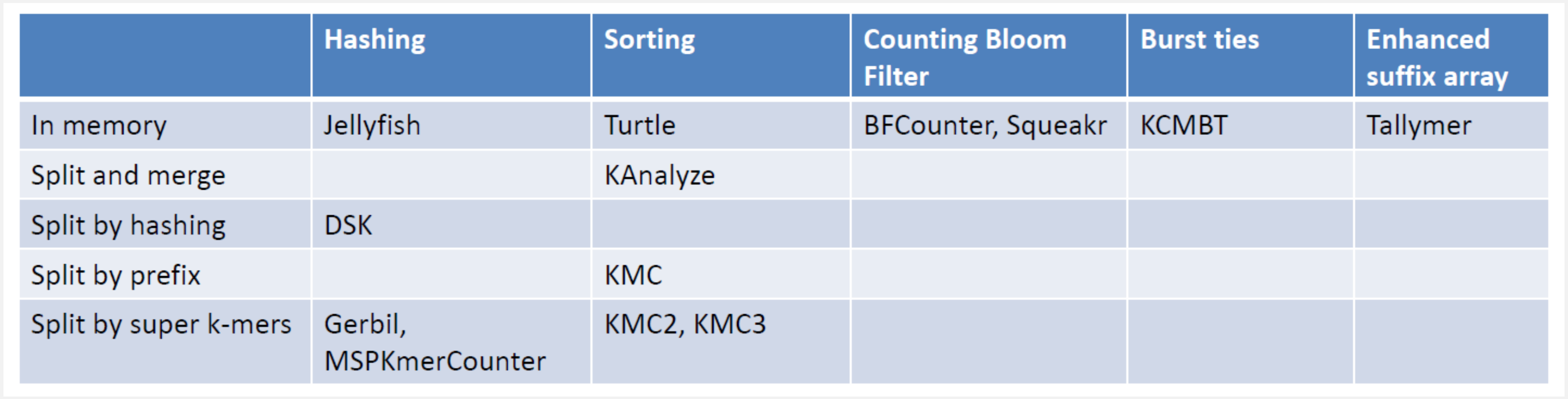
Disk-based techniques

Split and merge,

Split by hashing,

Hash by super K-mers,

Hash by prefix



	Hashing	Sorting	Counting Bloom Filter	Burst ties	Enhanced suffix array
In memory	Jellyfish	Turtle	BFCounter, Squeakr	KCMBT	Tallymer
Split and merge		KAnalyze			
Split by hashing	DSK				
Split by prefix		KMC			
Split by super k-mers	Gerbil, MSPKmerCounter	KMC2, KMC3			

Good to read

KMC1 & 2

Deorowicz et al., “Disk-based k-mer counting on a PC”, *BMC Bioinformatics* 14:160, 2013. https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-14-160

Deorowics et al., “KMC2: Fast and resource-frugal k-mer counting”, Bioinformatics 31(10):1569-1576, 2015 https://doi.org/10.1093/bioinformatics/btv022

Good to read

KAnalyze

Audano & Vannberg, “KAnalyze: A fast versatile pipelined K-mer toolkit”, *Bioinformatics* 30(14):2070-2072, 2014 https://doi.org/10.1093/bioinformatics/btu152

DSK

Rizk et al., “DSK: k-mer counting with very low memory usage”, *Bioinformatics* 29(5):652-653, 2013 https://doi.org/10.1093/bioinformatics/btt020

MSPKmerCounter

Li & Yan, “MSPKmerCounter: A fast and memory efficient approach for K-mer counting”, 2015 https://doi.org/10.48550/arXiv.1505.06550

Encoding of counted K-mers

K-mers are useful in many genomic applications: genome assembly, error correction, repeat detection, ...

K-mers and their counts sometimes cannot fit into memory directly; e.g., the 31-mers with frequency ≥ 2 in HapMap sample NA12878 is 90GB

How to encode K-mers and their counts so that they can be used in memory at will?

Good to read, for counted K-mer encoding

K-mer sparsification

Pellow et al., “Improving Bloom filter performance on sequence data using k-mer Bloom filters”, *JCB* 24(6):547-557, 2017
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5467106/

kmcEx

Jiang et al., “kmcEx: Memory-frugal and retrieval efficient encoding of counted k-mers”, *Bioinformatics* 35(23):4871-4878, 2019
https://doi.org/10.1093/bioinformatics/btz299