CS4330: Combinatorial Methods in Bioinformatics

K-mers counting in memory

Wong Limsoon

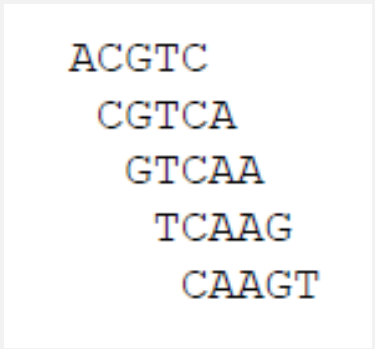
Acknowledgement: This set of slides were adapted from Ken Sung's

K-mer counting

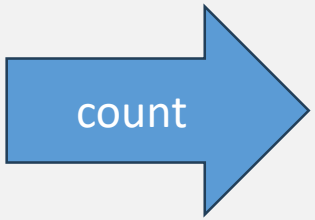
Given a set Z of K-mers that appear in a set of reads $\mathcal{R}$

Count # of occurrences of each K-mer in Z in $\mathcal{R}$

Let N = | Z |



$\mathcal{R}$, a set of 5 reads



k-mer	count
ACG	1
CGT	2
GTC	3
TCA	3
CAA	3
AAG	2
AGT	1

Reverse complement

DNA is double-stranded

The double-stranded DNA below can be read as CGT or ACG:

```
5′-CGT-3′
3′-GCA-5′
```

The lexicographically smaller one is chosen as the canonical form

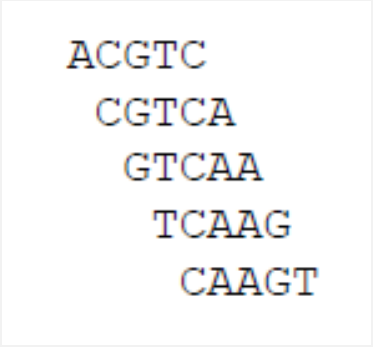
For the example above, the canonical form is ACG

K-mer counting, considering canonical form

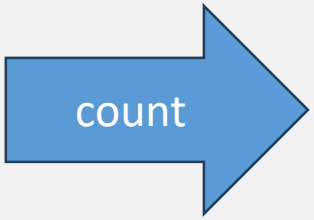
Given a set Z of K-mers that appear in a set of reads $\mathcal{R}$

Count # of occurrences of canonical K-mers in Z in R

Let N = | Z |



$\mathcal{R}$, a set of 5 reads



k-mer	count
ACG	1
CGT	2
GTC	3
TCA	3
CAA	3
AAG	2
AGT	1

noncanonical

k-mer	count
ACG	3
GAC	3
TCA	3
CAA	3
AAG	2
ACT	1

canonical

Exercise

For counting canonical K-mers, would it be better to use odd K or even K? Why?

Counting techniques

Hashing (used by Jellyfish)

Radix sorting (used by KMC & KMC2)

Counting Bloom filter (used by BFCounter)

Burst ties (used by KCMBT)

Enhanced suffix array (used by Tallymer)

Check out the Wikipedia entry on hash tables, linear probing, etc. if anyone does not know these mean …

Hash with count table of size 4^k

Build hash table Count[ ]

Initialize Count[t] = 0 for every K-mer t

For every read and every K-mer t occurrence in the read

    Count[t] ++

Report t and Count[t] for each Count[t] > 0

Time complexity = $O(4^K + N)$, N = # of K-mer occurrences

Space complexity = $O(4^K)$

When K is large, this needs too much space

Example

```
S=AGCAAGCTACC
  AGC          → AGC (9)
   GCA         → GCA (36)
    CAA        → CAA (16)
     AAG       → AAG (2)
      AGC      → AGC (9)
       GCT     → AGC (9)
        CTA    → CTA (28)
         TAC   → GTA
          ACC  → ACC (5)
```

Exercise ☺

canonical

hash value

i	3-mer	Count[i]
0	AAA	0
1	AAC	0
2	AAG	1
…	…	
5	ACC	1
…	…	
9	AGC	3
…	…	
16	CAA	1
…	…	
28	CTA	1
…	…	
36	GCA	1
…	…	
44	GTA	1
…	…	
63	TTT	0

Jellyfish: Using smaller hash table

Build hash table H[1..N/α] and count table C[1..N/α], where α = *Load factor, C[i] stores count of K-mer H[i]*

Hash each K-mer into H[] using hash function h()

Resolve collision by linear probing

When $\alpha < 0.7$, expected # of collisions is low

Expected time complexity = O(N)

Space complexity = (2K + 32) N / α bits

H[]

C[]

Used by Jellyfish with bells & whistles

Exercise

Table size = 11

h(z) = z mod 11

```
S=TAGCAAGCTACC
  TAG          → CTA (28)   h(28)=6
   AGC         → AGC (9)    h(9)=9
    GCA        → GCA (36)   h(36)=3
     CAA       → CAA (16)   h(16)=5
      AAG      → AAG (2)    h(2)=2
       AGC     → AGC (9)    h(9)=9
        GCT    → AGC (9)    h(9)=9
         CTA
          TAC
           ACC
```

Fill in these

i	H[i]	Count[i]
0	GTA	1
1		
2	AAG	1
3	GCA	1
4		
5	CAA	1
6	CTA	1
7		
8		
9	AGC	1+1+1
10		

Lock-free hash table access in Jellyfish enables high parallelism

"Compare and swap" (CAS) assembly instruction in modern multicore CPU

```
1 currentvalue ← read at location;
2 if currentvalue = oldvalue then
3 |   set location to newvalue;
4 end
5 return currentvalue
```

Algorithm 1. CAS(*location*, *oldvalue*, *newvalue*)

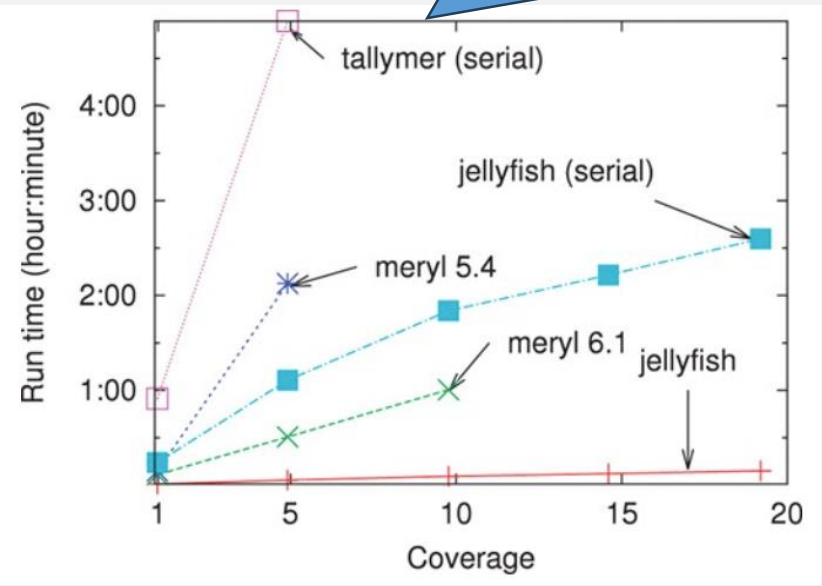
Data: K the array where the keys are stored
Data: V the array where the values are stored
// Claim key
1 $i \leftarrow 0$
2 **repeat**
3 | **if** $i \geq max\_reprobe$ **then return False**
4 | $x \leftarrow \text{pos}(key, i)$
5 | $i \leftarrow i+1$
6 | $current\_key \leftarrow \text{CAS}(K[x], \text{EMPTY}, key)$
7 **until** $current\_key = EMPTY$ **or** $current\_key = key$
// Increment value
8 $cval \leftarrow V[x]$
9 **repeat**
10 | $oval \leftarrow cval$
11 | $cval \leftarrow \text{CAS}(V[x], oval, oval + value)$
12 **until** $cval = oval$
13 **return True**

Algorithm 2. Increment(*key*, *value*)

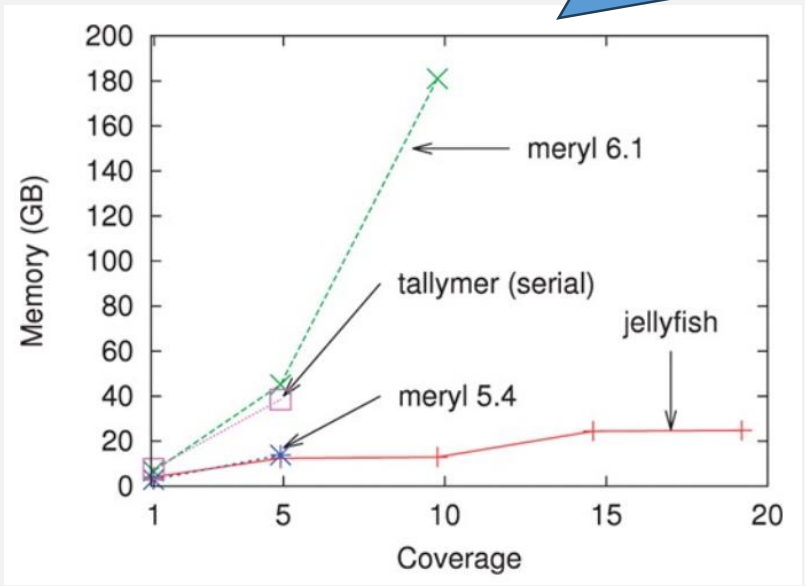
Marcais & Kingsford, "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers", *Bioinformatics* 27(6):764-770, 2011

Performance of Jellyfish

Jelyfish benefits from lock-free hash table access



Jelyfish benefits from smaller hash table



Computation time & memory usage for various levels of sequencing coverage on reads generated during the Turkey genome project when counting 22-mers. Except for Tallymer which is single-threaded, all programs were ran using 32 threads.

Marcais & Kingsford, "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers", *Bioinformatics* 27(6):764-770, 2011

Check out the Wikipedia entry on radix sort, if anyone does not know what this is …

LSD radix sort

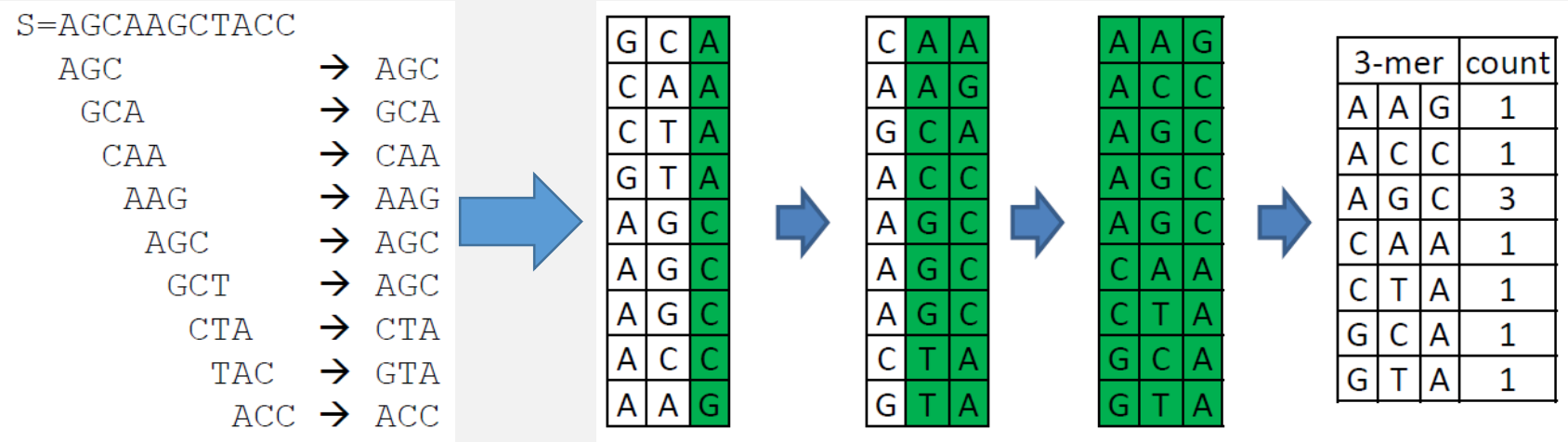
Sort K-mers from LSD to MSD

Time complexity = O(NK)

Slow for big K

Used in KMC & KMC2

1. **Distribute:**
   - Elements are initially distributed into buckets based on the least significant digit.
   - The order of elements in each bucket is preserved.
2. **Collect:**
   - Elements are then collected back from the buckets, and the process is repeated for the next significant digit.
3. **Repeat:**
   - The distribution and collection steps are repeated for each digit, moving from the least significant to the most significant.
4. **Result:**
   - After processing all digits, the array is sorted.



Deorowicz et al., "Disk-based k-mer counting on a PC", *BMC Bioinformatics* 14:160, 2013

MSD radix sort

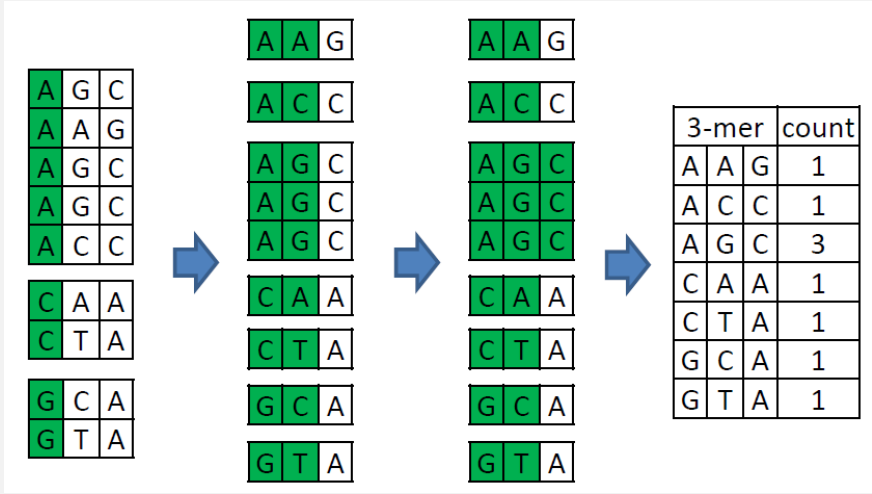
Sort K-mers from MSD to LSD

Time complexity = O(NK), but ideal case is $\Omega(N \log_4 N)$

Used in KMC3

1. **Distribute:**
   - Elements are initially distributed into buckets based on the most significant digit.
   - The order of elements in each bucket is preserved.
2. **Sort Buckets:**
   - Each bucket is recursively sorted using MSD radix sort.
   - The process continues until all digits are considered.
3. **Collect:**
   - Elements are collected back from the buckets in a way that preserves the order of the digits.
4. **Repeat:**
   - The process is repeated for each digit, moving from the most significant to the least significant.
5. **Result:**
   - After processing all digits, the array is sorted.

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



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	11994/12730	62	364	11968/12469
Jellyfish 2	84	251	38338/20284	104	636	31783/31345
KMC 2	64	551	10777/9036	72	381	13774/11804
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

Counting Bloom filter

Some applications only interested in K-mers occurring at least q times

Can use counting Bloom filters for counting

e.g., BFCounter

Melsted & Pritchard, "Efficient counting of k-mers in DNA sequences using a bloom filter", *BMC Bioinformatics* 12:333, 2011

Bloom filter

Consider a set X of elements

And we want to support these operations on X:

Insert(w, X) – insert element w into X

Query(w, X) – check whether X contains an element w

When | X | is small, maintain it using hash table

When | X | is big, hash table cannot fit into memory; maintain it using Bloom filter

A Bloom filter is a space-efficient probabilistic data structure designed for membership testing

Bloom filter consists of
Bit array B[0...n –1]
Hash functions h_1, h_2, …, h_k

Initialization: B[j] = 0 for $0 \le j < n$

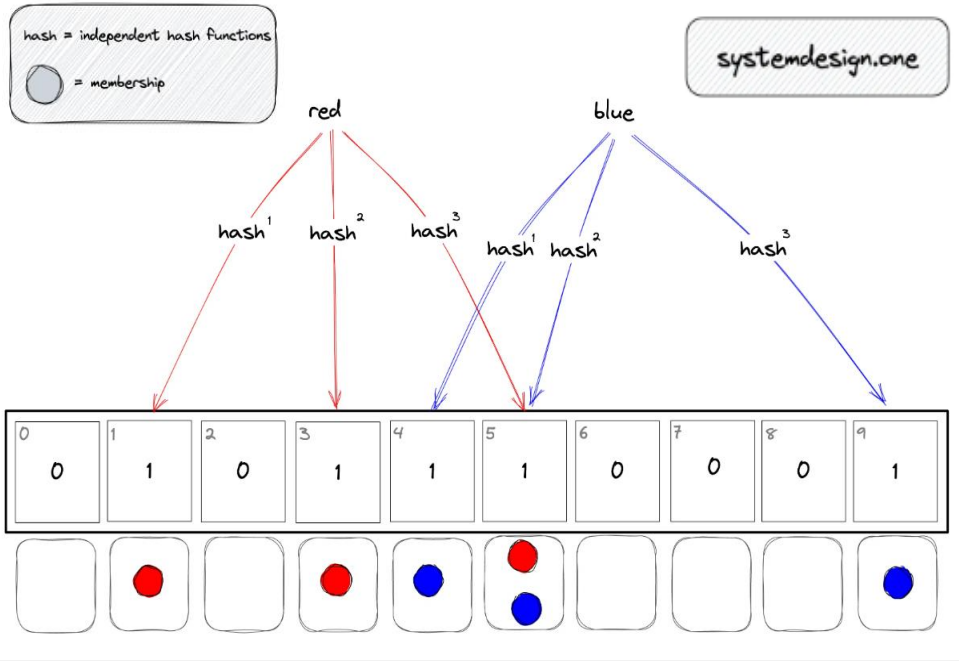
Insert(w, X): $B[h_i(w)] = 1$ for i = 1, 2, …, k

Query(w, x): $\prod_{i = 1, 2, \ldots, k} B[h_i(w)]$

Insert/query is O(1) time

May give false positive but never give false negative

- **Basic Idea:**
  - At its core, a Bloom filter consists of an array of bits and multiple hash functions. This combination allows it to represent set membership in a highly efficient manner.
- **Multiple Hash Functions:**
  - The key innovation of Bloom filters lies in the use of multiple hash functions. Each element is hashed by several independent hash functions, and the corresponding bits in the array are set to 1.
- **Purpose of Multiple Hash Functions:**
  - The use of multiple hash functions is essential to minimize the risk of false positives. By distributing the bits across the array using different hash functions, Bloom filters achieve a balanced and efficient representation of set membership.



Example

Assume k=3, n=13

Initialization:

B[j] = 0 for 0 ≤ j < 13

Hash functions:

$h_1[w] = w \bmod 13$

$h_2[w] = w^2 \bmod 13$

$h_3[w] = (w + w^2) \bmod 13$

	0	1	2	3	4	5	6	7	8	9	10	11	12
B	0	0	0	0	0	0	0	0	0	0	0	0	0

(a)

Insert 12

	0	1	2	3	4	5	6	7	8	9	10	11	12
B	1	1	0	0	0	0	0	0	0	0	0	0	1

(b)

Insert 4

	0	1	2	3	4	5	6	7	8	9	10	11	12
B	1	1	0	1	1	0	0	1	0	0	0	0	1

(c)

Insert 31

	0	1	2	3	4	5	6	7	8	9	10	11	12
B	1	1	0	1	1	1	0	1	0	0	0	0	1

(d)

Insert 27

	0	1	2	3	4	5	6	7	8	9	10	11	12
B	1	1	1	1	1	1	0	1	0	0	0	0	1

(e)

Example

X = { 4, 12, 27, 31 }

	0	1	2	3	4	5	6	7	8	9	10	11	12
B	1	1	1	1	1	1	0	1	0	0	0	0	1

Query(4, X):

$h_1(4) = 4 \bmod 13 = 4$

$h_2(4) = 4^2 \bmod 13 = 3$

$h_3(4) = (4^2 + 4) \bmod 13 = 7$

$B[4] = B[3] = B[7] = 1 \Rightarrow Yes$

Correct!

Query(35, X):

$h_1(35) = 35 \bmod 13 = 9$

$h_2(35) = 35^2 \bmod 13 = 3$

$h_3(35) = (35^2 + 35) \bmod 13 = 12$

$B[9] = 0 \Rightarrow No$

Correct!

Query(40, X):

$h_1(40) = 40 \bmod 13 = 1$

$h_2(40) = 40^2 \bmod 13 = 1$

$h_3(40) = (40^2 + 40) \bmod 13 = 2$

$B[1] = B[2] = 1 \Rightarrow Yes$

Wrong!

False positive rate

Useful identity:

$$\lim_{m\to\infty}\left(1-\frac{1}{m}\right)^{m}=\frac{1}{e}$$

Assume the hash functions are truly random

Prob(B[j] = 0) = $(1 – 1/n)^{km}$

Prob(B[j] = 1) = $1 – (1 – 1/n)^{km}$

= $1 – ((1 – 1/n)^{n})^{km/n}$

≈ $1 – e^{–km/n}$, for large n

n = size of hash table
k = # of hash functions
m = # of elements inserted

False positive exists if w ∉ X and $\Pi_{i = 1..k}$ $B[h_i(w)] = 1$

Prob($\Pi_{i = 1..k}$ $B[h_i(w)] = 1$) ≈ $(1 – e^{–km/n})^{k}$

∴ When km << n, false positive rate is low

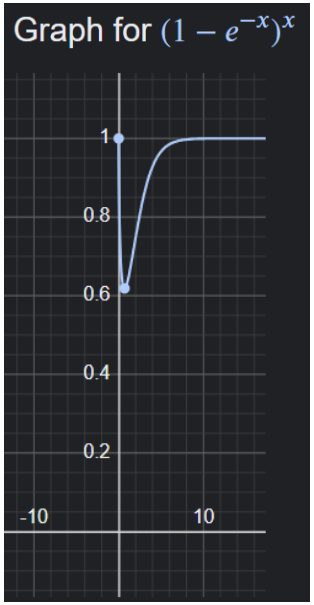
Exercise

n = size of hash table
k = # of hash functions
m = # of elements inserted

False positive rate $\varepsilon = (1 - e^{-km/n})^k$

It is minimized when $k = (n/m)(\ln 2)$

What is the optimal number of mutually independent hash functions to achieve a target false positive rate ε ?



Exercise

Suppose a Bloom filter stores K-mers from a set of sequencing reads

Can you reduce the number of false positives returned by this Bloom filter without using additional data structures or storing extra information?

Counting Bloom filter

Consider a bag X of elements

And we want to support these operations on X:

Insert(w, X) – insert an element w into X

Delete(w, X) – remove an element w from X

Count(w, X) – count occurrences of w in X

Counting Bloom filter can be used

Counting Bloom filter extends Bloom filter to maintain counters for each element

Counting Bloom filter consists of:

Hash functions h_1, h_2, .., h_k

Integer array B[0..n – 1]

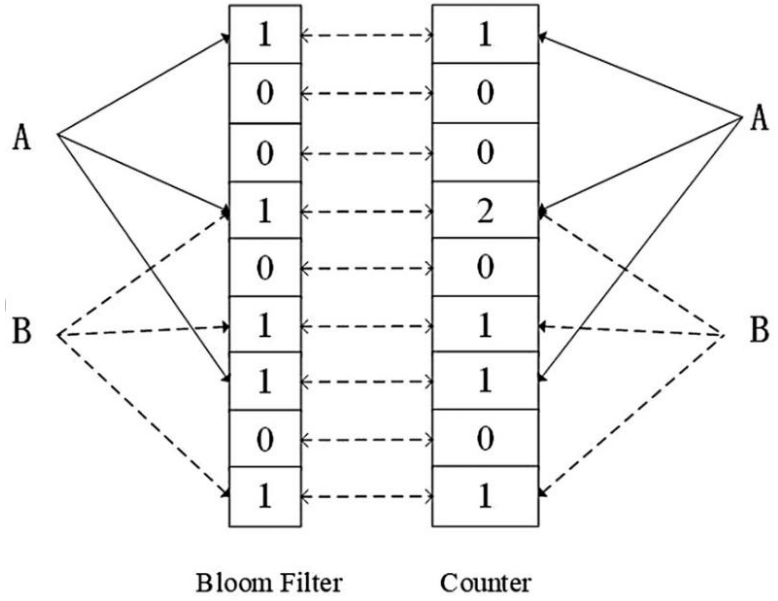
Initialization: B[j] = 0 for $0 \le j < n$

Insert(w, X): $B[h_i(w)] \mathrel{+}= 1$, $i = 1..k$

Delete(w, X): $B[hi(w)] \mathrel{-}= 1$, $i = 1..k$

Query(w, X): $\min_{i = 1..k} B[h_i(w)]$

- **Basic Idea:**
  - The fundamental concept remains similar to a Bloom filter, where elements are hashed using multiple hash functions. However, instead of a binary 'presence or absence,' Counting Bloom Filters maintain a count of how many times each element has been inserted.
- **Array of Counters:**
  - The core structure consists of an array of counters. Each counter corresponds to a position in the array, and the hash functions determine which counters are incremented or decremented during insertions and deletions.



Example

Assume k=3, n=13

Initialization:

B[j] = 0 for 0 ≤ j < 13

Hash functions:

$h_1[w] = w \bmod 13$

$h_2[w] = w^2 \bmod 13$

$h_3[w] = (w + w^2) \bmod 13$

	0	1	2	3	4	5	6	7	8	9	10	11	12
B	0	0	0	0	0	0	0	0	0	0	0	0	0

(a)

Insert 4

	0	1	2	3	4	5	6	7	8	9	10	11	12
B	0	0	0	1	1	0	0	1	0	0	0	0	0

(b)

Insert 31

	0	1	2	3	4	5	6	7	8	9	10	11	12
B	0	0	0	1	2	1	0	1	0	0	0	0	1

(c)

Insert 4

	0	1	2	3	4	5	6	7	8	9	10	11	12
B	0	0	0	2	3	1	0	2	0	0	0	0	1

(d)

Delete 31

	0	1	2	3	4	5	6	7	8	9	10	11	12
B	0	0	0	2	2	0	0	2	0	0	0	0	0

(e)

Example

X = { 4, 4, 31 }

	0	1	2	3	4	5	6	7	8	9	10	11	12
B	0	0	0	2	3	1	0	2	0	0	0	0	1

Count(4, X):

$h_1(4) = 4 \bmod 13 = 4$

$h_2(4) = 4^2 \bmod 13 = 3$

$h_3(4) = (4^2 + 4) \bmod 13 = 7$

$min \{ B[4], B[3], B[7] \} = 2$

Count(31, X):

$h_1(31) = 31 \bmod 13 = 5$

$h_2(31) = 31^2 \bmod 13 = 12$

$h_3(31) = (31^2 + 31) \bmod 13 = 4$

$min \{ B[5], B[12], B[4] \} = 1$

Query(33, X):

$h_1(33) = 33 \bmod 13 = 7$

$h_2(33) = 33^2 \bmod 13 = 10$

$h_3(33) = (33^2 + 33) \bmod 13 = 0$

$min \{ B[7], B[10], B[0] \} = 0$

BFCounter

BFCounter reports counts of K-mers occurring ≥ q times in a set of reads S

Create empty counting Bloom filter X & empty hash table H

For every K-mer w occurrence in S:

If Count(w, X) < q – 1, then insert(w, X)

If Count(w, X) = q – 1, then insert(w, X); H[w] = q

If Count(w, X) = q, then H[w]++

Return H

Melsted & Pritchard, "Efficient counting of k-mers in DNA sequences using a bloom filter", *BMC Bioinformatics* 12:333, 2011

Exercise

What is the advantage of BFCounter as opposed to the following codes?

Create empty hash tables G and H

For each K-mer w occurrence in S: G[w]++

For each entry w in G: if G[w] $\geq$ q, then H[w] = G[w]

Return H

Exercise

Let w = a K-mer,

Let X = a counting Bloom filter constructed from reads S

Suppose Count(w, X) < q

Could w have occurred q or more times in S?

Suppose Count(w, X) ≥ q

Could w have occurred less than q times in S?

Good to read

Jellyfish

Marcais & Kingsford, “A fast, lock-free approach for efficient parallel counting of occurrences of k-mers”, *Bioinformatics* 27(6):764-770, 2011.
https://doi.org/10.1093/bioinformatics/btr011

BFCounter

Melsted & Pritchard, “Efficient counting of k-mers in DNA sequences using a bloom filter”, *BMC Bioinformatics* 12:333, 2011
https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-12-333

Good to read

Radix sort according to Wikipedia

https://en.wikipedia.org/wiki/Radix_sort

Bloom filter according to Wikipedia

https://en.wikipedia.org/wiki/Bloom_filter