meryl

Release 1.3

unknown

Sep 25, 2023

CONTENTS

1 Quick Start **1**

2 What is Meryl? **9**

3 Databases **11**

4 Counting K-mers **13**
4.1 Counting Small k-mers (k < 17) . . . 13
4.2 Counting Large k-mers (k > 15) . . . 14

5 Actions **15**

6 Assignments **21**
6.1 Value Assignment . . . 21
6.2 Label Assignment . . . 22

7 Selectors **25**
7.1 Value Selectors . . . 25
7.2 Label Selectors . . . 26
7.3 Base Composition Selectors . . . 31
7.4 Input Selectors . . . 32

8 Processing Trees **33**

9 Command Line Options **35**
9.1 meryl2 Global Options . . . 35
9.2 meryl2-lookup Usage . . . 36

10 History **39**

11 Frequently Asked Questions **53**
11.1 No questions! . . . 53

12 Publications **55**

13 Install **57**

14 Learn **59**

CHAPTER

ONE

QUICK START

To give a quick introduction to meryl2, we'll use three random Escherichia coli assemblies pulled from GenBank. They're not *quite* random; they're flagged as `complete` and have the highest number of `scaffolds` - which, we hope, means they have a complete nuclear genome and a few handfuls of plasmids.

We'll use EC931, SCEC020022 and MSB1_4I-sc-2280412, but give them shorter names for convenience.

Listing 1: Downloading data.

```
curl -LRO ftp://ftp.ncbi.nlm.nih.gov/genomes/all/GCA/014/522/225/GCA_014522225.1_
↪ASM1452222v1/GCA_014522225.1_ASM1452222v1_genomic.fna.gz
curl -LRO ftp://ftp.ncbi.nlm.nih.gov/genomes/all/GCA/002/165/095/GCA_002165095.2_
↪ASM216509v2/GCA_002165095.2_ASM216509v2_genomic.fna.gz
curl -LRO ftp://ftp.ncbi.nlm.nih.gov/genomes/all/GCA/905/071/835/GCA_905071835.1_MSB1_4I/
↪GCA_905071835.1_MSB1_4I_genomic.fna.gz

mkdir data

mv -i GCA_014522225.1_ASM1452222v1_genomic.fna.gz data/ec.fna.gz
mv -i GCA_002165095.2_ASM216509v2_genomic.fna.gz  data/sc.fna.gz
mv -i GCA_905071835.1_MSB1_4I_genomic.fna.gz      data/ms.fna.gz
```

First, lets count the kmers in each and save the results in meryl databases.

Listing 2: Counting k-mers in a single file.

```
% meryl2 -k 42 count data/ec.fna.gz output=ec.meryl

Found 1 command tree.

|- TREE 0: action #1 count kmers
   |> database 'ec.meryl'
   |- SET value to that of the kmer in the first input
   |- SET label to first -- constant 0
   ^- INPUT @1: sequence file 'data/ec.fna.gz'
|- TREE 0 ends.

Counting 4475 (estimated) thousand canonical 42-mers from 1 input file:
    sequence-file: data/ec.fna.gz

[...]
```

We can show the k-mers in the database by printing the output to the screen (or to a file with *print=42-mers.txt*).

Listing 3: Displaying k-mers.

```
% meryl2 -t 1 -Q print ec.meryl | head

AAAAAAAAAAACGGATCTGCATCACAACGAGAGTGATCCCAC      1
AAAAAAAAAACTTCCACCAATATGATGGGTGGCGTACAGTAA      1
AAAAAAAAAACGGATCTGCATCACAACGAGAGTGATCCCACC      1
AAAAAAAAAATAGACAAAAAATACTTTATCAAAACATACATA      1
AAAAAAAAACCATCCAAATCTGGATGGCTTTTCATAATTCTG      1
AAAAAAAAACCCGATTTTATCAGGCGTCAACCTCTGAATTGT      1
AAAAAAAAACCCGCTGATTAAGCGGGTTTTGAATTCTTGCTG      1
AAAAAAAAACCTGAAAAAACGGCCTGACGTGAATCAAGCAAT      1
AAAAAAAAACCTGCCATCGCTGGCAGGTTTTTTATGACTAAA      1
AAAAAAAAACCGACCAAGGGTCGGGGCAAGAATCAGAGTCTG      1
```

(Messy detail: Option *-Q* turns off the report of the command tree. Option *-t 1* is supplied to prevent meryl from processing its 64 data chunks in parallel; for the print operation this results in the output kmers being unsorted. See {INSERT LINK TO THREADS AND PRINT HERE}.)

Let's now count the other two databases and combine the kmers from all three genomes into one database, all in one command.

Listing 4: Counting two FASTA files and merging with a third database.

```
% meryl2 \
    union-sum \
    output=union.meryl \
    [count data/sc.fna.gz output=sc.meryl] \
    [count output=ms.meryl \
           data/ms.fna.gz] \
    ec.meryl

Found 1 command tree.

|- TREE 0: action #1 filter kmers
   |> database 'union.meryl'
   |- SET value to the sum of all kmers and constant 0
   |- SET label to or -- constant 0
   |- FILTER 1
      |- EMIT if <index filter not described>
   ^- INPUT @1: action #2 count kmers
      |> database 'sc.meryl'
      |- SET value to that of the kmer in the first input
      |- SET label to first -- constant 0
      ^- INPUT @1: sequence file 'data/sc.fna.gz'
   ^- INPUT @2: action #3 count kmers
      |> database 'ms.meryl'
      |- SET value to that of the kmer in the first input
      |- SET label to first -- constant 0
      ^- INPUT @1: sequence file 'data/ms.fna.gz'
   ^- INPUT @3: meryl database 'ec.meryl'
|- TREE 0 ends.

[...]
```

There's a lot going on here. There are three *actions* (lines 11, 17 and 22), three *output* files (lines 12, 18 and 23), and three *inputs* (lines 17, 22, and 27). Each action decribes how to combine the k-mers in its inputs into an output k-mer, then passes those k-mers to its destination actions. Here, *action #1* takes input k-mers from *INPUT @1* (which itself is *action #2*), *INPUT @2* (*action #3*) and *INPUT @3* (which is a pre-computed database of k-mers).

The *union-sum* action in the command line is *action #1* in the tree. Lines 13 and 14 describe how this action is computing the output value and label of each k-mer. Line 16 will {EVENTUALLY} describe the conditions that must be met for a k-mer to be output. For *union*, there is only one condition: the k-mer must be in at least one input.

The end result of this is to independently count kmers in *sc.fna.gz* and *ms.fna.gz*, writing the kmers from each to outpout databases {SEE COUNTING} *sc.meryl* and *ms.meryl*, respectively. When the counting operations are done, those two new databases and the third pre-computed database are sent as input to the first action which will combine all k-mers, summing their values, into one output.

Notice that the *-k 42* option is not present. Meryl will determine the k-mer size from the *ec.meryl* input database and use that for all operations. However, if a k-mer size is supplied, it must match the size of ALL input databases, and ALL input databases must have k-mers of the same size.

A meryl database also stores the histogram of kmer values. This can be displayed:

Listing 5: A k-mer value histogram.

```
% meryl2 -Q histogram ec.meryl

1       4911809
2       37336
3       7632
4       1217
5       2705
6       2232
7       4544
8       384
9       862
11      3
12      4
15      967
16      230
18      1
19      1
21      81
22      3
27      39
29      4
30      21
31      19
32      5
37      39
48      3
49      42
50      38
51      15
52      493
53      13
```

Which hints there is a 52 copy repeat of around 500 bases in Escherichia coli EC931. Histograms from the other two genomes show either no high copy repeat (Escherichia coli SCEC020022, *sc.meryl*) or a potential 64 copy repeat

(Escherichia coli MSB1_4I-sc-2280412, *ms.meryl*). Let's now extract those kmers and see where they are on the genomes.

Listing 6: Extracting high-value k-mers.

```
% meryl2 print=ec-repeats.dump at-least 48 ec.meryl output=ec-repeats.meryl

Found 1 command tree.

|- TREE 0: action #1 filter kmers
   |> database 'ec-repeats.meryl'
   |> text file 'ec-repeats.dump'
   |- SET value to that of the kmer in the first input
   |- SET label to first -- constant 0
   |- FILTER 1
      |- EMIT if output kmer value     is-more-or-equal constant value 48
   ^- INPUT @1: meryl database 'ec.meryl'
|- TREE 0 ends.

[...]

% wc -l ec-repeats.dump
     604 ec-repeats.dump
```

The *meryl-lookup* tool compares FASTA/FASTQ sequences against a meryl database (or several databases) and generates various reports about how the k-mers in the database(s) "paint" onto the input sequences. We'll use it to generate a bed file of the bases covered by kmers in our E.coli database.

Listing 7: Finding runs of high-value k-mers in a genome.

```
% meryl2-lookup -sequence data/ec.fna.gz -mers ec-repeats.meryl -bed-runs > ec-repeats.
↪bed
--
-- Estimating memory usage for 'ec-repeats.meryl'.
--

 p       prefixes             bits gigabytes (allowed: 63 GB)
-- -------------- ---------------- ---------
 2              4            53408     0.000
 3              8            53060     0.000
 4             16            52968     0.000
 5             32            53388     0.000
 6             64            54832     0.000 (smallest)
 7            128            58324     0.000
 8            256            65912     0.000
 9            512            81692     0.000 (faster)
10           1024           113856     0.000
11           2048           178788     0.000
12           4096           309256     0.000
13           8192           570796     0.000
-- -------------- ---------------- ---------
              604 total kmers

--
```

(continues on next page)

(continued from previous page)

```
-- Minimal memory needed: 0.000 GB
-- Optimal memory needed: 0.000 GB  enabled
-- Memory limit           63.907 GB
--
--
-- Loading kmers from 'ec-repeats.meryl' into lookup table.
--

For 604 distinct 42-mers (with 9 bits used for indexing and 75 bits for tags):
    0.000 GB memory for kmer indices -          512 elements 64 bits wide)
    0.000 GB memory for kmer tags    -          604 elements 75 bits wide)
    0.000 GB memory for kmer values  -          604 elements  6 bits wide)
    0.000 GB memory

Will load 604 kmers.  Skipping 0 (too low) and 0 (too high) kmers.
Allocating space for 16732 suffixes of 75 bits each -> 1254900 bits (0.000 GB) in blocks␣
↪of 32.000 MB
                     16732 values   of 6 bits each -> 100392 bits (0.000 GB) in blocks␣
↪of 32.000 MB
Loaded 604 kmers.  Skipped 0 (too low) and 0 (too high) kmers.
-- Opening input sequences 'data/ec.fna.gz'.
-- Opening output file '-'.
Bye!

% head ec-repeats.bed
CP049118.1      46087   46370
CP049118.1      46409   46729
CP049118.1      46729   46856
CP049118.1      140430  140592
CP049118.1      140592  140713
CP049118.1      140752  141072
CP049118.1      141072  141199
CP049118.1      270969  271131
CP049118.1      271131  271252
CP049118.1      271291  271611
```

From this we see that there is not a single 52-copy repeat, but several shorter repeats. Curiously, there are several instances of a 447 base repeat with single base differences:

Listing 8: Curiously similar repeats.

```
CP049118.1      3782667 3782794
CP049118.1      3782794 3783114

CP049118.1      3762937 3763257
CP049118.1      3763257 3763384
```

Though this isn't really part of meryl, the high-count kmers can be passed to a greedy assembler with nice results (the greedy assembler is included in the meryl source code, but isn't installed in the binary directory).

Listing 9: Greedily assembling high-value k-mers.

```
% perl $MERYL/scripts/greedy-assemble-kmers.pl < ec-repeats.dump
>1
AGCCTGTCATACGCGTAAAACAGCCAGCGCTGGCGCGATTTAGCCCCGACATAGCCCCACTGTTCGTCCATTTCCGC
GCAGACGATGACGTCACTGCCCGGCTGTATGCGCGAGGTTACCGACTGCGGCCTGAGTTTTTTAAGTGACGTAAAAT
CGTGTTGAGGCCAACGCCCATAATGCGGGCTGTTGCCCGGCATCCAACGCCATTCATGGCCATATCAATGATTTTCT
GGTGCGTACCGGGTTGAGAAGCGGTGTAAGTGAACTGCAGTTGCCATGTTTTACGGCAGTGAGAGCAGAGATAGCGC
TGATGTCCGGC
>2
ATGGCGACGCTGGGGCGTCTTATGAGCCTGCTGTCACCCTTTGACGTGGTGATATGGATGACGGATGGCTGGCCGCT
GTATGAATCCCGCCTGAAGGGAAAGCTGCACGTAATCAGCAAGCGATATACGCAGCGAATTGAGCGGCATAACCTGA
ATCTGAGGCAGCACCTGGCACGGCTGGGACGGAAGTCGCTGTCGTTCTCAAAATCGGTGGAGCTGCATGACAAAGTC
ATCGGGCATTATCTGAACATAAAACACTATCAATAAGTTGGAGTCATTACC
>3
GTGCTTTTGCCGTTACGCACCACCCCGTCAGTAGCTGAACAGGAGGGACAGCTGATAGAAACAGAAGCCACTGGAGC
ACCTCAAAAACACCATCATACACTAAATCAGTAAGTTGGCAGCATCACC
```

Dropping the kmer threshold to 10 (*at-least 10*) and assembling those repeat k-mers finds 11 repeat sequences, one of length 770 bp and one of length 1195 bp.

For our final example, let's find the high-value k-mers common to EC931 and MSB1_4I-sc-2280412 and assemble those.

Listing 10: Greedily assembling medium-and-high-value k-mers.

```
% meryl2 -Q \
    print \
    intersect \
      [at-least 48 ec.meryl] \
      [at-least 55 ms.meryl] \
  | \
  perl $MERYL/scripts/greedy-assemble-kmers.pl
>1
GGTAATGACTCCAACTTATTGATAGTGTTTTATGTTCAGATAATGCCCGATGACTTTGTCATGCAGCTCCACCGATT
TTGAGAACGACAGCGACTTCCGTCCCAGCCGTGCCAGGTGCTGCCTCAGATTCAGGTTATGCCGCTCAATTCGCTGC
GTATATCGCTTGCTGATTACGTGCAGCTTTCCCTTCAGGCGGGATTCATACAGCGGCCAGCCATCCGTCATCCATAT
CACCACGTCAAAGGGTGACAGCAGGCTCATAAGACGCCCCAGCGTCGCCAT
>2
AGCCTGTCATACGCGTAAAACAGCCAGCGCTGGCGCGATTTAGCCCCGACATAGCCCCACTGTTCGTCCATTTCCGC
GCAGACGATGACGTCACTGCCCGGCTGTATGCGCGAGGTTACCGACTGCGGCCTGAGTTTTTTAAGTGACGTAAAAT
CGTGTTGAGGCCAACGCCCATAATGCGGGCTGTTGCCCGGCATCCAACGCCATTCATGGCCATATCAATGATTTTCT
GGTGCGTACCGGGTTGAGAAGCGGTGTAAGTGAACTGCAGTTGCCATGTTTTACGGCAGTGAGAGCAGAGATAGCGC
TGATGTCCGGC
>3
GTGCTTTTGCCGTTACGCACCACCCCGTCAGTAGCTGAACAGGAGGGACAGCTGATAGAAACAGAAGCCACTGGAGC
ACCTCAAAAACACCATCATACACTAAATCAGTAAGTTGGCAGCATCACC
```

Aside from a strand and sequence order difference caused by the assemlber, they're the same!

Listing 11: The same!

```
% minimap2 --eqx -Y -c ec.fasta ec+ms.fasta
1 282 0 282 - 2 282 0 282 282 282 60 NM:i:0 ms:i:564 AS:i:564 nn:i:0 tp:A:P cm:i:49␣
```

(continues on next page)

(continued from previous page)

```
 ↪s1:i:274 s2:i:0 de:f:0 rl:i:0 cg:Z:282=
2 319 0 319 + 1 319 0 319 319 319 60 NM:i:0 ms:i:638 AS:i:638 nn:i:0 tp:A:P cm:i:58␣
 ↪s1:i:311 s2:i:0 de:f:0 rl:i:0 cg:Z:319=
3 126 0 126 + 3 126 0 126 126 126 60 NM:i:0 ms:i:252 AS:i:252 nn:i:0 tp:A:P cm:i:16␣
 ↪s1:i:115 s2:i:0 de:f:0 rl:i:0 cg:Z:126=
```

Listing 12: But rearranged.

```
% minimap2 -x sr --eqx -Y -c data/ec.fna.gz ec.fasta
1 319 0 319 + CP049118.1 4767973 2542965 2543284 319 319 0 NM:i:0 ms:i:638 AS:i:638␣
 ↪nn:i:0 tp:A:P cm:i:44 s1:i:304 s2:i:304 de:f:0 rl:i:0 cg:Z:319=
2 282 0 282 - CP049118.1 4767973 3430238 3430520 282 282 0 NM:i:0 ms:i:564 AS:i:564␣
 ↪nn:i:0 tp:A:P cm:i:39 s1:i:266 s2:i:266 de:f:0 rl:i:0 cg:Z:282=
3 126 0 126 - CP049118.1 4767973 4539117 4539243 126 126 0 NM:i:0 ms:i:252 AS:i:252␣
 ↪nn:i:0 tp:A:P cm:i:19 s1:i:123 s2:i:123 de:f:0 rl:i:0 cg:Z:126=

% minimap2 -x sr --eqx -Y -c data/ms.fna.gz ec.fasta
1 319 0 319 - LR898874.1 5278133   11725   12044 319 319 0 NM:i:0 ms:i:638 AS:i:638␣
 ↪nn:i:0 tp:A:P cm:i:44 s1:i:304 s2:i:304 de:f:0 rl:i:0 cg:Z:319=
2 282 0 282 - LR898874.1 5278133 1418500 1418782 282 282 0 NM:i:0 ms:i:564 AS:i:564␣
 ↪nn:i:0 tp:A:P cm:i:39 s1:i:266 s2:i:266 de:f:0 rl:i:0 cg:Z:282=
3 126 0 126 - LR898874.1 5278133 1510654 1510780 126 126 0 NM:i:0 ms:i:252 AS:i:252␣
 ↪nn:i:0 tp:A:P cm:i:19 s1:i:123 s2:i:123 de:f:0 rl:i:0 cg:Z:126=
```

We'll stop the quick-start here, before we barrel out of control into repeats.

CHAPTER

TWO

WHAT IS MERYL?

Meryl is a tool for creating and working with DNA sequence k-mers. K-mers are typically created by **counting** how many times each k-mer sequence occurs in some collection of sequences. Meryl refers to this as the **value** of the k-mer. Each k-mer can also be annotated with a **label** of up to 64-bits of arbitrary binary data. The label can be interpreted as a collection of yes/no flags or binary-coded data, for example, 7 bits could be used to indicate which of 7 samples the k-mer is present in, or a set of 10 bits could be used to unary encode the melting temperature of the k-mer. Databases are filtered and combined using **actions**. An action tells how to **assign** the output value and label of a k-mer, and how to **select** desired k-mers for output to an output **database**.

Note: The original meryl used order ACTG for a reason that turned out to be incorrect. It was believed that complementing a binary sequence would be easier in that order, but it is just as easy in the normal order. The revised order does have the appealing property that GC content can be computed by counting the number of low-order bits set in each base, where the more standard ACGT order requires additional operations.

In the revised (ACTG) order, flipping the first bit of each two-bit encoded base will complement the base. This can be done with an exclusive or against b101010.

```
A  C  T  G
00 01 10 11
v| v| v| v| -- flip first bit to complement
10 11 00 01
 T  G  A  C

compl = kmer ^ 0b101010
NumGC = popcount(kmer & b010101)
```

In the usual (ACGT) order, complementation can be accomplished by flipping every bit; an exclusive-or against b111111. GC content can be computed by counting bits also, but we need to count the number of two-bit encoded bases where the first and second bits differ. This requires shifting the encoded k-mer one place, comparing the – now overlapping – adjacent bits with XOR, and finally counting the number of set bits.

```
A  C  G  T
00 01 10 11
vv vv vv vv -- flip every bit to complement
11 10 01 00
 T  G  C  A

compl = kmer ^ 0b111111
NumGC = popcount( (kmer>>1 ^ kmer) & 0b010101 )
```

It is yet to be decided if meryl2 will also use the same order (maintaining compatibility with meryl1) or if it will use

the more typical order (maintaining compatibility with the rest of the world)..

Note: A k-mer is a short sequence of nucleotide bases. The **forward k-mer** is from the supplied sequence orientation, while the **reverse-k-mer** is from the reverse-complemented sequence. The **canonical k-mer** is the lexicographically smaller of the forward-mer and reverse-mer.

For example, the sequence GATCTCA has five forward 3-mers: GAT, ATC, TCT, CTC and TCA. The canonical k-mers are found by reverse-complementing each of those and picking the lexicographically smaller:

Listing 1: Forward, reverse and canonical 3-mers.

```
G
A (GAT, ATC) -> ATC
T (ATC, GAT) -> ATC
C (TCT, AGA) -> AGA
T (CTC, GAG) -> CTC
C (TCA, TGA) -> TCA
A
```

In meryl, k-mers can be up to 64 bases long and are canonical by default.

Given at least one sequence file, meryl will find the list of k-mers present in it and count how many times each one occurs. The count becomes the `value` of the k-mer. These are stored in a meryl database. The example in the sidebar would store:

```
ATC 2
AGA 1
CTC 1
TCA 1
```

While values are typically interpreted as the frequency of the k-mer in some set of sequences, they are simply unsigned 32-bit integers (a maximum value of 4,294,967,295) and cane be used to store any arithmetic data.

The (optional) label of a k-mer can contain up to 64 bits worth of non-arithmetic data. The label can, for example, be used to assign a `type` or `class` to certain k-mers, or to mark k-mers as coming from a specific source, etc. Labels are operated on by the binary logic operations (AND, OR, XOR, NOT) and can also be shifted to the left or right.

The primary purpose of meryl is to combine multiple k-mer databases into a single output database by computing new values and labels and filtering k-mers based on their value, label, base composition and presence or absence from specific databases. It does this by passing each k-mer through a tree of `actions`. A leaf node of the tree reads k-mers from input databases (or by counting k-mers in input sequence files), filtering k-mers via an action, and emitting k-mers to other nodes or output databases.

CHAPTER

THREE

DATABASES

Database Implementation Details

Each k-mer is stored by breaking the binary represntation into three pieces: a file prefix, a block prefix, and a suffix. A k-mer needs 2*k bits to represent it. The file prefix is 6 bits wide, representing one of the 64 possible files in a database. Inside a file, k-mers are stored in blocks, each k-mer in a block will have the same block prefix. The suffix data for a block is stored using Elias-Fano encoding (CITE) where each suffix is split into two pieces. The first piece is encoded as a unary offset from the last piece, and the second piece is a N-log2(N)+1 binary value. At present, values are stored as plain 32-bit integers, stored immediately after the k-mer data.

A set of k-mers, each k-mer with a value and a label, is stored in a **database**. The database is a directory with 129 binary files in it – 64 data files, 64 index files and one master index. This division lets meryl easily process each of these files independently, making effective use of up to 64 compute threads.

Databases also store the k-mer size (**-k** option), label size (**-l** option), and any simple sequence reductions (**–compress** and **–ssr** options) applied. It is not possible to combine databases with different parameters.

Each k-mer is stored at most once per database - thus a k-mer cannot have multiple values of labels associated with it (though we did envision doing this at one time).

CHAPTER

FOUR

COUNTING K-MERS

The **count** action reads sequence from any number of input files and counts the number of times each (canonical) k-mer occurs. Actions **count-forward** and **count-reverse** will instead count k-mers as they are oriented in the input sequence or the reverse-complement sequence, respectively.

Input sequences can be in either FASTA, FASTQ, raw bases, or if compiled with Canu support, in a Canu seqStore database. Sequence files can be gzip, bzip2 or xz compressed.

An output database must be supplied to all count actions. K-mers are both written to the output database and provided as input to destination actions.

Count actions, unless accompanied by an action that reads input from an existing database, MUST specify the k-mer size on the command line with the **-k** option.

Count actions can include a value or label assignment, but cannot include any selectors. A value assignment could be used to assign each k-mer a constant value instead of the count; a label assignment could be used to assign each k-mer a constant representing the input file.

Counting is resource intense. Meryl will use memory and threads up to a limit supplied by: the operating system (usually physical memory and the number of CPUs), a grid manager (such as Slurm, PBS or SGE) or a command line option (**-m** and **-t**).

Two algorithms are used for counting k-mers. The algorithm that is expected to use the least memory is used. The choice depends on the size of the input sequences and the k-mer size.

4.1 Counting Small k-mers (k < 17)

Warning: does count really only use one thread here?

Warning: is this method used even for small amount of input sequence?

Small k-mer Counting Implementation Details

Each integer counter is initially a 16-bit value. Once any count exceeds 2^{16} = 65,535 another bit is added to all value, resulting in 17-bit values for every k-mer. Once any count then exceeds 2^{17} = 131,072, another bit is added, and so on. Thus, memory usage is 512 MB * $\log_2$ maximum_count_value

For k at most 16, meryl counts k-mers directly, that is, by associating an integer count with each possible k-mer. This has the benefit of being simple and uses a constant amount of memory regardless of the size of the input, but quickly exhausts memory for even moderate k-mer sizes.

There are 4^k k-mers of size k; for k=16, there are 4,294,967,296 possible k-mers. Counting 16-mers with this method will use at least 8 GB of memory, independent of input size: counting 16-mers in an E.coli genome will use 8 GB of memory, despite there being only 5 million or so k-mers. Further, memory usage can increase depending on the maximum count value.

This method uses only a single thread to read the input sequence and increment counters in the array, but multiple threads can be used to generate the output database.

4.2 Counting Large k-mers (k > 15)

Large k-mer Counting Implementation Details

Each k-mer is split into a prefix and a suffix. The prefix is used to select a list to which the suffix is added. When the (approximate) size of all lists exceeds a user-supplied threshold, each list is sorted, the suffixes are counted, and this subset of counted k-mers is output to an intermediate database. After all k-mers are processed, the intermediate databases are merged into one.

For k larger than 15, or for small amounts of input sequence, meryl counts k-mers by first converting the sequence to a list of k-mers, duplicates included, then sorts the list, then scans the list to count the number of times each k-mer is present.

If all k-mers in an input sequence do not fit in memory, a partial result is written to disk. After all input sequences have been processed, the partial results are combined into a single output database. In practice, this method requires several additional gigabytes of memory to minimize the overhead of writing and merging partial results.

This method can use multiple threads for every stage.

CHAPTER

FIVE

ACTIONS

Meryl processing is built around **actions**. An action loads a k-mer from one or multiple databases (or, for counting actions, computes the k-mer from a sequence file) computes value and label for it (based on the input kmer values and labels), decides if it should be output or discarded (e.g., "if the new value is more than 100, output the k-mer"), and saves it to an output database or text file or displays it on the screen or passes it to another action for further processing.

An action is specified as an alias (listed below) or by explicitly stating all parameters. The parameters describe:

- how to compute the value of the k-mer
- how to compute the label of the k-mer
- **conditions when a k-mer should be output or discarded:**
  - based on which input databases it came from
  - based on the input and/or output values of the k-mer
  - based on the input and/or output labels of the k-mer
  - based on the sequence of the k-mer
- **what to do with output k-mers**
  - output them to a new database
  - print them to ASCII output files
  - display them on the terminal
  - pass them to other actions

Note: K-mers are read "in order" from the inputs. If an input does not contain the "next" k-mer, it does not participate in the action processing. For example, suppose we have three input databases with the following 4-mers and their counts:

Listing 1: Sample databases.

```
input-1  input-2  input-3
AAAA/1   AAAA/2   AAAA/3
AAAC/1   CAAT/2   CCCC/3
CAAT/1            GGGG/3
GGGG/1
```

A `union-sum` action with these three databases as input will output:

Listing 2: Sample output from union-sum action.

```
AAAA/6 (using the k-mer from input-1, input-2 and input-3)
AAAC/1 (... from input-1)
CAAT/3 (... from input-1 and input-2)
CCCC/3 (... from input-3)
GGGG/4 (... from input-1 and input-3)
```

An **assignment** computes the output value (label) for each k-mer from among the input values (labels). At most one assignment can be supplied for the value and one for the label.

A **selector** decides if the k-mer should be output or discarded. Selectors can use input values (labels), the computed output value (label), the base composition of the k-mer and which specific inputs the k-mer was present in. Any number of selectors can be supplied, linked with **and**, **or** and **not** operators. See SELECTORS.

Though it is possible to specify all those choices explicitly, **aliases** are provided for most common operations.

Aliases exist to support common operations. An alias sets the `value`, `label` and `input` options and so these are not allowed to be used with aliases. Examples of aliases and their explicit configuration:

Table 1: Action Aliases

Alias	Output k-mer if...	Sets value to the...
union	...k-mer is in any input database.	...number of databases the k-mer is in.
union-min		...smallest input value.
union-max		...largest input value.
union-sum		...sum of the input values.
intersect	...k-mer is in all input databases.	...value of the k-mer in the first database.
intersect-min		...smallest input value.
intersect-max		...largest input value.
intersect-sum		...sum of the input values.
subtract	...k-mer is in the first database.	...value of the k-mer in the first database minus all other values.
difference		...value of the k-mer in the first database.
less-than X	...k-mer is in the first and only database and the value meets the speficied condition.	...value of the k-mer.
greater-than X		
at-least X		
at-most X		
equal-to X		
not-equal-to X		
increase X	...k-mer is in the first and only database.	...value of the k-mer modified by the specified operation. (divide-round rounds 0 up to 1)
decrease X		
multiple X		
divide X		
divide-round X		
modulo X		

Warning: This table has not been verified!

Table 2: Action Aliases

Alias	Action				
	Assignment		Selector		
union	value=sum	label=or	input:any	value:	label:
union-min	value=min	label=min-value	input:any	value:	label:
union-max	value=max	label=max-value	input:any	value:	label:
union-sum	value=sum	label=or	input:any	value:	label:
intersect	value=first	label=and	input:all	value:	label:
intersect-min	value=min	label=min-value	input:all	value:	label:
intersect-max	value=max	label=max-value	input:all	value:	label:
intersext-sum	value=sum	label=and	input:all	value:	label:
subtract	value=sub	label=first	input:first	value:	label:
difference	value=sub	label=first	input:first	value:	label:
less-than X	value=first	label=first	input:only	value:<X	label:
greater-than X	value=first	label=first	input:only	value:>X	label:
at-least X	value=first	label=first	input:only	value:>=X	label:
at-most X	value=first	label=first	input:only	value:<=X	label:
equal-to X	value=first	label=first	input:only	value:==X	label:
not-equal-to X	value=first	label=first	input:only	value:!=X	label:
increase X	value=@1+X	label=first	input:only	value:	label:
decrease X	value=@1-X	label=first	input:only	value:	label:
multiply X	value=@1*X	label=first	input:only	value:	label:
divide X	value=@1/X	label=first	input:only	value:	label:
divide-round X	value=@1/X[1]	label=first	input:only	value:	label:
modulo X	value=@1%X	label=first	input:only	value:	label:

A full action is:

Listing 3: Fully general action template.

```
[ action-name
    output:database=<database.meryl>            # to a binary database
    output:list=<files.##.mers>                 # to ascii files
    output:show                                 # to stdout
    output:pipe=<label>                         # to a 'named pipe' action input
    output:histogram=<hist-file>                # compute a histogram
    output:statistics=<stats-file>              # compute statistics
    assign:value=<rule-to-create-output-value>
    assign:label=<rule-to-create-output-label>
    [not] select:value:<rule-to-select-k-mer-for-output> [or | and]  # if none specified
    [not] select:label:<rule-to-select-k-mer-for-output> [or | and]  # and is assumed
    [not] select:bases:<rule-to-select-k-mer-for-output> [or | and]
```

(continues on next page)

[1] The `divide-round` alias rounds values of 0 up to 1.

(continued from previous page)

```
    [not] select:input:<rule-to-select-k-mer-for-output>
    [input:database] input-1                          # from a database or
    [input:list]     input-2                          # list file
    [input:pipe]     input-3                          # 'named pipe' label
    [input:action]   [ action-name ... ]
    ...
]
```

The 'action-name' is used to either set default values from an alias, or to define a new alias (when no inputs are supplied, see ALIASES).

Notice that each parameter is described by a parameter-class ('output') and a parameter-name ('database'). There are four parameter-classes: 'output', 'assign', 'select' and 'input'. Parameter-class 'input' is optional as meryl can query the input source to determine what type it is. Parameter-classes and parameter-names may be shortened to any prefix of the class or name, all the way to a single letter. In some cases, a different word can be substituted, however, these may not be shortened and must be used as shown:

Table 3: Parameter class abbreviations and aliases

Parameter Class	Suggested Abbreviations	Aliases
output	o, out,	
assign	a	set
select	s, sel	get
input	i, inp, in	

Table 4: Parameter name abbreviations and aliases

Parameter Name	Suggested Abbreviations	Aliases
database	d, dat	db
list	l, lis	t, txt, text
show	s, sho	display, dis, print, stdout
pipe	p, pip	
histogram	h, hist, histo	
statistics	st, stats	NOTE: 's' is NOT an abbrev.
action	a, act	
value	v, val	
label	l, lab	
bases	b, bas	acgt, bp
input	i, inp, in	

Note: Selectors are 'positive-sense', meaning they initially assume every k-mer is to be discarded, then 'select' only those k-mers that (positively) match some set of conditions. A 'negative-sense' scheme would initially assume all k-mers are selected, and then de-select those that match the set of conditions. One can always convert selectors between senses, but the two styles cannot be mixed. Here's why.

We'll use 'select' for a positive-sense selector and 'reject' for a negative-sense selector. These following two sets are equivalent:

```
select:value:>=3 and select:value:<=10
reject:value:<3  or  reject:value:>10
```

either will unambiguously discard items with value less than three, accept items with value between 3 and 10, and discard items with value more than 10.

De Morgan rests comfortably; these are all equivalent:

```
1 select:value:>=3 and select:value:<=10
2 not (not (select:value:>=3 and select:value:<=10))
3 not (not select:value:>=3) or (not select:value:<=10)
4 not (select:value:<3) or (select:value:>10)
```

But something odd happens if we replace a single 'select' term with a seemingly equivalent 'reject' term. Replcing the first term (*select:value:>=3*) in the positive-sense selector (line 1 in both examples above) with an 'equivalent' reject term (*reject:value:<3*) results in a contradiction:

```
1 reject:value:<3 or select:value:<=10
```

For values less than three, we're told to both 'reject' and 'select' the k-mer! We could probably devise rules of precedence to resolve this problem, but that would quickly lead to confusion.

A more sinister contradiction is that the 'reject' term implicitly accepts items with value 3 or larger unless some other term says to discard it; the 'select' term does the opposite - it implicitly discards items with value more than ten. Values greater than ten are both (implicitly) discarded and accepted! discarding and implicitly accepting them.

For simplicity, we chose to implement only one sense of select term. It might be possible to later implement the other sense, or even add 'exceptions' of the form *select:value>=3 except reject:label=2*.

Square brackets MUST surround every action (exception: the first action in a command tree can omit the brackets).

`output:database` (`o:d`) is optional, but may occur at most once per action. If present, the k-mers generated by this action will be written to the specified meryl database. If an existing database is supplied, it will be overwritten.

`output:list` (`o:l`) is optional. If present, the k-mers generated by this action will be written to ASCII file(s) in the format `<k-mer><tab><value><tab><label>`, one k-mer per line. The k-mers will be in sorted order: A, C, T, G. If the file name includes the string `##`, the data will be written to 64 files, in parallel, using up to 64 threads. Appending suffix `.gz`, `.bz2` or `.xz` will cause the output file to be compressed.

`output:show` (`o:s`) is optional. It behaves like `output:list`, except the k-mers are written to the `stdout`, the terminal, unless redirected.

`output:pipe` (`o:p`) is optional. If present, the k-mers generated by this acction will be supplied to other meryl actions that read input from the same pipe name.

Note: `listACGT` is the same as `list`, but modifies the ordering of k-mers from `A < C < T < G` to `A < C < G < T` when forming canonical k-mers. While this generates correct canonical k-mers, the output k-mers are not sorted.

Consider 3-mers from string `GGAGAGCT`:

Table 5: ACTG order vs ACGT order

GGAGCT	forward	reverse	canonical k-mer in	
			ACTG order	ACGT order
GGA...	GGA	TCC	TCC	GGA
.GAG..	GAG	CTC	CTC	CTC
..AGC.	AGC	GCT	AGC	AGC
...GCT	GCT	AGC	AGC	AGC

When meryl builds the datase, it uses the `A < C < T < G` order. These k-mers will be stored in the database in order: `AGC`, `CTC`, `TCC`, `GCT`. But when output using `listACGT`, the k-mers will be reported as `AGC`, `CTC`, `GGA`, `GCT`. Notice that because of the change in canonical k-mer from `TCC` to `GGA` the last k-mer is not in sorted order.

`assign:value=` (`a:v=`) and `assign:label=` (`a:l=`) describe how to combine the input values and labels into a single output value and label.

`select:value:` (`s:v`), `select:label:` (`s:l`), `select:bases:` (`s:b`) and `select:input:` (`s:i`) describe the conditions required for a k-mer to output. Any number of these may be supplied.

An `input` is either a meryl database, a list of k-mers, or another meryl action.

Some actions require exactly one input, others require more than one - this is specified in the `select:input:` rule.

CHAPTER

SIX

ASSIGNMENTS

Warning: HOW IS THIS IMPLEMENTED?

When value:#c, value:first, value:min or value:max are used, the label operation acts ONLY on the k-mers matching the value selection. For example, if value:min finds value=5 is the minimu, label=or would combine the labels of all k-mers with value=5. Contrast this with value:add (which would set the output value to the sum of the k-mer values in all databases) and label:and (which would set each bit in the output label to true if the corresponding bit is true in all inputs).

Likewise for label:#c, label:first, label:minweight and label:maxweight. For example, when label:#c is used, value:add would sum the values of all labels that are the same as constant c.

6.1 Value Assignment

A **value assignment** computes the output value of the k-mer based on the input values and possibly a single integer constant.

Note: The optional parameter `(#X)` means to also include constant `X` in the computation.

Note: Constants can be decimal integers (`123` or `123d`), hexadecimal (`abch`), octal (`147o`) or binary (`0101010b`). SI suffixes can be used on plain decimal integers (`123k` == 123,000; `1mi` == 1,048,576). For example, `value=add#10` would set the output value to the sum of the input values plus ten; `value=min#10` would set the output value to the smallest input value or 10 if all input values are larger than 10.

Warning: How to form complex expressions?

Warning: Things like value=@1-@2 are NOT supported. Even the potentially useful value=@1 isn't supported (though it is listed below).

Warning: value=selected isn't implemented.

Table 1: Value Selectors

Assignment	Set value to . . .
value=#X	. . . constant X.
value=@X	. . . that of the k-mer in the Xth input
value=first	. . . that of the k-mer in the first input.
value=selected	. . . that of the k-mer selected by the label= selector. When multiple k-mers are selected, the value of the first is used.
value=min(#X)	. . . the minimum of all input values.
value=max(#X)	. . . the maximum of all input values.
value=add(#X)	. . . the sum of all input values.
value=sum(#X)	. . . the sum of all input values.
value=sub(#X)	. . . the value of the k-mer in the first input minus all other values.
value=dif(#X)	. . . the value of the k-mer in the first input minus all other values.
value=mul(#X)	. . . the product of all input values.
value=div(#X)	. . . the value of the k-mer in the first input divided by all other values.
value=divzero(#X)	. . . the value of the k-mer in the first input divided by all other values, rounding zero up to one.
value=mod(#X)	. . . the remainder after the value of the k-mer in the first input is divided by all other values.
value=rem(#X)	. . . the remainder after the value of the k-mer in the first input is divided by all other values.
value=count	. . . the number of inputs the k-mer is present in.

6.2 Label Assignment

A **label assignment** computes the output label of the k-mer based on the input label and possibly a single 64-bit constant.

Warning: How to form complex expressions?

Table 2: Value Assignments

Assignment	Set label to . . .
label=#X	. . . constant X.
label=@X	. . . that of the k-mer in the Xth input
label=first	. . . that of the k-mer in the first input.
label=selected	. . . that of the k-mer selected by the value= selector. When multiple k-mers are selected, the label of the first is used.
label=min(#X)	. . . the minimum of all input labels.
label=max(#X)	. . . the maximum of all input labels.
label=and(#X)	. . . the bitwise AND of all input labels.
label=or(#X)	. . . the bitwise OR of all input labels.
label=xor(#X)	. . . the bitwise XOR of all input labels.
label=difference(#X)	. . . that of the k-mer in the first input, with all bits set in other inputs turned off.
label=lightest(#X)	. . . the label with the fewest bit set.
label=heaviest(#X)	. . . the label with the most bits set.
label=invert(#X)	. . . the bitwise invert of the first input.
label=shift-left(#X)	. . . the first input shifted left by X places.
label=shift-right(#X)	. . . the first input shifted right by X places.
label=rotate-left(#X)	. . . the first input rotated left by X places.
label=rotate-right(#X)	. . . the first input rotated right by X places.

CHAPTER

SEVEN

SELECTORS

A selector decide if the k-mer should be output. They can use the values and labels of the input k-mers, the computed value and label of the k-mer to be output, the number and location of inputs that supplied an input k-mer, and the base composition of the k-mer. A single select term tests one condition, e.g., `value:>3`, and multiple terms are connected together in a sum-of-products form (e.g., 'and' has higher precedence than 'or'):

Listing 1: Sum-of-Products filters.

```
value:@1>=20 or value:@2>=20 or value:>30 and input:#2
```

will output a k-mer if it has a value of at least 20 in either input database, or the output value is more than 30 and the k-mer occurs in both inputs, or both conditions are met.

The 'not' keyword has highest precedence and can be used to invert the sense of the next term, and only the next term. While this seems restrictive, De Morgan's laws are useful:

Listing 2: De Morgan's laws

```
  not (A and B) = (not A) or  (not B)
  not (A or  B) = (not A) and (not B)
```

Do not confuse selectors (which use a `:` - 'select:value:', 'select:label:', 'select:input:', 'select:bases:') with assignments (which use an = - 'assign:value=' and 'assign:label=').

7.1 Value Selectors

A value selector discards the k-mer from output if the input or output values are undesired. When the selector is TRUE the k-mer is output. The syntax and options are similar to **label selectors**, but value selectors are typically integer functions.

```
value:<ARG1><OP><ARG2>
```

ARG1 and ARG2 can be an input file (`@3`), a constant (`#4` or `4`), a special function (ARG2 only) or empty (ARG1 only).

ARG1	OP	ARG2	Meaning
@n		@n	Use the value from the k-mer in the nth input.
#n or n		#n or n	Use the constant n.
<not-present>			Use the value of the selected output k-mer.
		distinct=f	Use the value such that f fraction of the distinct k-mers have at most this value. That is, value:ge:distinct=0.9 will output the 10% most repetitive k-mers in the database.
		word-freq=f	Compute the word-frequency of a k-mer as its value divided by the total number of k-mers in the database.
		threshold=n	Use the constant n.
	== = eq		TRUE if ARG1 equals ARG2.
	!= <> ne		TRUE if ARG1 does not equal ARG2.
	<= le		TRUE if ARG1 is less than or equal to ARG2.
	>= ge		TRUE if ARG1 is greater than or equal to ARG2.
	< lt		TRUE if ARG1 is less than ARG2.
	> gt		TRUE if ARG1 is greater than ARG2.

Note that @1 is not necessarily the first file supplied to the action. If the k-mer occurs only in the last file, @1 will be the value of the k-mer in that file.

Examples:

	TRUE if . . .
value:>5	. . . the output value is more than 5.
value:@2<=#52o	. . . the value of the second input is at most 52_8 (or 42_{10}).
value:4>@2	. . . 4 is larger than the value of the second input.
value:@1>@2	. . . the value of the first input is more than the second input.

7.2 Label Selectors

A label selector discards the k-mer from output if the input or output labels are undesired. When the selector is TRUE the k-mer is output. The syntax and options are similar to **value selectors**, but label selectors are typically binary functions.

```
label:<ARG1><OP><ARG2>
```

ARG1 and ARG2 can be an input file (@3), a constant (#0100b or 4h), a special function (ARG2 only) or empty (ARG1 only).

ARG1	OP	ARG2	Meaning
@n		@n	Use the label from the k-mer in the nth input.
#n or n		#n or n	Use the constant n.
<not-present>			Use the label of the selected output k-mer.
		distinct=f	Use the label such that f fraction of the distinct k-mers have at most this label.
		word-freq=f	(same, but for total k-mers?)
		threshold=n	Use the constant n.
	== = eq		TRUE if ARG1 equals ARG2.
	!= <> ne		TRUE if ARG1 does not equal ARG2.
	<= le		TRUE if ARG1 is less than or equal to ARG2.
	>= ge		TRUE if ARG1 is greater than or equal to ARG2.
	< lt		TRUE if ARG1 is less than ARG2.
	> gt		TRUE if ARG1 is greater than ARG2.

Note that @1 is not necessarily the first file supplied to the action. If the k-mer occurs only in the last file, @1 will be the label of the k-mer in that file.

Table 1: Proposed Flters

Selector	TRUE if...
label:all#c label:and#c	...all bits set in c are also set in the label equivalent to l & c == c or l | c == l equivalent to ~l & c == 0 (not expressible in meryl)
label:any#c label:or#c	...any bits set in c are also set in the label equivalent to l & c != 0
label:none#c label:not#c	...no bits set in c are set in the label equivalent to l & c == 0
label:only#c label:xor#c ??	...only the bits set in c are set in the label equivalent to l | c == c or l && c == l equivalent to none#~c
label:and#c=d	...not expressible in meryl
label:or#c=d	...not expressible in meryl

Examples:

We want to find k-mers in an that are in none, one or two different read sets. We'll assign distinct indicator bits to each input, union everything together, then pick out k-mers that have the *assembly* indicator set.

Listing 3: Finding unsupported k-mers, version 1.

```
meryl \
  union \
    output=labeled.meryl \
    value=@3 \
    label=or \
    label:and#0b100 \
    [ label=0b001 db1.meryl ] \
    [ label=0b010 db2.meryl ] \
    [ label=0b100 asm.meryl ]
```

The result will by k-mers with the value from the assembly and labeled with:

0b0..	(label never occurs)
0b100	appears only in asm
0b101	appears in asm and db1
0b110	appears in asm and db2
0b111	appears in asm and both db1 and db1

An alternate method is to first intersect the read k-mers with the assembly, then merge those two sets:

Listing 4: Finding unsupported k-mers, version 2.

```
meryl \
  union \
  output=labeled.meryl \
    value=sum \
    label=or \
    [ intersect value=@2 label=0b01 asm.meryl db1.meryl ] \
    [ intersect value=@2 label=0b10 asm.meryl db2.meryl ]
```

The result is slightly different. We no longer output k-mers that exist only in the assembly; the value of k-mers will by the sum of the values in the read databases, and the label will be:

0b00	(label never occurs)
0b01	appears in the assembly and db1
0b10	appears in the assembly and db2
0b11	appears in the assembly and both db1 and db1

Find k-mers with a value of at least ten that exist in two or more databases, report which and how many databases contained the k-mer.

Listing 5: Finding supported k-mers from multiple databases.

```
meryl \
  display \
  value=count \
  label=or \
  inputs:2-4 \
  [ value:>=10 label:0001b a.meryl ] \
  [ value:>=10 label:0010b b.meryl ] \
  [ value:>=10 label:0100b c.meryl ] \
  [ value:>=10 label:1000b d.meryl ]
```

Output a k-mer if it exists in at least three databases with count greater than 100, but output the minimum count the k-mer has in any input database.

Listing 6: Minimum value of well-supported k-mers.

```
meryl \
intersect \
  value:@2 \
  [ input:3-5 \
    [ value:>100 a.meryl ] \
    [ value:>100 b.meryl ] \
    [ value:>100 c.meryl ] \
```

(continues on next page)

(continued from previous page)

```
    [ value:>100 d.meryl ] \
    [ value:>100 e.meryl ] \
  ]
  [ union-min \
    a.meryl \
    b.meryl \
    c.meryl \
    d.meryl \
    e.meryl \
  ] \
```

The first sub-action generates a list of k-mers that are well-supported in at least three inputs. Its sub-actions return lists of k-mers with value greater than 100. The second sub-action returns a list of all k-mers with their actual value. Finally, `intersect` returns a list of k-mers that are both "well-supported in at least three inputs" and "in any input" and sets the output value to whatever was in the second input.

7.2.1 A Generalized Label Selector

A general label selector can be devised by supplying a function that converts each bit in the label to some testable output bit, then testing those output bits.

An example will follow the tables.

The four functions are:

function	code	output value	
zero(bit)	0	0	always false
one(bit)	1	1	always true
pass(bit)	+	bit	true if label is set
flip(bit)	-	!bit	true if label is unset

And the five tests are:

test	code	
all-must-be-true	T	All 'T' bits must be true.
any-must-be-true	t	At least one 't' bit must be true.
any-must-be-false	f	At least one 'f' bit must be false.
all-must-be-false	F	All 'F' bits must be false.
don't care	x	Bit is not tested.

Example: With a five-bit label, the selector `label:+--++:FttTT` will output the k-mer if its label begins with `0`, has at least one `0` in the next two bits, and ends with `11`.

Aliases `all` (all tests are `T`), any (all tests are `t`), and none (all tests are `F`) exist.

The default function is + and the default test is `T`.

A selector `label:0101011` needs to be special case interpreted to mean "the label equals 0101011".

A selector `label:....011` likewise should be special cased to mean "the label ends in 011".

Examples on 2-bit labels:

Selector	Meaning	Label Example			
		00	01	10	11
label:00:all	always false	F	F	F	F
label:11:all	always true	T	T	T	T
label:-+:all	label must be 01	F	T	F	F
label:1+:all	label must be x1	F	T	F	T
label:00:any	always false	F	F	F	F
label:11:any	always true	T	T	T	T
label:-+:any	label cannot be 10	T	T	F	T
label:0+:any	label cannot be x0	F	T	F	T
label:00:none	always true	T	T	T	T
label:11:none	always false	F	F	F	F
label:-+:none	label must be 10	F	F	T	F
label:0+:none	label cannot be x1	T	F	T	F

Examples:

```
'equal' label:+-+-:all   --> true if the label is exactly 1010
'all'   label:+1+1:any   --> true if none of the '+' bits are set
'any'   label:+0+0:any   --> true if any of the '+' bits are set
'none'  label:-1-1:all   --> true if none of the '-' bits are set
'only'  label:1-1-:all   --> true if the '-' bits are all zero
```

Examples:

```
label:101011

  SPECIAL CASE, outputs k-mer if the label is exactly 101011.

label:+-+-++:all

  Outputs k-mer if the label is exactly 101011.

label=and label:+11+:all label:@2:1++1:all

  Compute the output label as the AND of all input labels.
  Require that the output label have the first and last bits set, AND
  require that the label on the second input have the middle two bits set.
```

Some tests that can be implemented:

A. `L == C` or `L != C`

   For testing equality, convert the constant into a string of +'s (for 1 bits) and -'s (for 0 bits) then check that all bits are set.

   For testing non-equality, invert the conversion so that 1's are set to - and 0's to +, then check that any bit is set.

   For testing (non-)equality of only a portion of the label, set the bits that should not be tested to 1 (for equality) or 0 (for non-equality).

   Proof: **not verified recently** if the label is the same as the constant, 1 bits in the label will be inverted (to 0) and 0 bits in the label will be output true (so also 0) resulting in a modified label of all 0's. If a 1 bit in the label

corresponds to a 0 bit in the constant, it will be passed true (a 1 in the modified label), similarly, a if a 0 bit in the label corresponds to a 1 bit in the constant it will be passed inverted (a 1 in the modified label), either of which will make the modified label not equal to zero.

B. `L & C == L` or `L | C == C`

   These test that C dominates L: that every bit set in L is also set in C, equivalently, that there is no bit set in L that is not set in C.

   Convert the constant c into a string of 0's (for 1 bits) and +'s (for 0 bits), then check that no bit is set.

   Where C is a 1, we don't care what L is; 'l&c == l' is true for both l=0 and l=1. By forcing these bits to 0 in the modified string, they will never result in the check failing.

   Where C is a 0, howeveer, L must be 0. Hence, passing the L bit true will result in a 1 bit output when L=1, which will cause the check to properly fail. When L=0, a 0 is outpout, and the check passes.

   If instead of testing that no bit is set, we test that any bit is set, the sense of the selector is inverted; we test that C does not dominate L; that there is a bit set in L that is not set in C.

C. `L & C == C` or `L | C == L`

   This is the dual of case B: L dominates C; that every bit set in C is also set in L.

   Convert the constant c into a string of +'s (for 1 bits) and 1's (for 0 bits), then check that all bits are set.

   The selector is inverted if we test that some testable bit is false.

D. `L & C == 0` or `L & C != 0`

   These test that L and C have no bits in common (or at least one bit in common).

   Convert the constant to +'s for 1's and 0's for 0's, then check that no bit is set (for no bits in common) or that some bit is set (for some bits in common).

7.3 Base Composition Selectors

The base composition selector selects k-mers for output based on the number of A's, C's, G's and T's in the k-mer sequence.

```
bases:<BASES>:<OP><NUMBER>
```

Where <BASES> is a string containing `A`, `C`, `G` and `T` letters; case, order and quantity are unimportant. The selector will count the number of the specified letters in the k-mer and compare aginst <NUMBER> using the specified numeric comparison operator <OP>.

BASES	OP	NUMBER	Meaning
A			Count the number of A's in the k-mer.
AC			Count the number of A's and C's in the k-mer.
GAAGAA			Count the number of A's and G's in the k-mer.
		#n or n	Use the constant n.
	== = eq		TRUE if the number of bases is equal to the constant.
	!= <> ne		TRUE if the number of bases is not equal to the constant.
	<= le		TRUE if the number of bases is less than or equal to the constant.
	>= ge		TRUE if the number of bases is greater than or equal to the constant.
	< lt		TRUE if the number of bases is less than the constant.
	> gt		TRUE if the number of bases is greater than the constant.

7.4 Input Selectors

The input selector selects k-mers for output based on which inputs supplied the k-mer.

```
input:<CONDITION>[:<CONDITION>[...]]
```

A <CONDITION> is either an input number (@n) or input count (n or n-m). For the selector to be TRUE, all the CONDITIONS must be met.

> **Warning:** Do input-counts require #n or just integers n?

Note that a k-mer is always present in at least one input.

Assuming 9 input files, some examples are:

Selector	Output k-mer if it is present in…
input:@1	…the first input file.
input:@1-@3	…the first three input files.
input:#4:#5:@1	…4 or 5 input files, including the first
input:#4-#6:#8	…4 or 5 or 6 or 8 input files.
input:#3-#9	…3 or more input files.
input:#1-#6	…at most 6 input files.

A few aliases exist:

Alias	Selector	Meaning
input:any	input:#1-#9	k-mer is in any number of inputs.
input:all	input:#9	k-mer is in all inputs.
input:only	input:@1:#1	k-mer is in the first input, and in exactly one input.
input:first	input:@1	k-mer is in the first input, and maybe other inputs.

The difference between 'only' and 'first' is subtle: 'only' is true if the k-mer is present exactly only in the firt file, while 'first' is true if the k-mer exists in the first file and any other files. 'only' will effect a set difference action, while 'first' is more akin to a set intersection.

CHAPTER

EIGHT

PROCESSING TREES

Meryl processes k-mers using a tree of actions. An action reads k-mers from multiple inputs, computes a function on the values and labels of all inputs with the same k-mer, and outputs a single k-mer with a single value and a single label.

(An action can also read sequence files and count the k-mers.)

Each action in the tree is enclosed in square brackets. Square brackets around the top-level / outermost action are optional.

The input to an action can be either a meryl database on disk or the output of a different action.

The 'union' action below reads input from meryl databases 'input-1.meryl' and 'input-2.meryl'. All three forms below are equivalent.

Listing 1: A simple union action reading from two inputs.

```
[ union input-1.meryl input-2.meryl ]
```

Listing 2: A simple union action reading from two inputs, but formatted.

```
  union
    input-1.meryl
    input-2.meryl
```

Listing 3: A simple union action reading from two inputs, as sub-actions.

```
  union
    [ input-1.meryl ]   //  This form technically makes input-1 and input-2 into
    [ input-2.meryl ]   //  sub-actions instead of direct inputs to 'union'.
```

Sub-actions can pre-process inputs. The 'intersect' action below reads input from two counting actions, and the one after computes a *union* before the *intersection*.

Listing 4: Sample databases.

```
intersect
  [ count input-1.fasta output=input-1.meryl ]
  [ count input-2.fasta output=input-2.meryl ]
```

Each action will automatically pass its output k-mers to the parent action, and can optionally write them to an output database.

Listing 5: Sample databases.

```
intersect output=abINT12.meryl
  [ union input-a.meryl input-b.meryl output=ab.meryl ]
  [ union input-1.meryl input-2.meryl output=12.meryl ]
```

The original meryl allowed sub-actions to be supplied without surrounding square brackets, but this led to great ambiguity in which action the output modifier was associated with. Without brackets, the following is ambiguous:

Listing 6: Sample databases.

```
meryl
  union
    intersect
      a.meryl
      b.meryl
    intersect
      c.meryl
      d.meryl
```

As written, the intent is clear, but meryl interprets the second 'intersect' action as an input to the first:

Listing 7: Sample databases.

```
meryl
  union
    intersect
      a.meryl
      b.meryl
      intersect
        c.meryl
        d.meryl
```

Therefore, meryl2 **requires** actions (except the very first) to be surrounded by square brackets.

CHAPTER

NINE

COMMAND LINE OPTIONS

9.1 meryl2 Global Options

In meryl2, global options apply to every action. Global options are expressed as command line switches. K-mer size, label size, memory usage, number of threads, and simple sequence reductions are all global options.

Processing is specified as a list of `actions` and are described elsewhere.

Listing 1: meryl2 usage.

```
KMER and LABEL SIZE:
  -k K             Set kmer size to K (6 <= K <= 64).  Legacy format "k=K".
  -l L             Set label size to L bits (0 <= L <= 64).

SIMPLE SEQUENCE REDUCTION:
  --compress       Homopolymer compress input sequence before forming kmers.
  --ssr ssr-spec   Apply 'ssr-spec' to input sequence before forming kmers.

RESOURCE USAGE:
  -m M             Use up to M GB memory for counting.  Legacy format "memory=M".
  --memory M       Use up to M GB memory for counting.

  -t        T      Use T threads for counting and processing.
  --threads T      Use T threads for counting and processing.

LOGGING:
  -V[V[V[...]]]    Increae verbosity by the length of this option.
  -Q               Be absolutely silent.
  -P               Show progress.
  -C               Show processing tree and stop.

USAGE:
  -h               Display command line help.
  --help           Display command line help.
  help             Display command line help.
```

Obsolete forms of some of these are still allowed. The kmer size can be set with *k*=<*kmer-size*>, memory limits with *memory*=<*memory-in-gigabytes*>, thread usage with *threads*=<*thread-count*>.

Obsolete option *-E* has been removed. This used to estimate the size of an imput that could be counted in some given memory size.

Listing 2: meryl2 debugging actions.

```
dumpIndex <meryl-database>
  Report the parameters used for storing data in this meryl database.

dumpFile <meryl-database-file>
  Dump the raw data from a merylData file in a meryl database.
```

9.2 meryl2-lookup Usage

Listing 3: meryl2-lookup usage.

```
usage: meryl2-lookup <report-type> \
         -sequence <input1.fasta> [<input2.fasta>] \
         -output   <output1>      [<output2>] \
         -mers     <input1.meryl> [<input2.meryl>] [...] [-estimate] \
         -labels   <input1name>   [<input2name>]   [...]

  Compare kmers in input sequences against kmers in input meryl databases.

  Input sequences (-sequence) can be FASTA or FASTQ, uncompressed, or
  compressed with gzip, xz, or bzip2.

  Report types:

  -bed:
     Generate a BED format file showing the location of kmers in
     any input database on each sequence in 'input1.fasta'.
     Each kmer is reported in a separate bed record.

  -bed-runs:
     Generate a BED format file showing the location of kmers in
     any input database on each sequence in 'input1.fasta'.
     Overlapping kmers are combined into a single bed record.

  -wig-count:
     Generate a WIGGLE format file showing the multiplicity of the
     kmer starting at each position in the sequence, if it exists in
     an input kmer database.

  -wig-depth:
     Generate a WIGGLE format file showing the number of kmers in
     any input database that cover each position in the sequence.

  -existence:
     Generate a tab-delimited line for each input sequence with the
     number of kmers in the sequence, in the database and common to both.

  -include:
  -exclude:
```

(continues on next page)

(continued from previous page)

```
     Copy sequences from 'input1.fasta' (and 'input2.fasta') to the
     corresponding output file if the sequence has at least one kmer
     present (include) or no kmers present (exclude) in 'input1.meryl'.

Run `meryl2-lookup <report-type> -help` for details on each method.
```

CHAPTER

TEN

HISTORY

Meryl was first imagined in late-2000/early-2001/mid-2001 while BPW as at Celera Genomics.

A 3 June 2001 journal entry is titled "merConting the genome", but makes no mention of motivation for doing so. The same journal has an 18 June entry outlining the algorithm, and this entry also strongly hints the application was to build a fast 20-mer to assembly coordinate lookup table. The "Meryl" name is mentioned on 16 August 2002, desiring to increment/decrement the counts. There is even *source code* dating back to 12 February 2001 deep in the dusty archives; as that's the oldest archive found, meryl is doubtless a little bit older yet. It was originally called "merMaid" or "merCounter", the "Meryl" name first appears on 11 June 2001, and the code looks reasonably mature.

Meryl was definitely used twice in the assembly of Anopheles gambiae [Holt, et al. 2002], once to find k-mers that occur frequently and exclude them from seeding overlaps (which is how it is still used in Canu [Koren and Walenz 2017]) and once to estimate repeat content of the assembly. It is mentioned, anonymously, the paper:

> By counting the number of times each 20-nucleotide oligomer in the Anopheles and Drosophila assemblies appeared in its corre- sponding whole-genome shotgun data, we con- firmed that simple repeats are not expanded in Anopheles

The supplement elaborates a bit:

> **METHODS FOR ESTIMATING REPEAT CONTENT:**
>
> The consensus sequence of scaffolded contigs was analyzed for repeat content as follows. For each 20mer of the consensus, we counted the number of times that that 20mer appeared in the set of approximately 4.5 million sequence reads. Out of 262.8 M consensus 20mers, 26.3M (10.0%) occurred more than 50 times (5 times what is expected for unique sequence, given 10- fold sequence coverage). For comparison, of 133.5M consensus 20mers from a recent reassembly of the D. melanogaster genome that was done using the same version of the assembly software that was used for mosquito , 13.0M (9.7%) were observed more than 60 times (5 times more than expected, given 12-fold sequence coverage of the Drosophila genome). Using a count cutoff of 1.5 times expected or 50 times expected gives a similar picture: the repetative fractions of the A. gambiae and D. melanogaster genomes are not strikingly different.

Unfortunately, the earliest versions of the software are not online, likely because they were never stored in a version control system. The earliest online version is found in the initial commit of the kmer subversion repository on 2 January 2003. The main function in meryl.C shows it supporting the "binary and mathematical" operations, including "increment/decrement" from the 16 August 2002 journal entry.

The Celera Assembler subversion repository also has a relatively early version. CA was publicly released to SourceForge on 14 April 2004 (revision 4 has the good stuff). Use of meryl is mentioned in wga.pl. The meryl code and a CA-specific function to access sequence data are nicely organized, the bulk of meryl in one file (AS_MER_meryl.cc).

The offline BPW archives contain several early versions, including one just nine days after the first journal entry mentioned above:

```
-rw-r--r--  1 bri  bri  5964 Jun 10  2001 ne-20010611-1812/near-identity/meryl/meryl.C
-rw-r--r--  1 bri  bri   180 Jul  3  2001 ne-20010703-1747/near-identity/meryl/meryl.C
```

(continues on next page)

(continued from previous page)

```
-rw-r--r--  1 bri  bri  1322 Jul  6  2001 ne-20010706-1813/near-identity/meryl/meryl.C
-rw-r--r--  1 bri  bri  4132 Jul  9  2001 ne-20010709-1647/near-identity/meryl/meryl.C
-rw-r--r--  1 bri  bri  4294 Jul 10  2001 ne-20010710-1842/near-identity/meryl/meryl.C
-rw-r--r--  1 bri  bri  4675 Jul 11  2001 ne-20010712-1907/near-identity/meryl/meryl.C
-rw-r--r--  1 bri  bri  4775 Jul 18  2001 ne-20010718-1745/near-identity/meryl/meryl.C
-rw-r--r--  1 bri  bri  4706 Jul 23  2001 ne-20010723-1800/near-identity/meryl/meryl.C
-rw-r--r--  1 bri  bri  4850 Aug 15  2001 ne-20010817-1800/near-identity/meryl/meryl.C
-rw-r--r--  1 bri  bri  4988 Feb 27  2005 ne-20010831-1745/near-identity/meryl/meryl.C
-rw-r--r--  1 bri  bri  4988 Aug 30  2001 ne-20010912-1829/near-identity/meryl/meryl.C
-rw-r--r--  1 bri  bri  5130 Feb 27  2005 ne-20010928-1826/near-identity/meryl/meryl.C
-rw-r--r--  1 bri  bri  5707 Feb 27  2005 ne-20011109-1837/near-identity/meryl/meryl.C
```

(I'm not sure where the 2005 time stamps came from; I vaguely remember replacing duplicate files with symlinks before some of these directories were tar'd and compressed.)

Peeking into that first version, we find:

```
d:.../ne-20010611-1812/near-identity/meryl> ls -l
total 88
-rw-r--r--  1 bri  bri   2187 Jun 10  2001 1.C
-rw-r--r--  1 bri  bri   1232 Jun 10  2001 2.C
-rw-r--r--  1 bri  bri   1582 Jun 10  2001 3.C
-rw-r--r--  1 bri  bri   2496 Jun 10  2001 4.C
-rw-r--r--  1 bri  bri    538 Jun 11  2001 Makefile
-rwxr-xr-x  1 bri  bri  50432 Jun 10  2001 meryl
-rw-r--r--  1 bri  bri   5964 Jun 10  2001 meryl.C
-rw-r--r--  1 bri  bri   1725 Jun 10  2001 meryl.H
```

Is that a binary?

```
d:.../ne-20010611-1812/near-identity/meryl> file meryl
meryl: ELF 32-bit LSB executable, \
       Intel 80386, \
       version 1 (FreeBSD), \
       dynamically linked, \
       interpreter /usr/libexec/ld-elf.so.1, \
       for FreeBSD 4.1, \
       not stripped
```

The algorithm is quite simple, but the details are complicated: it simply builds a list of all the kmers in the input sequences, sorts, then counts how many times each kmer is in the list and write that to the output file.

The details are to split the kmer into a prefix and a suffix. The prefix points to a bucket into which all the suffixes are listed. Once all kmers are stored, each bucket is sorted then scanned to count the kmers. Bit-packed integers are used throughout to minimize memory usage.

In all it's embarassing glory, here is the first ever version of meryl (omitting the functions that do the actual work).

```
#include "meryl.H"

//  theSeq is a compressed sequence.  Three bits per character, the first bit
//  telling us if the next two bits are valid sequence.
//
u64bit  *theSeq       = 0L;
```

(continues on next page)

(continued from previous page)

```
u64bit   theSeqLen    = 0;
u64bit   numberOfMers = 0;

//  theCounts is a packed-word array of the number of mers
//  that all have the same TABLEBITS bits on the left side of
//  the mer.
//
//  theMers is a list of the other bits in the mers.
//
u64bit  *theCounts       = 0L;
u64bit   theCountsWords  = 0;
u64bit  *theMers         = 0L;
u64bit   theNumberOfMers = 0;

u64bit  approxMers = 1073741824;
u64bit  merSize    = 20;
u64bit  merMask    = u64bitMASK(merSize);

u64bit  tblBits = 20;
u64bit  tblMask = u64bitMASK(tblBits);

u64bit  chkBits = 0;
u64bit  chkMask = 0;

u64bit  bitsPerIndex = 0;

bool    beVerbose           = true;
bool    doReverseComplement = false;

char   *seqFileName = 0L;
FILE   *seqFile     = 0L;

void
usage(char *name) {
  fprintf(stderr, "usage: %s [-m merSize] [-v] [-r] [-f seqFile] [-s seqFileSize] [-h]\n
↪", name);
  fprintf(stderr, "       -m   Sets the size of mer to count.  Must be less than 32.\n");
  fprintf(stderr, "       -v   Be noisy about it.\n");
  fprintf(stderr, "       -r   Also counts the reverse complement.\n");
  fprintf(stderr, "       -f   File to count mers in.  MultiFastA\n");
  fprintf(stderr, "       -s   Upper limit on the number of mers.\n");
  fprintf(stderr, "       -h   This.\n");
}


int
main(int argc, char **argv) {

  int   arg = 1;
  while (arg < argc) {
    switch (argv[arg][1]) {
      case 'v':
```

(continues on next page)

(continued from previous page)

```
      beVerbose = true;
      break;
    case 'r':
      doReverseComplement = true;
      break;
    case 'f':
      arg++;
      seqFileName = argv[arg];
      break;
    case 'm':
      arg++;
      merSize = atol(argv[arg]);
      break;
    case 's':
      arg++;
      approxMers = atol(argv[arg]);
      break;
    case 'h':
      usage(argv[0]);
      exit(1);
      break;
    default:
      usage(argv[0]);
      fprintf(stderr, "Unknown option '%s'\n", argv[arg]);
      exit(1);
      break;
  }
  arg++;
}

if ((seqFileName == 0L) ||
    ((seqFileName[0] == '-') && (seqFileName[1] == 0)))
  seqFile = stdin;
else
  seqFile = fopen(seqFileName, "r");
if (seqFile == 0L) {
  fprintf(stderr, "Couldn't open the sequence file '%s'.\n", seqFileName);
  exit(1);
}


////////////////////////////////////////////////////////
//
//  Based on the approximate number of mers given, set the
//  size of the table and number of bits per each table entry.
//

//  Not the greatest way to find the number of bits needed
//  to encode approxMers, but easy
//
bitsPerIndex = 1;
while (approxMers > u64bitMASK(bitsPerIndex))
```

(continues on next page)

(continued from previous page)

```
  bitsPerIndex++;

//  Determine the size of the table to reduce the memory footprint.
//  This is computed backwards, so that we pick the best smallest
//  table size.
//
u64bit   bestMem = ~u64bitZERO;
u64bit   bestSiz =  u64bitZERO;
u64bit   mem;

for (u64bit siz=32; siz>18; siz-=2) {
  mem   = bitsPerIndex * (u64bitONE << siz) + approxMers * (2*merSize - siz);
  mem >>= 23;

  if (mem < bestMem) {
    bestMem = mem;
    bestSiz = siz;
  }
}

tblBits = bestSiz;
tblMask = u64bitMASK(tblBits);

chkBits = 2 * merSize - tblBits;
chkMask = u64bitMASK(chkBits);


fprintf(stderr, "You told me there will be no more than %lu mers.\n", approxMers);
fprintf(stderr, "Using %2lu bits to encode positions.\n", bitsPerIndex);
fprintf(stderr, "Using %2lu bits of the mer for the count.\n", tblBits);
fprintf(stderr, "Using %2lu bits of the mer for the check.\n", chkBits);




////////////////////////////////////////////////////////
//
//  Allocate and clear the counts
//
theCountsWords = bitsPerIndex * (u64bitONE << tblBits) / 64 + 2;

if (beVerbose) {
  fprintf(stderr, "Allocating and clearing the counting table.\n");
  fprintf(stderr, "  %lu entries at %lu bits == %lu MB.\n",
          u64bitONE << tblBits,
          bitsPerIndex,
          theCountsWords * 8 >> 20);
}

theCounts      = new u64bit [ theCountsWords ];
for (u64bit i=theCountsWords; i--; )
```

(continues on next page)

(continued from previous page)

```
    theCounts[i] = u64bitZERO;


  ////////////////////////////////////////////////////////////
  //
  //
  theSeqLen    = 0;
  numberOfMers = 0;
  theSeq       = new u64bit [ 3 * (u64bitONE << bitsPerIndex) / 64 + 1 ];

  if (beVerbose)
    fprintf(stderr, "Computing the bucket sizes (pass one through the sequence).\n");
  readSequenceAndComputeBucketSize();

  if (theNumberOfMers > approxMers) {
    fprintf(stderr, "Sorry.  You need to increase your estimate of the number of mers!\n
↪");
    fprintf(stderr, "I found %lu mers.\n", theNumberOfMers);
    exit(1);
  }


  ////////////////////////////////////////////////////////////
  //
  //
  if (beVerbose)
    fprintf(stderr, "Converting counts to offsets (%lu bits or %lu Mbuckets).\n",
            tblBits,
            (u64bitONE << tblBits) >> 20);
  convertCounts();


  ////////////////////////////////////////////////////////////
  //
  //  Allocate theMers, but don't clear them.
  //
  if (beVerbose) {
    fprintf(stderr, "Allocating space for the mers.\n");
    fprintf(stderr, "  %lu mers at %lu bits each == %lu MB.\n",
            theNumberOfMers,
            chkBits,
            chkBits * theNumberOfMers >> 23);
  }
  theMers = new u64bit [ ((chkBits * theNumberOfMers) >> 6) + 1 ];


  ////////////////////////////////////////////////////////////
  //
  //
  if (beVerbose)
    fprintf(stderr, "Filling the buckets with mers (pass two through the sequence).\n");
  fill();
```

(continues on next page)

(continued from previous page)

```
  ////////////////////////////////////////////////////////
  //
  //  Sort each bucket
  //
  if (beVerbose)
    fprintf(stderr, "Sorting buckets (%lu total).\n", u64bitONE << tblBits);
  sort();


  //
  //  Yeah, I didn't close the stupid input file.  So what?
  //
}
```

The first code block reads input sequence from disk, converts each base into a 3-bit code, the counts the size of each bucket.

```
#include "meryl.H"

void
readSequenceAndComputeBucketSize(void) {
  double  startTime = getTime() - 0.1;
  u64bit  count     = 0;

  theNumberOfMers = u64bitZERO;

  s32bit  timeUntilValid      = 0;
  u64bit  substring           = u64bitZERO;
  u64bit  partialEncoding     = u64bitZERO;
  u32bit  partialEncodingSize = u32bitZERO;

  for (unsigned char ch=nextCharacter(seqFile); ch != 255; ch=nextCharacter(seqFile)) {
    if (ch > 127) {
      timeUntilValid = merSize;
      ch = 0x07;
    } else {
      ch = compressSymbol[ch];

      substring <<= 2;
      substring  |= ch;
      timeUntilValid--;

      if (beVerbose && ((++count & (u64bit)0x3fffff) == u64bitZERO)) {
        fprintf(stderr, " %4ld Mbases -- %8.5f Mb per second\r",
                count >> 20,
                count / (getTime() - startTime) / (1000000.0));
        fflush(stderr);
      }

      if (timeUntilValid <= 0) {
```

(continues on next page)

(continued from previous page)

```
        timeUntilValid = 0;

        theNumberOfMers++;

        u64bit I = ((substring >> chkBits) & tblMask) * bitsPerIndex;

        preIncrementDecodedValue(theCounts,
                                 I >> 6,
                                 I & 0x000000000000003f,
                                 bitsPerIndex);
      }
    }

    //  Place the character into the encoded sequence.  If the character was
    //  not a valid base, ch will have the "I'm a crappy character" flag set,
    //  otherwise, ch is a valid base encoding.
    //
    partialEncoding <<= 3;
    partialEncoding  |= ch;

    partialEncodingSize++;

    //  We can fit 21 bases (at 3 bits per base, 63 bits) per 64 bit word
    //  without going to any trouble.
    //
    if (partialEncodingSize == 21) {
      theSeq[theSeqLen++] = partialEncoding;
      theSeq[theSeqLen]   = ~u64bitZERO;

      partialEncoding     = u64bitZERO;
      partialEncodingSize = u32bitZERO;
    }
  }

  //  Push on the last partialEncoding, if one exists.
  //
  if (partialEncodingSize > 0) {
    while (partialEncodingSize != 21) {
      partialEncoding <<= 3;
      partialEncoding  |= 0x07;
      partialEncodingSize++;
    }

    theSeq[theSeqLen++] = partialEncoding;
  }

  if (beVerbose)
    fprintf(stderr, "\n");
}
```

The second block of code converts the bucket sizes into an offset to the start of each bucket.

```
#include "meryl.H"

void
convertCounts(void) {
  double  startTime = getTime() - 0.1;
  u32bit  count     = 0;

  //  Convert the counts into offsets into theMers
  //
  //  We use the trick pointed out by Liliana of setting the offset to the LAST
  //  entry in the bucket, then decrementing when filling.  This also makes
  //  the code below easier -- we don't need to store the size of the bucket
  //  so we can increment the sum after we set the bucket offset.
  //

  u64bit   i = 0;
  u64bit   m = (u64bitONE << tblBits) + 1;
  u64bit   s = 0;
  while (i < m) {

    if (beVerbose && ((++count & (u64bit)0x3fffff) == u64bitZERO)) {
      fprintf(stderr, " %4ld Mbuckets -- %8.5f Mbuckets per second\r",
              count >> 20,
              count / (getTime() - startTime) / (1000000.0));
      fflush(stderr);
    }

    u64bit I = i * bitsPerIndex;

    s = sumDecodedValue(theCounts,
                        I >> 6,
                        I & 0x000000000000003f,
                        bitsPerIndex,
                        s);

    i++;
  }

  if (s != theNumberOfMers) {
    fprintf(stderr, "internal error in stage 2: s != theNumberOfMers.\n");
    exit(1);
  }

  if (beVerbose)
    fprintf(stderr, "\n");
}
```

The third block of code makes a second pass through all the kmers in the sequence, adding suffixes to the bucket indicated by the prefix.

```
#include "meryl.H"

void
```

(continues on next page)

(continued from previous page)

```
fill(void) {
  double  startTime = getTime() - 0.1;
  u32bit  count     = 0;

  s32bit  timeUntilValid      = 0;
  u64bit  substring           = u64bitZERO;

  //  Stream again, filling theMers
  //

  for (u64bit i=0; i<theSeqLen; i++) {
    u64bit  t = theSeq[i];
    u64bit  I;
    u64bit  J;
    u64bit  ch;

    for (u32bit j=0; j<21; j++) {
      ch   = t;
      ch >>= 60;
      ch  &= 0x07;

#if 0
      ch   = t & 0x7000000000000000;
      ch >>= 60;
#endif

      t <<= 3;

      if (ch & 0x04) {
        timeUntilValid = merSize;
      } else {
        substring <<= 2;
        substring  |= ch;
        timeUntilValid--;

        if (beVerbose && ((++count & (u64bit)0x3fffff) == u64bitZERO)) {
          fprintf(stderr, " %4ld Mbases -- %8.5f Mb per second\r",
                  count >> 20,
                  count / (getTime() - startTime) / (1000000.0));
          fflush(stderr);
        }

        if (timeUntilValid <= 0) {
          timeUntilValid = 0;

          I = ((substring >> chkBits) & tblMask) * bitsPerIndex;

          J = chkBits * preDecrementDecodedValue(theCounts,
                                                 I >> 6,
                                                 I & 0x000000000000003f,
                                                 bitsPerIndex);
```

(continues on next page)

(continued from previous page)

```
          setDecodedValue(theMers,
                          J >> 6,
                          J & 0x3f,
                          chkBits,
                          substring & chkMask);
        }
      }
    }
  }
  if (beVerbose)
    fprintf(stderr, "\n");
}
```

The fourth block of code sorts the kmers and writes output. It is using its own implementation of *heap sort <https://dl.acm.org/doi/10.1145/512274.512284>*_, likely beacuse the C sort() implementation is not in place. Output is written to `stdout`.

```
#include "meryl.H"

typedef u64bit heapbit;

void
adjustHeap(heapbit *M, s64bit i, s64bit n) {
  heapbit     m = M[i];

  s64bit     j = (i << 1) + 1;  //  let j be the left child

  while (j < n) {

    if (j<n-1 && M[j] < M[j+1])
      j++;       //  j is the larger child

    if (m >= M[j])   //  a position for M[i] has been found
      break;

    //  Move larger child up a level
    //
    M[(j-1)/2]      = M[j];

    j = (j << 1) + 1;
  }

  M[(j-1)/2]      = m;
}


void
sortBucket(u64bit b) {
  u64bit  st = getCount(b);
  u64bit  ed = getCount(b+1);
  u64bit  sz = ed - st;
```

(continues on next page)

(continued from previous page)

```
if (sz == 0)
  return;

heapbit *a = new heapbit [sz + 1];

for (u64bit i=st; i<ed; i++) {
  u64bit J = i * chkBits;

  a[i-st] = getDecodedValue(theMers,
                            J >> 6,
                            J & 0x3f,
                            chkBits);
}

u64bit  substring;
char    mer[merSize+1];
u64bit  count     = u64bitONE;
u32bit  i;
u32bit  m;

//  Sort if there is more than one item
//
if (sz > 1) {
  s64bit  i;

  //  Create the heap of lines.
  //
  for (i=(sz-2)/2; i>=0; i--)
    adjustHeap(a, i, sz);

  //  Interchange the new maximum with the element at the end of the tree
  //
  for (i=sz-1; i>0; i--) {
    heapbit m  = a[i];
    a[i]       = a[0];
    a[0]       = m;

    adjustHeap(a, 0, i);
  }
}

for (i=1; i<sz; i++) {
  if (a[i] == a[i-1]) {
    count++;
  } else {
    if (count > 100) {
      substring = b << chkBits | a[i-1];
      for (m=0; m<merSize; m++)
        mer[merSize-m-1] = decompressSymbol[(substring >> (2*m)) & 0x03];
      mer[m] = 0;
      printf("%s %lu\n", mer, count);
    }
```

(continues on next page)

(continued from previous page)

```
      count = 1;
    }
  }

  //  Output the last one
  //
  if (count > 100) {
    substring = b << chkBits | a[i-1];
    for (m=0; m<merSize; m++)
      mer[merSize-m-1] = decompressSymbol[(substring >> (2*m)) & 0x03];
    mer[m] = 0;
    printf("%s %lu\n", mer, count);
  }

  delete [] a;
}


void
sort(void) {
  double  startTime = getTime() - 0.1;
  u32bit  count     = 0;
  u64bit  m         = u64bitONE << tblBits;

  for (u64bit i=0; i<m; i++) {
    if (beVerbose && ((++count & (u64bit)0x3fff) == u64bitZERO)) {
      fprintf(stderr, "             %4lu buckets -- %8.5f buckets per second\r",
              count,
              count / (getTime() - startTime));
      fflush(stderr);
    }

    sortBucket(i);
  }
  if (beVerbose)
    fprintf(stderr, "\n");
}
```

CHAPTER

ELEVEN

FREQUENTLY ASKED QUESTIONS

- *No questions!*

11.1 No questions!

And no answers yet.

Meryl is a tool for working with sets of kmers. A set of kmers, when annotated with an integer value (typically the number of times the kmer occurs in a set of sequences) and optionally a label, is termed a database. Meryl comprises three (modules), one for generating kmer databases, one for filtering and combining databases, and one for searching databases. A simple, but yet to be documented, C++ API to access kmer databases exists.

CHAPTER

TWELVE

PUBLICATIONS

Rhie A, Walenz BP, Koren S, Phillippy AM. Merqury: reference-free quality, completeness, and phasing assessment for genome assemblies. Genome Biology 21, 245 (2020).

CHAPTER

THIRTEEN

INSTALL

The easiest way to get started is to download a release. If you encounter any issues, please report them using the github issues page.

Alternatively, you can also build the latest unreleased from github:

```
git clone https://github.com/marbl/meryl.git
cd meryl/src
make -j <number of threads>
```

CHAPTER

FOURTEEN

LEARN

NOTE! This documentation applies to meryl2. The original meryl is quite similar, but doesn't support filters or labels, and the specification of actions is much simpler.

- *Quick Start* - no experience or data required, download and analyze *Escherichia coli* repeats today!
- *Usage* - command line usage cheat sheet
- *Reference* - all the details
- *History* - the history of meryl
- *FAQ* - Frequently asked questions