Shotgun Metagenomics Pseudo-alignment

Table of content:

1. Introduction
   1.1 Biological background:
   1.2 Metagenomic sequencing by NGS
   1.3 Reverse complementation
2. A Pseudo-alignment Algorithm
      2.1 Preprocess - Building a reference.
      2.2 Pseudo-alignment Algorithm
      2.3 Pseudo-alignment examples
         2.3.1. Case 1: Unmapped Read
         2.3.2.Case 2: Uniquely Mapped Read
         2.3.3. Case 3: Ambiguous Mapping (With Multiple Genomes)
         2.3.4. Case 4: Initially Unique but Changed to Ambiguous (With Multiple Genomes)
3. Core requirements:
   3.0 Command line interface
   3.1 Building a reference.
   3.2 Dumping a reference database.
   3.3 Building **and** Dumping a reference database.
   3.4 Pseudo-alignment with a pre-built reference
   3.5 Building a reference **+** Pseudo-alignment
   3.6 Dumping an align output file.
   3.7 Pseudo-alignment with a pre-built reference + Dumping the result.
   3.8 Building a reference **+** Pseudo-alignment + Dumping the result.
4. Extensions
   4.1. (EXTQUALITY) Pseudo-alignment with quality filtering
   4.2. (EXTREVCOMP) Support for reverse complement read analysis
   4.3. (EXTCOVERAGE) For each genome maintain a coverage map of unique/ambiguous mapping k-mers.
   4.4. (EXTSIM) Detect and filter highly similar genomes during construction

1. Introduction

Shotgun metagenomics is a method for studying the DNA from an entire community of organisms—such as those found in soil, water, or the human gut—without first separating each organism. In this approach, researchers extract all the DNA from the sample and sequence it directly, generating millions of short DNA fragments ('reads') from multiple species. Given such a sample, the main challenge is identifying the composition and abundance of the organisms whose DNA is in the sample by comparing these reads to a set of previously identified genome sequences, called reference genomes.

In this project, you will create a simple "pseudo-alignment" tool to help with metagenomic analysis. Pseudo-alignment is the process of assigning each DNA fragment to one or more possible reference genomes by a fast heuristic process. Alignment (or more exactly sequence-alignment) is the task of finding an optimal string matching between two DNA sequences (given a set of matching-rules), which is a more algorithmically complex and computationally intensive task. Instead, in pseudo-alignment, we scan for short stretches of DNA sequence (called "k-mers") within each fragment. By finding these k-mers in known reference genomes, we can quickly identify which organisms a DNA fragment might have come from.

1.1 Biological background:

1.1.1 DNA and Alphabet:

DNA (Deoxyribonucleic acid) is the hereditary material in all living organisms. DNA is composed of long chains of nucleotides, and each such nucleotide is one of just four types: Adenine (A), Thymine (T), Cytosine (C), and Guanine (G). For sequencing and computational analysis, DNA sequences are represented as strings composed of the letters `{A, T, C, G}`. Sometimes, when the sequencing process is uncertain, an ambiguous character `N` (meaning "no call") is used to represent a nucleotide that could not be confidently identified. Note that N calls are in some sense a wildcard (*), and require different handling than confident calls (of `{A, T, C, G}`), these will not be considered here.

1.1.2. Genomes, Chromosomes and Complementarity

A genome is the complete set of genetic instructions for an organism, stored as DNA. In bacteria, this DNA is organized into a single chromosome - a long molecule twisted into a circular shape. The chromosome contains all the genes and other sequences that make up the organism's genome.
The DNA molecule itself has a ladder-like structure with two strands running in parallel. These strands are held together by connections between pairs of bases: A always pairs with T, and C always pairs with G. This creates a natural complementarity - when one strand has the sequence

"ATCG", its partner strand must have "TAGC". This pairing is essential for DNA to replicate and function in living cells.

1.1.3 k-mers:

A **k-mer** is a short DNA sequence of length **k**. In other words a short string of length K composed of the letters {A,T,C,G, and possibly N}.

1.1.4 Bacterial Genomes:

A bacterial genome typically consists of a single circular chromosome composed of millions of bases DNA sequence. The genome encodes the bacterium's genetic information. Although much smaller and simpler than a human genome, bacterial genomes are still large enough that analyzing them computationally requires efficient algorithms and data structures. Such genomes are usually stored in FASTA files.

1.1.5 FASTA Format (usually .fa or .fa.gz):

The FASTA format is a simple text-based representation of nucleotide sequences, it is the standard format used to store previously identified "reference genomes".

A FASTA file contains one or more records. Each record starts with a header line beginning with `>`, followed by a description or identifier. Subsequent lines contain the raw sequence data (A, T, C, G, and possibly N). Whitespace and newline characters outside the header should be ignored.

Example:

```
>genome1
ATCGATCGAAATTTCCGG...
```

1.2 Metagenomic sequencing by NGS

Next-Generation Sequencing (NGS) refers to a set of modern, DNA sequencing technologies, that enable us to detect the exact sequence nucleotide bases of relatively short DNA fragments in a cost effective manner. NGS devices can produce millions to billions of short reads, which are short fragments of DNA (often 30-300 bases), from a biological sample in a matter of hours.

1.2.1 FASTQ file format (usually .fq or fq.gz):

The FASTQ format is commonly used to store raw reads from NGS machines. Each record in a FASTQ file consists of four lines:

- **Header:** A header line beginning with `@` followed by an identifier.
- **Read sequence:** A line with the raw DNA sequence (A, T, C, G).

- **Space:** A line starting with `+`.
- **Read quality sequence:** A line with the quality scores corresponding to each base in the sequence.

Example:

```
@EAS139:136:FC706VJ:2:2104:15343:197393                    1:N:18:1
GGGTGATGGCCGCTGCCGATGGCGTCAAATCCCACC
+
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII9IG9IC
```

1.2.1.1 Quality

The **quality string** consists of ASCII characters, each representing a numerical quality score for the corresponding nucleotide in the sequence. In this scheme, the ASCII value of each character minus 33 yields the quality score (called a Phred33 score). For example, the character 'I' has an ASCII value of 73; thus, its corresponding score is 73 - 33 = 40.

(See also: https://en.wikipedia.org/wiki/FASTQ_format)

1.3 Reverse complementation

In DNA analysis, reverse complementation refers to taking a given DNA sequence, finding its complementary bases (A↔T and C↔G), and then reversing the order of the resulting sequence.

To create the reverse complement of a DNA sequence, follow these steps:

1. **Reverse the sequence**
   - Take the original DNA sequence and reverse the order of its nucleotides.
   - For instance, if the sequence is `ACGTGG`, reversing it yields `GGTGCA`.
2. **Find the complement of each nucleotide**
   - Replace each base with its complement:
     - A ↔ T
     - C ↔ G
   - For the example `GGTGCA`, its complement is `CCACGT`.

Together, these steps produce the reverse complement. So, if the original sequence was `ACGTGG`, its reverse complement is `CCACGT`.

2. A Pseudo-alignment Algorithm

In this section, we will describe a general Pseudo-alignment Algorithm that you will implement in the commands specified in the next section.

Input :

- A collection of "reference" genomes (".fa", fasta format)
- A collection of reads (from NGS machines) to process (".fq", sequencing data).
- A number k.
- Parameters m and p (default: m=1, p=1)

Output :

A mapping for each read in the input collection.
The mapping of each read can be either:

- Uniquely Mapped → found a match to a single genome in the reference.
- Ambiguously Mapped → found a match to several genomes in the reference.
- Unmapped → did not find a match in the reference.

2.1 Preprocess - Building a reference.

Goal: Create an efficient lookup data-structure mapping between k-mers and genomes in the references set (from .fa file).

- Extract all k-mers of length k from each "reference" genome. (see k-mer definition)
- Build a data structure linking each k-mer to:
  - The genome(s) in which it appears
  - The positions at which it appears.
- In the next part we will refer to this data structure as the “k-mer reference”

Notes:

- Some k-mers may appear ***specifically*** in one genome; others may appear in multiple genomes ***(unspecific k-mers)***.
- For most interesting k values, covering all possible k-mers is difficult (and unnecessary). Note that there are 4^k possible k-mers for any k value.
- Kmers of size 20-31 should be supported but you may choose to support additional sizes.

- In the basic implementation you may ignore the presence of "N" unknown nucleotides in both the reference and NGS reads data. You may simply skip any k-mer that contains an "N".

2.2 Pseudo-alignment Algorithm

Goals:
(1) Use k-mers counting to create a mapping between each NGS read in the input (.fq file).
(2) Count how many *reads* map uniquely and ambiguously to each reference genome.

<u>For *each read* in the FASTQ File:</u>

Rationale: The process defined in the steps below uses k-mer lookup and a set of rules to assign each read to a single genome (if possible). K-mer lookup is a fast and simple way to identify a "good" mapping for each read.

Step 1. Extract all k-mers and find their genome matches:

1.1. Split read into all possible k-mers of length k.
Example: read "ATGGC" with k=3 gives k-mers: ["ATG", "TGG", "GGC"]
(You may skip k-mers containing *"N"* unknown values).

1.2. For each k-mer:

1.2.1. Look up which genomes contain this k-mer (in the k-mer reference data-structure you created in the reference build step)

1.2.2. Classify k-mer as either:

- **Specific: k-mer** appears in exactly one genome.
- **Unspecific:** k-mer appears in multiple genomes.

Step 2. Determine Best Mapping
(Uniquely Mapped, Ambiguously Mapped, Unmapped):
Rationale: we map reads using the specific kmers.

2.1. If no k-mers match any genome → define read as ***Unmapped***.

2.2. Check specific K-mers

2.2.1. If all ***specific*** (step 1.2.2) k-mers point to same genome (in the k-mer reference)

→ ***uniquely map*** read to this genome

2.2.2. If ***specific*** k-mers point to different genomes:

2.2.2.1. Count how many specific k-mers (from this read) match each genome
2.2.2.2. Find the genome with the highest specific k-mer count
2.2.2.3. Find the genome with the second-highest specific k-mer count.

2.2.2.4. Find the difference between the counts of the highest and second-highest unique k-mer count.

2.2.2.5. If the difference is greater or equal to ***m*** → ***uniquely map*** read to the highest count genome

2.2.2.6. Otherwise → ***ambiguously map*** read to all genomes with specific mapping.

Increment the count of all (ambiguously) mapping genomes.

2.3. Validate *unique* mappings with unspecific K-mers (if *p* ≥ 0)

Rationale: we validate our read mapping using the unspecific k-mers - we make sure that the genome selected as a unique mapping is also supported by the ***unspecific*** (step 1.2.2) k-mer counts. To do so, we test that the genome mapped in step 2.2, also has enough ***unspecific*** k-mers mapping to it (from the read).

2.3.1. If the read was **not *uniquely mapped***, skip.

2.3.2. If the read was ***uniquely mapped*** (step 2.2), we examine the subset of genomes with specific k-mers:

2.3.2.1. Count how many total k-mers each genome in the set has (total k-mers = specific + unspecific k-mers, from this read).

2.3.2.2. Let **MaxCount** = the highest total k-mer counts.

2.3.2.3. Let **MapCount** = the total k-mer count of the genome uniquely mapped the read to in step 2.2.

2.3.2.4. If (MaxCount - MapCount **>** p) → ***ambiguously map***

In other words, if the difference between the counts of the selected unique mapping and highest count mapping is more than p, change the mapping of the read to ambiguous.
Increment the count of all genomes with total counts greater than or equal to MapCount.

2.4. Final read mapping assignment

Note: *All reads belong to one of 3 states below.*

2.4.1. Unmapped: no k-mers match any genome.

2.4.2. Uniquely Mapped: has unique mapping in step 2.2 and was not changed by total-counts in step 2.3 **(*p* ≥ 0)**.

2.4.3. Ambiguously Mapped: all other cases.

Note: Ambiguously reads increment the ambiguous mapping count of at least 2 genomes.

2.3 Pseudo-alignment examples

The following sections are meant to illustrate the kmer pseudo-alignment algorithm described above, with parameters m=1 and p=1.

2.3.1. Case 1: Unmapped Read

```
Read: TAGGCAT
k-mers (k=4): [TAGG, AGGC, GGCA, GCAT]

Reference Genomes:
genome1: contains none of these k-mers
genome2: contains none of these k-mers

Result: UNMAPPED
Reason: No k-mers found in any reference genome
```

2.3.2.Case 2: Uniquely Mapped Read

```
Read: ATGGCTAT
k-mers (k=4): [ATGG, TGGC, GGCT, GCTA, CTAT]

Reference Genomes:
genome1: contains [ATGG:1, TGGC:2, GGCT:1, GCTA:1]  (4 specific k-mers)
genome2: contains [CTAT:1]  (1 specific k-mer)

Specific k-mer counts:
- genome1: 4 k-mers
- genome2: 1 k-mer

Difference between highest and second highest = 4 - 1 = 3
Since 3 > m (where m=1), this read is UNIQUELY MAPPED to genome1
```

2.3.3. Case 3: Ambiguous Mapping (With Multiple Genomes)

```
Read: ATGCCTAT
k-mers (k=4): [ATGC, TGCC, GCCT, CCTA, CTAT]

Reference Genomes:
genome1: contains [ATGC:1, TGCC:1]  (2 specific k-mers)
genome2: contains [GCCT:1, CCTA:1]  (2 specific k-mers)
```

```
genome3: contains [TGCC:1, CTAT:1]  (2 specific k-mers)
genome4: contains [ATGC:1]  (1 specific k-mer)

Specific k-mer counts:
- genome1: 2 k-mers
- genome2: 2 k-mers
- genome3: 2 k-mers
- genome4: 1 k-mer

Analysis:
1. Three genomes tie for highest specific k-mer count (2 each)
2. Difference between highest and second highest = 2 - 2 = 0
3. Since 0 < m (where m=1), this read is AMBIGUOUSLY MAPPED to genome1,
genome2 and genome3
Result: AMBIGUOUSLY MAPPED
Increment ambiguous counts for: genome1, genome2, and genome3
(Note: genome4 is not included in ambiguous mapping as it has fewer
specific k-mers)
```

2.3.4. Case 4: Initially Unique but Changed to Ambiguous (With Multiple Genomes)

```
Read: ATGCCGGGGCTAA
k-mers (k=4): [ATGC, TGCC, GCCG, CCGG, CGGG, GGGG, GGGC, GGCT, GCTA, CTAA]

Reference Genomes:
genome1: contains [ATGC:1, TGCC:1]  (2 specific k-mers)
genome2: contains [GCCG:1]  (1 specific k-mer)
genome3: contains [CCGG:1]  (1 specific k-mer)
genome4: contains no specific k-mers
genome5: contains no specific k-mers

Genome1 and Genome2 [CGGG:1, GGGG:1]  (2 unspecific k-mers)
Genome2 and genome4 contains [GGGC:1, GGCT:1]  (2 unspecific k-mer)
Genome2 and genome5 contains [GCTA:1, CTAA:1]  (2 unspecific k-mer)

Step 1: Initial Mapping Based on Specific k-mers
- genome1: 2 specific k-mers
- genome2: 1 specific k-mer
- genome3: 1 specific k-mer
- genome4: 0 specific k-mers
```

```
- genome5: 0 specific k-mers

Difference between highest (genome1) and second highest = 2 - 1 = 1
Since 1 ≥ m (where m=1), initially UNIQUELY MAPPED to genome1

Step 2: Validation with Total k-mer Counts
Total k-mer counts (specific + unspecific):
- genome1: 2 specific + 2 unspecific = 4 total
- genome2: 1 specific + 6 unspecific = 7 total
- genome3: 1 specific + 0 unspecific = 0 total
- genome4: 0 specific + 2 unspecific = 3 total
- genome5: 0 specific + 2 unspecific = 3 total

Difference between MaxCount and MapCount = 7 - 4 = 3
Since 3 > p (where p=1), changes to AMBIGUOUSLY MAPPED to genome1 and
genome2.

Result: AMBIGUOUSLY MAPPED
Increment ambiguous counts for: genome1, genome2.
(Note: genome3, genome4, genome5 are not included as their total k-mer
count is less than genome1's)
```

3. Core requirements:

3.0 Command line interface

The tool will be run in python from the command line. It should strictly adhere to the following interface:

General Structure:

```
python main.py -t <command> [options...]
```

The specific commands and options will be detailed in the next subsections. The core deliverables of the project are to implement solutions of 4 basic tasks:

1. `reference` - for building a k-mer reference database. (subsection 3.1)
2. `dumpref` - for "dumping" a k-mer reference database. (subsections 3.2,3.3)
3. `align` - for running the pseudo-alignment algorithm. (subsections 3.4, 3.5)
4. `dumpalign` - for "dumping" a result of the pseudo-alignment algorithm (subsections 3.6, 3.7, 3.8)

Notes:

- You may use any packages installed by default on the lab machines. The use of any other module will require explicit permission on the project forum (and is unlikely).
- You are encouraged to use the following packages: "sys", "json", "argparse", "csv", "pickle", "gzip", "numpy".
- The "argparse" module is the default recommended standard library module for implementing basic command line applications with both required and optional arguments. See the documentation here.
- The pseudo alignment tools you will implement in the project will all be called from main.py. The specific tool is implemented using the -t <command> flag.
- All outputs will be printed to screen (stdout) in json format.
- Each command can be run with different parameters as detailed in each subsection.
- Some commands will be run on full inputs, and some will be run with results produced by running previous commands. (This will allow us to test separate parts of your implementation separately)
- For the base implementation, you can assume that the input does not include the unknown "N" ambiguous character.
- For this project's scope, no strict upper bound on memory usage is enforced, the code should run on typical lab machine configurations (e.g., under 4 GB RAM, for the moderate example inputs).

3.1 Building a reference.

Run the preprocessor defined in section 2.1

Usage:.

```
python3 main.py -t reference -g GENOMEFILE.fa -k KMER-SIZE -r REFERENCEFILE.kdb
```

Options (required):

- `-g GENOMEFILE.fa`: A genome fasta file listing one or more records (genomes). The order of genomes in this file is the reference order used in all future outputs.
- `-k KMER-SIZE`: Integer specifying the length of the k-mers (e.g. 31).
- `-r REFERENCEFILE.kdb`: Output filename for the k-mer data structure.

Input: A file containing a collection of "reference" genomes in FASTA format and a number k.

Process: Run the preprocess defined in section 2.1

Output: A file containing a reference data structure file (referred to as a “k-mer reference database” or `.kdb` file).

Notes:

- The `.kdb` file size should be on the order of the O(n), n being the total size, not exceeding a few hundred MB for typical input sets in the core project.
- You may choose to implement your kdb file as you see fit. We recommend using a serializable object and a compressed pickle file.

3.2 Dumping a reference database.

Produce a flat summary of a given “k-mer reference database” (i.e., kdb) file and print in a json format. This process is also referred to as “dumping”.

The dumping mechanism creates two human-readable text outputs that allow us to inspect and verify the contents of our k-mer reference database. Think of these outputs as a way to "peek inside" our database and make sure everything was built correctly.

Usage:

```
python3 main.py -t dumpref -r REFERENCEFILE.kdb
```

Options (required):

- `-r REFERENCEFILE.kdb`: filename for the reference k-mer database, built with the reference command.

Input: A file containing a reference k-mer database, built with the reference command.

Process:

1. **k-mer-details :** For each k-mer in the input file, construct a set showing where this k-mer appears
2. **Genome-summary:** For each genome in the in the input file, calculate the following statistics:
    a. **total_bases:** Total length of this genome's sequence
    b. **unique_kmers:** Number of k-mers that appear ONLY in this genome
    c. **multi_mapping_kmers:** Number of k-mers that appear in this genome AND at least one other genome

Output: Print the following in a json format:

```
{"Kmers": k-mer-details,
"Summary": genome-summary}
```

Where k-mer-details and genome-summay are json object defined below:

a. k-mer-details:
A json object representing the **k-mer-details** defined above.

Example:

```
{ "ATCG": { "genome1": [0, 5], "genome2": [10] } },
{ "GCTA": { "genome1": [3] } }
```

The output is a list of entries, separated by a comma (,) - one per k-mer composed of:

- **kmer:** The actual k-mer sequence (e.g., "ATCG")
- **genome_locations**: A set showing where this k-mer appears
    - Different genomes are separated by commas
    - For each genome, we have a list of positions
    - Positions are separated by semicolons

Additional rules for formatting:

- Positions must be sorted numerically within each genome

b. genome-summary:

A json object that holds statistics about each genome. Each subsequent line contains summary information for one genome

Example:

```
{"genome1":{"total_bases":1000,"unique_kmers":150,"multi_mapping_kmers":50},
"genome2": {...}}
```

The output is a list of entries, separated by a comma (,) - one per genome composed of:

1. **genome:** The genome identifier exactly as it appears in the FASTA file
2. **total_bases:** Total length of this genome's sequence
3. **unique_kmers:** Number of k-mers that appear ONLY in this genome
4. **multi_mapping_kmers:** Number of k-mers that appear in this genome AND at least one other genome

Additional rules for formatting:

- Numbers should be written as plain integers (no commas or scientific notation)

3.3 Building **and** Dumping a reference database.

Run the preprocess defined in section 2.1, produce a flat summary of the “k-mer reference database” and print it in a json format, i.e., “dumping”.

The dumping definition is identical to that of the dumping described in section 3.2.

Usage:

```
python3 main.py -t dumpref -g GENOMEFILE.fa -k KMER-SIZE
```

Options (required):

- `-g GENOMEFILE.fa`: A genome fasta file listing one or more records. The order of genomes in this file is the reference order used in all future outputs.
- `-k KMER-SIZE`: Integer specifying the length of the k-mers (e.g. 31).

Input: Same as the “reference” command.

Process:

a. Run the preprocess defined in section 2.1
b. Calculate:
    1. **K-mer-details**
    2. **Genome-summary**

Output: Print the following in a json format:

```
{"Kmers": k-mer-details,
"Summary": genome-summary}
```

Where k-mer-details and genome-summary are JSON objects defined in section 3.2.

3.4 Pseudo-alignment with a pre-built reference

Perform pseudo-alignment algorithm from section 2.2 on input reads against a pre-built "k-mer reference database".

Usage:

```
python3 main.py -t align -r REFERENCEFILE.kdb -a ALIGNFILE.aln --reads
READS.fastq [-m UNIQE_THRESHOLD] [-p AMBIGUOUS_THRESHOLD]
```

Options (required):

- `-r REFERENCEFILE.kdb`: filename for the reference k-mer database, built with the reference command.
- `-a ALIGNFILE.aln`: filename for output
- `--reads READS.fastq`: FASTQ file containing reads to be pseudo-aligned.

Options (optional):

- `-m UNIQE_THRESHOLD`: Integer specifying the ***specific*** k-mers threshold. Default: 1.
- `-p AMBIGUOUS_THRESHOLD`: Integer specifying the ***Unspecific*** k-mers threshold. Default: 1.

Input:.

- A k-mer reference database (.kdb file)
- A FASTQ file containing reads to process
- Optional Parameters m and p (default: m=1, p=1)

Output:

- A pseudo-alignment mapping file containing serialized data structure containing the mapping statistics for each read in the FASTQ file.
- You may choose to implement your aln file as you see fit. We recommend using a serializable object and a compressed pickle file.

3.5 Building a reference + Pseudo-alignment

Runs the preprocessor (section 2.1) to built a “k-mer reference database”, and performs a pseudo-alignment algorithm (section 2.2) on input reads

Usage:

```
python3 main.py -t align -g GENOMEFILE.fa -k KMER-SIZE -a ALIGNFILE.aln --
reads READS.fastq [-m UNIQE_THRESHOLD] [-p AMBIGUOUS_THRESHOLD]
```

Options (required):

- `-g GENOMEFILE.fa`: A genome fasta file listing one or more. The order of genomes in this file is the reference order used in all future outputs.
- `-k KMER-SIZE`: Integer specifying the length of the k-mers (e.g. 31).
- `-a ALIGNFILE.aln`: filename for output
- `--reads READS.fastq`: FASTQ file containing reads to be pseudo-aligned.

Options (optional):

- `-m UNIQE_THRESHOLD`: Integer specifying the ***Specific*** k-mers threshold. Default: 1.
- `-p AMBIGUOUS_THRESHOLD`: Integer specifying the ***Unspecific*** k-mers threshold. Default: 1.

Input:

- A file containing a collection of "reference" genomes in FASTA format
- a number k.
- A FASTQ file containing reads to process
- Parameters m and p (default: m=1, p=1)

Process: Run the preprocess defined in section 2.1, and then run the algorithm from section 2.2.

Output:

- A pseudo-alignment mapping file containing serialized data-structure containing the mapping statistics for each read in the FASTQ file. Same as section 3.4.

3.6 Dumping an align output file.

Produce a flat summary (dump) of a given ALIGNFILE, and print it in a JSON format.

The dumping mechanism creates two human-readable text outputs that allow us to inspect and verify the contents of our ALIGNFILE. Think of these outputs as a way to "peek inside" our result from the aline command.

Usage:

```
python3 main.py -t dumpalign -a ALIGNFILE.aln
```

Options (required):

- `ALIGNFILE.aln`: align result file, built with the align command.

Process:

1. **reads-stats:** Calculate the following statistics for the whole aline process:
    a. unique_mapped_reads - total number of reads that were uniquely mapped.
    b. ambiguous_mapped_reads - - total number of reads that were ambiguously mapped.
    c. unmapped_reads - total number of reads that were not mapped.
2. **Genome-mapping-summary:** For each genome that appears in the align result, calculate the following statistics:
    a. **unique_reads:** number of reads that where uniquely mapped to this genome.
    b. **ambiguous_reads:** number of reads that where ambiguously mapped to this genome.

Output: Print the following in a JSON format:

```
{"Statistics":reads-stats,
"Summary": genome-mapping-summary}
```

Where reads-stats and genome-mapping-summary are JSON objects defined below:

a. **Reads-stats:**

A json object that holds statistics of the mapping summary: the counts of uniquely assigned, ambiguous, and unmapped reads.

Example:

```
{"unique_mapped_reads":1200 ,"ambiguous_mapped_reads":300
,"unmapped_reads":150}
```

- **Note:** The total should match the number of input reads.

b. Genome-mapping-summary:

A json object that holds a genome mapping counts summary

Example:

```
{"genome1":{"unique_reads":1000,"ambiguous_reads":200},
"genome2": {"unique_reads":150,"ambiguous_reads":80}}
```

The output is a list of entries, separated by a comma (,) - one per genome composed of:

1. **genome:** The genome identifier exactly as it appears in the FASTA file
1. **unique_reads**: the total number of reads that were uniquely assigned to this genome.
2. **ambiguous_reads**: the total number of reads that were ambiguously assigned to this genome (i.e., also validly mapped to other genomes).

3.7 Pseudo-alignment with a pre-built reference + Dumping the result.

Runs the Pseudo-alignment algorithm on a pre-built reference, produces a flat summary (dump) of the result, and prints in a json format.

Usage:

```
python3 main.py -t dumpalign -r REFERENCEFILE.kdb --reads READS.fastq
[-m UNIQE_THRESHOLD] [-p AMBIGUOUS_THRESHOLD]
```

Options (required):

- `-r  REFERENCEFILE.kdb`: filename for the reference k-mer database, built with the reference command.
- `--reads READS.fastq`: FASTQ file containing reads to be pseudo-aligned.

Options (optional):

- `-m UNIQE_THRESHOLD`: Integer specifying the *unique* k-mers threshold. Default: 1.
- `-p AMBIGUOUS_THRESHOLD`: Integer specifying the *ambiguous* k-mers threshold. Default: 1.

Input:.

- A k-mer reference database (.kdb file)
- A FASTQ file containing reads to process
- Optional Parameters m and p (default: m=1, p=1)

Process:

a. Run the algorithm from section 2.2.
b. Calculate:
    1. **reads-stats**
    2. **Genome-mapping-summary**

Output: Print the following in a JSON format:

```
{"Statistics":reads-stats,
"Summary": genome-mapping-summary}
```

Where reads-stats and genome-mapping-summary are JSON objects defined in section 3.6.

3.8 Building a reference + Pseudo-alignment + Dumping the result.

Runs the preprocessor (section 2.1) to built a “k-mer reference database”, performs a pseudo-alignment (section 2.2) on input reads, produces a flat summary of the result, and print in a json format

Usage:

```
python3 main.py -t dumpalign -g GENOMEFILE.fa -k KMER-SIZE --reads
READS.fastq [-m UNIQE_THRESHOLD] [-p AMBIGUOUS_THRESHOLD]
```

Options (required):

- `-g GENOMEFILE.fa`: A genome fasta file listing one or more. The order of genomes in this file is the reference order used in all future outputs.
- `-k KMER-SIZE`: Integer specifying the length of the k-mers (e.g. 31).
- `--reads READS.fastq`: FASTQ file containing reads to be pseudo-aligned.

Options (optional):

- `-m UNIQE_THRESHOLD`: Integer specifying the ***Specific*** k-mers threshold. Default: 1.
- `-p AMBIGUOUS_THRESHOLD`: Integer specifying the ***Unspecific*** k-mers threshold. Default: 1.

Input:

- A file containing a collection of "reference" genomes in FASTA format
- a number k.
- A FASTQ file containing reads to process
- Parameters m and p (default: m=1, p=1)

Process:

a. Run the preprocess defined in section 2.1
b. Run the algorithm from section 2.2.
c. Calculate:
    1. **reads-stats**
    2. **Genome-mapping-summary**

Output: Print the following in a JSON format:

```
{"Statistics":reads-stats,
"Summary": genome-mapping-summary}
```

Where reads-stats and genome-mapping-summary are JSON objects defined in section 3.6.

4. Extensions

4.1. (EXTQUALITY) Pseudo-alignment with quality filtering

Goal: Perform pseudo-alignment of reads (as described in section 2 *Pseudo-alignment*) in addition to taking into consideration the quality of sequencing reads in the input file.

Command: Add support of the following optional arguments to the **`align`** and **`dumpalign`** commands that run on the *Pseudo-alignment Algorithm* (subsections 3.4, 3.5, 3.7, 3.8):

1. `--min-read-quality MRQ` - Minimum mean quality for a read to be considered. Default: no filtering if not provided.
2. `--min-kmer-quality MKQ` - Minimum mean quality for an individual k-mer from a read to be considered. Default: no filtering if not provided.
3. `--max-genomes MG` - Ignore k-mers that map to more than this number of genomes. Default: no limit if not provided.

Process:

The process is the same as the standard pseudo-alignment process defined above with the following additions for filtering reads or parts of reads (k-mers) with a high-likelihood of error.

1. **Read-Level Quality Filter (**do not map reads below mean quality threshold)

   Calculate the mean quality of the entire read.

   - Calculate the arithmetic mean. Sum all per-base quality scores, then divide by the read length.
   - **Compare** this mean quality to a specified threshold (e.g., `MRQ`).
   - **Ignore/skip** the read entirely if its mean quality is **below** this threshold.

2. **K-mer-Level Quality Filter**
   - For each k-mer, calculate its mean quality by averaging the quality scores of its constituent bases.
   - If a k-mer's mean quality falls **below** a specified threshold (e.g., `MKQ`), **ignore** that k-mer and do not include it in subsequent mapping steps (**in step 1.2 of the algorithm**).

3. **Ignore highly redundant k-mers**

   Ignore (in step 1.2 of the algorithm) any k-mer that maps to **more than** d different genomes (where d is a user-defined integer, e.g., `MG`). This helps reduce noise from k-mers that are too common or non-discriminatory.

Output:

For the dumpalign command, also includes the following:
For each optional filter included add another summary stats, with the appropriate counts, as follows:

a) **filtered_quality_reads**
b) **filtered_quality_kmers**
c) **filtered_hr_kmers**

4.2. (EXTREVCOMP) Support for reverse complement read analysis

Goal : In DNA sequencing, reads can come from either strand of the DNA double helix. Since DNA strands are complementary and run in opposite directions, a particular sequence fragment might be read in either its forward orientation or its reverse complement form. This extension enhances the pseudo-aligner to consider both possibilities when mapping reads, potentially improving mapping accuracy.

Command: Add support of the following optional argument to the **`align`** and **`dumpalign`** commands as extensions of the core *Pseudo-alignment Algorithm* (subsections 3.4, 3.5, 3.7, 3.8):

- --reverse-complement - Perform pseudo-alignment on both the original read and its reverse complement separately. Choose the orientation (forward or reverse) that gives the better mapping result (in terms of unique k-mers count).

Process:

- **Sequence Processing**
  - For each read, analyze two sequences:
    - Original sequence (forward orientation)
    - Reverse complement sequence (see section 1.3 for details)
- **Mapping Process**
  - Process the reads in each orientation (regular and reversed) independently:
    - Extract k-mers from each orientation
    - Find genome matches for each k-mer
    - Count unique and ambiguous k-mer matches per genome
  - Store two separate result sets:
    - Forward results: {genome_id_F → (unique_kmers, ambiguous_kmers)}
    - Reverse results: {genome_id_R → (unique_kmers, ambiguous_kmers)}

- **Orientation Selection Rules**
  Compare mapping results from both orientations, as if considering different reference genomes, using the pseudo-alignment algorithm.
- **Additional Implementation Notes**
  - Don't combine k-mer counts between orientations
  - Each read must be assigned exactly one orientation (forward or reverse)
  - Only use k-mer counts from the selected orientation for final mapping
  - Apply thresholds m and p only to the selected orientation's counts

    Example:

    Read: ATGGCC
    Forward k-mers (k=3): [ATG, TGG, GGC, GCC]
    Reverse k-mers (k=3): [GGC, GCC, CAT, GCC]

    If forward - matches genome1 with 2 unique k-mers, and reverse - matches genome2 with 1 unique k-mer

    → Choose forward orientation, map to genome1

4.3. (EXTCOVERAGE) For each genome maintain a coverage map of unique/ambiguous mapping k-mers.

Goal: This extension enhances the pseudo-aligner to track and report the coverage of reads across each reference genome (or a specified set of reference genomes). For each position in a genome, it records:

- How many uniquely mapped reads cover that position
- How many ambiguously mapped reads cover that position
- Summary statistics about coverage distribution

Command: Add support of the following optional argument to the **`align`** and **`dumpalign`** commands that run on the *Pseudo-alignment Algorithm* (subsections 3.4, 3.5, 3.7, 3.8):

- `--coverage`
- `--genomes GENOMES` - A genome or a comma separated list of genomes to calculate coverage for. Default: output the coverage results of all genomes in reference.
- `--window-size WS`: Size of windows for summarizing coverage (default: 100)
- `--min-coverage MC`: Minimum coverage depth to report (default: 1)
- `--full-coverage:` Output the detailed Position Coverage

Process:

1. **Coverage Analysis Process**
   - For each mapped read in the alignment file:
     - Record the start and end positions of the mapping
     - Track unique and ambiguous mappings separately
     - Increment coverage counters for all positions spanned by the read
2. **Coverage Rules**
   - A position is considered "covered" if at least one k-mer from a mapped read overlaps it
   - For uniquely mapped reads, increment unique coverage counter
   - For ambiguously mapped reads, increment ambiguous coverage counter
   - Coverage depth at each position = number of reads covering that position

Output:

In addition to the basic command (**`align/dumpalign`**) output, print the following in a JSON format:

```
{
"Summary": coverage-statistics-summary,
“Details”: detailed-position-coverage,
“Window”: window-coverage-summary}
```

Where coverage-statistics-summary, detailed-position-coverage, and window-coverage-summary are json object defined below:

a. coverage-statistics-summary

A json object representing the **coverage-statistics-summary**.

Example:

```
{"genome1":{"total_bases":1000000,   "covered_bases_unique":800000
,"covered_bases_ambiguous”:150000   ,   "mean_coverage_unique”:2.5,
“mean_coverage_ambiguous”:0.3,          “median_coverage_unique”:2.0,
“median_coverage_ambiguous”:0.0      ,      “max_coverage_unique”:8,
“max_coverage_ambiguous”:3},
"genome2":...}
```

Notes:
- Coverage values rounded to one decimal place
- Base counts as integers
- Only requested genomes included

b. detailed-position-coverage

A json object representing the **detailed-position-coverage**.

Example:

```
{"genome1":{"position":0, "unique_depth":3 ,"ambiguous_depth”:1},
"genome1":{"position":1, "unique_depth":3 ,"ambiguous_depth”:0},
"genome1":{"position":2, "unique_depth":4 ,"ambiguous_depth”:0}}
```

Notes:

- Only positions with coverage ≥ min-coverage included
- Positions sorted numerically within each genome
- All depth values as integers
- Only positions within specified regions if regions provide

c. window-coverage-summary

A json object representing the **window-coverage-summary**.

Example:

```
{"genome1":{"window_start":0,     "window_end":99     ,"mean_unique”:2.5,
"mean_ambiguous”:0.3, “max_unique”:4, “max_ambiguous”:2},
"genome1":{"window_start":100,    "window_end":199    ,"mean_unique”:1.8,
"mean_ambiguous”:0.5, “max_unique”:3, “max_ambiguous”:2}}
```

Notes:

- Window boundaries align with window_size parameter
- Mean values rounded to one decimal place
- Max values as integers

- **Edge Cases**
    - Empty regions or regions with no coverage must appear in output
    - Partial windows at boundaries must be handled correctly
    - Zero coverage positions excluded from _coverage_detail.tsv
    - Coverage calculation must account for reads spanning window boundaries. Reads are assigned to only one window.

4.4. (EXTSIM) Detect and filter highly similar genomes during construction

Goal: This extension enhances the build process by detecting and optionally filtering out highly similar genomes from the reference database. This helps prevent ambiguous mappings caused by nearly identical genomes and improves the accuracy of downstream pseudo-alignment.

Command: Add support of the following optional arguments to the `reference` and `dumpref` commands that bulid the k-mer reference database (subsections 3.1, 3.3):

- --filter-similar: Flag to remove similar genomes from the reference database
- --similarity-threshold THRESHOLD: Threshold for considering genomes similar (between 0 and 1, default: 0.95)

Process:

1. **Initial Genome Ranking**
    - For each genome, calculate:
        - Count of unique k-mers (appearing only in this genome)
        - Total k-mer count
        - Genome length
    - Sort genomes in descending order by:
        - Unique k-mer count
        - If tied: Total k-mer count
        - If still tied: Genome length
        - If still tied: Input order of genome ID

2. **(Greedy) Filtering Process**
    - Initialize empty set of kept genomes
    - For each genome G in sorted order:
        - Compare G to all previously kept genomes
        - Calculate similarity score: $|K1 \cap K2| / \min(|K1|, |K2|)$
        - If similarity < threshold for all kept genomes:
            - Add G to kept set
        - Else:
            - Add G to filtered set, noting which kept genome caused removal

Output:

In addition to the basic command (`reference/dumpref`) output, print the following in a JSON format:

```
{"Similarity": similarity-results}
```

Where similarity-results is a json object defined below:

similarity-results:

Example:

```
{"genome1":{"kept":yes,     "unique_kmers":15000     ,"total_kmers”:50000,
"genome_length”:1200000, “similar_to”:NA, “similarity_score”:NA},
"genome2":{"kept":no,      "unique_kmers":14000      ,"total_kmers”:49000,
"genome_length”:1180000, “similar_to”:genome1, “similarity_score”:0.97},
"genome3":{"kept":yes,      "unique_kmers":13000      ,"total_kmers”:48000,
"genome_length”:1150000, “similar_to”:NA, “similarity_score”:NA},
"genome4":{"kept":no,      "unique_kmers":12000      ,"total_kmers”:47000,
"genome_length”:1100000, “similar_to”:genome1, “similarity_score”:0.96}}
```

Fields explained:

- kept: Whether genome was kept ("yes") or filtered ("no")
- unique_kmers: Number of k-mers unique to this genome
- total_kmers: Total number of k-mers in genome
- genome_length: Length of genome in bases
- similar_to: ID of kept genome that caused filtering (NA if kept)
- similarity_score: Similarity score to similar_to genome (NA if kept)

Impact on the basic command Output

- The k-mer reference database built will only include kept genomes.

Additional Requirements:

- The greedy approach ensures reproducibility as the order of processing is deterministic
- Processing in order of unique k-mers helps maintain diversity in the reference