Introduction to Focus Area: Advanced Algorithms

Group: The Sequence Seekers (Group 11)
Members: Anh Vo, Nicolò Retis, Rinoshan Parameswaran

Assignment 1

GitHub: https://github.com/vmeomeo/FUB_FocusArea_AdvAlg/

1 Background

In the field of computational biology, the search for specific markers within genomic data is a critical task with applications in vaccine development, disease tracking, and personalized medicine. Imagine a fictional world where a virus is rapidly spreading across the globe, and scientists are working against the clock to develop an effective vaccine. As part of this effort, virologists need to locate specific markers in the human genome to evaluate the vaccine's efficacy. This pressing need for accurate and efficient genomic analysis is the foundation of the study presented here.

The assignment at hand replicates this scenario, where the goal is to identify specific marker sequences within a reference genome (a partial segment of the human genome). Two search algorithms are explored:

1. **Naive Search:** A simple brute-force approach that slides a window across the genome to locate marker matches.
2. **FM-Index Search:** A state-of-the-art algorithm leveraging a compressed data structure derived from a suffix array for efficient searches.

These algorithms are benchmarked under varying conditions to assess their runtime and memory performance. Specifically, the study investigates:

- The **runtime and memory usage** of the naive and FM-index search algorithms for increasing query sizes (1,000, 10,000, 100,000, 1,000,000 queries) with a fixed marker length (100 base pairs).
- The **runtime performance** of both algorithms for queries of different lengths (40, 60, 80, and 100 base pairs) using a fixed number of queries (50 per length).

Through this analysis, we aim to identify the trade-offs between the simplicity of naive search and the efficiency of FM-index search, and how these trade-offs impact real-world applications like genomic marker identification.

2 Methods

2.1 Naive Search

- The naive search algorithm is a brute-force method that iterates through the reference genome and compares substrings of equal length to the markers.

- **Time Complexity:**
  - For a reference genome of length n and markers of total length m, the complexity is O(n*m).
- **Memory Usage:**
  - Minimal memory overhead as no additional data structures are used beyond the input sequences.

2.2 FM-Index Search

- The FM-index is a space-efficient data structure derived from the Burrows-Wheeler Transform and a suffix array. It enables fast substring searches through backward search algorithms.
- **Preprocessing**
  - The FM-index preprocesses the reference genome into a compressed data structure.
  - Preprocessing involves sorting suffixes and constructing auxiliary arrays.
- **Query Search:**
  - Uses backward search to find markers in O(k) time, where k is the marker length.
- **Time Complexity:**
  - The index construction is done in $O(n \log n)$.
  - Search: O(k) for each query.

2.3 Benchmarking Setup

- Queries were divided into subsets with specific lengths: 40, 60, 80, and 100.
- For each query length, the first 1000 queries were extracted from the respective files.
- Performance metrics:
  - **Runtime:** Measured in seconds for both Naive Search and FM-Index Search.
  - **Memory Usage:** Measured during runtime using the "memory profiler" library.
- For plots 4 and 5, we only use C++ because the Python code of fm-index also has a C++ backend, so we believe running C++ in the server would give reliable results, enough for us to see the difference when applying different parameters (query sequence length, number of queries, k number of errors)

2.4 Experimental Design

- The reference genome used is a subset of the human genome (`hg38_partial.fasta.gz`). For fm-index search, we also perform with full human genome (genome assembly GRCh38 GCF_000001405.26)

- Queries of different lengths were benchmarked using the following files:
  - `illumina_reads_40.fasta.gz`
  - `illumina_reads_60.fasta.gz`
  - `illumina_reads_80.fasta.gz`
  - `illumina_reads_100.fasta.gz`
- The FM-index was saved to disk to avoid redundant preprocessing for repeated searches.
- For FM-index search, we benchmark runtime of 10000 queries with k = 0, k = 1 and k = 2 errors of length 40, 60, 80, and 100.

2.5 Which algorithm is best suited?

For question 4 in assignment 2, we would say fm-index, firstly, because of its flexibility (it can align even with errors). Second, after having the index file, it's very fast to run, even with big queries like 1000000. Third, the size of the index file is much smaller compared to the suffix array-based search, thus, we can use fm-index for the whole human genome.

3 Implementations details

3.1 Helper function

Read fasta

Using IV2py package (Python)

```
def read_fasta_iv2py(file_path, limit=None):
    """
    Reads sequences from a FASTA file using iv2py with an optional limit.

    Args:
        file_path (str): Path to the FASTA file.
        limit (int): Maximum number of sequences to read. If None, reads all
sequences. Default: None

    Returns:
        list: List of sequences from the file.
    """
    sequences = []
    for i, record in enumerate(iv.fasta.reader(file=file_path)):
        if limit and i >= limit:
            break
        sequences.append(record.seq)
    return sequences
```

Without using IV2py package (Python)

3.2 Native Search

- A sliding window approach is used to extract substrings of the same length as the marker and compare them for equality.

```
def read_fasta(file_path, limit=None):
    """
    Reads sequences from a FASTA file.

    Args:
        file_path (str): Path to the FASTA file.
        limit (int): Maximum number of sequences to read. If None, reads all
sequences. Default: None

    Returns:
        list: List of sequences from the file.

    """
    sequences = []
    with gzip.open(file_path, 'rt') if file_path.endswith('.gz') else
open(file_path, 'r') as f:
        current_sequence = []
        for i, line in enumerate(f):
            if line.startswith(">"):
                if current_sequence:
                    sequences.append("".join(current_sequence))
                    current_sequence = []
                if limit and len(sequences) >= limit:
                    break
            else:
                current_sequence.append(line.strip())
        if current_sequence and (not limit or len(sequences) < limit):
            sequences.append("".join(current_sequence))
    return sequences
```

- Each marker is searched individually, and positions of matches are stored in a dictionary.

Code Details:

- **Input:**
  - The reference genome (`hg38_partial.fasta.gz`) was loaded into memory as a single string.
  - Markers loaded as a list of sequences from query files.
- **Output:**
  - A dictionary where keys are markers and values are lists of starting positions.
- **Benchmarking:**
  - Wrapped in a function to measure runtime and memory usage.

Python

```
# Naive search function
def naive_search(reference, markers):
	"""
	Naive search for markers in the reference genome.

	Args:
	reference (str): The reference genome sequence.
	markers (list): List of marker sequences to search for.

	Returns:
	dict: A dictionary where keys are markers, and values are lists of start
positions.
	"""
	results = {}
	for marker in markers:
	positions = []
	marker_len = len(marker)
	for i in range(len(reference) - marker_len + 1):
		if reference[i:i + marker_len] == marker:
		positions.append(i)
	results[marker] = positions
	return results
```

C++

```
// prints out all occurences of query inside of ref
void findOccurences(std::vector<seqan3::dna5> const& ref,
std::vector<seqan3::dna5> const& query) {
   //!TODO ImplementMe
   std::vector<size_t> positions; // Store the start pos of matches
   size_t query_size = query.size();
   // Naive search: Slide a window over the ref & compare to queries
   for (size_t i = 0; i <= ref.size() - query_size; i++) {
       bool match = true;

       //Compare each char in the window to the query
       for (size_t j = 0; j < query_size; j++) {
           if (ref[i + j] != query[j]) {
               match = false;
               break;
           }
       }

       if (match) {
           positions.push_back(i); // Record the start pos of match
       }
   }
   // Output the results
   if (positions.empty()) {
       seqan3::debug_stream << "Query not found in the reference.\n";
   } else {
       seqan3::debug_stream << "Query found at positions: ";
       for (auto pos : positions) {
           seqan3::debug_stream << pos << " ";
       }
       seqan3::debug_stream << "\n";
   }
}
```

3.3 Suffix Array Based Search

Code Details:

- Utilized the `iv2py` library for FM-index construction and query searching.
- Preprocessing involved:
  - The reference genome was normalized using `iv.normalize`.
  - Constructing the FM-index with a sampling rate of 16 for a balance between query efficiency and memory usage.
- For each query, the FM-index backward search was applied, returning the positions of matches.
  - Wrapped in a function to measure runtime and memory usage.

Code Details:

- **FM-Index Construction:**
  - **Input:** Reference genome.
  - **Output:** Serialized FM-index stored as `fmindex` on disk.
- **Query Search:**

- **Input:** FM-index object and a list of markers.
- **Output:** Dictionary mapping markers to lists of starting positions.

- **Benchmarking:**
  - Both FM-index construction and query search were benchmarked separately.

Python

```
import iv2py as iv
import os
import argparse
import time
from memory_profiler import memory_usage

# Helper function for reading reference genome
def load_reference(reference_file, fmindex_file):
	"""
	Load reference genome and build/load FM-index.
	"""
	if os.path.exists(fmindex_file):
	# Load FM-index from disk
	print("Loading FM-index from disk...")
	index = iv.fmindex(path=fmindex_file)
	else:
	print("Building FM-index...")
	reference = [iv.normalize(rec.seq) for rec in
iv.fasta.reader(reference_file)]
	index = iv.fmindex(reference=reference, samplingRate=16)
	index.save(fmindex_file)
	print("FM-index saved to disk.")
	return index

# Helper function for searching queries
def search_queries(index, query_file, output_file):
	"""
	Search for query sequences in the FM-index and save results to an output
file.
	"""
	with open(output_file, "w") as out_file:
	for record in iv.fasta.reader(file=query_file):
		res = index.search(record.seq)
		out_file.write(f"Marker '{record.seq}' found at positions: {res}\n")
	print(f"Results saved to {output_file}")

# Helper function for benchmarking
def benchmark_function(function, *args, **kwargs):
	"""
	Benchmark runtime and memory usage of a function.
	"""
	start_time = time.time()
	memory_before = memory_usage()[0]
	function(*args, **kwargs)
	memory_after = memory_usage()[0]
	print(f"Runtime: {time.time() - start_time:.2f} seconds")
	print(f"Memory used: {memory_after - memory_before:.2f} MB")

# Main execution
def main(reference_file, query_file, fmindex_file="fmindex",
output_file="search_results.txt"):
	# Load reference genome and build/load FM-index
	index = load_reference(reference_file, fmindex_file)

	# Benchmark search process
	print("Starting query search...")
	benchmark_function(search_queries, index, query_file, output_file)

# Specify file paths
reference_file = "hg38_partial.fasta.gz"
query_file = "illumina_reads_40.fasta.gz"
fmindex_file = "fmindex"
output_file = "search_results.txt"

# Run the main program
main(reference_file, query_file, fmindex_file, output_file)
```

C++

```
// Array that should hold the future suffix array
   std::vector<saidx_t> suffixarray;
   suffix array.resize(reference.size()); // resizing the array, so it can hold the complete SA`
   sauchar_t const *str = reinterpret_cast<sauchar_t const *>(reference.data());
   divsufsort(str, suffixarray.data(), reference.size()); // construct suffix array

   // Binary search for each query
   for (auto &q : queries)
   {
       auto compare_lower = [&](int suffix_start, const auto &query_str)
       {
           // return ref_str.compare(suffix_start, query_str.size(), query_str) <   0;
        for (int i = 0; i < query_str.size() && (suffix_start + i)reference.size(); i++)
           {
               if (reference[suffix_start + i] < query_str[i])
               {
                   return true;
               }
               if (reference[suffix_start + i] > query_str[i])
               {
                   return false;
               }
           }
           return false;
       };

       auto compare_upper = [&](const auto &query_str, int suffix_start)
       {
           // return ref_str.compare(suffix_start, query_str.size(), query_str) < 0;
           for (int i = 0; i < query_str.size() && (suffix_start + i) < reference.size(); i++)
           {
               if (reference[suffix_start + i] > query_str[i])
               {
                   return true;
               }
               if (reference[suffix_start + i] < query_str[i])
               {
                   return false;
               }
           }
           return false;
       };

       auto lower = std::lower_bound(
           suffixarray.begin(), suffixarray.end(), q, compare_lower);
       auto upper = std::upper_bound(
           suffixarray.begin(), suffixarray.end(), q, compare_upper);

       if (lower == upper)
       {
           seqan3::debug_stream << "Query not found.\n";
       }
       else
       {
           seqan3::debug_stream << "Query found at positions: ";
           for (auto it = lower; it != upper; ++it)
           {
               seqan3::debug_stream << *it << " ";
           }
           seqan3::debug_stream << "\n";
       }
   }
```

Benchmark

1. **Query Size Benchmarking**

   - Queries are read from the file `illumina_reads_100.fasta.gz`. Only the first 1,000, 10,000, 100,000, or 1,000,000 queries are processed for each benchmark.
   - Runtime and memory usage were recorded and plotted for comparison.

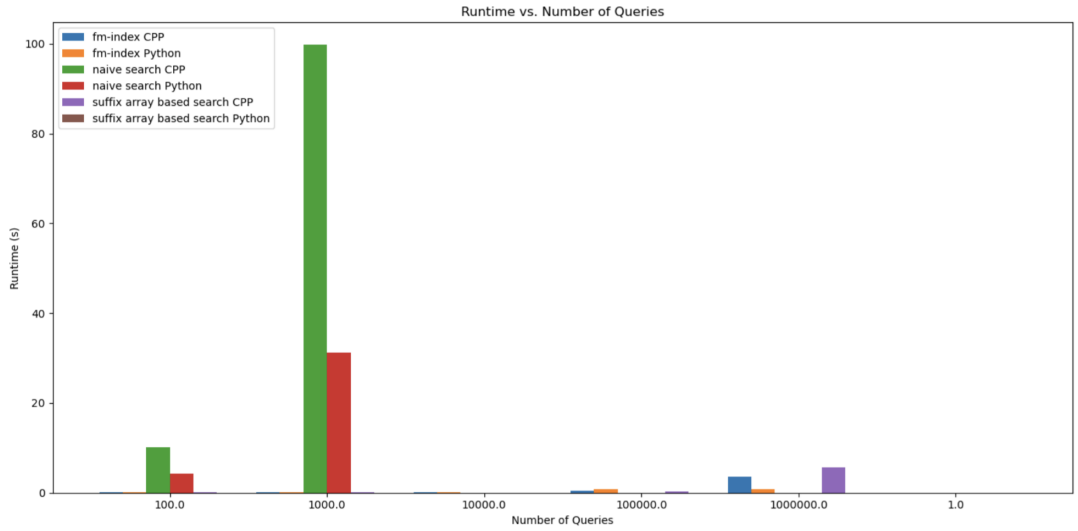
2. **Query Length Benchmarking**

   - Benchmarks were conducted for both algorithms on:
     - Query lengths: 40, 60, 80, 100.
     - Query sizes: The first 50 queries from each length-specific file.
   - Runtime and memory usage were recorded and plotted for comparison.

Code and plots are in jupyter notebook in our GitHub repository

3.4 Benchmark results

1. **Benchmark (runtime and memory) the naive, suffix array based search and fm index for 1'000, 10'000, 100'000 1'000'000 queries of length 100**



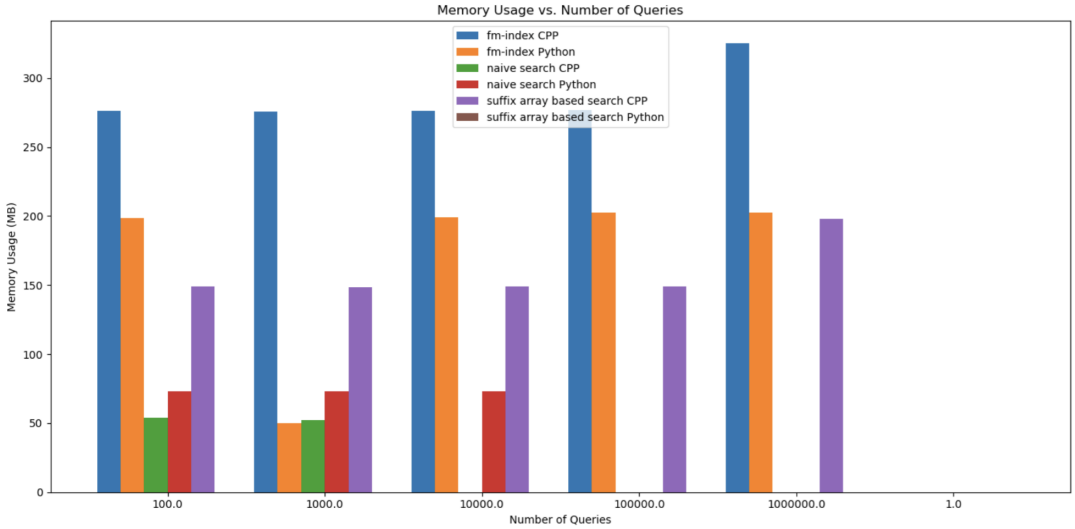
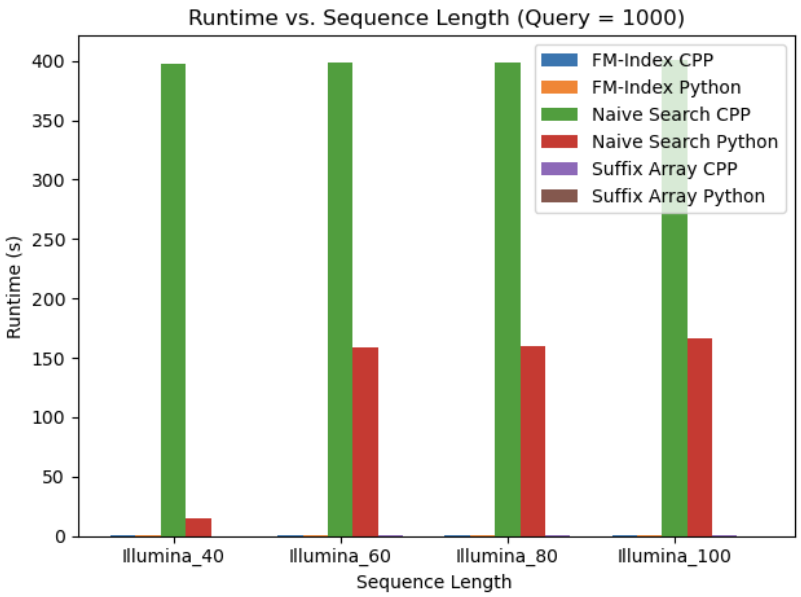


Table 1: Performance of Search Algorithms

Algorithms	Language	Number of Queries	Input (Seq Length)	Memory (KB)	Time (s)
Naive Search	Python	100	illumina_100	298484	17.14
Naive Search	CPP	100	illumina_100	220816	40.78
Naive Search	Python	1000	illumina_100	298696	165.9
Naive Search	CPP	1000	illumina_100	212456	401.12
Naive Search	Python	10000	illumina_100	298680	1540
Naive Search	CPP	10000	illumina_100	Forgot to run	Forgot to run
Naive Search	Python + CPP	Over 100000	illumina_100	Not measured	Not measured
Suffix Array Based Search	CPP	100	illumina_100	609668	0.18
Suffix Array Based Search	CPP	1000	illumina_100	610236	0.11
Suffix Array Based Search	CPP	10000	illumina_100	609372	0.15
Suffix Array Based Search	CPP	100000	illumina_100	609648	1.15
Suffix Array Based Search	CPP	1000000	illumina_100	810152	11.59
FM-Index	Python	1000	illumina_40	138039	0.23
FM-Index	CPP	1000	illumina_40	1127576	0.16
FM-Index	Python	1000	illumina_60	138003.9	0.23
FM-Index	CPP	1000	illumina_60	1128852	0.19
FM-Index	Python	1000	illumina_80	138015.6	0.23
FM-Index	CPP	1000	illumina_80	1130412	0.21
FM-Index	Python	1000	illumina_100	138039	0.25
FM-Index	CPP	1000	illumina_100	1131976	0.2
FM-Index	Python	10000	illumina_100	814320	0.56
FM-Index	CPP	10000	illumina_100	1132080	0.2
FM-Index	Python	100000	illumina_100	830195	3.3
FM-Index	CPP	100000	illumina_100	1132444	1.83
FM-Index	Python	1000000	illumina_100	830207	3.35
FM-Index	CPP	1000000	illumina_100	1331372	14.62

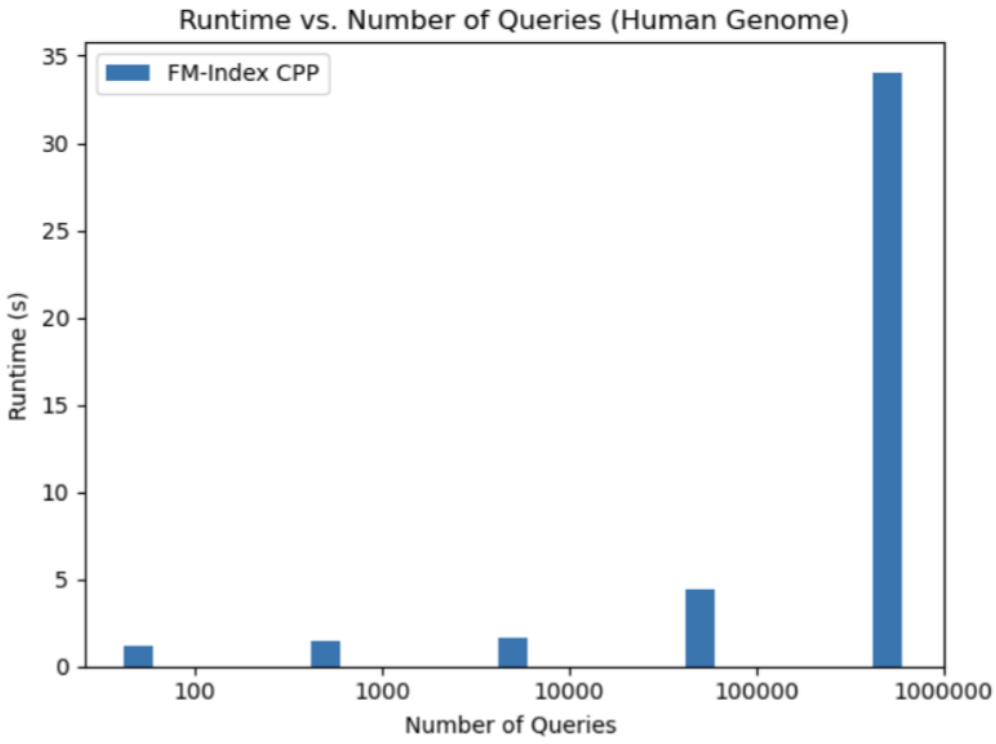
2. **Benchmark (runtime) queries of the length 40, 60, 80, and 100 with a suitable number of queries (1000)**



3. **Benchmark (runtime) fm index with human genome as reference for 1'000, 10'000, 100'000 1'000'000 queries**

Table 2: Performance Metrics of Search Algorithms

Algorithms	Language	Number of Queries	Input (Seq Length)	Memory (KB)	Time (s)
Naive Search	Python	1000	illumina_40	298596	15.02
Naive Search	CPP	1000	illumina_40	212460	397.92
Naive Search	Python	1000	illumina_60	298604	158.73
Naive Search	CPP	1000	illumina_60	215564	398.34
Naive Search	Python	1000	illumina_80	298436	159.54
Naive Search	CPP	1000	illumina_80	215526	398.89
Naive Search	Python	1000	illumina_100	298696	165.9
Naive Search	CPP	1000	illumina_100	212456	401.12
Suffix Array Based Search	CPP	1000	illumina_40	605028	0.05
Suffix Array Based Search	CPP	1000	illumina_60	607792	0.84
Suffix Array Based Search	CPP	1000	illumina_80	609284	0.09
Suffix Array Based Search	CPP	10000	illumina_100	609372	0.15
FM-Index	Python	1000	illumina_40	138039	0.23
FM-Index	CPP	1000	illumina_40	1125756	0.16
FM-Index	Python	1000	illumina_60	138003.9	0.16
FM-Index	CPP	1000	illumina_60	1128852	0.23
FM-Index	Python	1000	illumina_80	138015.6	0.23
FM-Index	CPP	1000	illumina_80	1130412	0.21
FM-Index	Python	1000	illumina_100	138039	0.25
FM-Index	CPP	1000	illumina_100	1131976	0.2



4. **Benchmark (runtime) of 10000 queries with k=0,k=1 and k=2 errors of length 40,60,80 and 100**

Table 3: Performance of FM-Index with Full Human Genome

Algorithms	Language	Number of Queries	Input (Seq Length)	Memory (KB)	Time (s)
FM-Index with Full Human Genome	CPP	100	illumina_100	45521604	1.21
FM-Index with Full Human Genome	CPP	1000	illumina_100	45521640	1.43
FM-Index with Full Human Genome	CPP	10000	illumina_100	45521660	1.6
FM-Index with Full Human Genome	CPP	100000	illumina_100	45521992	4.39
FM-Index with Full Human Genome	CPP	1000000	illumina_100	45720568	34.05

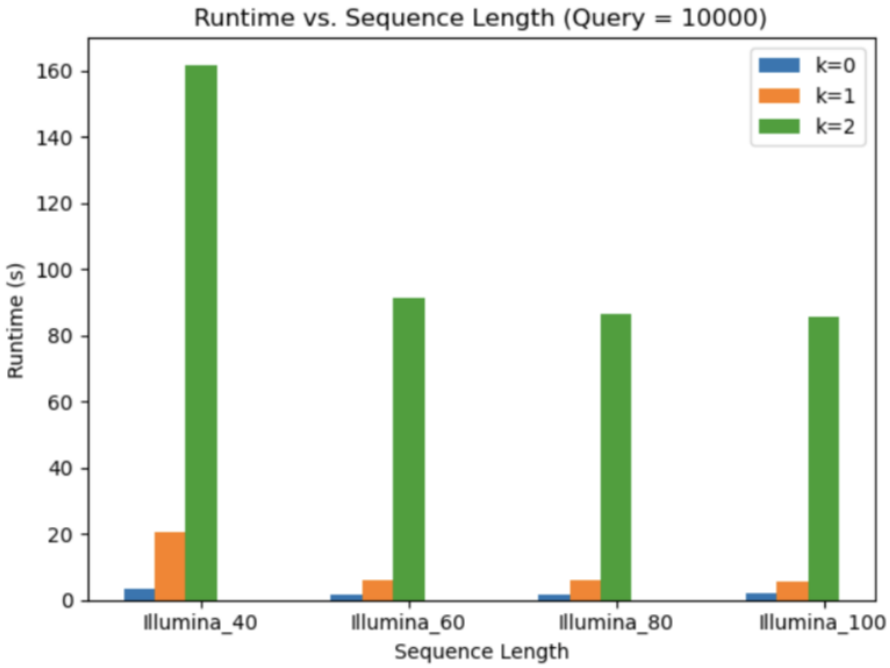


Table 4: Performance of FM-Index with Full Human Genome for Different k Values

Algorithms	Language	Number of Queries	Input (Seq Length)	Memory (KB)	Time (s)
FM-Index with Full Human Genome, $k = 0$	CPP	10000	illumina_40	45515084	3.31
FM-Index with Full Human Genome, $k = 0$	CPP	10000	illumina_60	45518012	1.59
FM-Index with Full Human Genome, $k = 0$	CPP	10000	illumina_80	45519776	1.44
FM-Index with Full Human Genome, $k = 0$	CPP	10000	illumina_100	45521308	1.82
FM-Index with Full Human Genome, $k = 1$	CPP	10000	illumina_40	45522360	20.71
FM-Index with Full Human Genome, $k = 1$	CPP	10000	illumina_60	45518188	6.07
FM-Index with Full Human Genome, $k = 1$	CPP	10000	illumina_80	45519812	6.06
FM-Index with Full Human Genome, $k = 1$	CPP	10000	illumina_100	45521388	5.4
FM-Index with Full Human Genome, $k = 2$	CPP	10000	illumina_40	45564920	161.78
FM-Index with Full Human Genome, $k = 2$	CPP	10000	illumina_60	45522544	91.18
FM-Index with Full Human Genome, $k = 2$	CPP	10000	illumina_80	45519796	86.54
FM-Index with Full Human Genome, $k = 2$	CPP	10000	illumina_100	45521400	85.48

Application of AI in this assignment

We utilized AI chat as support for Python coding and for rewriting and rephrasing sentences. Chatbots were also employed in this project for benchmarking purposes, as we were unfamiliar with how to perform it.