IFA Algo Final Report

Charlie Rettig, Johannes von Kleist, Marcus Wenau

January 2025

1 Scientific Background

The rapid identification of genetic markers within the human genome plays a crucial role in modern vaccine development. This computational challenge involves locating specific short DNA sequences (markers) within the much larger reference genome sequence (hg38).

The problem is essentially one of pattern matching on a large scale: searching for thousands or even millions of short sequences (ca. 100 base pairs) within a reference text of billions of characters. Four main computational approaches are commonly used:

1. Naive string matching, which directly compares sequences at each possible position, offering simplicity but requiring O(nm) time for each marker (where n is the reference length and m is the marker length).

2. Suffix array-based matching, which preprocesses the reference sequence into a searchable index. While requiring initial O(n log n) preprocessing time and O(n) space, it enables faster O(m log n) searches per marker.

3. FM-index based matching, which combines the Burrows-Wheeler transform with auxiliary data structures. It provides space-efficient indexing with O(n) construction time and supports O(m) time exact matching queries, making it particularly suitable for large-scale genomic data.

4. Pigeonhole-approach, which increases the speed with which matches with k errors can be found by splitting the query into k + 1 segments, ensuring that at least one segment must match exactly within the reference genome. This method significantly reduces the search space for potential matches and is particularly useful when combined with suffix arrays or FM-index structures. The pigeonhole principle enables rapid approximate matching, helping to identify genetic variations, minor mutations, or sequencing errors that would otherwise be missed in exact matching approaches.

In real-world applications, perfect matches are rarely sufficient due to genetic variations and sequencing errors. Approximate matching (allowing for k mismatches or errors) becomes crucial for robust marker identification. This capability to detect sequences with small variations (typically k = 1 or 2 errors) helps identify relevant markers despite minor mutations or sequencing artifacts, enhancing the sensitivity of vaccine effectiveness monitoring.

The choice between these approaches involves trade-offs between preprocessing time, memory usage, and query performance, particularly important given the time-sensitive nature of vaccine development. While exact matching provides a foundation for marker identification, the ability to perform approximate matching with controlled error tolerance adds an essential layer of flexibility for practical applications in genomic analysis.

2 Methods

We evaluated two DNA sequence search methods: a naive search and a suffix array-based search. Both approaches were implemented in C++ using the SeqAn3 library and tested on reference and query DNA sequences provided in FASTA format.

Naive Search
The naive search method sequentially scans the reference DNA sequence to locate each occurrence of the query sequence. Matches are identified by direct comparison of substrings, and the position of the each match is reported, or the absence of a match is noted.

Suffix Array-Based Search
The suffix array-based approach leverages a precomputed suffix array for efficient searching. This data structure allows the method to quickly narrow down the range of potential matches using binary search, significantly improving search performance compared to the naive approach.

FM Index Search
The FM Index approach was utilized to perform efficient pattern searches of nucleotide query sequences within a reference dataset. Query sequences were loaded from an input file, and an FM Index constructed from the reference was deserialized from a binary file. To ensure sufficient data for analysis, queries were duplicated to reach the desired number, and the seqan3::search function was applied with a configurable error tolerance. The resulting match positions were recorded and output to a file for further analysis.

Experimental Setup
Both methods were parameterized to accept input reference and query files, as well as the number of queries to process. To ensure sufficient query data, the input queries were duplicated as needed. The naive method was applied directly to the reference, while the suffix array method required preprocessing to construct the suffix array prior to querying.

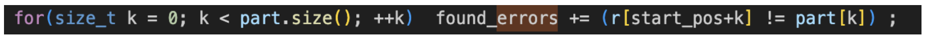
Pigeonhole approach
The Pigeonhole search approach was used to efficiently identify approximate matches of nucleotide query sequences within the reference data. Given the specified errors k, each query sequence was divided into k + 1 segments, ensuring that at least one segment remained error-free. The FM Index was used to search for these segments in the reference. We implemented two different variants of searching using this principle: First, the PEX algorithm where matching segment positions were then extended and refined by iteratively merging adjacent segments and verifying full-sequence alignment within an allowable error threshold or the split. Second, a simpler search without indels where the pieces of the query are matched to the pieces of the text bordering on the original match. The final match positions were recorded and output to a file for further analysis.

3 Implementation Details

For the python programs, the two non FM-index-based programs are named `IFA_ALGO1_NAIVE.py` and `IFA_ALGO1_SUFFIX.py`. The FM-index programs are named `IFA_ALGO_FM.py`, `IFA_ALGO_FM_CON.py`, and . They can be run in an environment which has iv2py and numpy linked (conda used locally). The programs then ask for user input for the reference and query files, which can be manually entered via Command Line. The FM-index programs additionally asks for error amount and query count. The program for the Pigeonhold FM Index-based search is named `FMIndex_Pigeon_Search.py`. It can be run in the same environment as the aforementioned programs. When running the script on the command line, the user must input 4 positional arguments: the reference FASTA file path (file1, the query fasta file path (file2), the number of queries to be used from the query file (repeats) and the number of errors tolerated in the search (errors). To prevent the repeated building of the FM index on every run of `FMIndex_Pigeon_Search.py`, we created a supplementary FM index-generating file `index_generation.py`. When run on the command line, the user must only give one positional argument: the reference genome file path.

For the C++ programs only the corresponding C++ files were edited, the code skeletons have been largely preserved. For the two non-FM-index based searches, `naive_search.cpp` and `suffixarray_search.cpp`, files and query counts can be passed to the program when calling it as specified in the code skeleton. The C++ programs write to a .txt file each called `naive_output.txt` and `output.txt` respectively. The file `fmindex_search.cpp` was edited for the FM-indices, as well as `fmindex_pigeon_search.cpp` for the pigeon-search. To run both, an index must first be constructed, whose file path can then also be passed as an argument to these programs. These programs also await user input and prompt the user for the arguments, including query files, reference files, query count, error count and FM-index.

The naive programs both implement a sliding window approach, where the query is attempted to be matched to a sliding window across the reference string.



```
for(size_t k = 0; k < part.size(); ++k)  found_errors += (r[start_pos+k] != part[k]) ;
```

Figure 1: For-loop that could be vectorized, which we think is also getting vectorized

These occurrences of matches are then reported. The suffix array programs implement the suffix array binary search approach described in the lecture, finding the left and right borders in the suffix array and then reporting the positions inside these bounds. Otherwise no really interesting additions or design decisions were made for these programs. All of the programs report the same occurrences, as such these are taken to be true occurrences.

The FM-index programs both implement the search schemes described in the lecture, using seqan3's existing tools as well. The direct FM-index search was not modified in any very interesting ways, and just used the constructed FM-index and seqan3's FM-index-search as described above. The pigeon search was implemented manually, with the splitting of the query as well as the verification being implemented by us. Despite some difficulties surrounding exact accessing and error matching, this implementation also returns the same, expected results as the direct FM-index search and the non-index search schemes. Additionally, we achieved relatively strong speedup in the verification step by simplifying one of our for-loops down to where it could be vectorized, which is visible in Fig. 1. However, the loading time of the human genome of about 20 seconds was irreducible for us.

4 Benchmarks

These methods were compared in terms of their ability to locate query sequences within the reference efficiently and accurately with benchmarked runtimes. All of the non-FM runs were done on an M3 Macbook Air, in part due to our inability to access the university's servers that week. All of the FM-index runs, unless indicated otherwise, were done on the FU's compute servers. A max time limit of 20 minutes was set for each amount of queries of length 100. Listed below are tables of the runtime and memory consumption for all the programs.

4.1 Non-FM programs

Table 1: Runtime of the various non-FM programs on queries of length 100

Amount of Queries/Program	**Py Naive**	**Py Suffix**	**C++ Naive**	**C++ Suffix**
1000	Timed out	156.54s	425.46s	138.65s
10.000	Timed out	Timed out	Timed out	Timed out (21 minutes)
100.000	Timed out	Timed out	Timed out	Timed out
1.000.000	Timed out	Timed out	Timed out	Timed out

Table 2: Memory consumption of the various non-FM programs on queries of length 100

Program	**Py Naive**	**Py Suffix**	**C++ Naive**	**C++ Suffix**
1000	Timed out	4.044GB	0.494 GB	1.032 GB
10.000	Timed out	Timed out	Timed out	0.925 GB
100.000	Timed out	Timed out	Timed out	Timed out
1.000.000	Timed out	Timed out	Timed out	Timed out

Table 3: Runtimes of the various non-FM programs on 1000 queries of various lengths

Program	**Py Naive**	**Py Suffix**	**C++ Naive**	**C++ Suffix**
40	Timed out	154.62s	425.40s	142.05s
60	Timed out	150.74s	427.69s	134.01s
80	Timed out	151.77s	426.81s	141.76s
100	Timed out	156.54s	425.46s	138.65s

As is visible above, all the programs time-out starting at 10.000 queries of length 100, with the suffixes being much faster, however they require some more memory than the sliding window approaches. Python also requires more time and more memory than C++.

Additionally, we benchmarked the performance of these programs on 1000 queries of length 40, 60, 80, and 100, again with a (soft) max time limit of 10 minutes. The naive solution could not be reasonably benchmarked, as a run of 10 queries took 8 minutes already.

We expected the runtime on the shorter queries to be a little shorter for the programs, but this turned out not to be the case, the length of the queries had an entirely negligible impact on the runtime of the python programs, with no clear trend visible for any of the programs. The fluctuations are probably due to random differences when running on the laptop, only one run each was done, so no average was made.

4.2 FM-programs

Table 4: Runtimes and memory usage of the construction of the FM-indices

	Memory Usage	**Runtime**
Python	N/A	10m22s
C++	43.2 GB	11m20s

For the FM-based methods, an FM-index first had to be constructed. We used the whole human genome, version GCF_000001405.26_GRCh38 and used

Table 5: Runtimes of the various FM-index-based programs on queries of length 100 against the full genome

Amount of Queries/Program	Py FM	C++ FM	Py Pigeon	C++ Pigeon
1000	Timed out	1.66s	N/A	21.16s
10.000	Timed out	1.99s	N/A	20.71s
100.000	Timed out	7.32s	N/A	21.09s
1.000.000	Timed out	33.59s	N/A	42.68s

Note: Python Pigeon query runtimes are not available for the full reference genome due to exceeded disk quota on servers.

Table 6: Runtimes of Python Pigeon on queries of length 100 against the partial genome

Amount of Queries/Program	Py Pigeon
1000	1.32s
10.000	9.21s
100.000	96.27s
1.000.000	932.1s

Table 7: Memory consumption of the FM-index-based programs on queries of length 100

Program	Py FM	C++ FM	Py Pigeon	C++ Pigeon
1000	Timed out	2.272904GB	N/A	5.654496GB
10.000	Timed out	2.272832GB	N/A	5.654824GB
100.000	Timed out	2.273540GB	N/A	5.654740GB
1.000.000	Timed out	2.471988GB	N/A	5.853700GB

Note: Python Pigeon query memory usage are not available for the full reference genome due to exceeded disk quota on servers.

Table 8: Memory consumption of Python Pigeon on queries of length 100 against the partial genome

Amount of Queries/Program	Py Pigeon
1000	14.22MB
10.000	14.25MB
100.000	14.25MB
1.000.000	14.25MB

pre-implemented methods for the construction. In Python this took x time and y memory, in C++ this took a little over 11 minutes and 43 GBs of memory.

The FM-index search for C++ had stellar performance: being far and away the fastest method and being capable of handling the full human genome. It's performance on queries of various lengths and different error counts is plotted in Fig. 2. The python FM-index implementation worked well on the partial human genome, however the suffix array construction timed out for the full human genome, even on the server. For the FM-indices, lower errors lead to exponentially shorter run-times over 100.000 queries, and the query length seemed to (very counterintuitively) anti-proportionately effect run time - shorter queries led to longer run times, even when keeping the amount of queries the same! An explanation offered during the tutorial was that the shorter queries had a higher "error density" than the longer ones, which made the FM-index approach slower on these as well. Some of this variation may also be a computing artifact, for the shorter queries with 0 errors less CPU usage was inexplicably afforded, which might explain the longer runtimes for this set in particular.

The pigeon search methods showed similar results, with the query length anti-proportionately affecting run time. Its performance on queries of various lengths and different error counts is plotted in Fig. 3. The pigeon-search was quite a bit faster, especially with a higher amount of errors, however with 0 errors the "naive" FM-index search was faster, due to each pigeon search loading the human genome which takes ca. 20 seconds. Outside this loading time, the search is blazingly fast, even for searches with 1.000.000 queries or 3 errors on queries of length 100 taking less than a minute.

As far as memory usage is concerned, the direct and pigeon search both require large amount of memory: C++ implementations take 2.2-2.5 GB and 5.6-5.9 GB respectively. Python implementations take 14.22-14.25 MB on the partial genome.

Table 9: Runtime of FM-Index Pigeon Search with 0-2 Errors over 10000 Queries for C++

Program	**C++ (0 Err)**	**C++ (1 Err)**	**C++ (2 Err)**	**Py (0 Err)**	**Py (1 Err)**	**Py (2 Err)**
40	0m25.75s	3m27.59s	24m37.76s	Timed out	Timed out	Timed out
60	0m5.89s	0m53.95	15m06.69s	Timed out	Timed out	Timed out
80	0m04.17s	0m46.10s	14m23.51s	Timed out	Timed out	Timed out
100	0m4.46s	0m44.9s	14m20s	Timed out	Timed out	Timed out

Table 10: Runtime of FM-Index Pigeon Search with 0-3 Errors over 10000 Queries against partial genome for C++

Length of the Queries / Amount of errors	**0 Errors**	**1 Error**	**2 Errors**	**3 Errors**
40	0m21.5s	1m9.1s	4m59s	26m03s
60	20.869s	27.252s	1m34.028s	5m41.001s
80	0m20.987s	0m22.187s	0m39.662s	2m02.978s
100	0m20.470s	0m20.792s	0m27.157s	0m55.313s

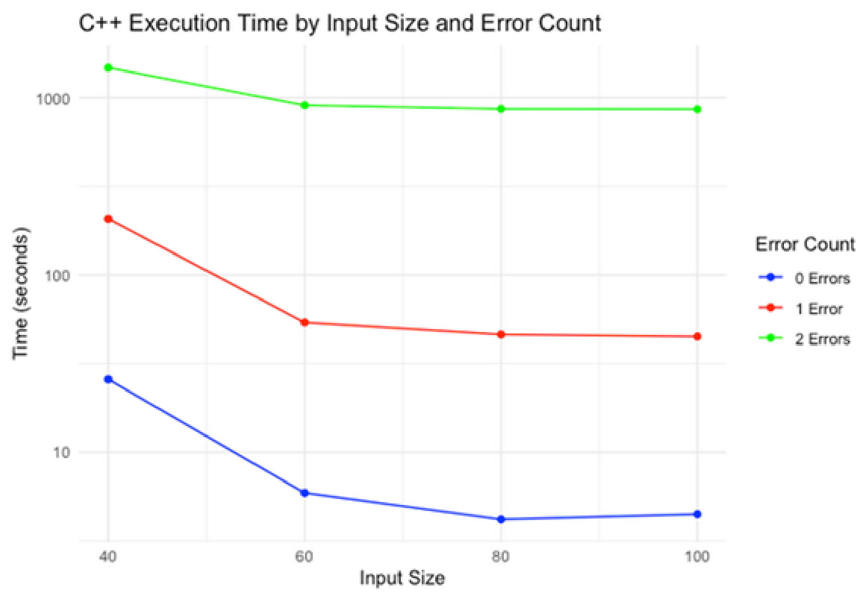


Figure 2: Performance of the C++ FM-Index Search, time in s plotted against query length

Table 11: Runtime of FM-Index Pigeon Search with 0-3 Errors over 10000 Queries against partial genome for Python

Length of the Queries / Amount of errors	**0 Errors**	**1 Error**	**2 Errors**	**3 Errors**
40	0m0.70s	4m7.2s	Timed out	Timed out
60	0m0.67s	47.50s	Timed out	Timed out
80	0m0.72s	0m14.87s	Timed out	Timed out
100	0m0.72s	0m9.81s	Timed out	Timed out

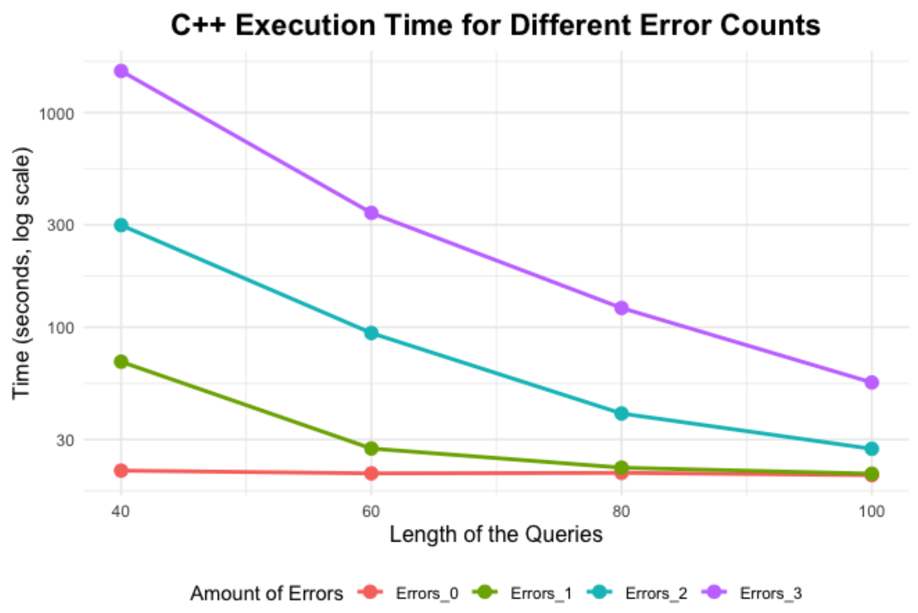


Figure 3: Performance of the C++ pigeon search, time in s plotted against query length

5 Conclusions

The index based methods showed the best performance, especially with their ability to readily integrate errors into their searches. Of the index-based methods, pigeon search performs the best, with the difference being especially noticeable with shorter query lengths and higher error counts. However, these methods require the construction of an FM-index, which takes up not only time (only once though) but also a great deal of space. As such there are scenarios in which the non-index methods may find some use. C++ was also found to be faster than python in general.

6 AI Usage

The LLM usage in this report was oriignally restricted to helping figure out the errors when trying to build via cmake and one line of C++ code: `int M = static_cast<int>(ceil((R + L) / 2.0));`.

This line was chosen to see what Claude 3.5 Sonnet (by Anthropic AI) would suggest, no other changes were taken from the suggestions.

For the pigeon search, Deepseek was used to debug a segfault error. Deepseek was also used to help format some of the TeX used in the creation of this report.