Phase 1 — Algorithm Description & Analysis

June 30, 2025

Contents

1 Initialisation and Main Driver

1.1 Driver Prototype

```
void analyzePair(const std::string& fileA,
                 const std::string& fileB);
```

1.2 Steps

1.1 Tokenisation

- Launch the stand-alone executable:
  `./tokenizer <file.cpp>`
- Each output line has the form `<kind> - <spelling> <line> <column>`

- Lines are parsed into the structure

```
struct TokenInfo {
    int kind; // CXCursorKind
    unsigned line; // 1based
    unsigned column; // 1based
};
using Tokens = std::vector<TokenInfo>;
```

1.2 Exact-run map

- Build `std::map<int, std::set<int>> submission2_positions` that stores *all* indices of every token-kind appearing in submission 2.

1.3 Main scan over submission 1

- For each index `i` in submission 1:
  1. Determine the longest exact run starting at `i`: `match_len = longestExactRunFrom(...)`
  2. If `match_len` ≥ 10
     3. Print a heading (once):
        `Pattern matches of length >= 10 from submission1:`
     4. Print the match information:
        `Match starting at submission1[i]:  Length = match_len`
     5. Add `match_len` to the global `SUM`.
     6. Skip ahead: `i += match_len - 1` (greedy, non-overlapping).

2 Core Functions

2.1 `Tokens tokenizeFile(const std::string& cppFile);`

Shells out to `./tokenizer` and fills `TokenInfo{kind,line,column}`.

2.2 `std::pair<int, Positions> lcsWithPositions(const Tokens& A, const Tokens& B);`

Classic $O(nm)$ dynamic-programming LCS that back-tracks to recover the vector of aligned positions.

2.3 `int longestExactRunFrom(...)`

Greedy search for the longest contiguous identical run starting at position `posA` in A.
If the run is ≥ 10 tokens, its indices are erased from the candidate map to prevent double-counting.

2.4 `int longestFuzzyBlock(const Positions& P, int& startA, int& startB);`

Scans P for the largest range $[i, j]$ such that

$$\begin{aligned}&\text{blockLen} \;\geq\; 24,\\ &\text{blockLen} > 0.8\,\big(P_j.first - P_i.first + 1\big),\\ &\text{blockLen} > 0.8\,\big(P_j.second - P_i.second + 1\big).\end{aligned}$$

Returns `blockLen` and start indices in each submission.

2.5 `double plagiarismScore(const Tokens& A, const Tokens& B, int SUM);`

Let $n_1 = |A|$ and $n_2 = |B|$. The score formula is

$$\text{score} = \frac{n_1}{n_1 + n_2}\,\frac{\text{SUM}}{n_2} + \frac{n_2}{n_1 + n_2}\,\frac{\text{SUM}}{n_1}.$$

3 Console Output Example

```
Length of LCS: 467
Pattern matches of length >= 10 from submission1:
Match starting at submission1[195]: Length = 101
Match starting at submission1[296]: Length = 260
Match starting at submission1[576]: Length = 17
Match starting at submission1[593]: Length = 45
plag : 1
sum_of_all_matched_patterns : 423
max_len : 432
position_1 : 128
position_2 : 77
Fuzzy block spans  fileA.cpp lines 37-52  and  fileB.cpp lines 19-34
```

If no ≥ 10-token runs exist, the checker prints:
`No pattern matches of length >= 10 were found in submission1.`

4 Complexity Analysis

4.1 Asymptotic Bounds

Stage	**Time Complexity**	**Space**
Tokenisation (external)	$O(N)$ per file	$O(1)$ (inside checker)
LCS DP	$O(n\,m)$	$O(n\,m)$
Fuzzy-block scan	$O(L^2)$	$O(1)$
Exact-run search	$\sum O(p_i \log p_j)$	$O(k+m)$
Overall (dom.)	$O(n\,m)$	$O(n\,m)$

Table 1: Worst-case asymptotic costs (notation defined in Table 2).

4.2 Symbol Definitions

Symbol	Meaning
n	$\lvert A\rvert$ — tokens in submission 1
m	$\lvert B\rvert$ — tokens in submission 2
k	$\lvert K\rvert$ — distinct cursor kinds
L	LCS length ($\leq \min\{n, m\}$)
R	number of ≥ 10-token exact runs
p_i	candidate positions for run r_i
$\sum n_i$	total length of exact runs (`SUM`)

Table 2: Notation used throughout the analysis.

4.3 Detailed Phase Costs

Step	Operation	Time
A-tokens	`./tokenizer A.cpp`	$O(n)$
B-tokens	`./tokenizer B.cpp`	$O(m)$
Build map	insert m tokens	$O(m \log d)$
LCS fill	DP table $(n+1) \times (m+1)$	$O(n\,m)$
Back-track	recover positions	$O(L)$
Fuzzy scan	double loop on L	$O(L^2)$
Exact runs	greedy + sets	$O(m \log d)$

Table 3: Stage-by-stage cost breakdown (d = average bucket size).

4.4 Memory Breakdown

Component	Size
Tokens vectors	$(n+m) \times$ `sizeof(TokenInfo)`
DP table (32-bit)	$(n+1)(m+1) \times 4$ bytes
Position map	$\approx 16m$ bytes

Table 4: Peak memory consumption inside the checker.

Typical scenarios feature $L \ll n, m$ and small bucket sizes, making the L^2 scan and set operations sub-dominant.