Project 1 Report: Global Alignment with Linear Gap Cost — Group 3

Authors: Andreas Anastasiou and Eduardo Iglesias
Date: 17 Feb 2026

Introduction

All components work as expected. Both programs produce correct results for all test cases provided by the course (`project1_examples.txt`) and for the evaluation sequences (`project1_eval.txt`).
The `global_linear` tool computes one optimal global alignment and reports its total cost (with optional traceback output).
The `global_count` tool computes the same optimal cost and the number of optimal alignments.
The scoring model is distance-like, so the implementation performs **cost minimization** (lower values are better).
Inputs are two sequences (typically FASTA), a Phylip-like cost matrix, and a linear gap penalty (`gap`).
The submission is packaged for easy one-click execution so a tutor can run it directly without code changes.

Methods

Core alignment model (minimize cost)

For two sequences `s1` and `s2` , dynamic programming (DP) is used.
Each DP cell `(i, j)` represents the best (minimum) alignment cost for prefixes `s1[0..i)` and `s2[0..j)` .

The recurrence is:

```
 dp[i][j] = min(
dp[i-1][j-1] + matrix.cost(s1[i-1], s2[j-1]),   // diagonal: match or substitution
dp[i-1][j]   + gap,                               // up:       gap in s2
dp[i][j-1]   + gap                                // left:     gap in s1
)
```

Plain-language meaning of the three options: - **Diagonal:** align one symbol from each sequence (match or substitution). - **Up:** align a symbol in `s1` to a gap (`-`) in `s2` . - **Left:** align a symbol in `s2` to a gap (`-`) in `s1` .

Backtracking in `global_linear`

After the DP cost table is filled, stored move directions are followed from the last cell back to `(0,0)` .
This reconstructs one optimal alignment by appending characters/gaps and reversing at the end.

Counting in `global_count`

A second DP table `count[i][j]` is maintained alongside the cost table `dp[i][j]` .
Base cases are initialized as `count[0][0] = 1` , `count[i][0] = 1` , and `count[0][j] = 1` , because there is

exactly one way to align any prefix with an empty sequence (all gaps).
For each interior cell `(i, j)`, the optimal cost is computed first (same three transitions as in the main DP).
Then we set `count[i][j] = 0` and add counts from every predecessor move that achieves that same optimal cost.
This correctly handles ties: if multiple moves are optimal, all corresponding path counts are summed.
The final answer is `count[n][m]`.
`BigInteger` is used for counts because values can become large (for example, `138240 for seq3 vs seq4`).

```
best = min(diag, up, left)
count[i][j] = 0
if diag == best: count[i][j] += count[i-1][j-1]
if up   == best: count[i][j] += count[i-1][j]
if left == best: count[i][j] += count[i][j-1]
```

File formats

FASTA

Expected structure: - Header line starts with `>` - Sequence lines follow - Multi-line sequence content is concatenated

Example (3 lines):

```
>seq1
ACGTTGCA
```

Matrix file (Phylip-like)

Format: 1. First non-empty line: alphabet size `n` 2. Next `n` lines: one symbol, then `n` integer costs

Small DNA example (`A/C/G/T` , values 0/5/2):

```
4
A 0 2 5 2
C 2 0 2 5
G 5 2 0 2
T 2 5 2 0
```

If an input sequence contains unknown symbols (not in the matrix alphabet), execution stops with an error message.

Design choices (module structure)

The implementation is split into three modules to make review and grading straightforward: - `core` : sequence objects, FASTA reading, matrix/alphabet parsing, and gap-cost utilities. - `pairwise-alignment` : global DP algorithms for optimal cost/alignment and optimal-alignment counting. - `cli` : command-line parsing and output formatting, with no algorithm logic.

This keeps biological/scoring logic separate from I/O and shell interaction, improving reproducibility and debugging.

Tests

Verification against course examples

The program was verified against all four test cases in `project1_examples.txt` provided by the course:

Case	Sequences	Expected cost	Our cost	Expected count	Our count
1	acgtgtcaacgt vs acgtcgtagcta	22	22	2	2
2	aataat vs aagg	14	14	1	1
3	tccagaga vs tcgat	20	20	4	4
4	(long seqs, 197/196 chars)	325	325	288	288

All cases match exactly, confirming correctness of both the cost computation and the optimal alignment counting.

The source snapshot contains JUnit tests covering key correctness points:

- `OptimalAlignmentCounterTest`
  Validates counting behavior for both tie cases (multiple optimal alignments) and unique-optimum cases.
- `ScoreMatrixTest`
  Verifies matrix-file parsing and selected pairwise cost lookups.
- `FastaReaderTest`
  Checks FASTA parsing behavior (including lowercase and multiline sequence input).
- `PairwiseSmokeTest`
  Confirms the pairwise module test pipeline executes successfully in the build.
- `TextWrapTest`
  Verifies CLI output wrapping behavior for readable traceback output formatting.

Together, these tests validate input parsing, cost lookup, DP-based outputs, and user-facing formatting.

Experiments

Runtime artifacts are included in: - `submissions/project-1-global-linear-gap/results/timings_project1.csv` - `submissions/project-1-global-linear-gap/results/timings_project1_plot.png`

Protocol

- Warmup: 3 runs on length 500 (ignored in reported results).

- Measured lengths: 1000, 2000, 3000, 4000, 5000.
- Repetitions per length: 5.
- Reported metric: median runtime (plus minimum runtime).
- Plot uses **DP cells (`n*m`)** on x-axis and runtime (seconds) on y-axis.

Exact measured values (from CSV)

length	cells	runtime_seconds_median	runtime_seconds_min
1000	1000000	0.172	0.153
2000	4000000	0.187	0.182
3000	9000000	0.234	0.229
4000	16000000	0.308	0.306
5000	25000000	0.392	0.382

Interpretation

- Runtime grows monotonically from 0.17s (1M cells) to 0.39s (25M cells).
- The growth is consistent with the expected (O(nm)) time complexity: a 25x increase in cells produces roughly a 2.3x increase in runtime, which aligns with linear scaling once JVM startup overhead is accounted for.
- The JVM startup cost (roughly 0.15s) is visible as a baseline offset; subtracting it reveals near-linear scaling with DP cell count.
- Warmup runs and median-of-5 reporting reduce JIT and system noise.

How to run

A. One-click run (recommended for tutor)

Use the folder: `submissions/project-1-global-linear-gap/run/`

- **Windows:** double-click:
- `RUN_GLOBAL_LINEAR.bat`
- `RUN_GLOBAL_COUNT.bat`
- **Mac/Linux:** run:
- `./RUN_GLOBAL_LINEAR.sh`
- `./RUN_GLOBAL_COUNT.sh`

Output files appear in: `submissions/project-1-global-linear-gap/run/output/`

B. Command-line run (reproducible)

From `submissions/project-1-global-linear-gap/` :

`global_linear` with traceback and custom output:

```
java -jar run/bioseq-cli.jar global_linear --fasta1 <file> --fasta2 <file> --matrix <file> --ga
```

`global_count` :

```
java -jar run/bioseq-cli.jar global_count --fasta1 <file> --fasta2 <file> --matrix <file> --gap
```

Example placeholders: - `<file>` for FASTA inputs can be paths such as `run/examples/seq1.fa` and `run/examples/seq2.fa` - matrix path can be `run/examples/dna_matrix.txt`

Build from bundled source snapshot

From `submissions/project-1-global-linear-gap/code/` :

Windows:

```
 mvnw.cmd -q test
mvnw.cmd -q -pl cli -am package
```

Mac/Linux:

```
 ./mvnw -q test
./mvnw -q -pl cli -am package
```

The runnable jar in `run/` was built from this source snapshot.

Checklist for submission

- `run/` contains runnable scripts and `bioseq-cli.jar` .
- `run/output/` contains example output files after execution.
- `code/` contains source snapshot (`core` , `pairwise-alignment` , `cli`).
- `results/` contains `timings_project1.csv` and `timings_project1_plot.png` .
- `project1_eval_answers.txt` is filled with Q1/Q2/Q3 answers.
- `report.md` is present and ready for export to PDF if needed.