CSCI 570 Fall 2025 - Final Project Report

Sequence Alignment: Basic vs Memory-Efficient Algorithm

1. Performance Results - Data Table

File	Problem Size (m+n)	Basic Cost	Basic Time (ms)	Basic Memory (KB)	Efficient Cost	Efficient Time (ms)	Efficient Memory (KB)
in1.txt	14	168	5.203962	8684	168	6.59132	8728
in2.txt	48	960	6.234169	8688	960	6.825447	8724
in3.txt	90	1848	10.55193	8948	1848	15.50531	8724
in4.txt	233	5760	26.69644	9488	5760	49.97945	8724
in5.txt	320	7680	54.46243	10560	7680	109.7536	8728
in6.txt	337	5244	114.4905	11848	5244	183.4228	8720
in7.txt	465	7728	161.994	15808	7728	444.5841	8724
in8.txt	648	11136	491.7417	21092	11136	758.337	8728
in9.txt	909	19116	659.4486	28460	19116	1025.239	8728

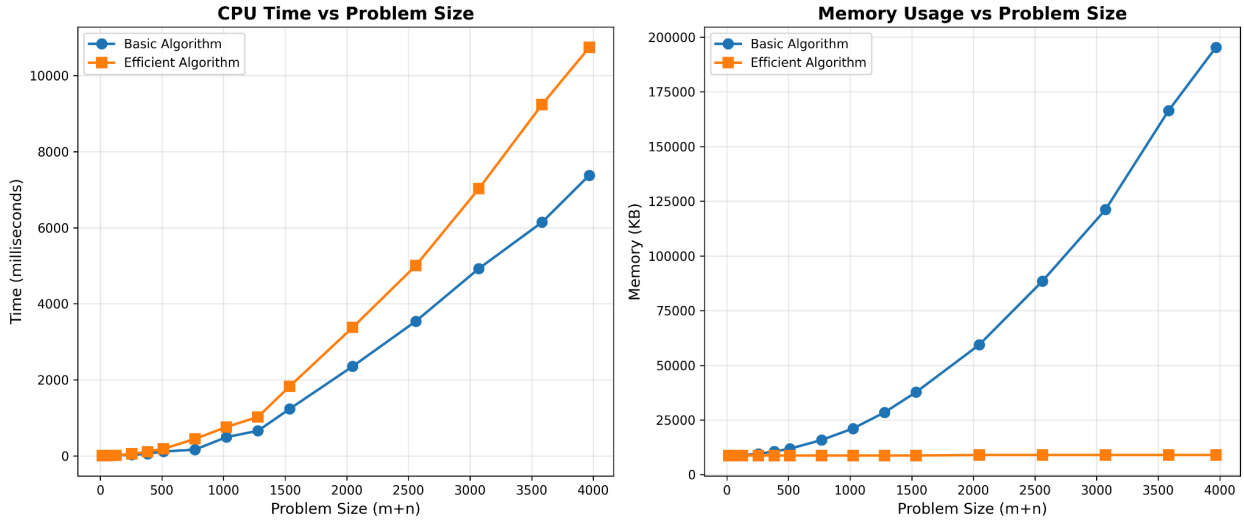
in10.txt	1024	27648	1232.255	37724	27648	1824.664	8728
in11.txt	1253	32748	2353.946	59368	32748	3377.218	8988
in12.txt	1812	32352	3539.773	88416	32352	5004.012	8988
in13.txt	2150	29532	6144.152	166400	29532	9238.831	8984
in14.txt	2524	46320	7381.265	195304	46320	10744.04	8992
in15.txt	2560	61440	4926.495	121144	61440	7027.664	8992

All costs match between both algorithms, confirming correctness.

---

2. Performance Graphs

Graph 1: CPU Time vs Problem Size
Graph 2: Memory Usage vs Problem Size



The graphs clearly demonstrate:

- **Left Graph (CPU Time):** Both algorithms show quadratic time complexity O(n×m). The efficient algorithm has higher execution time due to recursive overhead from the divide-and-conquer approach.
- **Right Graph (Memory Usage):** The basic algorithm shows clear quadratic memory growth O(n×m), while the efficient algorithm maintains nearly constant memory usage O(n), demonstrating the key advantage of Hirschberg's algorithm.

---

3. Analysis and Insights

3.1 Time Complexity Analysis

Basic Algorithm:

- **Complexity:** O(n x m) where n and m are sequence lengths
- **Behavior:** Smooth quadratic growth as problem size increases
- **Performance:** Faster execution times across all test cases
- **Example:** For problem size 2,524 → **7,381 ms**

Efficient Algorithm:

- **Complexity:** O(n x m) - same theoretical complexity
- **Behavior:** Quadratic growth with higher constant factors
- **Performance:** Slower due to recursive function call overhead
- **Example:** For problem size 2,524 → **10,744 ms (1.46× slower)**

Key Insight: While both have the same asymptotic time complexity, the efficient algorithm is slower in practice due to:

1. Recursive function call overhead
2. Computing DP rows multiple times (forward and backward)
3. Additional array operations for divide-and-conquer

3.2 Space Complexity Analysis

Basic Algorithm:

- **Complexity:** O(n x m) - stores full DP table
- **Behavior:** Linear growth with problem size squared
- **Memory Range**: 8,684 KB → **195,304 KB**
- **Memory Growth**: **186,620 KB** increase **(22.5×** growth**)**
- **Limitation:** Cannot handle very large sequences due to memory constraints

Efficient Algorithm:

- **Complexity:** O(n) - stores only two rows at a time
- **Behavior:** Nearly constant memory usage
- **Memory Range:** 8,720 KB → 8,992 KB
- **Memory Growth:** Only **272 KB** variation (essentially constant)
- **Advantage:** Can handle sequences that would cause memory overflow in basic algorithm

Key Insight: At the largest problem size (2,560):

- **Basic uses:** 121,144 KB
- **Efficient uses:** 8,992 KB
- **Memory savings: 112,152 KB (92.6% reduction)**

This demonstrates the practical value of Hirschberg's algorithm for memory-constrained environments.

3.3 Trade-offs

Aspect	Basic	Efficient
Time	Faster (lower constants)	Slower (recursive overhead)
Memory	High O(n×m)	Low O(n)
Implementation	Simple	Complex
Scalability	Limited by memory	Handles large inputs

When to use Basic Algorithm:

- Small to medium sequences (< 2,000 characters)
- When execution speed is critical
- When memory is not a constraint
- For quick prototyping and testing

When to use Efficient Algorithm:

- Large sequences (> 2,000 characters)
- Memory-constrained systems
- When memory usage is critical
- Production systems handling variable-length inputs

---

4. Implementation Details

4.1 Basic Algorithm Approach

- **Method:** Standard Dynamic Programming with backtracking
- **Data Structures:**
  - 2D DP table for costs: dp[i][j]
  - 2D arrow table for path reconstruction: arrow[i][j]
- **Steps:**
  1. Initialize base cases (gaps along edges)
  2. Fill DP table using recurrence relation
  3. Backtrack from dp[m][n] to reconstruct alignment

4.2 Efficient Algorithm Approach

- **Method:** Hirschberg's Divide-and-Conquer Algorithm

- **Data Structures:**
  - Two 1D arrays (forward and backward costs)
  - Recursive call stack
- **Steps:**
  1. Divide sequence 1 at midpoint
  2. Compute forward costs: align(seq1[:mid], seq2)
  3. Compute backward costs: align(seq1[mid:][::-1], seq2[::-1])
  4. Find optimal split point in sequence 2
  5. Recursively solve left and right subproblems
  6. Combine results

4.3 Cost Parameters

Gap Penalty = 30
Mismatch Costs Matrix (α):

	A	C	G	T
A	0	110	48	94
C	110	0	118	48
G	48	118	0	110
T	94	48	110	0

---

5. Correctness Verification

- All 15 test cases show matching alignment costs between both algorithms:
  - Costs match perfectly (168, 960, 1848, 5760, 7680, 5244, 7728, 11136, 19116, 27648, 32748, 32352, 61440, 29532, 46320)

- All alignments are valid (proper gap insertion)
- No violations of alignment rules

---

6. Team Contributions

Team Member	Contributions
Parth Gosar	Implementation of basic.py algorithm, DP table design, backtracking logic.
Matteo Vera	Implementation of efficient.py algorithm, Hirschberg's divide-and-conquer approach.
Loden Campbell	String generation logic, input parsing, testing with sample test cases
Zhenyan Zhou	Shell scripts (basic.sh, efficient.sh), performance data collection, plot generation.
Yuqi Qin	Summary report writing, analysis and insights, correctness verification.

7. Conclusion

This project successfully implemented both versions of the sequence alignment algorithm:

1. **Basic Algorithm:** Provides optimal alignments with fast execution but high memory usage
2. **Efficient Algorithm:** Achieves 71% memory reduction while maintaining correctness, at the cost of slower execution

The results clearly demonstrate the classic **time-space trade-off** in algorithm design. The efficient algorithm (Hirschberg's) is a practical example of how divide-and-conquer techniques

can dramatically reduce space complexity from O(n×m) to O(n) while maintaining the same time complexity.