Solution Report

1 Dataset Generation

I generated a random dataset as specified:

- R_1: 100 tuples (i, x) where $i \in [1, 100]$ and x is random in $[1, 5000]$.
- R_2: 100 tuples (y, j) where $j \in [1, 100]$ and y is random in $[1, 5000]$.
- R_3: 100 tuples (l, l) where $l \in [1, 100]$.

This dataset structure creates a scenario where joins are selective. R_1 joins with R_2 on A_2 (values x vs y), and R_2 joins with R_3 on A_3 (values j vs l). Since x and y are drawn from a large range $[1, 5000]$ but only 100 tuples exist, the join selectivity between R_1 and R_2 is likely low. R_2 and R_3 join on $j = l$, which is a 1-to-1 match for all 100 values.

2 Comparison Results

I ran both algorithms on the generated dataset. The code can be found in `Solution.js`.

Algorithm	**Execution Time**	**Number of Results**
Yannakakis (Problem 2)	~ 0.225 ms	4
Sequential Join (Problem 3)	~ 0.141 ms	4

(Note: Exact execution times vary by run, but are generally very fast for this small dataset size.)

Do they return the same results?

Yes. The result sets are identical. Both algorithms correctly compute the natural join of the three relations.

Running Time Analysis

For this specific small dataset ($N = 100$), the **Sequential Join** was slightly faster.

- **Reason**: The overhead of the reduction phase in the Yannakakis algorithm (building indices and filtering multiple times) might outweigh its benefits when the dataset is small and the intermediate join results are not exploding.
- **Scalability**: On much larger datasets or datasets where intermediate results would be huge (e.g., many-to-many joins), the Yannakakis algorithm would likely outperform the Sequential Join because it avoids generating dangling intermediate tuples. However, in this specific random dataset, the selectivity is high (few matches), so the intermediate result size for Sequential Join stays small, keeping it efficient.