**Thilok Reddy Anugu**

**Sree Chandana Kurella**

**Program**

The program is designed to find a valid mathematical expression with N numbers and arithmetic operators that evaluates to a target value X. It offers multiple search algorithms, including Depth-First Search (DFS), Breadth-First Search (BFS), and A\*, to find a valid solution to the puzzle.

**Input Data**

The program takes two inputs:

N: The number of values to be used in the expression.

X: The target value that the expression should evaluate to.

**Output Data**

The program provides the following output:

If a valid solution is found, it prints the valid mathematical expression that meets the criteria.

If no valid solutions are found, it prints a message indicating that no valid solutions were found.

**Report**

**Basic Introduction**

The primary objective of this project is to solve the problem of expressing a target number, X, using a combination of N integers drawn from the range [1, 9] and basic arithmetic operations, which include addition (+), subtraction (-), multiplication (\*), and division (/). This problem is fundamentally mathematical and computational, with practical applications in various fields, including education and computer science. The ability to determine if such an expression exists and, if so, finding it, is crucial for enhancing numerical problem-solving skills and algorithmic thinking.

**Algorithm Pseudocode**

***Depth-First Search (DFS):***

DFS(expression, length):

if length is equal to N:

if expression evaluates to X:

return expression

else:

return None

for num in numbers:

for op in operators:

new\_expression = expression + num + op

result = DFS(new\_expression, length + 1)

if result is not None:

return result

return None

***Breadth-First Search (BFS):***

BFS(N, X):

Initialize a queue with (expression, length) pairs

while the queue is not empty:

Dequeue the first pair

if length is equal to N:

if expression evaluates to X:

return expression

if length is less than N:

Generate new expressions by adding numbers and operators

Enqueue the new expressions with updated lengths

return None

***A\* Search:***

A\*(N, X):

Initialize a priority queue with (priority, g\_value, expression) tuples

while the priority queue is not empty:

Dequeue the top tuple

if length is equal to N:

if expression evaluates to X:

return expression

Generate new expressions by adding numbers and operators

Calculate heuristic values and update priorities

Enqueue the new tuples with updated g\_values and priorities

return None

***Brute-Force:***

BruteForce(N, X):

Generate all combinations of N numbers and operators

for each combination:

Evaluate the expression

If the result is equal to X, return the expression

return None

**Test Case Generation**

Test cases can be generated by specifying different values of N and X. The program tests various combinations of numbers and operators to find valid solutions.

The following factors should be considered when creating test cases:

* Varying N and X values: Generate test cases with different values of N and X to assess the algorithm's flexibility.
* Patterns and constraints: Include test cases where the solution is relatively simple (e.g., N = 2 and X = 4, representing multiplication) or more complex (e.g., N = 3 and X = 9, potentially involving exponentiation).
* Randomization: Incorporate random test cases to assess the algorithm's ability to handle unforeseen scenarios.

**Positive Test Cases (Valid Solutions):**

| **Test Case** | **N** | **X** | **Expression** |
| --- | --- | --- | --- |
| 1 | 2 | 4 | 2 + 2 |
| 2 | 3 | 16 | 2 \* 2 \* 2 |
| 3 | 4 | 18 | 2 + 2 \* 2 \* 2 |
| 4 | 5 | 81 | 3 \* 3 \* 3 \* 3 |

**Negative Test Cases (No Valid Solutions):**

| **Test Case** | **N** | **X** | **Reason** |
| --- | --- | --- | --- |
| 1 | 2 | 50 | No valid solution |
| 2 | 3 | 728 | No valid solution |
| 3 | 1 | 12 | No valid solution |

Top of Form

**Bottom of Form**

**Search Space Analysis**

The search space for DFS and BFS grows exponentially with N. The search space for this problem is determined by the number of integers (N) and the number of operations. With N integers and N-1 operations, the search space is exponential, specifically **O(4^(N-1)).** This implies that as N increases, the search space grows exponentially. The exponential nature of the search space becomes more evident as N and X values increase, making the problem computationally challenging for large instances. In the worst case, they explore all possible combinations.

A\* utilizes a heuristic to explore more promising paths first, reducing the search space.

Brute-force explores the entire search space, making it inefficient for large N.

**Performance Analysis**

In our performance analysis, we evaluated the algorithms across a range of N and X values to understand their efficiency and scalability. The key observations are as follows:

* *DFS outperforms A and BFS:*\* For the given problem, Depth First Search (DFS) emerges as the most efficient algorithm, especially in comparison to A\* and Breadth First Search (BFS).
* **Small N and X Values:** All algorithms, including DFS, A\*, BFS, and the Brute Force method, work effectively and yield results in a reasonable time frame for small N and X values.
* **Large N and X Values:** When it comes to larger values of N and X, DFS stands out as the only algorithm that remains practical and capable of delivering results within a reasonable time. The other algorithms, including A\* and BFS, tend to become computationally intensive and less efficient for larger problem instances.

This analysis highlights the relative strengths of the DFS algorithm, making it the preferred choice for addressing the problem, particularly when dealing with larger values of N and X.

**Possible Suggestions for Your Algorithm to Do Next:**

* **Pruning Techniques:** Implement pruning techniques to reduce the search space by excluding invalid combinations early in the search process. This can significantly enhance algorithm efficiency.
* **Alternative Algorithms:** Experiment with other search algorithms like A\* and dynamic programming to determine if they offer faster solutions for larger N and X values. Different algorithms may excel in different scenarios.
* **Heuristics:** Develop heuristics to guide the search process more efficiently. Heuristics can help identify promising paths in the search space and focus computational efforts on more likely solutions.
* **Parallel and Distributed Computing:** Explore parallel computing or distributed computing to accelerate the search for large problem instances. Distributing the computation across multiple processors or machines can lead to faster results.

This detailed version of the report should provide a comprehensive overview of your project, your algorithm, and potential areas for improvement. If you have any further questions or need additional information, please feel free to ask.