CQUPT – University at Albany

Computer Science – International College

**ICSI 403 --- Design and Analysis of Algorithms**

**Project 3 --- Spring 2025**

**Table of Contents**

1. **System documentation -------------------------------------------------------------1**
   * A high-level data flow diagram for the system-----------------------------1
   * A list of routines and their brief descriptions-------------------------------2
   * Implementation details --------------------------------------------------------3
2. **Test documentation-----------------------------------------------------------------10**
   * How you tested your program ----------------------------------------------12
   * Testing outputs ----------------------------------------------------------------17
3. **User documentation ----------------------------------------------------------------17**
   * How to run your program ---------------------------------------------------17
   * Describe parameter (if any) -------------------------------------------------18
4. **Source Code--------------------------------------------------------------------------18**
   * Correctness --------------------------------------------------------------------18
   * Programming style -----------------------------------------------------------21

**I.** **System documentation**

**1.** **A high-level data flow diagram for the system**

图表, 图示, 箱线图

AI 生成的内容可能不正确。

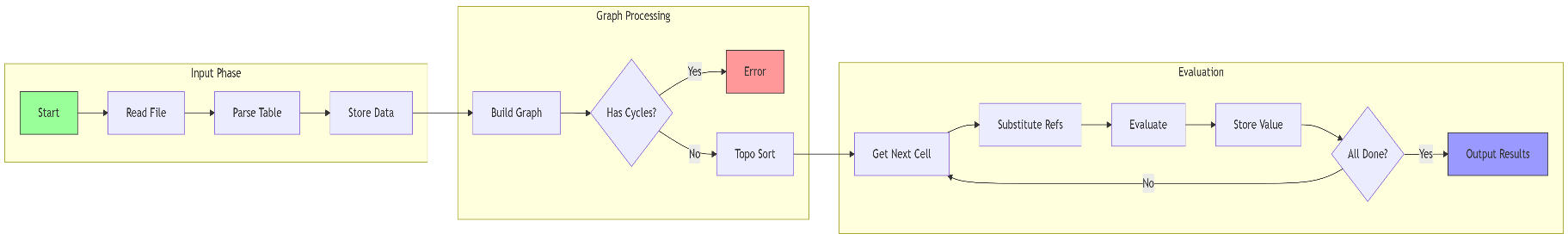
Data flow diagram

**2. A list of routines and their brief descriptions**

| **Function Name** | **Description** |
| --- | --- |
| getCellName(int row, int col) | Converts row and column indices to a spreadsheet cell name (e.g., A1, B2). |
| readInput(const string &filename) | Reads the input file and stores table dimensions and cell expressions. |
| extractDependencies(const string &expr) | Extracts referenced cell names (e.g., A1, C3) from an expression. |
| buildGraph() | Constructs a dependency graph for all cells based on their expressions. |
| dfs(const string &node) | Performs depth-first search for topological sorting of the dependency graph. |
| topologicalSort() | Returns the cells sorted in topological order based on dependencies. |
| substituteVariables(const string &expr, const map<string, int> &values) | Replaces cell references in an expression with their computed values. |
| evaluateSimpleExpression(const string &expr) | Evaluates a basic arithmetic expression (only +, -, \*, / supported). |
| main() | Program entry point: reads input, builds graph, evaluates expressions, outputs results. |

**3. Implementation details.**

This program implements a simple spreadsheet expression evaluator, supporting cell references, basic arithmetic operations, and dependency resolution between cells. The key components and their detailed functionalities are described below.



Flowchart

**3.1 Data Structure**

vector<vector<string>> table;    // Stores the original spreadsheet

map<string, vector<string>> adj; // Adjacency list for dependency graph

map<string, bool> visited;       // Tracks visited nodes during DFS

map<string, bool> onPath;        // Tracks nodes on current DFS path (for cycle detection)

stack<string> topoStack;         // Stack to store topological order

map<string, string> expressions; // Maps cell names to original expressions

map<string, int> values;         // Stores computed values of cells

**Data Structure Overview**

| **Name** | **Type** | **Declared In** | **Purpose** |
| --- | --- | --- | --- |
| table | vector<vector<string>> | Global | Stores the original 2D spreadsheet read from the input file. Each cell contains either a number or an expression (e.g., A1+2). |
| expressions | map<string, string> | Global | Maps each cell name (e.g., "A1") to its original expression from the input file. |
| adj | map<string, vector<string> | Global | Represents the dependency graph using an adjacency list. For example, if A1 = B1 + 2, then there is an edge from B1 to A1. |
| visited | map<string, bool> | DFS & Topo Sort | Marks whether a cell has been visited during DFS traversal for topological sorting. |
| onPath | map<string, bool> | DFS (Cycle Detection) | Tracks whether a node is currently in the recursion stack, used for cycle detection (circular dependencies). |
| topoStack | stack<string> | Topological Sort | Temporarily stores the topologically sorted cells (in reverse order) while performing DFS. |
| values | map<string, int> | Evaluation Phase | Stores the final evaluated integer values of each cell after substituting and calculating expressions. |
| order | vector<string> | main() function | Stores the final topological order of cells, determining the order of expression evaluation. |

**3.2 Utility Function**

// Convert row and column index to cell name (e.g., A1, B2)

string **getCellName**(int row, int col)

{

    char colChar = 'A' + col;

    return **string**(1, colChar) **+** **to\_string**(row + 1);

}

**Purpose:**  
Convert a row-column index to a spreadsheet-style cell name. For example: (0, 0) → "A1"

**3.3 Input Reading**

// Read spreadsheet data from input file

void **readInput**(const string &filename)

{

    ifstream **file**(filename);

    if (**!**file)

    {

        cerr **<<** "Cannot open file.\n";

**exit**(1);

    }

    file **>>** rows **>>** cols;

    table.**resize**(rows, **vector**<string>(cols));

    string line;

**getline**(file, line); // Skip the rest of the first line

    for (int i = 0; i < rows; ++i)

    {

**getline**(file, line);

        stringstream **ss**(line);

        for (int j = 0; j < cols; ++j)

        {

            ss **>>** table**[**i**][**j**]**;

            string cell = **getCellName**(i, j);

            expressions**[**cell**]** **=** table**[**i**][**j**]**;

        }

    }

}

**Purpose:**

* Read the spreadsheet content from a file
* Fill expressions with cell names mapped to their raw expressions

**3.4 Extract Dependencies**

// Extract all cell dependencies from a given expression

vector<string> **extractDependencies**(const string &expr)

{

    vector<string> deps;

    regex **cellRegex**("([A-Z][1-9][0-9]\*)"); // Matches A1 to Z99 etc.

    auto begin = **sregex\_iterator**(expr.**begin**(), expr.**end**(), cellRegex);

    auto end = **sregex\_iterator**();

    for (auto it = begin; it **!=** end; **++**it)

    {

        deps.**push\_back**(it**->str**());

    }

    return deps;

}

**Purpose:**

* Use regular expressions to find all references like A1, B2 in an expression
* For example: "A1+B2" → ["A1", "B2"]

**3.5 Build the Dependency Graph**

// Build a directed graph based on cell dependencies

void **buildGraph**()

{

    for (const auto &[cell, expr] : expressions)

    {

        vector<string> deps = **extractDependencies**(expr);

        for (const string &dep : deps)

        {

            adj**[**dep**]**.**push\_back**(cell); // dep → cell

        }

        if (adj.**find**(cell) **==** adj.**end**())

        {

            adj**[**cell**]** **=** {}; // Ensure all cells appear in the graph

        }

    }

}

**Purpose:**

* Build a graph from referenced cells to dependent cells
* E.g., A2 = A1 + B1 leads to: A1 → A2 and B1 → A2

**3.6 Topological Sort**

// Perform topological sort using DFS

vector<string> **topologicalSort**()

{

    visited.**clear**();

    onPath.**clear**();

    // Initialize visited and onPath maps

    for (const auto &pair : adj)

    {

        visited**[**pair.first**]** = false;

        onPath**[**pair.first**]** = false;

    }

    // Visit all nodes

    for (const auto &pair : adj)

    {

        if (!visited**[**pair.first**]**)

        {

            if (!**dfs**(pair.first))

            {

                cerr **<<** "Topological sort failed due to circular dependency.\n";

**exit**(1);

            }

        }

    }

    // Collect the result

    vector<string> order;

    while (!topoStack.**empty**())

    {

        order.**push\_back**(topoStack.**top**());

        topoStack.**pop**();

    }

    return order;

}

**Purpose:**

* Set all nodes as unvisited and not currently on the DFS path
* For each unvisited node, call dfs() to explore its neighbors
* If a node is visited again while on the current path, report a cycle and stop
* Each cell is pushed to the stack after all its dependencies are visited
* Pop from the stack to get a valid order for evaluating expressions (dependencies first)

**3.7 Substitute Variable Names with Values**

// Replace cell names in the expression with their actual values

string **substituteVariables**(const string &expr, const map<string, int> &values)

{

    string result = expr;

    regex **cellRegex**("([A-Z][1-9][0-9]\*)");

    smatch match;

    string output;

    size\_t pos = 0;

    auto begin = **sregex\_iterator**(expr.**begin**(), expr.**end**(), cellRegex);

    auto end = **sregex\_iterator**();

    size\_t last\_pos = 0;

    for (auto it = begin; it **!=** end; **++**it)

    {

        string var = it**->str**();

        int val = values.**at**(var); // Replace with actual value

        output **+=** expr.**substr**(last\_pos, it**->position**() - last\_pos);

        output **+=** **to\_string**(val);

        last\_pos = it**->position**() + var.**length**();

    }

    output **+=** expr.**substr**(last\_pos);

    return output;

}

**Purpose:**

* Replace variable names (e.g., A1, B1) with actual evaluated numbers
* Example: A1 + B1 → 1 + 2

**3.8 Expression Evaluation**

// Evaluate a basic arithmetic expression (support +, -, \*, /)

int **evaluateSimpleExpression**(const string &expr)

{

    istringstream **ss**(expr);

    int result;

    ss **>>** result;

    char op;

    int num;

    while (ss **>>** op **>>** num)

    {

        if (op == '+')

            result += num;

        else if (op == '-')

            result -= num;

        else if (op == '\*')

            result \*= num;

        else if (op == '/')

            result /= num;

    }

    return result;

}

**Purpose:**

* Replace variable names (e.g., A1, B1) with actual evaluated numbers, Variables like A1 and B1 are replaced with their evaluated values from the values map.
* Extract the first number (or variable) from the string. The first part of the string is extracted as the initial number or variable.
* Extract the next number (or variable) from the string.The next element (either a number or a variable) is extracted after identifying the first one.
* For each operator (+, -, , /), process the next number and apply the operation. The program processes each operator in the expression, one by one.
* Depending on the operator, perform addition, subtraction, multiplication, or division
  + If the operator is +, it adds the next number to the result.
  + If the operator is -, it subtracts the next number from the result.
  + If the operator is \*, it multiplies the result by the next number.
  + If the operator is /, it divides the result by the next number.
* After processing the entire expression, return the computed value as an integer. After applying all operations, the final result is returned as an integer.

**3.9 DFS**

// Depth-First Search with cycle detection

bool **dfs**(const string &node)

{

    visited**[**node**]** = true;

    onPath**[**node**]** = true;

    for (const string &neighbor : adj**[**node**]**)

    {

        if (!visited**[**neighbor**]**)

        {

            if (!**dfs**(neighbor))

                return false;

        }

        else if (onPath**[**neighbor**]**)

        {

            cerr **<<** "Error: Circular dependency detected involving " **<<** node **<<** " and " **<<** neighbor **<<** ".\n";

            return false;

        }

    }

    onPath**[**node**]** = false;

    topoStack.**push**(node); // Add to topological sort

    return true;

}

**Purpose:**

* Mark current node as visited and on the current path
* For each neighbor: Recurse if unvisited. If on current path → cycle detected → return false
* After visiting neighbors, unmark current path
* Push node to stack (for topological sort)
* Return true if no cycle detected

**3.10 Main Execution Logic**

int **main**()

{

    string filename;

    cout **<<** "Please enter the input file name (for example, input.txt): ";

    cin **>>** filename;

**readInput**(filename);                      // Step 1: Read table

**buildGraph**();                             // Step 2: Build dependency graph

    vector<string> order = **topologicalSort**(); // Step 3: Topological sort

    cout **<<** "Topological Order:\n";

    for (const string &cell : order)

    {

        cout **<<** cell **<<** " ";

    }

    cout **<<** "\n\nEvaluated Results:\n";

    // Step 4: Evaluate expressions in topological order

    for (const string &cell : order)

    {

        string expr = expressions**[**cell**]**;

        string substituted = expr;

        // Replace cell references with values

        if (!**extractDependencies**(expr).**empty**())

        {

            substituted **=** **substituteVariables**(expr, values);

        }

        // Evaluate the final expression

        int val = **evaluateSimpleExpression**(substituted);

        values**[**cell**]** = val;

        // Display full evaluation trace

        cout **<<** cell **<<** " = " **<<** expr;

        if (expr **!=** substituted)

        {

            cout **<<** " = " **<<** substituted;

        }

        cout **<<** " = " **<<** val **<<** **endl**;

    }

    // Step 5: Output final values sorted by cell name

    cout **<<** "\nFinal Values (sorted by cell name):\n";

    vector<pair<string, int>> **sortedValues**(values.**begin**(), values.**end**());

**sort**(sortedValues.**begin**(), sortedValues.**end**());

    for (const auto &[cell, val] : sortedValues)

    {

        cout **<<** cell **<<** " = " **<<** val **<<** **endl**;

    }

    return 0;

}

**Purpose:**

* Prompt user for input filename (e.g., input.txt)
* Call readInput() to load and store table data
* Call buildGraph() to create dependency graph between cells
* Perform topological sort to get evaluation order of cells
* For each cell in order:
  + Replace variables in the expression with actual values
  + Evaluate the final numeric expression
  + Store and print result
* Finally, print all cell values sorted by cell name

**II Test documentation**

**1.** **How you tested your program**

I execute my program in **Vscode**, the program structure as follows

屏幕上有字

AI 生成的内容可能不正确。

The input files are: **input1.txt, input2.txt,input3.txt,input4.txt，**The answer files are: **answer1.txt, answer2.txt, answer3.txt, answer4.txt**

1. Compiling and running **code.c**, then **input** the testcase file’s name
2. The console will show the result
3. Results have **three parts**, organe show the **Topological Order**, blue show the **calculate process** and the write show the **final values( sorted by name)**

文本

AI 生成的内容可能不正确。

Output layout

I using **excel** to calculate the corret answer

图片包含 图表

AI 生成的内容可能不正确。

Result calculate by excel

Then compare the result with my output one by one

**2. Testing outputs**

There are four testcases

**2.1 Testcase1**

文本

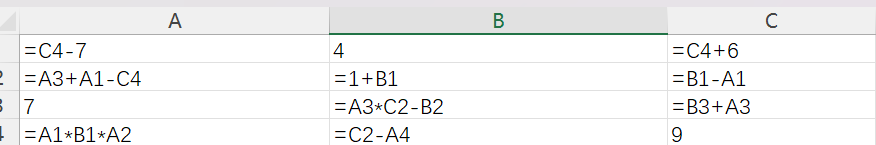
AI 生成的内容可能不正确。

Input1.txt

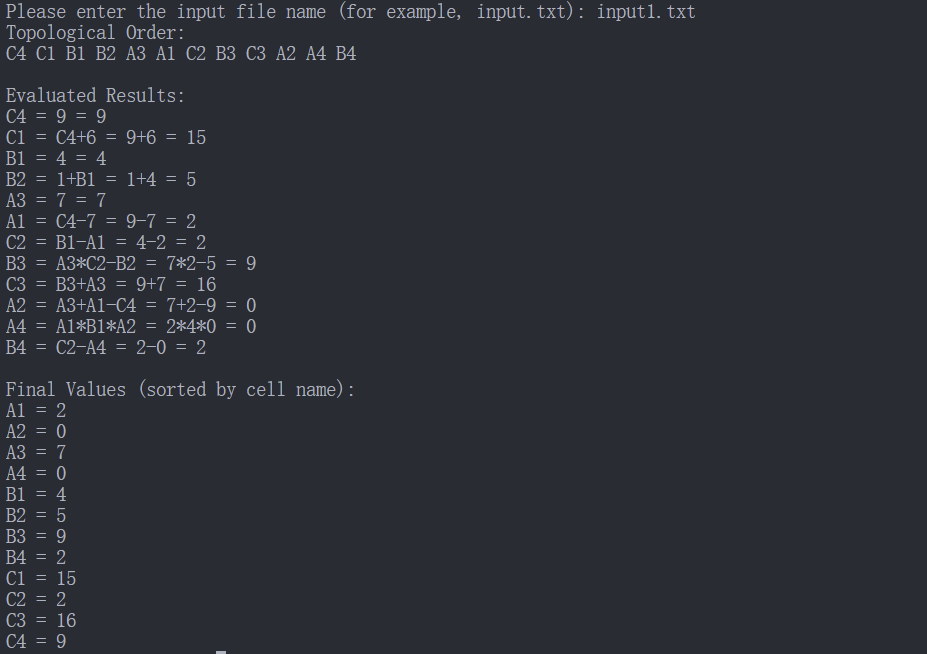
图示

AI 生成的内容可能不正确。

Input1 Illustration



Correct result1



My output1

**2.2 Testcase2**

图形用户界面, 文本, 应用程序, 聊天或短信

AI 生成的内容可能不正确。

Input2.txt

图示

AI 生成的内容可能不正确。

Input2 Illustration

表格

AI 生成的内容可能不正确。

Correct result2

文本

AI 生成的内容可能不正确。

My output2

**2.3 Testcase3**

This is a big input, It contains 26 different letters and each letter has four rows

电脑键盘

AI 生成的内容可能不正确。

Input3.txt

图示

AI 生成的内容可能不正确。

Input3 Illustration

图片包含 室内, 桌子, 行, 夹

AI 生成的内容可能不正确。

Correct result3

文本

AI 生成的内容可能不正确。

文本

AI 生成的内容可能不正确。

形状

AI 生成的内容可能不正确。

形状

AI 生成的内容可能不正确。

My output3

**2.4 Testcase4**

手机屏幕的截图

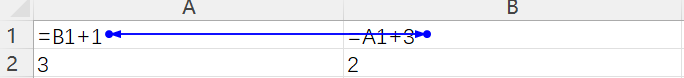
AI 生成的内容可能不正确。

Input4.txt

图示

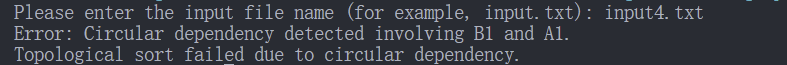
AI 生成的内容可能不正确。

Input4 Illustration



Correct result4

It discovered the error in the input(**a cycle**)



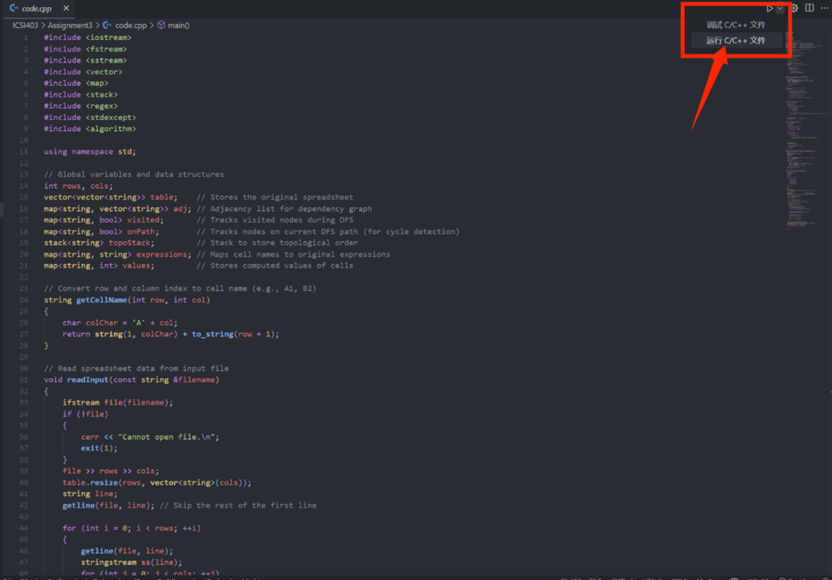
My output4

**III. User documentation**

1. **How to run your program**

The input file to be tested should placed in the **same directory** as the program file

There are a lot of IDE, I will show how to run it in **Vscode**, the processes are same



Input the file name

****

**2.** **Describe parameter (if any)**

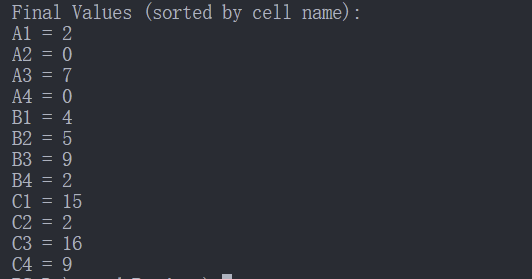
The program receive a filename to find the input file

**IV. Source Code**

1. **Correctness**

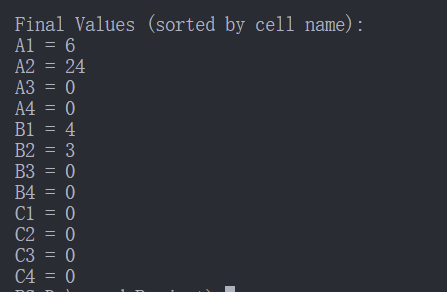
**1.1 Testcase1**

表格

AI 生成的内容可能不正确。

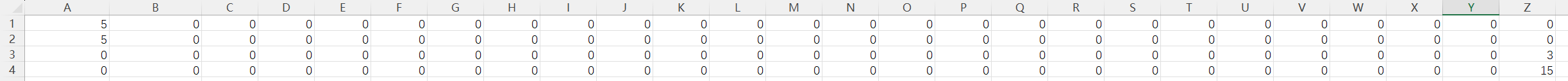
**Correct!**

**1.2 Testcase2**

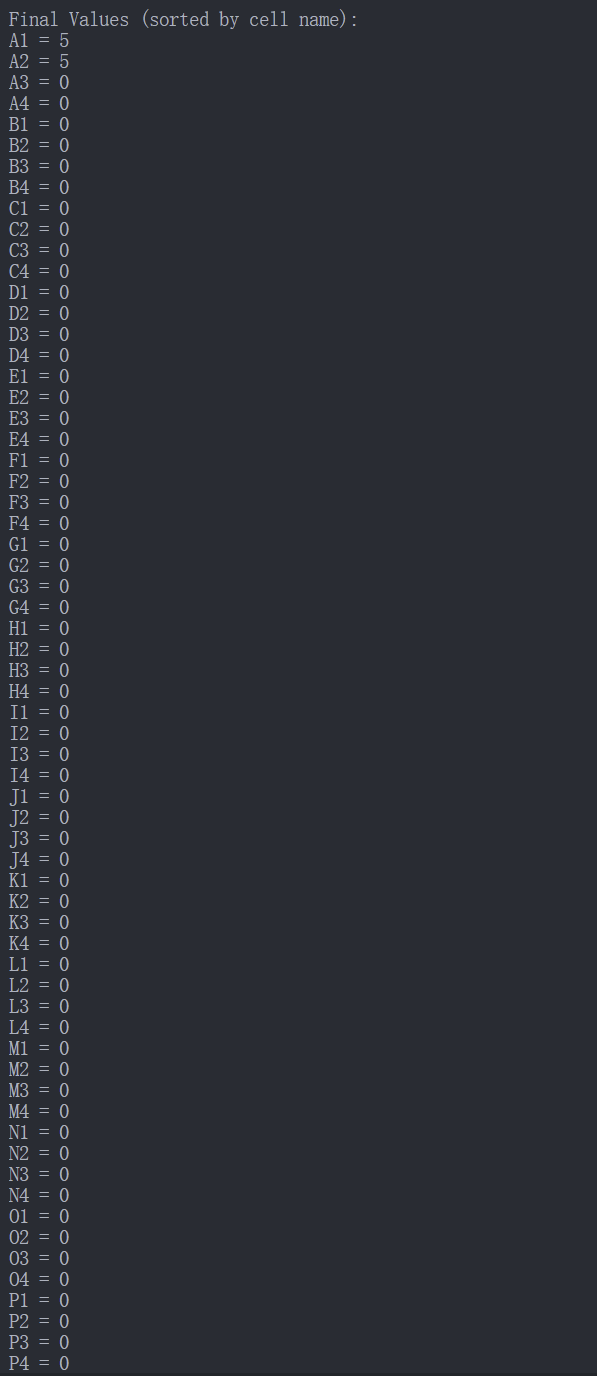
****表格

AI 生成的内容可能不正确。

**Correct!**

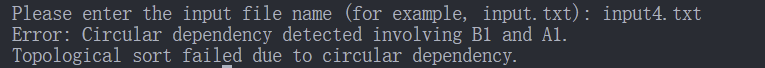
**1.3 Testcase3、**

电脑萤幕

AI 生成的内容可能不正确。

**Correct!**

**1.4 Testcase4**



**Correct!**

It find the error in test case which is a cycle

1. **Programming style**

**Layering | Readability | Comments | Efficiency** are showing as follows

#include <iostream>

#include <fstream>

#include <sstream>

#include <vector>

#include <map>

#include <stack>

#include <regex>

#include <stdexcept>

#include <algorithm>

using namespace std;

// Global variables and data structures

int rows, cols;

vector<vector<string>> table;    // Stores the original spreadsheet

map<string, vector<string>> adj; // Adjacency list for dependency graph

map<string, bool> visited;       // Tracks visited nodes during DFS

map<string, bool> onPath;        // Tracks nodes on current DFS path (for cycle detection)

stack<string> topoStack;         // Stack to store topological order

map<string, string> expressions; // Maps cell names to original expressions

map<string, int> values;         // Stores computed values of cells

// Convert row and column index to cell name (e.g., A1, B2)

string **getCellName**(int row, int col)

{

    char colChar = 'A' + col;

    return **string**(1, colChar) **+** **to\_string**(row + 1);

}

// Read spreadsheet data from input file

void **readInput**(const string &filename)

{

    ifstream **file**(filename);

    if (**!**file)

    {

        cerr **<<** "Cannot open file.\n";

**exit**(1);

    }

    file **>>** rows **>>** cols;

    table.**resize**(rows, **vector**<string>(cols));

    string line;

**getline**(file, line); // Skip the rest of the first line

    for (int i = 0; i < rows; ++i)

    {

**getline**(file, line);

        stringstream **ss**(line);

        for (int j = 0; j < cols; ++j)

        {

            ss **>>** table**[**i**][**j**]**;

            string cell = **getCellName**(i, j);

            expressions**[**cell**]** **=** table**[**i**][**j**]**;

        }

    }

}

// Extract all cell dependencies from a given expression

vector<string> **extractDependencies**(const string &expr)

{

    vector<string> deps;

    regex **cellRegex**("([A-Z][1-9][0-9]\*)"); // Matches A1 to Z99 etc.

    auto begin = **sregex\_iterator**(expr.**begin**(), expr.**end**(), cellRegex);

    auto end = **sregex\_iterator**();

    for (auto it = begin; it **!=** end; **++**it)

    {

        deps.**push\_back**(it**->str**());

    }

    return deps;

}

// Build a directed graph based on cell dependencies

void **buildGraph**()

{

    for (const auto &[cell, expr] : expressions)

    {

        vector<string> deps = **extractDependencies**(expr);

        for (const string &dep : deps)

        {

            adj**[**dep**]**.**push\_back**(cell); // dep → cell

        }

        if (adj.**find**(cell) **==** adj.**end**())

        {

            adj**[**cell**]** **=** {}; // Ensure all cells appear in the graph

        }

    }

}

// Depth-First Search with cycle detection

bool **dfs**(const string &node)

{

    visited**[**node**]** = true;

    onPath**[**node**]** = true;

    for (const string &neighbor : adj**[**node**]**)

    {

        if (!visited**[**neighbor**]**)

        {

            if (!**dfs**(neighbor))

                return false;

        }

        else if (onPath**[**neighbor**]**)

        {

            cerr **<<** "Error: Circular dependency detected involving " **<<** node **<<** " and " **<<** neighbor **<<** ".\n";

            return false;

        }

    }

    onPath**[**node**]** = false;

    topoStack.**push**(node); // Add to topological sort

    return true;

}

// Perform topological sort using DFS

vector<string> **topologicalSort**()

{

    visited.**clear**();

    onPath.**clear**();

    // Initialize visited and onPath maps

    for (const auto &pair : adj)

    {

        visited**[**pair.first**]** = false;

        onPath**[**pair.first**]** = false;

    }

    // Visit all nodes

    for (const auto &pair : adj)

    {

        if (!visited**[**pair.first**]**)

        {

            if (!**dfs**(pair.first))

            {

                cerr **<<** "Topological sort failed due to circular dependency.\n";

**exit**(1);

            }

        }

    }

    // Collect the result

    vector<string> order;

    while (!topoStack.**empty**())

    {

        order.**push\_back**(topoStack.**top**());

        topoStack.**pop**();

    }

    return order;

}

// Replace cell names in the expression with their actual values

string **substituteVariables**(const string &expr, const map<string, int> &values)

{

    string result = expr;

    regex **cellRegex**("([A-Z][1-9][0-9]\*)");

    smatch match;

    string output;

    size\_t pos = 0;

    auto begin = **sregex\_iterator**(expr.**begin**(), expr.**end**(), cellRegex);

    auto end = **sregex\_iterator**();

    size\_t last\_pos = 0;

    for (auto it = begin; it **!=** end; **++**it)

    {

        string var = it**->str**();

        int val = values.**at**(var); // Replace with actual value

        output **+=** expr.**substr**(last\_pos, it**->position**() - last\_pos);

        output **+=** **to\_string**(val);

        last\_pos = it**->position**() + var.**length**();

    }

    output **+=** expr.**substr**(last\_pos);

    return output;

}

// Evaluate a basic arithmetic expression (support +, -, \*, /)

int **evaluateSimpleExpression**(const string &expr)

{

    istringstream **ss**(expr);

    int result;

    ss **>>** result;

    char op;

    int num;

    while (ss **>>** op **>>** num)

    {

        if (op == '+')

            result += num;

        else if (op == '-')

            result -= num;

        else if (op == '\*')

            result \*= num;

        else if (op == '/')

            result /= num;

    }

    return result;

}

int **main**()

{

    string filename;

    cout **<<** "Please enter the input file name (for example, input.txt): ";

    cin **>>** filename;

**readInput**(filename);                      // Step 1: Read table

**buildGraph**();                             // Step 2: Build dependency graph

    vector<string> order = **topologicalSort**(); // Step 3: Topological sort

    cout **<<** "Topological Order:\n";

    for (const string &cell : order)

    {

        cout **<<** cell **<<** " ";

    }

    cout **<<** "\n\nEvaluated Results:\n";

    // Step 4: Evaluate expressions in topological order

    for (const string &cell : order)

    {

        string expr = expressions**[**cell**]**;

        string substituted = expr;

        // Replace cell references with values

        if (!**extractDependencies**(expr).**empty**())

        {

            substituted **=** **substituteVariables**(expr, values);

        }

        // Evaluate the final expression

        int val = **evaluateSimpleExpression**(substituted);

        values**[**cell**]** = val;

        // Display full evaluation trace

        cout **<<** cell **<<** " = " **<<** expr;

        if (expr **!=** substituted)

        {

            cout **<<** " = " **<<** substituted;

        }

        cout **<<** " = " **<<** val **<<** **endl**;

    }

    // Step 5: Output final values sorted by cell name

    cout **<<** "\nFinal Values (sorted by cell name):\n";

    vector<pair<string, int>> **sortedValues**(values.**begin**(), values.**end**());

**sort**(sortedValues.**begin**(), sortedValues.**end**());

    for (const auto &[cell, val] : sortedValues)

    {

        cout **<<** cell **<<** " = " **<<** val **<<** **endl**;

    }

    return 0;

}