631. Design Excel Sum Formula

June 26, 2017 | 4.9K views

Previous Next

Your task is to design the basic function of Excel and implement the function of sum formula. Specifically, you need to implement the following functions:

`Excel(int H, char W)`: This is the constructor. The inputs represents the height and width of the Excel form. **H**, a positive integer, range from 1 to 26. It represents the height. **W** is a character range from 'A' to 'Z'. It represents that the width is the number of characters from 'A' to **W**. The Excel form content is represented by a height * width 2D integer array `C`, it should be initialized to zero. You should assume that the first row of `C` starts from 1, and the first column of `C` starts from 'A'.

`void Set(int row, char column, int val)`: Change the value at `C(row, column)` to be val.

`int Get(int row, char column)`: Return the value at `C(row, column)`.

`int Sum(int row, char column, List of Strings : numbers)`: This function calculate and set the value at `C(row, column)`, where the value should be the sum of cells represented by `numbers`. This function return the sum result at `C(row, column)`. This sum formula should exist until this cell is overlapped by another value or another sum formula.

`numbers` is a list of strings that each string represent a cell or a range of cells. If the string represent a single cell, then it has the following format : `ColRow`. For example, "F7" represents the cell at (7, F).

If the string represent a range of cells, then it has the following format : `ColRow1:ColRow2`. The range will always be a rectangle, and ColRow1 represent the position of the top-left cell, and ColRow2 represents the position of the bottom-right cell.

Example 1:

```
Excel(3,"C");
// construct a 3*3 2D array with all zero.
//   A B C
// 1 0 0 0
// 2 0 0 0
// 3 0 0 0

Set(1, "A", 2);
// set C(1,"A") to be 2.
//   A B C
// 1 2 0 0
// 2 0 0 0
// 3 0 0 0

Sum(3, "C", ["A1", "A1:B2"]);
// set C(3,"C") to be the sum of value at C(1,"A") and the values sum of the rectangle
//   A B C
// 1 2 0 0
// 2 0 0 0
// 3 0 0 4

Set(2, "B", 2);
// set C(2,"B") to be 2. Note C(3, "C") should also be changed.
//   A B C
// 1 2 0 0
// 2 0 2 0
// 3 0 0 6
```

Note:

1. You could assume that there won't be any circular sum reference. For example, A1 = sum(B1) and B1 = sum(A1).
2. The test cases are using double-quotes to represent a character.
3. Please remember to **RESET** your class variables declared in class Excel, as static/class variables are **persisted across multiple test cases**. Please see here for more details.

Solution

Approach 1: Topological Sort

Before discussing the required design, we'll discuss some prerequisites to help ease the understanding of the solution.

Firstly, we can note that once a formula is applied to any cell in excel, let's say $C1 = C2 + C3$. If any change is made to $C2$ or $C3$, the result to be put into $C1$ needs to be evaluated again based on the new values of $C2$ and $C3$. Further, suppose some other cell, say $D2$ is also dependent on $C1$ due to some prior formula applied to $D2$. Then, when any change is made to, say, $C2$, we re-evaluate $C1$'s value. Further, since $D2$ is dependent on $C1$, we need to re-evaluate $D2$'s value as well.

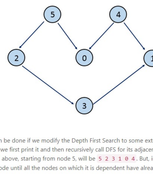
Thus, whenever, we make any change to any cell, x, we need to determine the cells which are dependent on x, and update these cells, and further determine the cells which are dependent on the changed cells and so on. We can assume that no cycles are present in the formula. i.e. Any cell's value won't directly or indirectly be dependent on its own value.

But, while doing these set of evaluations of the cells to determine their updated values, we need to update the cells in such an order that the cell on which some other cell is dependent is always evaluated prior to the cell which is dependent on the former cell.

In order to do so, we can view the dependence between the cells in the form of a dependency graph, which can be a Directed Graph. Since, no cycles are allowed between the formulas, the graph reduces to a Directed Acyclic Graph. Now, to solve the problem of evaluating the cells in the required order, we can make use of a very well known method specifically used for such problems in Directed Acyclic Graphs, known as the Topological Sorting.

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge ut, vertex u comes before v in the ordering. For example, a topological sorting of the following graph is `5 4 2 3 1 0`.

There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is `4 5 2 3 1 0`. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no in-coming edges).



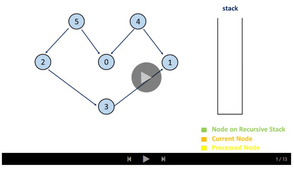
Topological Sorting can be done if we modify the Depth First Search to some extent. In Depth First Search, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. Thus, the DFS obtained for the graph above, starting from node 5, will be `5 2 3 1 0 4`. But, in the case of a topological sort, we can't print a node until all the nodes on which it is dependent have already been printed.

To solve this problem, we make use of a temporary stack. We do the traversals in the same manner as in DFS, but we don't print the current node immediately. Instead, for the current node we do as follows:

- Recursively call topological sorting for all the nodes adjacent to the current node.
- Push the current node onto a stack.
- Repeat the above process till all the nodes have been considered atleast once.
- Print the contents of the stack.

Note that a vertex is pushed to stack only when all of its adjacent(dependent) vertices (and their adjacent(dependent) vertices and so on) are already in stack. Thus, we obtain the correct ordering of the vertices.

The following animation shows an example of topological sorting for the graph above.



We can make use of the same concept while evaluating the cell values to determine the order in which they need to be evaluated.

Now, let's discuss how we implement the various required functions. We make use of a simple structure(Class), $Formula$, which contains two elements. First, the value of the cell which it represents, val, and a HashMap, $cells$. It is a list of cells on which the current cell's value is dependent. This $cells$ hashmap stores the data in the form $(cellName, count)$. $cellName$ has the format $ColRow$. $count$ refers to the number of times the current cell directly or indirectly comes in the current cell's formulas. e.g. $C1 = C2 + C3 + C2$. In this case, the frequency of $C3$ is 1 and that of $C2$ is 2.

- `Excel(int H, char W)` : We simply need to initialize an array of $Formula$ with H rows and the required number of columns corresponding to W.
- `set(int row, char column, int val)` : For setting the value of the cell corresponding to the given row and $column$, we can simply change the value , val, in the $Formula$ array at the indices corresponding to the current cell. Further, if any new formula is applied to a particular cell, we need to remove the previously applied formulas on the same cell. This is because two formulas can't be used to determine the value of a cell simultaneously. Now, setting a cell to a particular value can also be seen as a formula e.g. $C1 = 2$. Thus, we remove all the $cells$ in the $Formula$ for the current cell. Further, when the current cell's value is changed, all the other cells which are dependent on it also need to be evaluated in the correct order. Thus, we make use of Topological Sorting starting with the current cell. We make use of a function `topologicalSort(r, c)` for this purpose.
- `topologicalSort(r, c)` : In every call to this function, we traverse over all the cells in the $Formula$ array and further apply topological sorting to all the cells which are dependent on the current cell(row=r, column=c). To find these cells, we can check the $cells$ in the $Formula$s associated with each cell and check if the current cell lies in it. After applying Topological sorting to all these dependent cells, we put the current cell onto a $stack$.

After doing the topological sorting, the cells on the $stack$ lie in the order in which their values should be evaluated given the current dependency chain based on the formulas applied. Thus, we pick up these cells one by one, and evaluate their values. To do the evaluation, we make use of `calculate_sum(r, c, cells)`. In this function, we traverse over all the $cells$ in the $Formula$ of the current cell(row=r, column=c), and keep on adding their values. When this summing has been done, we update the current cell's value, val, to the sum just obtained. We keep on doing so till all the cells in the $stack$ have been evaluated.

- `get(int row, char column)` : We can simply obtain the value(val) associated with the current cell's $Formula$. If the cell has never been initialized previously, we can return a 0 value.
- `sum(int row, char column, List of Strings : numbers)` : To implement this function, firstly, we need to expand the given `numbers` to obtain all the cells which need to be added in the current formula. We obtain them, by making use of a `convert` function, which extracts all these cells by doing appropriate expansions based on `:` values. We put all these cells in the $cells$ associated with the current cell's $Formula$s. We also need to set the current cell's value to a new value based on the current formula added. For this, we make use of `calculate_sum` function as discussed above. We also need to do the topological sorting and evaluate all the cells dependent on the current cell. This is done in the same manner as in the `set` function discussed above. We also need to return the value to which the current cell has been set.

Java

```
public class Excel {
    Formula[][] Formulas;
    class Formula {
        Formula(HashMap < String, Integer > c, int v) {
            val = v;
            cells = c;
        }
        HashMap < String, Integer > cells;
        int val;
    }
    Stack < int[] > stack = new Stack < > ();
    public Excel(int H, char W) {
        Formulas = new Formula[H][W - 'A' + 1];
    }

    public int get(int r, char c) {
        if (Formulas[r - 1][c - 'A'] == null)
            return 0;
        return Formulas[r - 1][c - 'A'].val;
    }
    public void set(int r, char c, int v) {
        Formulas[r - 1][c - 'A'] = new Formula(new HashMap < String, Integer > (), v);
        topologicalSort(r - 1, c - 'A');
        execute_stack();
    }

    public int sum(int r, char c, String[] strs) {
```

Performance Analysis

- `set` takes $O((r * c)^2)$ time. Here, r and c refer to the number of rows and columns in the current Excel Form. There can be a maximum of $O(r * c)$ formulas for an Excel Form with r rows and c columns. For each formula, $r * c$ time will be needed to find the dependent nodes. Thus, in the worst case, a total of $O((r * c)^2)$ will be needed.
- `sum` takes $O((r * c)^2 + 2 * r * c * l)$ time. Here, l refers to the number of elements in the the list of strings used for obtaining the cells required for the current sum. In the worst case, the expansion of each such element requires $O(r * c)$ time, leading to $O(l * r * c)$ time for expanding l such elements. After doing the expansion, `calculate_sum` itself requires $O(l * r * c)$ time for traversing over the required elements for obtaining the sum. After this, we need to update all the dependent cells, which requires the use of `set` which itself requires $O((r * c)^2)$ time.
- `get` takes $O(1)$ time.
- The space required will be $O((r * c)^2)$ in the worst case. $O(r * c)$ space will be required for the Excel Form itself. For each cell in this form, the $cells$ list can contain $O(r * c)$ cells.

Analysis written by: @vinod23

Rate this article: ★ ★ ★ ★ ★

Previous Next