Facebook   Run   Submit

Description   Solutions   Submissions   Editorial

711. Number of Distinct Islands II Premium

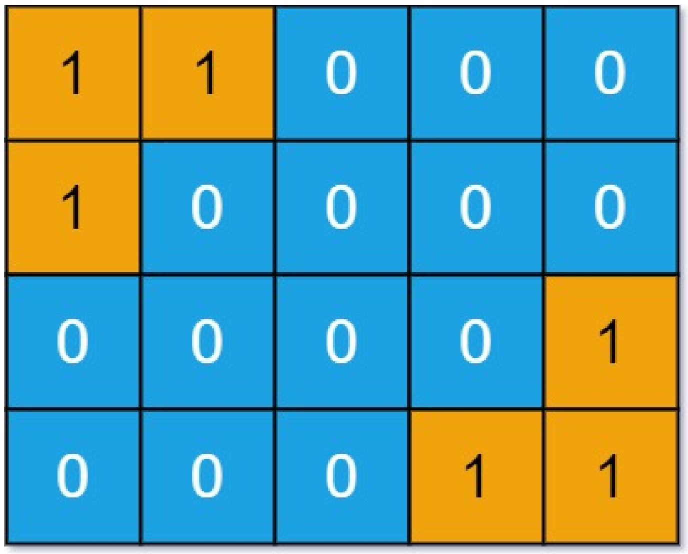
Hard   Topics   Companies

You are given an `m x n` binary matrix `grid`. An island is a group of `1`'s (representing land) connected **4-directionally** (horizontal or vertic
all four edges of the grid are surrounded by water.

An island is considered to be the same as another if they have the same shape, or have the same shape after **rotation** (90, 180, or 270 degr
reflection (left/right direction or up/down direction).

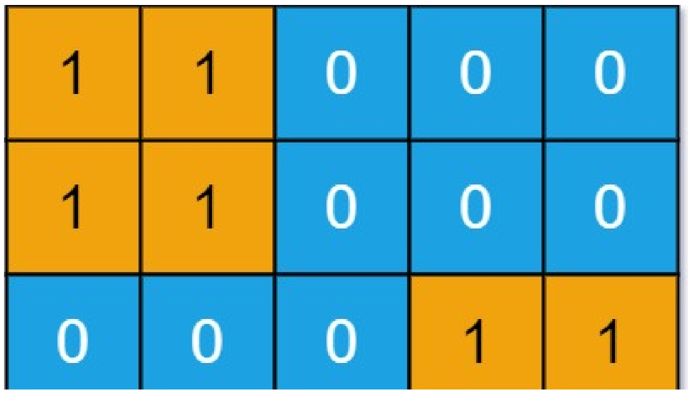
Return *the number of **distinct** islands*.

Example 1:



```
Input: grid = [[1,1,0,0,0],[1,0,0,0,0],[0,0,0,0,1],[0,0,0,1,1]]
Output: 1
Explanation: The two islands are considered the same because if we make a 180 degrees clockwise rota
first island, then two islands will have the same shapes.
```

Example 2:



Consise C++ solution, using DFS +sorting to find canonical representation for each island

imrusty
496 12386 Oct 17, 2017

After we get coordinates for an island, compute all possible rotations/reflections (https://en.wikipedia.org/wiki/Dihedral_group) of it and then sort them using the default comparison. The first representation in this order is then the canonical one.

```
class Solution {
public:
    map<int, vector<pair<int,int>>> mp;

    void dfs(int r, int c, vector<vector<int>> &g, int cnt) {
        if ( r < 0 || c < 0 || r >= g.size() || c >= g[0].size()) return;
        if (g[r][c] != 1) return;
        g[r][c] = cnt;
        mp[cnt].push_back({r,c});
        dfs(r+1,c,g,cnt);
        dfs(r-1,c,g,cnt);
        dfs(r,c+1,g,cnt);
        dfs(r,c-1,g,cnt);
    }

    vector<pair<int,int>> norm(vector<pair<int,int>> v) {
        vector<vector<pair<int,int>>> s(8);
        // compute rotations/reflections.
        for (auto p:v) {
            int x = p.first, y = p.second;
            s[0].push_back({x,y});
            s[1].push_back({x,-y});
            s[2].push_back({-x,y});
            s[3].push_back({-x,-y});
            s[4].push_back({y,-x});
            s[5].push_back({-y,x});
            s[6].push_back({-y,-x});
            s[7].push_back({y,x});
        }
        for (auto &l:s) sort(l.begin(),l.end());
        for (auto &l:s) {
            for (int i = 1; i < v.size(); ++i)
                l[i] = {l[i].first-l[0].first, l[i].second - l[0].second};
            l[0] = {0,0};
```

Facebook Run Submit

Description Solutions Submissions Editorial

All Solutions

```
            dfs(r+1,c,g,cnt);
            dfs(r-1,c,g,cnt);
            dfs(r,c+1,g,cnt);
            dfs(r,c-1,g,cnt);
        }

        vector<pair<int,int>> norm(vector<pair<int,int>> v) {
            vector<vector<pair<int,int>>> s(8);
            // compute rotations/reflections.
            for (auto p:v) {
                int x = p.first, y = p.second;
                s[0].push_back({x,y});
                s[1].push_back({x,-y});
                s[2].push_back({-x,y});
                s[3].push_back({-x,-y});
                s[4].push_back({y,-x});
                s[5].push_back({-y,x});
                s[6].push_back({-y,-x});
                s[7].push_back({y,x});
            }
            for (auto &l:s) sort(l.begin(),l.end());
            for (auto &l:s) {
                for (int i = 1; i < v.size(); ++i)
                    l[i] = {l[i].first-l[0].first, l[i].second - l[0].second};
                l[0] = {0,0};
            }
            sort(s.begin(),s.end());
            return s[0];
        }

        int numDistinctIslands2(vector<vector<int>>& g) {
            int cnt = 1;
            set<vector<pair<int,int>>> s;
            for (int i = 0; i < g.size(); ++i) for (int j = 0; j < g[i].size(); ++j) if (g[i][j] =
                dfs(i,j,g, ++cnt);
                s.insert(norm(mp[cnt]));
            }

            return s.size();
        }
    };
```

Most Straightforward Solution | No Rotation or Reflection Calculation | Java

LeetCoding_Pro
1739 1456 Jan 29, 2022

How to tell if island A has the same shape as island B? A suffcient way is that **all the connections between any "position pair" are the same**. For example, island A has [0, 0], [0, 1], [1, 0]. The connections are length 1 between [0, 0], [0, 1]; another length 1 between [0, 0], [1, 0], length 1.414 between [0, 1], [1, 0]; Similarly, island B has [5, 5], [4, 5], [5, 4]. The connections are length 1 between [5, 5], [4, 5]; another length 1 between [5, 5], [5, 4], length 1.414 between [4, 5], [5, 4]. So all the connections are the same length! They are in the same shape! Mission complete.

Tip: No need to calculate exact length, just keep the length with power of 2 to make sure no double value showing up.

```
class Solution
{
    private final int[][] dirs = new int[][]{{0, 1},{0, -1},{1, 0},{-1, 0}};

    public int numDistinctIslands2(int[][] mat)
    {
        Set<Map<Integer, Integer>> allDistinctIslands = new HashSet<>();
        int rows = mat.length;
        int cols = mat[0].length;
        for (int r = 0; r < rows; r++)
        {
            for (int c = 0; c < cols; c++)
            {
                if (mat[r][c] == 1)
                {
                    List<int[]> positions = new ArrayList<>();
                    getIsland(mat, r, c, positions);
                    Map<Integer, Integer> allDistanceCountMap = new HashMap<>();
                    for (int i = 0; i < positions.size(); i++)
                    {
                        for (int j = i + 1; j < positions.size(); j++)
                        {
                            int dist = (int)Math.pow(positions.get(i)[0] - positions.get(j)[0]
                            allDistanceCountMap.put(dist, allDistanceCountMap.getOrDefault(dis
                        }
                    }
                    allDistinctIslands.add(allDistanceCountMap);
                }
```

Facebook Run Submit

Description Solutions Submissions Editorial

All Solutions

```
            for (int c = 0; c < cols; c++)
            {
                if (mat[r][c] == 1)
                {
                    List<int[]> positions = new ArrayList<>();
                    getIsland(mat, r, c, positions);
                    Map<Integer, Integer> allDistanceCountMap = new HashMap<>();
                    for (int i = 0; i < positions.size(); i++)
                    {
                        for (int j = i + 1; j < positions.size(); j++)
                        {
                            int dist = (int)Math.pow(positions.get(i)[0] - positions.get(j)[0]
                            allDistanceCountMap.put(dist, allDistanceCountMap.getOrDefault(dis
                        }
                    }
                    allDistinctIslands.add(allDistanceCountMap);
                }
            }
        }
        return allDistinctIslands.size();
    }

    private void getIsland(int[][] mat, int r, int c, List<int[]> positions)
    {
        positions.add(new int[]{r, c});
        mat[r][c] = 0;
        for (int[] dir : dirs)
        {
            int rNext = r + dir[0];
            int cNext = c + dir[1];
            if (rNext < 0 || rNext >= mat.length || cNext < 0 || cNext >= mat[0].length || mat
            {
                continue;
            }
            getIsland(mat, rNext, cNext, positions);
        }
    }
}
```

Previous
Consise C++ solution, using DFS +sorting ...

Next
Python, normalize all pixels of an island in...

716. Max Stack Premium

Hard Topics Companies

Design a max stack data structure that supports the stack operations and supports finding the stack's maximum element.

Implement the `MaxStack` class:

- `MaxStack()` Initializes the stack object.
- `void push(int x)` Pushes element `x` onto the stack.
- `int pop()` Removes the element on top of the stack and returns it.
- `int top()` Gets the element on the top of the stack without removing it.
- `int peekMax()` Retrieves the maximum element in the stack without removing it.
- `int popMax()` Retrieves the maximum element in the stack and removes it. If there is more than one maximum element, only remove th

You must come up with a solution that supports `O(1)` for each `top` call and `O(logn)` for each other call.

Example 1:

```
Input
["MaxStack", "push", "push", "push", "top", "popMax", "top", "peekMax", "pop", "top"]
[[], [5], [1], [5], [], [], [], [], [], []]
Output
[null, null, null, null, 5, 5, 1, 5, 1, 5]

Explanation
MaxStack stk = new MaxStack();
stk.push(5);   // [5] the top of the stack and the maximum number is 5.
stk.push(1);   // [5, 1] the top of the stack is 1, but the maximum is 5.
stk.push(5);   // [5, 1, 5] the top of the stack is 5, which is also the maximum, because it is the
stk.top();     // return 5, [5, 1, 5] the stack did not change.
stk.popMax();  // return 5, [5, 1] the stack is changed now, and the top is different from the max.
stk.top();     // return 1, [5, 1] the stack did not change.
stk.peekMax(); // return 5, [5, 1] the stack did not change.
stk.pop();     // return 1, [5] the top of the stack and the max element is now 5.
stk.top();     // return 5, [5] the stack did not change.
```

Constraints:

- $-10^7 <= x <= 10^7$
- At most 10^5 calls will be made to `push`, `pop`, `top`, `peekMax`, and `popMax`.
- There will be **at least one element** in the stack when `pop`, `top`, `peekMax`, or `popMax` is called.

Python using stack + heap + set with explanation and discussion of performance

Hai_dee
4380 14037 Jun 11, 2019

I love implementations that combine a bunch of primitive data structures to make something awesome happen :D

Anyway, I used a **heap** for efficient finding of the max, and a **stack** for efficient finding of the most recent. Each new item was added onto both of these data structures.

I also assigned a decreasing ID to each added value so that they were all uniquely indentifible, as the same value could be added more than once, and we need to know which were more recently added. The reason for making them decreasing was so that they could be used to ensure the more recent items came up higher on the heap in the case of ties.

On `pop` operations, I directly popped from the data structure the operation was on, and then put the identifier into a **set** called `soft_deleted` . An identifier in `soft_deleted` represents an item that has been popped from one data structure, but not yet located and removed in the other. This avoids doing linear searches to try and find items that need deleting. I noticed a few sample solutions used 2 sets for this purpose, however I don't feel this is necessary. It's fine for them to share, and with O(1) removal from a set, we're not getting performance gains by seperating them.

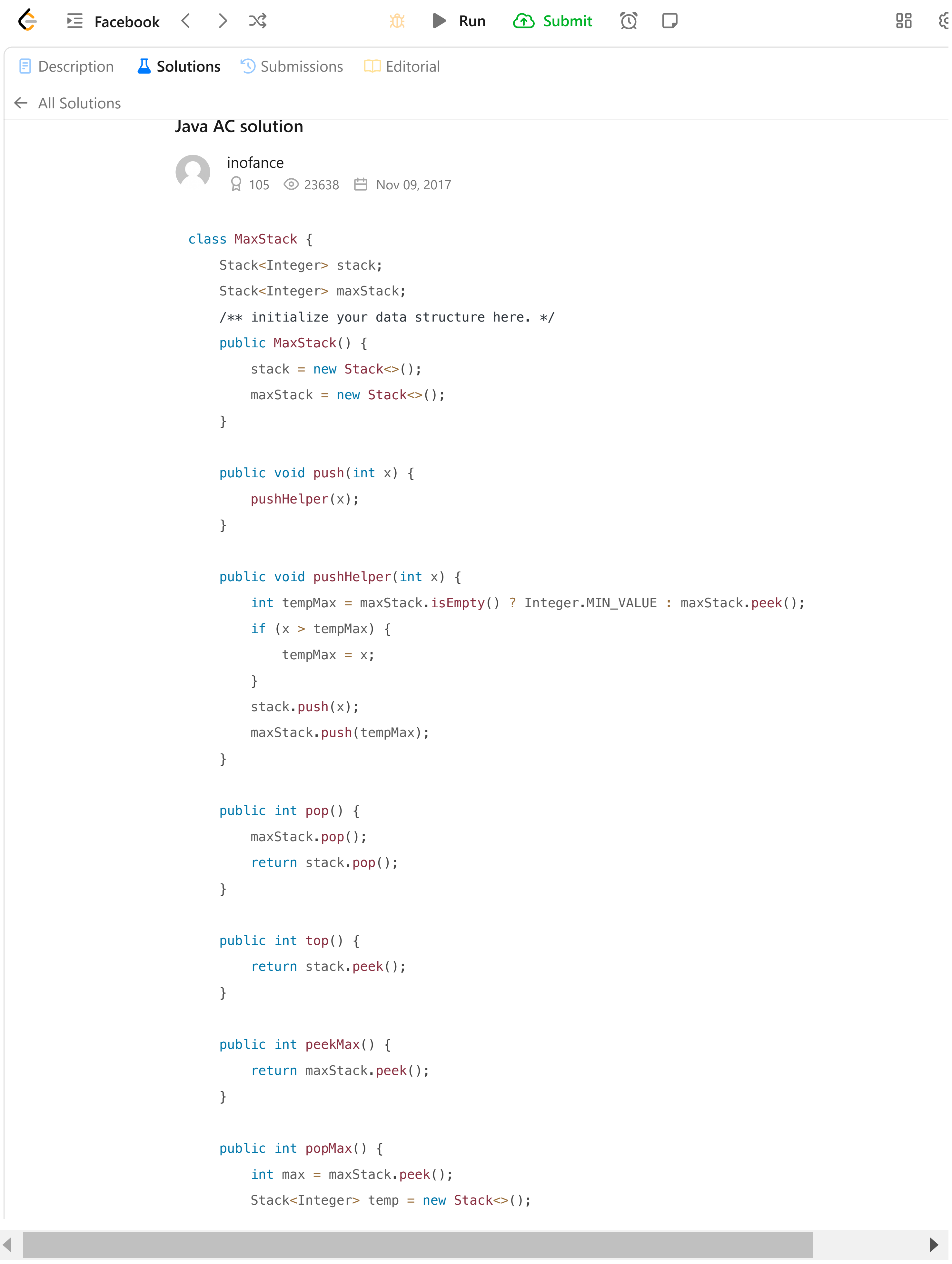
I defined a private function called `_clean_up` which checks the tops of the data structures and iteratively removes any soft deleted items from them. When a soft deleted item is removed, the identifier is also removed from `soft_deleted` as it no longer needs to be there, due to now being deleted from both sets.

So, we need to ensure that before we do a peek or a pop, that a clean up operation has been run to ensure that the tops of our data structures are clean. An interesting question is where this should be done. I decided to do it after a pop, so that then we wouldn't need to call it on peeks, although I need to think more about whether or not this is optimal. It's important to clean both data structures, because either could be "dirty" from the pop operation. On the one that had the pop done, the pop might have exposed a soft deleted element. On the other, it's possible that the element we just popped was also the top on it.

For the rather little test cases given, we can speed it up by not removing stuff from the soft deleted set. But if we have very large amounts of data going in and out, then in essence it'd be unbounded and thus a nasty memory leak. For this reason, I think it's better design to be doing those deletes.

Inorder to scale better and reduce memory wastage, we could do a "full clean" whenever the soft deleted set got above a certain size. This would involve rebuilding the stack and the heap with only the non deleted items.

I'm dubious about this being an easy-level question. Seems much more like a medium to me.



Java AC solution

inofance
105 23638 Nov 09, 2017

```
class MaxStack {
    Stack<Integer> stack;
    Stack<Integer> maxStack;
    /** initialize your data structure here. */
    public MaxStack() {
        stack = new Stack<>();
        maxStack = new Stack<>();
    }

    public void push(int x) {
        pushHelper(x);
    }

    public void pushHelper(int x) {
        int tempMax = maxStack.isEmpty() ? Integer.MIN_VALUE : maxStack.peek();
        if (x > tempMax) {
            tempMax = x;
        }
        stack.push(x);
        maxStack.push(tempMax);
    }

    public int pop() {
        maxStack.pop();
        return stack.pop();
    }

    public int top() {
        return stack.peek();
    }

    public int peekMax() {
        return maxStack.peek();
    }

    public int popMax() {
        int max = maxStack.peek();
        Stack<Integer> temp = new Stack<>();
```

```
            }
            stack.push(x);
            maxStack.push(tempMax);
        }

        public int pop() {
            maxStack.pop();
            return stack.pop();
        }

        public int top() {
            return stack.peek();
        }

        public int peekMax() {
            return maxStack.peek();
        }

        public int popMax() {
            int max = maxStack.peek();
            Stack<Integer> temp = new Stack<>();

            while (stack.peek() != max) {
                temp.push(stack.pop());
                maxStack.pop();
            }
            stack.pop();
            maxStack.pop();
            while (!temp.isEmpty()) {
                int x = temp.pop();
                pushHelper(x);
            }
            return max;
        }
    }
```

Previous
C++ O(logN) for write ops, O(1) for reads

Next
Fast and Simple Python Solution (lazy upd...

Comments (14)   Sort by: Best

Facebook Run Submit

Description Solutions Submissions Editorial

727. Minimum Window Subsequence Premium

Hard Topics Companies Hint

Given strings `s1` and `s2`, return *the minimum contiguous substring part of* `s1`*, so that* `s2` *is a subsequence of the part*.

If there is no such window in `s1` that covers all characters in `s2`, return the empty string `""`. If there are multiple such minimum-length wi
one with the **left-most starting index**.

Example 1:

```
Input: s1 = "abcdebdde", s2 = "bde"
Output: "bcde"
Explanation: 
"bcde" is the answer because it occurs before "bdde" which has the same length.
"deb" is not a smaller window because the elements of s2 in the window must occur in order.
```

Example 2:

```
Input: s1 = "jmeqksfrsdcmsiwvaovztaqenprpvnbstl", s2 = "u"
Output: ""
```

Constraints:

- `1 <= s1.length <= 2 * 10`^{4}
- `1 <= s2.length <= 100`
- `s1` and `s2` consist of lowercase English letters.

Seen this question in a real interview before? 1/4

Yes No

Accepted 89K Submissions 205.6K Acceptance Rate 43.3%

Topics

Companies

Hint 1

Similar Questions

Discussion (1)

dp[i][j] stores the starting index of the substring where T has length i and S has length j.

So dp[i][j would be:
if T[i - 1] == S[j - 1], this means we could borrow the start index from dp[i - 1][j - 1] to make the current substring valid;
else, we only need to borrow the start index from dp[i][j - 1] which could either exist or not.

Finally, go through the last row to find the substring with min length and appears first.

```
public String minWindow(String S, String T) {
    int m = T.length(), n = S.length();
    int[][] dp = new int[m + 1][n + 1];
    for (int j = 0; j <= n; j++) {
        dp[0][j] = j + 1;
    }
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (T.charAt(i - 1) == S.charAt(j - 1)) {
                dp[i][j] = dp[i - 1][j - 1];
            } else {
                dp[i][j] = dp[i][j - 1];
            }
        }
    }

    int start = 0, len = n + 1;
    for (int j = 1; j <= n; j++) {
        if (dp[m][j] != 0) {
            if (j - dp[m][j] + 1 < len) {
                start = dp[m][j] - 1;
                len = j - dp[m][j] + 1;
            }
        }
    }
    return len == n + 1 ? "" : S.substring(start, start + len);
}
```

Next
JAVA two pointer solution (12ms, beat 100%) with explaination

Comments (29) Sort by: Best

Facebook Run Submit

Description Solutions Submissions Editorial

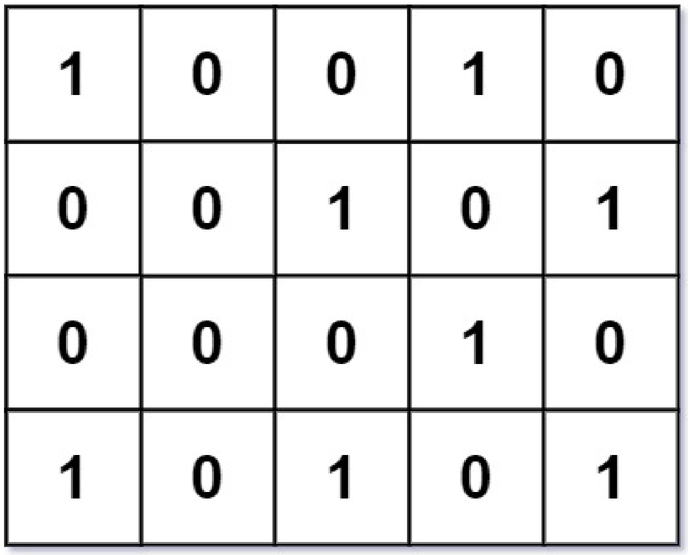
750. Number Of Corner Rectangles Premium

Medium Topics Companies Hint

Given an `m x n` integer matrix `grid` where each entry is only `0` or `1`, return *the number of* ***corner rectangles***.

A **corner rectangle** is four distinct `1`'s on the grid that forms an axis-aligned rectangle. Note that only the corners need to have the value
used must be distinct.

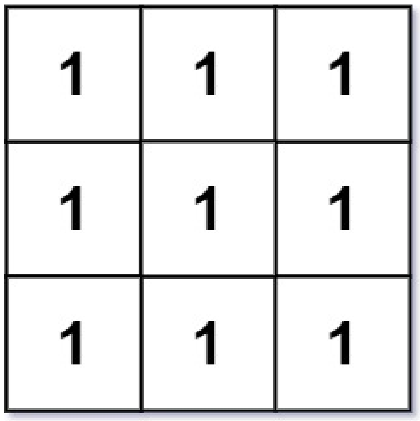
Example 1:



1	0	0	1	0
0	0	1	0	1
0	0	0	1	0
1	0	1	0	1

```
Input: grid = [[1,0,0,1,0],[0,0,1,0,1],[0,0,0,1,0],[1,0,1,0,1]]
Output: 1
Explanation: There is only one corner rectangle, with corners grid[1][2], grid[1][4], grid[3][2], gr
```

Example 2:



1	1	1
1	1	1
1	1	1

```
Input: grid = [[1,1,1],[1,1,1],[1,1,1]]
Output: 9
```

short JAVA AC solution (O(m^2 * n)) with explanation.

jun1013
678 12864 Dec 16, 2017

To find an axis-aligned rectangle, my idea is to fix two rows (or two columns) first, then check column by column to find "1" on both rows. Say you find n pairs, then just pick any 2 of those to form an axis-aligned rectangle (calculating how many in total is just high school math, hint: combination).

```java
class Solution {
    public int countCornerRectangles(int[][] grid) {
        int ans = 0;
        for (int i = 0; i < grid.length - 1; i++) {
            for (int j = i + 1; j < grid.length; j++) {
                int counter = 0;
                for (int k = 0; k < grid[0].length; k++) {
                    if (grid[i][k] == 1 && grid[j][k] == 1) counter++;
                }
                if (counter > 0) ans += counter * (counter - 1) / 2;
            }
        }
        return ans;
    }
}
```

Next
Summary of three solutions based on three different ideas

Comments (26) Sort by: Best

Type comment here... (Markdown supported)

Preview Comment

john221 Jan 06, 2018

Thanks OP for sharing the great solution.

I just want to add a little idea of optimization purposes:

772. Basic Calculator III Premium

Hard Topics Companies

Implement a basic calculator to evaluate a simple expression string.

The expression string contains only non-negative integers, `'+'`, `'-'`, `'*'`, `'/'` operators, and open `'('` and closing parentheses `')'`. Th
should **truncate toward zero**.

You may assume that the given expression is always valid. All intermediate results will be in the range of `[-2^31, 2^31 - 1]`.

Note: You are not allowed to use any built-in function which evaluates strings as mathematical expressions, such as `eval()`.

Example 1:

```
Input: s = "1+1"
Output: 2
```

Example 2:

```
Input: s = "6-4/2"
Output: 4
```

Example 3:

```
Input: s = "2*(5+5*2)/3+(6/2+8)"
Output: 21
```

Constraints:

- `1 <= s <= 10^4`
- `s` consists of digits, `'+'`, `'-'`, `'*'`, `'/'`, `'('`, and `')'`.
- `s` is a **valid** expression.

Seen this question in a real interview before? 1/4

Yes No

Accepted 121.1K Submissions 241.1K Acceptance Rate 50.2%

Topics

Companies

Similar Questions

Development of a generic solution for the series of the calculator problems

fun4LeetCode
24958 54730 Jan 25, 2018

So far, we have encountered the following series of calculator problems:

1. 224. Basic Calculator
2. 227. Basic Calculator II
3. 772. Basic Calculator III
4. 770. Basic Calculator IV

Though each of them may be solved using different methodologies, in this post I'd like to sort out the complexities and develop one "generic" solution to help us approach these problems.

Note that this post is **NOT** intended to deal with general calculator problems that involve complicated operands/operators. The analyses below are limited to the scope as specified by the problem descriptions of the above four problems.

```
I -- Definitions and terminology
```

In this section I will spell out some definitions to facilitate the explanation.

- `Expression` : An expression is a string whose value is to be calculated. Every expression must be alternating between **operand**s and **operator**s.
- `Operand` : An operand is a "generalized value" that can be the target of an operator. It must be one of the following three types -- **number**, **variable**, **subexpression**.
- `Number` : A number consists of digits only. The value of an operand of this type is given by the literal value of the number.
- `Variable` : A variable consists of lowercase letters only. It can be either a free variable whose value is unknown, or a bound variable whose value is mapped to some number (can be looked up). Here we define the value of an operand of free variable type is given by the variable itself, while that of bound variable type is given by the value of the number to which the variable is mapped.
- `Subexpression` : A subexpression is a valid expression enclosed in parentheses (which implies a recursive definition back to `Expression`). The value of an operand of this type is given by the calculated value of the subexpression itself.
- `Operator` : An operator is some prescribed action to be taken on the target operands. It must be one of the following four types: `+` , `-` , `*` , `/` , corresponding to addition, subtraction,

- `Operand` : An operand is a "generalized value" that can be the target of an operator. It must be one of the following three types -- **number**, **variable**, **subexpression**.
- `Number` : A number consists of digits only. The value of an operand of this type is given by the literal value of the number.
- `Variable` : A variable consists of lowercase letters only. It can be either a free variable whose value is unknown, or a bound variable whose value is mapped to some number (can be looked up). Here we define the value of an operand of free variable type is given by the variable itself, while that of bound variable type is given by the value of the number to which the variable is mapped.
- `Subexpression` : A subexpression is a valid expression enclosed in parentheses (which implies a recursive definition back to `Expression`). The value of an operand of this type is given by the calculated value of the subexpression itself.
- `Operator` : An operator is some prescribed action to be taken on the target operands. It must be one of the following four types: `+` , `-` , `*` , `/` , corresponding to addition, subtraction, multiplication and integer division, respectively. Operators have precedences, where `+` and `-` have **level one** precedence, while `*` and `/` have **level two** precedence (the higher the level is, the higher the precedence is).

`II -- Rules for calculations`

In this section, I will specify the general rules for carrying out the actual evaluations of the expression.

Separation rule:

- We separate the calculations into two different levels corresponding to the two precedence levels.
- For each level of calculation, we maintain two pieces of information: the *partial result* and the *operator in effect*.
- For level one, the partial result starts from `0` and the initial operator in effect is `+` ; for level two, the partial result starts from `1` and the initial operator in effect is `*` .
- We will use `l1` and `o1` to denote respectively the partial result and the operator in effect for level one; `l2` and `o2` for level two. The operators have the following mapping:
  `o1 == 1` means `+` ; `o1 == -1` means `-` ;
  `o2 == 1` means `*` ; `o2 == -1` means `/` .
  By default we have `l1 = 0` , `o1 = 1` , and `l2 = 1` , `o2 = 1` .

In this section, I will specify the general rules for carrying out the actual evaluations of the expression.

Separation rule:

- We separate the calculations into two different levels corresponding to the two precedence levels.
- For each level of calculation, we maintain two pieces of information: the *partial result* and the *operator in effect*.
- For level one, the partial result starts from `0` and the initial operator in effect is `+`; for level two, the partial result starts from `1` and the initial operator in effect is `*`.
- We will use `l1` and `o1` to denote respectively the partial result and the operator in effect for level one; `l2` and `o2` for level two. The operators have the following mapping:
  `o1 == 1` means `+`; `o1 == -1` means `-` ;
  `o2 == 1` means `*`; `o2 == -1` means `/`.
  By default we have `l1 = 0`, `o1 = 1`, and `l2 = 1`, `o2 = 1`.

Precedence rule:

- Each operand in the expression will be associated with a precedence of level two by default, meaning they can only take part in calculations of precedence level two, not level one.
- The operand can be any of the aforementioned types (number, variable or subexpression), and will be evaluated together with `l2` under the action prescribed by `o2`.

Demotion rule:

- The partial result `l2` of precedence level two can be demoted to level one. Upon demotion, `l2` becomes the operand for precedence level one and will be evaluated together with `l1` under the action prescribed by `o1`.
- The demotion happens when either a level one operator (i.e., `+` or `-`) is hit or the end of the expression is reached. After demotion, `l2` and `o2` will be reset for following calculations.

```
III -- Algorithm implementations
```

In this section, I will lay out the general structure of the algorithm using pseudo-codes. From section `I`, we know there are at most five different types of structs contained in the expression: number, variable, subexpression, level one operators, level two operators. We will check each of them and proceed accordingly.

Demotion rule:

- The partial result `l2` of precedence level two can be demoted to level one. Upon demotion, `l2` becomes the operand for precedence level one and will be evaluated together with `l1` under the action prescribed by `o1` .
- The demotion happens when either a level one operator (i.e., `+` or `-`) is hit or the end of the expression is reached. After demotion, `l2` and `o2` will be reset for following calculations.

`III -- Algorithm implementations`

In this section, I will lay out the general structure of the algorithm using pseudo-codes. From section `I` , we know there are at most five different types of structs contained in the expression: number, variable, subexpression, level one operators, level two operators. We will check each of them and proceed accordingly.

```
public int calculate(String s) {
    int l1 = 0, o1 = 1; // Initialization of level one
    int l2 = 1, o2 = 1; // Initialization of level two

    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);

        if (c is a digit) {

           --> we have an operand of type number, so find its value "num"
           --> then evaluate at level two: l2 = (o2 == 1 ? l2 * num : l2 / num);

        } else if (c is a lowercase letter) {

           --> we have an operand of type variable, so find its name "var"
           --> then look up the variable mapping table to find its value "num";
           --> lastly evaluate at level two: l2 = (o2 == 1 ? l2 * num : l2 / num);

        } else if (c is an opening parenthesis) {

           --> we have an operand of type subexpression, so find its string representation
           --> then recursively call the "calculate" function to find its value "num";
           --> lastly evaluate at level two: l2 = (o2 == 1 ? l2 * num : l2 / num);
```

```
public int calculate(String s) {
    int l1 = 0, o1 = 1; // Initialization of level one
    int l2 = 1, o2 = 1; // Initialization of level two

    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);

        if (c is a digit) {

           --> we have an operand of type number, so find its value "num"
           --> then evaluate at level two: l2 = (o2 == 1 ? l2 * num : l2 / num);

        } else if (c is a lowercase letter) {

           --> we have an operand of type variable, so find its name "var"
           --> then look up the variable mapping table to find its value "num";
           --> lastly evaluate at level two: l2 = (o2 == 1 ? l2 * num : l2 / num);

        } else if (c is an opening parenthesis) {

           --> we have an operand of type subexpression, so find its string representation
           --> then recursively call the "calculate" function to find its value "num";
           --> lastly evaluate at level two: l2 = (o2 == 1 ? l2 * num : l2 / num);

        } else if (c is a level two operator) {

            --> o2 needs to be updated: o2 = (c == '*' ? 1 : -1);

        } else if (c is a level one operator) {

            --> demotion happens here: l1 = l1 + o1 * l2;
            --> o1 needs to be updated: o1 = (c == '+' ? 1 : -1);
            --> l2, o2 need to be reset: l2 = 1, o2 = 1;

       }

       return (l1 + o1 * l2); // end of expression reached, so demotion happens again
}
```

IV -- List of solutions

It is now straightforward to adapt the algorithm in section III to tackle each of the calculator

`IV -- List of solutions`

It is now straightforward to adapt the algorithm in section `III` to tackle each of the calculator problems. Note that not all the checks are needed for a particular version of the calculator problems. More specifically,

- Basic Calculator I does not have variables and level two operators;
- Basic Calculator II does not contain variables as well as subexpressions;
- Basic Calculator III does not have variables;
- Basic Calculator IV is the most general form but its level two operators do not include division (`/`).

In this section, I will list two solutions for Basic Calculator III (recursive and iterative), and one solution for Basic Calculator IV (recursive).

Basic Calculator III: Solutions for this version pretty much follow the general structure in section `III` , except that we do not need to check for variables since the input expression does not contain any.

- Recursive solution: `O(n^2)` time, `O(n)` space

```
public int basicCalculatorIII(String s) {
    int l1 = 0, o1 = 1;
    int l2 = 1, o2 = 1;

    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);

        if (Character.isDigit(c)) {
            int num = c - '0';

            while (i + 1 < s.length() && Character.isDigit(s.charAt(i + 1))) {
                num = num * 10 + (s.charAt(++i) - '0');
            }

            l2 = (o2 == 1 ? l2 * num : l2 / num);

        } else if (c == '(') {
            int j = i;
```

Facebook Run Submit

Description Solutions Submissions Editorial

← All Solutions

III , except that we do not need to check for variables since the input expression does not contain any.

- Recursive solution: `O(n^2)` time, `O(n)` space

```
public int basicCalculatorIII(String s) {
    int l1 = 0, o1 = 1;
    int l2 = 1, o2 = 1;

    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);

        if (Character.isDigit(c)) {
            int num = c - '0';

            while (i + 1 < s.length() && Character.isDigit(s.charAt(i + 1))) {
                num = num * 10 + (s.charAt(++i) - '0');
            }

            l2 = (o2 == 1 ? l2 * num : l2 / num);

        } else if (c == '(') {
            int j = i;

            for (int cnt = 0; i < s.length(); i++) {
                if (s.charAt(i) == '(') cnt++;
                if (s.charAt(i) == ')') cnt--;
                if (cnt == 0) break;
            }

            int num = basicCalculatorIII(s.substring(j + 1, i));

            l2 = (o2 == 1 ? l2 * num : l2 / num);

        } else if (c == '*' || c == '/') {
            o2 = (c == '*' ? 1 : -1);

        } else if (c == '+' || c == '-') {
            l1 = l1 + o1 * l2;
            o1 = (c == '+' ? 1 : -1);

            l2 = 1; o2 = 1;
        }
```

- Iterative solution: O(n) time, O(n) space

```
public int basicCalculatorIII(String s) {
    int l1 = 0, o1 = 1;
    int l2 = 1, o2 = 1;

    Deque<Integer> stack = new ArrayDeque<>(); // stack to simulate recursion

    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);

        if (Character.isDigit(c)) {
            int num = c - '0';

            while (i + 1 < s.length() && Character.isDigit(s.charAt(i + 1))) {
                num = num * 10 + (s.charAt(++i) - '0');
            }

            l2 = (o2 == 1 ? l2 * num : l2 / num);

        } else if (c == '(') {
            // First preserve current calculation status
            stack.offerFirst(l1); stack.offerFirst(o1);
            stack.offerFirst(l2); stack.offerFirst(o2);

            // Then reset it for next calculation
            l1 = 0; o1 = 1;
            l2 = 1; o2 = 1;

        } else if (c == ')') {
            // First preserve the result of current calculation
            int num = l1 + o1 * l2;

            // Then restore previous calculation status
            o2 = stack.poll(); l2 = stack.poll();
            o1 = stack.poll(); l1 = stack.poll();

            // Previous calculation status is now in effect
            l2 = (o2 == 1 ? l2 * num : l2 / num);

        } else if (c == '*' || c == '/') {
            o2 = (c == '*' ? 1 : -1);
```

Facebook   Run   Submit

Description   Solutions   Submissions   Editorial

All Solutions

```
            while (i + 1 < s.length() && Character.isDigit(s.charAt(i + 1))) {
                num = num * 10 + (s.charAt(++i) - '0');
            }

            l2 = (o2 == 1 ? l2 * num : l2 / num);

        } else if (c == '(') {
            // First preserve current calculation status
            stack.offerFirst(l1); stack.offerFirst(o1);
            stack.offerFirst(l2); stack.offerFirst(o2);

            // Then reset it for next calculation
            l1 = 0; o1 = 1;
            l2 = 1; o2 = 1;

        } else if (c == ')') {
            // First preserve the result of current calculation
            int num = l1 + o1 * l2;

            // Then restore previous calculation status
            o2 = stack.poll(); l2 = stack.poll();
            o1 = stack.poll(); l1 = stack.poll();

            // Previous calculation status is now in effect
            l2 = (o2 == 1 ? l2 * num : l2 / num);

        } else if (c == '*' || c == '/') {
            o2 = (c == '*' ? 1 : -1);

        } else if (c == '+' || c == '-') {
            l1 = l1 + o1 * l2;
            o1 = (c == '+' ? 1 : -1);

            l2 = 1; o2 = 1;
        }
    }

    return (l1 + o1 * l2);
}
```

Basic Calculator IV: Solutions for this version, however, require some extra effort apart from the general structure in section `III`. Due to the presence of variables (free variables, to be exact), the partial results for each level of calculations may not be pure numbers, but instead expressions (simplified, of course). So we have to come up with some structure to represent these partial results.

Though multiple options exist as to designing this structure, we do want it to support operations like addition, subtraction and multiplication with relative ease. Here I represent each expression as a collection of mappings from terms to coefficients, where each term is just a list of free variables sorted in lexicographic order. Some quick examples:

1. `"2 * a * b - d * c"` : this expression contains two terms, where the first term is `["a", "b"]` and the second is `["c", "d"]`. The former has a coefficient of `2` while the latter has a coefficient of `-1`. So we represent the expression as two mappings: `["a", "b"] --> 2` and `["c", "d"] --> -1`.

2. " `4` ": this expression contains a single term, where the term has **zero** free variables and thus will be written as `[]`. The coefficient is `4` so we have the mapping `[] --> 4`. More generally, for any expression formed only by a pure number `num`, we have `[] --> num`.

See below for a detailed definition of the `Term` and `Expression` classes.

The `Term` class will support operations for comparing two terms according to their degrees as well as for generating a customized string representation.

The `Expression` class will support operations for adding additional terms to the existing mapping.

Addition of two expressions is done by adding all mappings in the second expression to those of the first one, and if possible, combine the coefficients of duplicate terms.

Subtraction of two expressions is implemented by first negating the coefficients of terms in the second expression and then applying addition (and thus can be combined with addition).

Multiplication of two expressions is done by collecting terms formed by merging every pair of terms from the two expressions (as well as their coefficients).

Lastly to conform to the format of the output, we have a dedicated function to convert the mappings in the result expression to a list of strings, where each string consists of the coefficient and the term connected by the `*` operator. Note that terms with `0` coefficient are ignored and terms without free variables contains numbers only.

As to complexity analysis, the nominal runtime complexity of this solution is similar to the recursive one for Basic Calculator III -- `O(n^2)`. It is also possible to use stacks to simulate the recursion process and cut the nominal time complexity down to `O(n)` (I will leave this as an exercise for

Addition of two expressions is done by adding all mappings in the second expression to those of the first one, and if possible, combine the coefficients of duplicate terms.

Subtraction of two expressions is implemented by first negating the coefficients of terms in the second expression and then applying addition (and thus can be combined with addition).

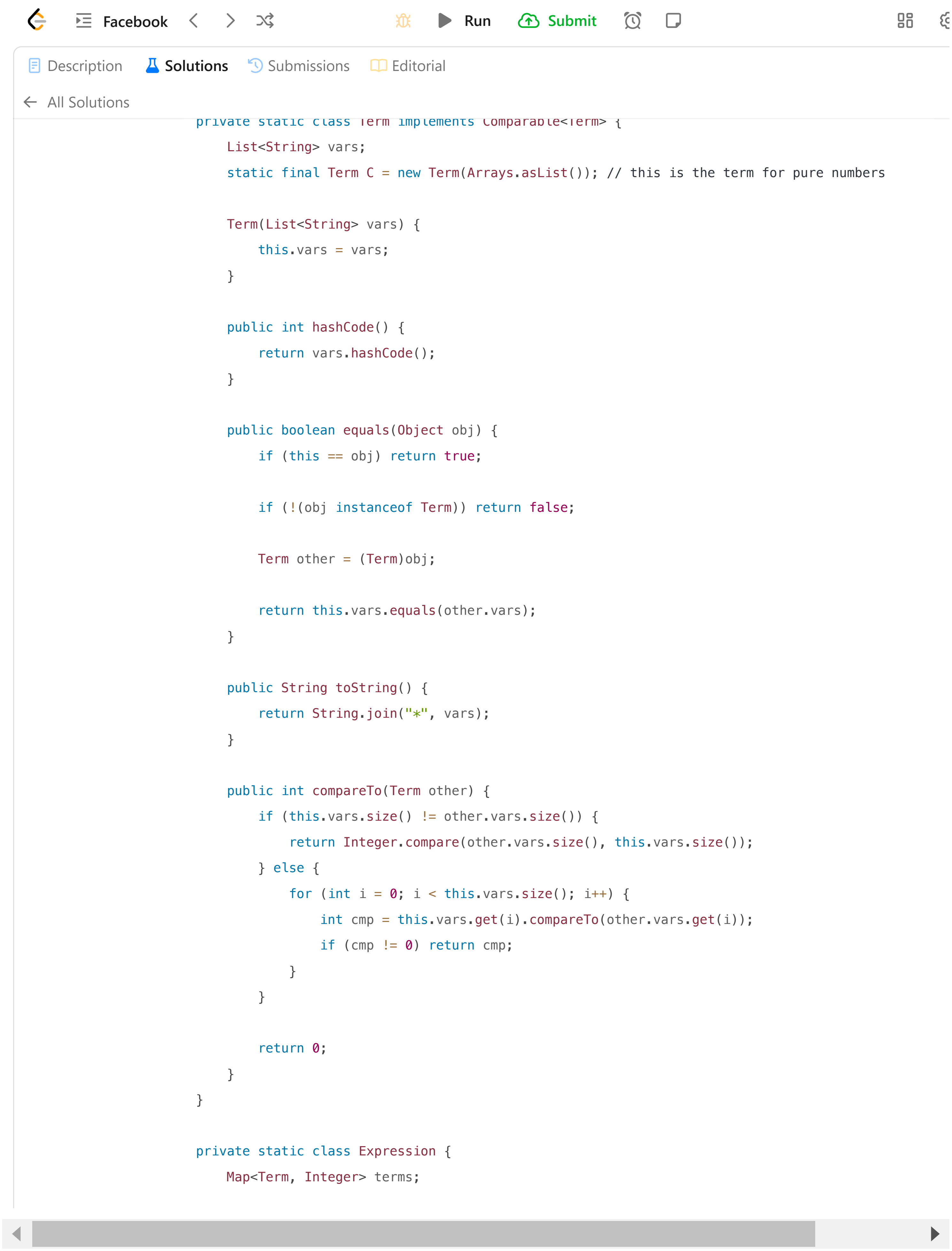
Multiplication of two expressions is done by collecting terms formed by merging every pair of terms from the two expressions (as well as their coefficients).

Lastly to conform to the format of the output, we have a dedicated function to convert the mappings in the result expression to a list of strings, where each string consists of the coefficient and the term connected by the `*` operator. Note that terms with `0` coefficient are ignored and terms without free variables contains numbers only.

As to complexity analysis, the nominal runtime complexity of this solution is similar to the recursive one for Basic Calculator III -- `O(n^2)` . It is also possible to use stacks to simulate the recursion process and cut the nominal time complexity down to `O(n)` (I will leave this as an exercise for you).

Here I used the word "nominal" because the above analyses assumed that the addition, subtraction and multiplication operations of two expressions take constant time, as is the case when the expressions are pure numbers. Apparently this assumption no longer stands true, since the number of terms may grow exponentially (think about this expression `(a + b) * (c + d) * (e + f) * ...`). Nevertheless, this solution should work against average/general test cases.

```
private static class Term implements Comparable<Term> {
    List<String> vars;
    static final Term C = new Term(Arrays.asList()); // this is the term for pure numbers

    Term(List<String> vars) {
        this.vars = vars;
    }

    public int hashCode() {
        return vars.hashCode();
    }

    public boolean equals(Object obj) {
        if (this == obj) return true;

        if (!(obj instanceof Term)) return false;

        Term other = (Term)obj;
```



Facebook Run Submit

Description Solutions Submissions Editorial

All Solutions

```
    private static class Term implements Comparable<Term> {
        List<String> vars;
        static final Term C = new Term(Arrays.asList()); // this is the term for pure numbers

        Term(List<String> vars) {
            this.vars = vars;
        }

        public int hashCode() {
            return vars.hashCode();
        }

        public boolean equals(Object obj) {
            if (this == obj) return true;

            if (!(obj instanceof Term)) return false;

            Term other = (Term)obj;

            return this.vars.equals(other.vars);
        }

        public String toString() {
            return String.join("*", vars);
        }

        public int compareTo(Term other) {
            if (this.vars.size() != other.vars.size()) {
                return Integer.compare(other.vars.size(), this.vars.size());
            } else {
                for (int i = 0; i < this.vars.size(); i++) {
                    int cmp = this.vars.get(i).compareTo(other.vars.get(i));
                    if (cmp != 0) return cmp;
                }
            }

            return 0;
        }
    }

    private static class Expression {
        Map<Term, Integer> terms;
```

```
            }

            return 0;
        }
    }

    private static class Expression {
        Map<Term, Integer> terms;

        Expression(Term term, int coeff) {
            terms = new HashMap<>();
            terms.put(term, coeff);
        }

        void addTerm(Term term, int coeff) {
            terms.put(term, coeff + terms.getOrDefault(term, 0));
        }
    }

    private Term merge(Term term1, Term term2) {
        List<String> vars = new ArrayList<>();

        vars.addAll(term1.vars);
        vars.addAll(term2.vars);
        Collections.sort(vars);

        return new Term(vars);
    }

    private Expression add(Expression expression1, Expression expression2, int sign) {
        for (Map.Entry<Term, Integer> e : expression2.terms.entrySet()) {
            expression1.addTerm(e.getKey(), sign * e.getValue());
        }

        return expression1;
    }

    private Expression mult(Expression expression1, Expression expression2) {
        Expression res = new Expression(Term.C, 0);

        for (Map.Entry<Term, Integer> e1 : expression1.terms.entrySet()) {
            for (Map.Entry<Term, Integer> e2 : expression2.terms.entrySet()) {
```

Facebook Run Submit

Description Solutions Submissions Editorial

All Solutions

```
        return expression1;
    }

    private Expression mult(Expression expression1, Expression expression2) {
        Expression res = new Expression(Term.C, 0);

        for (Map.Entry<Term, Integer> e1 : expression1.terms.entrySet()) {
            for (Map.Entry<Term, Integer> e2 : expression2.terms.entrySet()) {
                res.addTerm(merge(e1.getKey(), e2.getKey()), e1.getValue() * e2.getValue());
            }
        }

        return res;
    }

    private Expression calculate(String s, Map<String, Integer> map) {
        Expression l1 = new Expression(Term.C, 0);
        int o1 = 1;

        Expression l2 = new Expression(Term.C, 1);
        // we don't need 'o2' because the precedence level two operators contain '*' only

        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);

            if (Character.isDigit(c)) {  // this is a number
                int num = c - '0';

                while (i + 1 < s.length() && Character.isDigit(s.charAt(i + 1))) {
                    num = num * 10 + (s.charAt(++i) - '0');
                }

                l2 = mult(l2, new Expression(Term.C, num));

            } else if (Character.isLowerCase(c)) { // this is a variable
                int j = i;

                while (i + 1 < s.length() && Character.isLowerCase(s.charAt(i + 1))) i++;

                String var = s.substring(j, i + 1);
                Term term = map.containsKey(var) ? Term.C : new Term(Arrays.asList(var));
```

Facebook Run Submit

Description Solutions Submissions Editorial

All Solutions

```
            while (i + 1 < s.length() && Character.isDigit(s.charAt(i + 1))) {
                num = num * 10 + (s.charAt(++i) - '0');
            }

            l2 = mult(l2, new Expression(Term.C, num));

        } else if (Character.isLowerCase(c)) { // this is a variable
            int j = i;

            while (i + 1 < s.length() && Character.isLowerCase(s.charAt(i + 1))) i++;

            String var = s.substring(j, i + 1);
            Term term = map.containsKey(var) ? Term.C : new Term(Arrays.asList(var));
            int num = map.getOrDefault(var, 1);

            l2 = mult(l2, new Expression(term, num));

        } else if (c == '(') { // this is a subexpression
            int j = i;

            for (int cnt = 0; i < s.length(); i++) {
                if (s.charAt(i) == '(') cnt++;
                if (s.charAt(i) == ')') cnt--;
                if (cnt == 0) break;
            }

            l2 = mult(l2, calculate(s.substring(j + 1, i), map));

        } else if (c == '+' || c == '-') { // level one operators
            l1 = add(l1, l2, o1);
            o1 = (c == '+' ? 1 : -1);

            l2 = new Expression(Term.C, 1);
        }
    }

    return add(l1, l2, o1);
}

private List<String> format(Expression expression) {
    List<Term> terms = new ArrayList<>(expression.terms.keySet());
```

Facebook Run Submit

Description Solutions Submissions Editorial

All Solutions

```
            l1 = add(l1, l2, o1);
            o1 = (c == '+' ? 1 : -1);

            l2 = new Expression(Term.C, 1);
        }
    }

    return add(l1, l2, o1);
}

private List<String> format(Expression expression) {
    List<Term> terms = new ArrayList<>(expression.terms.keySet());

    Collections.sort(terms);

    List<String> res = new ArrayList<>(terms.size());

    for (Term term : terms) {
        int coeff = expression.terms.get(term);

        if (coeff == 0) continue;

        res.add(coeff + (term.equals(Term.C) ? "" : "*" + term.toString()));
    }

    return res;
}

public List<String> basicCalculatorIV(String expression, String[] evalvars, int[] evalints) {
    Map<String, Integer> map = new HashMap<>();

    for (int i = 0; i < evalvars.length; i++) {
        map.put(evalvars[i], evalints[i]);
    }

    return format(calculate(expression, map));
}
```

Next
Java and Python O(n) Solution Using Two Stacks