Problem Description

comments from the code: Line comments, which start with // and continue until the end of the line.

- a non-overlapping */ is found.
- A few important points to consider while solving this problem:

To solve this problem, we iterate through each line of the source code and use markers to identify the presence of a block comment.

If we are inside a block comment, we check for the closing */ sequence. If found, we ignore everything up to this point and mark that we've exited the block comment. If we are not in a block comment, we check for the start of a block comment /* or a line comment //. Upon finding a //, we can ignore the rest of the line. For a /*, we set the marker indicating we are inside a block comment, and

continue scanning from there. The key is to process characters based on the current context (inside a block comment or not).

Here's the approach broken down: 1. Initialize a list ans to collect the processed lines and a temporary list t that will hold the characters for the current line being processed. 2. Use a boolean block_comment to track whether we are inside a block comment.

5. If we're inside a block comment (block_comment is True), we look for the closing */. Once found, we set block_comment to False and continue scanning the rest of the line.

Solution Approach

3. Iterate through each line in the source code.

4. For each line, iterate through all characters in the line.

by additional amounts to skip over comment markers.

break out of the loop as the rest of the line is a comment and should be ignored.

into a string and add it to ans. We then clear t to start processing the next line.

- 6. If we're not inside a block comment, we look for either /* to start a block comment, // to skip the rest of the line, or any other character to add to our temporary list t.
- 7. After we've processed a line, if we are not in a block comment and t is not empty, join t into a string and add it to ans. Then,
- clear t for the next line of code. 8. Once all lines are processed, return ans.
- By following this approach, we systematically scan through the source code, correctly handling both line and block comments, and accumulating only the non-commented parts of the code.
- The Reference Solution Approach involves using a simple state machine with two states: inside a block comment, and outside of a block comment. The implementation leverages basic string manipulation techniques and a simple control flow to parse the lines of
- code and remove comments. Here is a walk-through of the implementation step-by-step:

1. Initialize Tracking Variables: We create a list ans to hold our output strings, and a list t to accumulate the characters for the

current line. In addition, we have a boolean flag block_comment to keep track of whether we are currently inside a block

2. Iterate Over Each Line: We loop over each line s in the input source. For each line, we process characters sequentially.

3. Iterate Over Characters in Line: Inside the line, we use a while loop to iterate over each character by index i. The index i is

temporary list t.

Example Walkthrough

" /* This is a",

" cout << a;",

1. Initialize Tracking Variables:

output while iterating through the source code.

" multiline comment */", // Line 3

" int a = 0; // variable", // Line 4

ans is initialized to an empty list [].

block_comment is initialized to False.

t is initialized to an empty list [].

// Line 5

comment.

characters form the */ sequence. If they do, we set block_comment to False and skip the */ by incrementing i by 2. Otherwise, we move on to the next character. 5. Process Outside Block Comment: When we're not in a block comment, we look for either the start of a new block comment /*

6. Accumulate Non-Comment Characters: If the character doesn't signal the beginning of a comment, we append it to the

increased when processing characters, and if certain conditions are met (like the end of a block comment), it may be increased

4. Process Inside Block Comment: If we're inside a block comment (when block_comment is True), we check if the current and next

or a line comment //. If we find /*, we set block_comment to True and skip the characters /* by incrementing i. If we find //, we

8. Return the Result: After processing all lines, ans will contain the source code with all comments removed. We return ans. By using this approach, we avoid the complexities of regex parsing or other more sophisticated text processing libraries and directly

implement a context-sensitive parser. The focus is on correctly adjusting the state (block_comment) and efficiently building the

7. Handle the End of Line: Once we reach the end of a line, if we are not inside a block comment and t is not empty, we convert t

C++ program: "int main() {", // Line 1 // Line 1 // Line 2

Let's illustrate the solution approach with a simple example. Suppose our source input is the following array of strings, representing a

Following the Reference Solution Approach, this is how the function will work:

We start with the first line "int main() {". Since there are no comments, each character is added to t.

2. Iterate Over Each Line:

3. Iterate Over Characters in Line:

and set block_comment to True. 4. Process Inside Block Comment:

For lines 5 and 6, since there are no comments, all characters go into t.

• After Line 5, t contains " cout << a;". This too is added to ans.</p>

Line 6 is just the closing brace, so it's added similarly.

def removeComments(self, source: List[str]) -> List[str]:

index += 1

index += 1

break

else:

 We continue to Line 3 while still inside the block comment. We find the ending */ sequence and set block_comment back to False after incrementing the index to skip it. 5. Process Outside Block Comment:

6. Accumulate Non-Comment Characters:

the rest of the line.

7. Handle the End of Line:

8. Return the Result:

Python Solution

1 class Solution:

2

6

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

3

10

11

12

13

14

15

16

17

18

19

20

21

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

5

9

10

11

12

14

15

16

17

18

19

20

21

22

23

24

25

33

34

35

36

37

38

39

41

40 }

};

We return ans.

 At the end of each line (except for lines 2 and 3, which are inside the block comment), we check t: • After Line 1, t contains "int main() {". We add it to ans and clear t. After Line 4, t contains " int a = 0; ". This is added to ans.

Moving to Line 2, we begin to iterate over the characters. Upon encountering /*, we recognize the start of a block comment

On Line 4, we are not in a block comment. We process characters until the line comment // is found. At this point, we skip

Here, we've successfully removed all comments (both line and block) from the C++ source code. The code now accurately reflects the logical part of the program without any comments.

final_output = []

current_line = []

for line in source:

else:

return final_output

in_block_comment = False

8 index, line_length = 0, len(line) 9 while index < line_length:</pre> 10 11 if in_block_comment: 12 # Check if the block comment ends on this line if index + 1 < line_length and line[index:index + 2] == "*/":</pre> 13

If not in a block comment and the line buffer contains characters, add it to the result

in_block_comment = False # End the block comment

if index + 1 < line_length and line[index:index + 2] == "/*":</pre>

elif index + 1 < line_length and line[index:index + 2] == "//":</pre>

current_line.clear() # Clear the line buffer for the next input line

// If in a block comment, look for the end of the block comment.

i++; // Skip the '/' character of the end block comment.

if $(i + 1 < lineLength \&\& line.charAt(i) == '*' && line.charAt(i + 1) == '/') {$

in_block_comment = True # Start the block comment

Check for the start of a block comment

Check for the start of a line comment

index += 1 # Move to the next character

final_output.append("".join(current_line))

if not in_block_comment and current_line:

public List<String> removeComments(String[] source) {

List<String> result = new ArrayList<>();

int lineLength = line.length();

if (inBlockComment) {

boolean inBlockComment = false;

for (String line : source) {

// A list to hold the final result without comments.

// Iterate over each line in the source code array.

inBlockComment = false;

for (int i = 0; i < lineLength; ++i) {</pre>

// Iterate over each line in the source code.

for (int i = 0; i < length; ++i) {</pre>

if (inBlockComment) {

break;

} else {

} else {

return cleanedSource;

for (const line of source) {

} else {

Time and Space Complexity

between /* and */) from a list of source code strings.

characters for comment patterns (// or /*, */).

Typescript Solution

// Iterate over each character in the line.

int length = line.size(); // Length of the current line

lineBuffer.push_back(line[i]);

if (!inBlockComment && !lineBuffer.empty()) {

// Return the source code with comments removed.

const result: string[] = []; // Array to store the results

function removeComments(source: string[]): string[] {

// Iterate through each line of the source code

// Iterate through characters of the line

for (let i = 0; i < lineLength; ++i) {</pre>

if (isBlockCommentActive) {

break;

cleanedSource.emplace_back(lineBuffer);

// then we have a line without comments to add to the result.

const temp: string[] = []; // Temporary array to collect non-comment characters

if (i + 1 < lineLength && line.slice(i, i + 2) === '*/') {</pre>

if $(i + 1 < lineLength && line.slice(i, i + 2) === '/*') {$

// If not in a block comment and temp array has collected characters

Considering n to be the total number of characters across all the strings in the source array:

temp.length = 0; // Clear temporary array for next line

isBlockCommentActive = false; // Turn off block comment flag

i++; // Skip next character as it's part of the comment end

isBlockCommentActive = true; // Turn on block comment flag

} else if $(i + 1 < lineLength && line.slice(i, i + 2) === '//') {$

// If line comment is found, ignore the rest of the line

result.push(temp.join('')); // Join characters and add to result array

i++; // Skip next character as it's part of the comment start

let isBlockCommentActive = false; // Flag to track block comment status

const lineLength = line.length; // Store current line length

// Check if the end of block comment is reached

// Check for start of a block comment

if (!isBlockCommentActive && temp.length > 0) {

return result; // Return the array with comments removed

for (const auto& line : source) {

List to store the output after removing comments

Ignore the rest of the line

current_line.append(line[index]) # Add the current character to the line buffer

Buffer to store characters of the current line being processed

Skip the ending characters of the block comment

Skip the starting characters of the block comment

Flag to indicate if we are inside a block comment or not

After all lines are processed, we have ans as ["int main() {", " int a = 0; ", " cout << a;", "}"].

6 // StringBuilder to construct strings without comments. StringBuilder lineWithoutComments = new StringBuilder(); 8 9 // A flag to indicate if we are inside a block comment.

Java Solution

1 class Solution {

```
22
                     } else {
 23
                         // If not in a block comment, look for the start of any comment.
 24
                         if (i + 1 < lineLength \&\& line.charAt(i) == '/' && line.charAt(i + 1) == '*') {
 25
                             inBlockComment = true;
 26
                             i++; // Skip the '*' character of the start block comment.
 27
                         } else if (i + 1 < lineLength && line.charAt(i) == '/' && line.charAt(i + 1) == '/') {
 28
                             // Found a line comment, break out of the loop to ignore the rest of the line.
 29
                             break;
                         } else {
 30
 31
                             // Character is not part of a comment; append it to the current line.
 32
                             lineWithoutComments.append(line.charAt(i));
 33
 34
 35
                 // If the current line is completed and we are not in a block comment, add the line to the result.
 36
 37
                 if (!inBlockComment && lineWithoutComments.length() > 0) {
                     result.add(lineWithoutComments.toString());
 38
 39
                     // Reset the StringBuilder for the next line.
                     lineWithoutComments.setLength(0);
 40
 41
 42
             // Return the list of lines without comments.
 43
 44
             return result;
 45
 46
 47
C++ Solution
  1 #include <vector>
  2 #include <string>
    class Solution {
    public:
         std::vector<std::string> removeComments(std::vector<std::string>& source) {
             std::vector<std::string> cleanedSource; // This will hold the final result without comments.
             std::string lineBuffer; // Buffer to construct a new string without comments for each line.
  8
             bool inBlockComment = false; // State variable to check if currently inside a block comment or not.
  9
```

// If currently inside a block comment, look for the end of the block comment.

if (i + 1 < length && line[i] == '*' && line[i + 1] == '/') {</pre>

++i; // Skip the '/' of the block comment end.

inBlockComment = false; // Found the end of block comment.

// If not in a block comment, check for the start of any comment.

inBlockComment = true; // Found the start of a block comment.

} else if (i + 1 < length && line[i] == '/' && line[i + 1] == '/') {</pre>

lineBuffer.clear(); // Clear the buffer for the next line of source code.

// Found the start of a line comment, ignore the rest of the line.

// This character is not part of a comment, add it to the buffer.

// If we're at the end of a line and we're not in a block comment and the buffer is not empty,

if (i + 1 < length && line[i] == '/' && line[i + 1] == '*') {</pre>

++i; // Skip the '*' of the block comment start.

26 } else { 27 // If no comment is found, add the character to the temp array 28 temp.push(line[i]); 29 30 31 32

The time complexity of the function is primarily determined by the number of characters in the input source array and the operations performed for each character. Here's the breakdown: The function loops through each string s in the source array.

For each string, a nested loop scans each character to identify and handle comments, with each iteration checking pairs of

The provided Python code defines a method to remove both line comments (starting with //) and block comments (enclosed

• The worst-case scenario occurs when there are no comments or only at the end, so we check every character pair, resulting in

Space Complexity:

Time Complexity:

O(n) time complexity. • Each character is assessed at most twice (for opening and closing block comments), maintaining the linear relationship. Hence, the overall time complexity of the function is O(n).

For space complexity, we consider any additional memory used by the algorithm besides the input itself: • A list t is maintained to construct the strings after removing comments, which in the worst case, will contain all characters that are not within comments. In the worst case, this is linear to the input, adding up to O(n).

The variable block_comment uses a constant amount of space, contributing an additional O(1) complexity.

• The list ans is used to store the final output strings. In the worst-case scenario where no comments are present, it will contain a copy of all the individual strings from the source, resulting in O(n) space.

Therefore, the total space complexity is 0(n) as the function stores intermediate and final results proportional to the input size.

- The problem requires us to write a function that removes comments from a given C++ program. The source code of the program is provided as an array of strings where each string represents a single line of code. The objective is to eliminate two types of
- The first occurrence of either type of comment takes precedence. Comments within comments are ignored (e.g., // within /*...*/ or /* within another /*...*/ or //). Lines that become empty after removal of comments should not be included in the output. Control characters, single or double quotes will not interfere with the comments.
- Intuition

- After processing the source code and removing the comments, the function should return the processed source code as an array of strings, with each string representing a non-empty line of the processed code.
- It is guaranteed that all block comments will be closed.
- Block comments, which start with /* and end with */. Block comments can span multiple lines and the block is closed only when