318. Maximum Product of Word Lengths

Medium Bit Manipulation <u>Array</u> String

Problem Description

do not have any letters in common. More formally, we want to find the maximum value of length(word[i]) * length(word[j]) where word[i] and word[j] are such that no letter appears in both words simultaneously. If there are no such pairs of words that satisfy this condition, the function should return 0.

Given an array of strings named words, the goal is to find the maximum product of the lengths of any two words in the array that

The straightforward approach to solve this problem would be to compare each pair of words and check if they have any common

ntuition

words and m is the average length of a word, rendering this approach inefficient for large input sizes. To optimize this, we can preprocess each word to create a bitmask that represents the set of characters of the word. In the bitmask, the i-th bit is set if the word contains the i-th letter of the alphabet. Since there are only 26 lowercase English letters, this bitmask can be represented by an integer. By comparing the bitmasks of two words, we can efficiently check if two words

letters. However, this would require checking every pair which leads to a time complexity of O(n^2 * m), where n is the number of

have common letters. If the result of a bitwise AND operation between two masks is zero, then the two words do not share any common letters. With the bitmask precomputation step, the time complexity is reduced significantly because the comparison of every pair now only takes constant time, O(1), leading to an overall time complexity of O(n^2 + n * m). Here's a step-by-step breakdown of the

solution approach: 1. Iterate through each word in the input, calculate its bitmask, and store these masks in an array.

2. Iterate through all pairs of words. For each pair, use the precomputed bitmasks to check if the words share common letters by performing a bitwise AND operation. 3. If the words do not share any common letters (bitwise AND result is 0), calculate the product of their lengths and update the answer with the

- maximum product found so far. 4. After checking all pairs, return the maximum product found.
- Solution Approach The solution uses a bit manipulation technique along with an array for storing the bitmasks. Here's a step-by-step breakdown of
- the implementation: **Initial Setup**

Calculate the number n of words in the input list.

o Initialize an array mask of length n to store the bitmasks which will represent the unique characters of each word by setting the bit position corresponding to each letter. **Creating Bitmasks**

- Initialize mask[i] to 0.
 - For each character ch in word[i], shift 1 left by ord(ch) ord('a') bits to create a bitmask for the letter. ■ Use the bitwise OR | operation to update mask[i] by turning on the bit corresponding to each character in the word.

Initialize a variable ans to 0 for storing the maximum product found.

After checking all pairs, return ans as the final result.

time, greatly enhancing performance.

- **Finding the Maximum Product**
- Iterate through all unique pairs of words with indices i and j (with j > i to avoid duplication of pairs), to compare their bitmasks. ■ If the bitwise AND & of mask[i] and mask[j] is 0, it means the words have no common letters.

Example Walkthrough

- Update ans with the maximum of itself and the calculated product. **Returning the Result**
- The code uses bitmasks stored in an integer to efficiently represent and compare the characters of the words. Rather than checking each letter individually for every pair, the solution leverages bitwise operations to check for common letters in constant

Let's take an example array of strings words = ["abc", "de", "fg", "hi", "aeh"].

The first step is to represent each word by a bitmask. Let's work through the words:

If so, calculate the product of their lengths len(words[i]) * len(words[j]).

• Iterate through the list of words with the index i. For each word word[i], do the following:

∘ For word "abc", the bitmask would be (set bit positions at 0, 1, 2): 0b111 which is 7 in decimal. ∘ For word "de", the bitmask would be (set bit positions at 3, 4): 0b11000 which is 24 in decimal. For word "fg", the bitmask would be (set bit positions at 5, 6): 0b1100000 which is 96 in decimal. For word "hi", the bitmask would be (set bit positions at 7, 8): 0b110000000 which is 384 in decimal.

In the next step, we will look for pairs of words with no overlapping bits in their bitmasks: • ○ Compare "abc" (7) and "de" (24): (7 & 24) equals Ø which means no common letters. Product of lengths equals 3 * 2 = 6.

• Compare "abc" (7) and "fg" (96): (7 & 96) equals 0; product is 3 * 2 = 6. Compare "abc" (7) and "hi" (384): (7 & 384) equals 0; product is 3 * 2 = 6.

For "aeh", the bitmask is (set bit positions at 0, 4, 7): 0b10010001 which is 145 in decimal.

 Compare "de" (24) and "hi" (384): (24 & 384) equals 0; product is 2 * 2 = 4. ○ Compare "de" (24) and "aeh" (145): (24 & 145) is not 0, so do not calculate product.

○ Compare "abc" (7) and "aeh" (145): (7 & 145) is not 0 as they share letters, so do not calculate product.

As a result, the mask array after processing the words array would be [7, 24, 96, 384, 145].

 Compare "fg" (96) and "hi" (384): (96 & 384) equals 0; product is 2 * 2 = 4. ○ Compare "fg" (96) and "aeh" (145): (96 & 145) equals 0; product of lengths is 2 * 3 = 6. Compare "hi" (384) and "aeh" (145): (384 & 145) equals 0; product is 2 * 3 = 6.

The final step is to find the maximum product from these comparisons:

"fg" with "aeh", and "hi" with "aeh".

def max_product(self, words: List[str]) -> int:

masks[i] |= 1 << (ord(ch) - ord('a'))

Compare every pair of words to find the maximum product of lengths

of two words which have no characters in common (no common bits in the bitmask).

Get the number of words in the list

num words = len(words)

Initialize max_product to 0

for i in range(num_words - 1):

Return the maximum product found

// Return the maximum product found

int maxProduct(vector<string>& words) {

for (int i = 0; i < wordsCount; ++i) {

for (int i = 0; i < wordsCount - 1; ++i) {

if (!(masks[i] & masks[j])) {

for (char ch : words[i]) {

// Return the maximum product found

function maxProduct(words: string[]): number {

// Return the maximum product found

def max_product(self, words: List[str]) -> int:

Get the number of words in the list

return maxProduct;

from typing import List

class Solution:

// Initialize all masks to 0

masks.fill(0);

return maxProduct;

// Compare each pair of words

int wordsCount = words.size(); // Count of the words in the vector

// Create bitmasks. Each bit in mask represents if a character ('a' to 'z') is in the word

masks[i] |= 1 << (ch - 'a'); // Set the bit corresponding to the current character

// If two words have no common letters, their masks will not share any common bits

maxProduct = max(maxProduct, (int)(words[i].size() * words[j].size()));

// Update max product if this pair gives us a bigger product

// Function to calculate the maximum product of lengths of two words that don't share common characters

const wordsCount: number = words.length; // Count of the words in the array

// Update max product if this pair gives us a bigger product

maxProduct = Math.max(maxProduct, words[i].length * words[j].length);

const masks: number[] = new Array(wordsCount); // Bitmasks for each word

vector<int> masks(wordsCount); // Bitmasks for each word

int maxProduct = 0; // Initialize max product to be 0

// The bitwise AND of their masks will be 0

for (int j = i + 1; j < wordsCount; ++j) {</pre>

return maxProduct;

Compare "de" (24) and "fg" (96): (24 & 96) equals 0; product is 2 * 2 = 4.

- The calculated products are 6, 6, 6, 4, 4, 4, 6, 6. • The maximum product is 6.
- bitwise operations and precomputation of the bitmasks for each word. Solution Implementation

Create a list to store the bitmask representation of each word masks = [0] * num_words # Generate a bitmask for each word where bit i is set if the # word contains the i-th letter of the alphabet for i, word in enumerate(words): for ch in word:

Return the maximum product, which is 6. The words yielding this product are "abc" with "de", "abc" with "fg", "abc" with "hi",

This illustrates how the bitmask approach greatly simplifies the computation and avoids the overhead of checking each letter

individually. The final solution would implement the process outlined in the "Solution Approach" section efficiently by leveraging

for j in range(i + 1, num_words): if masks[i] & masks[j] == 0: # No common characters # Update max_product if this pair has a larger product max_product = max(max_product, len(words[i]) * len(words[j]))

max_product = 0

return max_product

Python

from typing import List

class Solution:

```
Java
class Solution {
    public int maxProduct(String[] words) {
       // Get the length of the words array
       int length = words.length;
       // Create an array to store the bitmask representation of each word
       int[] bitMasks = new int[length];
       // Convert each word into a bitmask representation and store it in the bitMasks array
        for (int i = 0; i < length; ++i) {</pre>
            for (char c : words[i].toCharArray()) {
                // Set the bit corresponding to the character 'c'
                bitMasks[i] |= (1 << (c - 'a'));
       // Initialize the maximum product to 0
       int maxProduct = 0;
       // Compare each pair of words to find the pair with the maximum product of lengths
       // where the words do not share any common characters
        for (int i = 0; i < length - 1; ++i) {
            for (int j = i + 1; j < length; ++j) {
                // Check if the two words share any common characters using the '&' bitwise operator
                if ((bitMasks[i] & bitMasks[j]) == 0) {
                    // Calculate the product of the lengths of the two words
                    int product = words[i].length() * words[j].length();
                    // Update maxProduct with the maximum value between the existing maxProduct and the current product
                    maxProduct = Math.max(maxProduct, product);
```

};

TypeScript

C++

public:

#include <vector>

#include <string>

class Solution {

#include <algorithm>

using namespace std;

// Create bitmasks. Each bit in a mask represents if a character ('a' to 'z') is in the word for (let i = 0; i < wordsCount; ++i) {</pre> for (const char of words[i]) { masks[i] |= 1 << (char.charCodeAt(0) - 'a'.charCodeAt(0)); // Set the bit corresponding to the current character let maxProduct = 0; // Initialize max product to be 0 // Compare each pair of words for (let i = 0; i < wordsCount - 1; ++i) {</pre> for (let j = i + 1; j < wordsCount; ++j) {</pre> // If two words have no common letters, their masks will not share any common bits if ((masks[i] & masks[j]) === 0) { // The bitwise AND of their masks will be 0

// Import statements are not necessary in plain TypeScript, and there are no direct equivalents for `vector` and `using namespace

```
num words = len(words)
# Create a list to store the bitmask representation of each word
masks = [0] * num words
# Generate a bitmask for each word where bit i is set if the
# word contains the i-th letter of the alphabet
for i, word in enumerate(words):
    for ch in word:
        masks[i] |= 1 << (ord(ch) - ord('a'))
# Initialize max_product to 0
max_product = 0
# Compare every pair of words to find the maximum product of lengths
# of two words which have no characters in common (no common bits in the bitmask).
for i in range(num_words - 1):
    for j in range(i + 1, num_words):
        if masks[i] & masks[j] == 0: # No common characters
            # Update max_product if this pair has a larger product
            max_product = max(max_product, len(words[i]) * len(words[j]))
# Return the maximum product found
return max_product
```

Calculating the bitmask for each word: The first loop goes through each word and each character within those words to create a bitmask. This process occurs in O(N * L) time, where N is the number of words, and L is the average length of the words.

Time Complexity

compared.

Space Complexity

Time and Space Complexity

Comparing the bitmasks: The nested loop compares each pair of generated bitmasks. This results in 0(N^2) comparisons, since each word's bitmask is compared with every other word's bitmask. Checking for common characters and updating the maximum product happens in constant time, 0(1), for each pair of words

The provided code calculates the maximum product of the lengths of two words such that the two words do not share any

Combining these steps, the overall time complexity is $0(N * L + N^2)$.

common characters. The analysis of time and space complexity is as follows:

- Since typically L <= 1000 and the main contributing factor for large inputs is N, and because N^2 grows faster than N * L, the N^2 term dominates for large N, so the overall time complexity can be considered $0(N^2)$.
- The space used to store the bitmasks: For N words, we store a bitmask for each, which uses O(N) space. No additional significant space is used, since other variables use constant space.

Hence, the space complexity of the code is O(N).