318. Maximum Product of Word Lengths

Average Rating: 4.28 (24 votes)

Given a string array `words`, find the maximum value of `length(word[i]) * length(word[j])` where the two words do not share common letters. You may assume that each word will contain only lower case letters. If no such two words exist, return 0.

Example 1:

```
Input: ["abcw","baz","foo","bar","xtfn","abcdef"]
Output: 16
Explanation: The two words can be "abcw", "xtfn".
```

Example 2:

```
Input: ["a","ab","abc","d","cd","bcd","abcd"]
Output: 4
Explanation: The two words can be "ab", "cd".
```

Example 3:

```
Input: ["a","aa","aaa","aaaa"]
Output: 0
Explanation: No such pair of words.
```

Solution

Overview

Let's start from the naive straightforward solution.

> Compare each word with all the following words one by one. If two words have no common letters, update the maximum product `maxProd`.

Let's omit for the moment the implementation of `noCommonLetters` function.

```
class Solution:
    def maxProduct(self, words: List[str]) -> int:
        for i in range(len(words), 4):
            # code
        n = len(words)
        max_prod = 0
        for i in range(n):
            for j in range(i + 1, n):
                if noCommonLetters(words[i], words[j]):
                    max_prod = max(max_prod, len(words[i]) * len(words[j]))
        return max_prod
```

The number of operations performed in the nested loops is

$(N-1) + (N-2) + \ldots + 2 + 1 = \frac{N(N-1)}{2}$

that results in $O(N^2 \times f(L_1, L_2))$ time complexity. Here $f(L_1, L_2)$ is a complexity of function `noCommonLetters(string s1, string s2)`, i.e. the price to compare two words of lengths L_1 and L_2.

> What could be done here?

What could be done ?

- Decrease number of word comparisons: Hashing
- Optimize noCommonLetters: Bitmasks + Precomputation

Bitmasks + Precomputation + Hashmap

- Approach 1: minimize time complexity $f(L_1, L_2)$ of `noCommonLetters` function.
- Approach 2: minimize the number of word comparisons. There is no need to always perform $O(N^2)$ comparisons. Among all the strings with the same set of letters (ah, aaaaaaaaaaah, bbdfhbdfbd) it's enough to keep the longest one (aaaaaaaaaaah).

Naive solution $O(L_1 + L_2)$ **time**

Bitmasks $O(L_1 + L_2)$ **time**

Bitmasks + Precomputation $O(L_1 + L_2)$ **time**

Bitmasks + Precomputation + Hashmap $O(1)$ **time**

In the previous approach one computes a bitmask of each word 9 times. In fact, each bitmask could be precomputed just once, memoized and then used for the runtime comparison in a constant time.

Let's use two integer arrays to store bitmasks and string lengths. That's Java-optimized way, since in general Java works faster with arrays than with hashmaps.

Algorithm

- Precompute bitmasks for all words and save them in the array `masks`. Use array `lens` to keep the lengths for all words.
- Compare each word with all the following words one by one. If two words have no common letters, update the maximum product `maxProd`. Perform "no common letters" check in a constant time with the help of precomputed `masks` array: `(masks[i] & masks[j]) == 0`.
- Return `maxProd`.

Implementation

```
class Solution:
    def maxProduct(self, words: List[str]) -> int:
        n = len(words)
        masks = [0] * n
        lens = [0] * n
        LOS_number = lambda ch: ord(ch) - ord('a')

        for i in range(n):
            bitmask = 0
            for ch in s:
                bitmask |= 1 << LOS_number(ch)
            masks[i] = bitmask
            lens[i] = len(words[i])

        max_prod = 0
        for i in range(n):
            for j in range(i + 1, n):
                if max_prod < lens[i] * lens[j]:
                    max_prod = lens[i] * lens[j]
        return max_prod
```

Complexity Analysis

- Time complexity : $O(N^2 \times L)$ where N is a number of words and L is a total length of all words together. The precomputation takes $O(L)$ time because we iterate over all characters in all words. The runtime word comparison takes $O(N^2)$ time. In total, that results in $O(N^2 \times L)$ time complexity.
- Space complexity : $O(N)$ to keep two arrays of N elements.

Approach 2: Optimise Number of Comparisons : Bitmasks + Precomputation + Hashmap

Now, when the comparison itself is optimised, one could optimize the number of comparisons. There is no need to always perform $O(N^2)$ comparisons. Among all the strings with the same set of letters (ah, aaaaaaaaaaah, bbdfhbdfbd) it's enough to keep the longest one (aaaaaaaaaaah).

For that, instead of two arrays of length N as in the Approach 1, one could use a hashmap : bitmask -> word with max length of the word.

Bitmask of 26 bits

bit number in bitmask

words with equal bitmasks

bit number for each letter : 0=('a') ... 25=('z')

Algorithm

- Precompute bitmasks for all words and save them in the hashmap `bmhash` -> max word length with such a bitmask. Check there for several words with the same bitmask, for example, "a" and "aaaaaa".
- Compare each word with all the following words one by one. If two words have no common letters, update the maximum product `maxProd`. Perform "no common letters" check in a constant time with the help of precomputed hashmap of bitmasks: `(x & y) == 0`.
- Return `maxProd`.

Implementation

```
from collections import defaultdict
class Solution:
    def maxProduct(self, words: List[str]) -> int:
        hashmap = defaultdict(int)
        LOS_number = lambda ch: ord(ch) - ord('a')

        for word in words:
            bitmask = 0
            for ch in word:
                # set LOS number in bitmask
                bitmask |= 1 << LOS_number(ch)
            # there could be different words with the same bitmask
            # so, do max here
            hashmap[bitmask] = max(hashmap[bitmask], len(word))

        max_prod = 0
        for x in hashmap:
            for y in hashmap:
                if x & y == 0:
                    max_prod = max(max_prod, hashmap[x] * hashmap[y])
        return max_prod
```

Complexity Analysis

- Time complexity : $O(N^2 \times L)$ where N is a number of words and L is a total length of all words together. If you want to have some fun, here is a bloody discussion that all this is for "small" N only, when $N \le 2^{26}$. The idea is that the number of bitmasks is not more than 2^{26} and hence for $N > 2^{26}$ the complexity is $O(L)$.
- Space complexity : $O(N)$ to keep a hashmap of N elements if $N < 2^{26}$. Otherwise, it's $O(2^{26}) = O(1)$.

Rate this article: ★ ★ ★ ★ ★

Previous Next

Comments: 9

Sort By

Type comment here... (Markdown is supported)

Preview Post

Show More

gopitaparar101 ★ 0 May 30, 2020 5:02 PM
Now explanation to approach a problem.
1 Reply

dr_dk ★ 187 April 17, 2020 9:11 AM
Great article that explains the flow of optimization thought process.
1 Reply

NarayanaChintak ★ 0 August 27, 2019 9:32 AM
Nevertheless, the word case time complexity has to Nsquared right? We just are trying to decrease computations in some case?
1 Reply

(edited 1 day)

vachon92 ★ 0 March 29, 2020 2:57 PM
I have created the bitmask the way it is described here for example for string = "bc" bitmask= 0 1 0 0 0