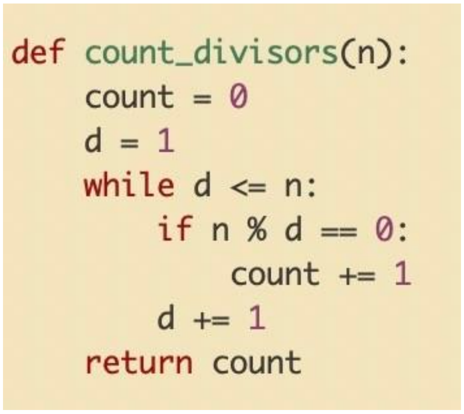
Complexity Analysis

▪ Problem:

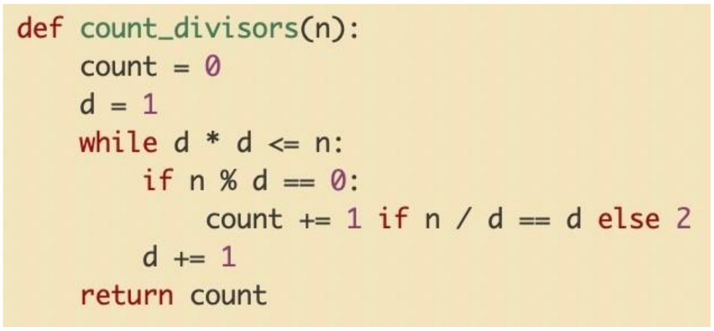
Given an integer n, count the number of its divisors.

Solution 1:



```python
def count_divisors(n):
    count = 0
    d = 1
    while d <= n:
        if n % d == 0:
            count += 1
        d += 1
    return count
```

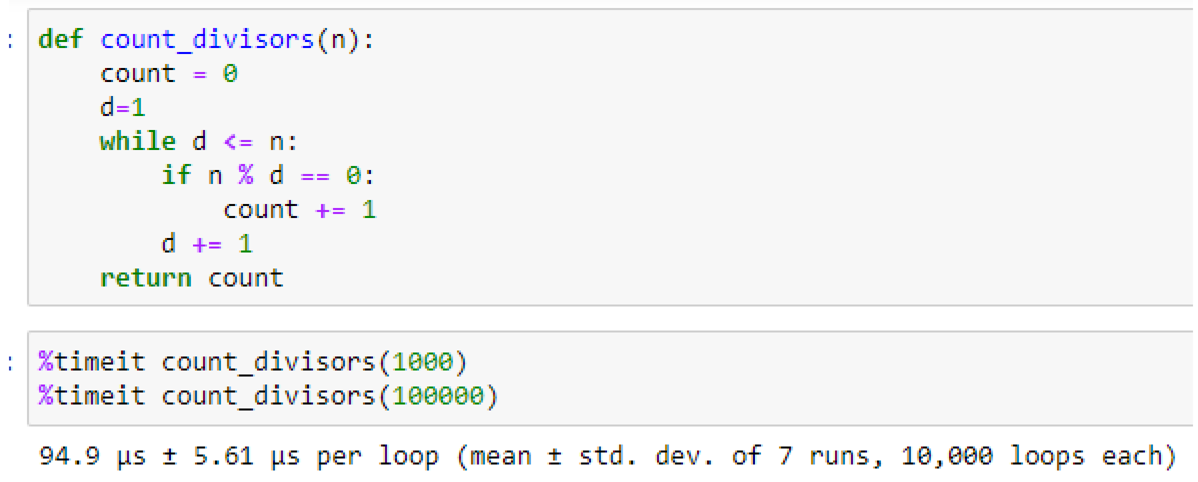
Solution 2:



```python
def count_divisors(n):
    count = 0
    d = 1
    while d * d <= n:
        if n % d == 0:
            count += 1 if n / d == d else 2
        d += 1
    return count
```

Introduction

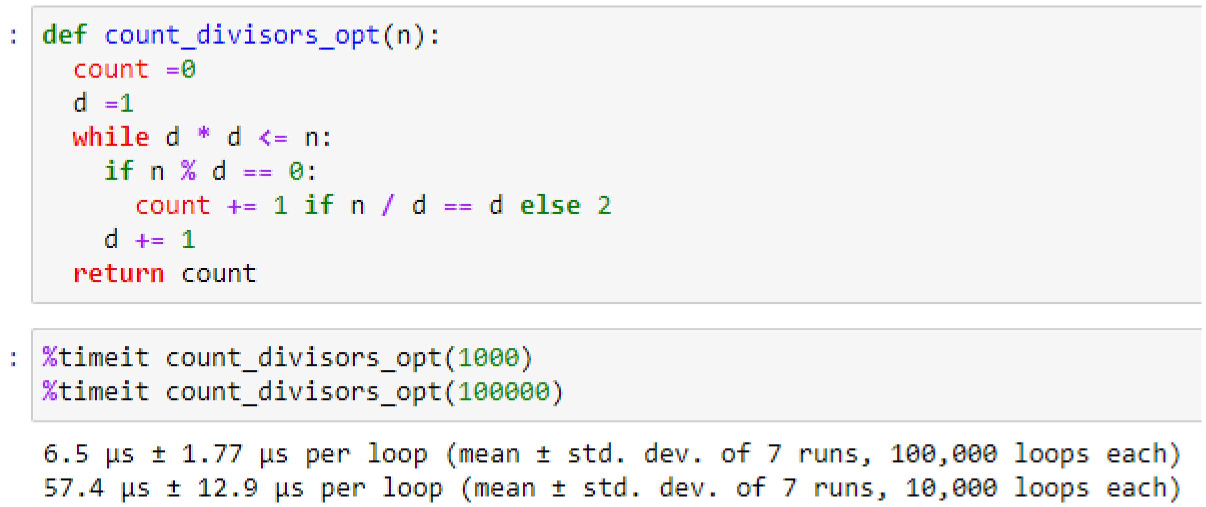
1) Describe solution 1
   Function count_divisors takes an integer n as input and returns the number of divisors that n has. It does so by initializing a counter variable to 0, iterating over all integers between 1 and n, and incrementing the counter for each integer that divides n without a rest.
2) Describe solution 2
   This function aims to count the number of divisors of n in a more optimized way than the previous implementation because it iterates only up to the square root of n (to determine the divisors of a number n, it is sufficient to check for values up to the square root of n.)
3) Run the two programs for different values of n and measure which algorithm is faster.



```python
: def count_divisors(n):
      count = 0
      d=1
      while d <= n:
          if n % d == 0:
              count += 1
          d += 1
      return count
```

```
: %timeit count_divisors(1000)
  %timeit count_divisors(100000)
```

```
94.9 µs ± 5.61 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```



```
: def count_divisors_opt(n):
    count =0
    d =1
    while d * d <= n:
      if n % d == 0:
        count += 1 if n / d == d else 2
      d += 1
    return count
```

```
: %timeit count_divisors_opt(1000)
  %timeit count_divisors_opt(100000)

  6.5 µs ± 1.77 µs per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
  57.4 µs ± 12.9 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

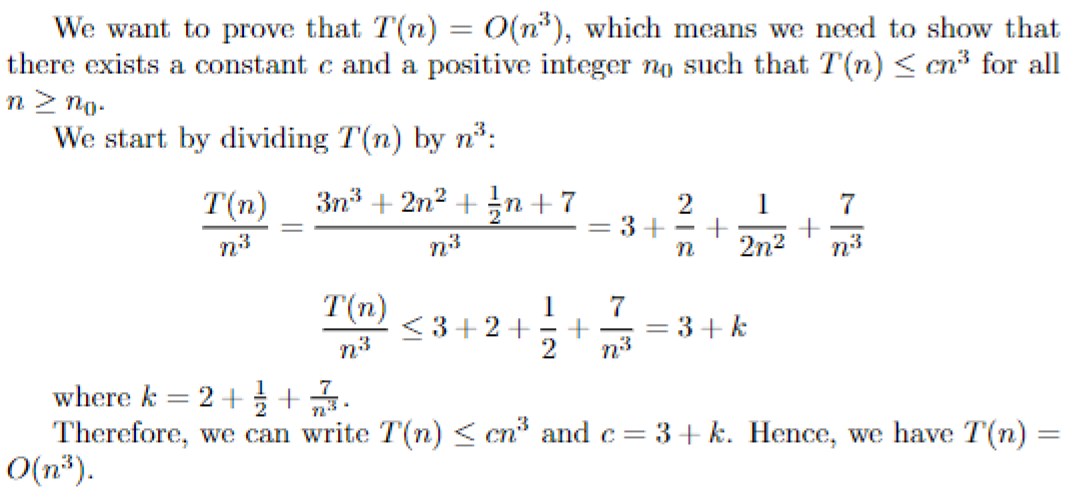
4) Calculate the number of operations executed by each of the programs for different values of n and generalize for any n.

For the first program count_divisors:
The number of operations executed by this program is proportional to n:
n iterations of the loop.
2 operations inside the loop (modulus and comparison).
Therefore, the total number of operations executed by the program is approximately 2n. This is a linear complexity O(n) algorithm.

For the second program count_divisors_opt:
The number of operations executed by this program is proportional to the square root of n:
sqrt(n) iterations of the loop.
3 operations inside the loop (modulus, comparison, and conditional increment).
Therefore, the total number of operations executed by the program is approximately 3*sqrt(n). This is a complexity of O(sqrt(n)).

Big-O notation

1) $T(n) = 3n^3 + 2n^2 + \frac{1}{2}n + 7$ prove that T(n) = O(n³)



We want to prove that $T(n) = O(n^3)$, which means we need to show that there exists a constant c and a positive integer n_0 such that $T(n) \leq cn^3$ for all $n \geq n_0$.

We start by dividing $T(n)$ by n^3:

$$\frac{T(n)}{n^3} = \frac{3n^3 + 2n^2 + \frac{1}{2}n + 7}{n^3} = 3 + \frac{2}{n} + \frac{1}{2n^2} + \frac{7}{n^3}$$

$$\frac{T(n)}{n^3} \leq 3 + 2 + \frac{1}{2} + \frac{7}{n^3} = 3 + k$$

where $k = 2 + \frac{1}{2} + \frac{7}{n^3}$.

Therefore, we can write $T(n) \leq cn^3$ and $c = 3 + k$. Hence, we have $T(n) = O(n^3)$.

2) Prove: ∀ k ≥ 1, n$ is not O?n$%1@

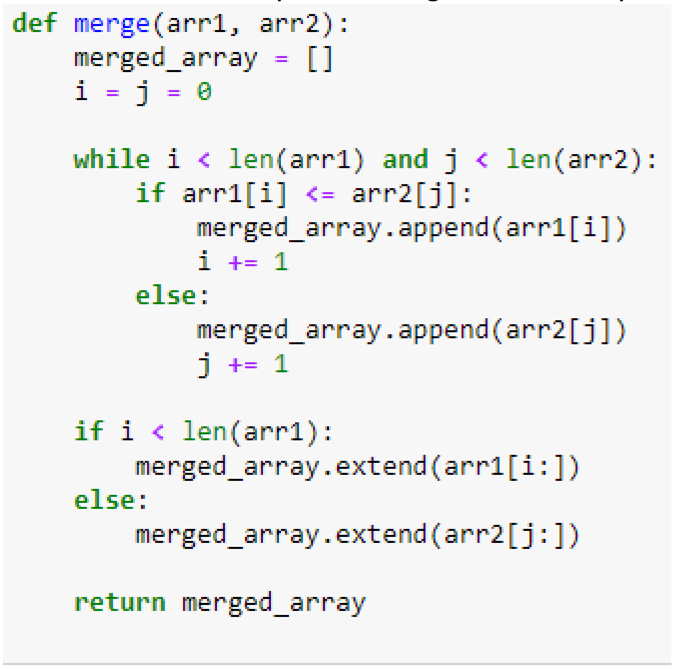
Similar to the above proof, dividing $T(n)$ by $f(n)$, we get:

$$\frac{T(n)}{f(n)} = \frac{n^k}{n^{k-1}} = n$$

Since n is not a constant, then n^k is not $O(n^{k-1})$

Merge sort

1) Given two sorted arrays, write a function (with a language of your choice) that merge the two arrays into a single sorted array.

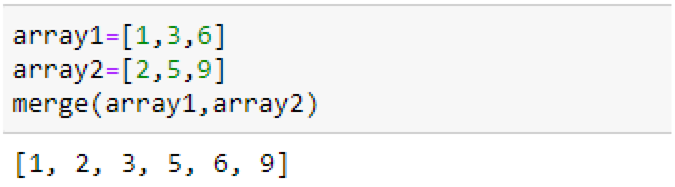


```python
def merge(arr1, arr2):
    merged_array = []
    i = j = 0

    while i < len(arr1) and j < len(arr2):
        if arr1[i] <= arr2[j]:
            merged_array.append(arr1[i])
            i += 1
        else:
            merged_array.append(arr2[j])
            j += 1

    if i < len(arr1):
        merged_array.extend(arr1[i:])
    else:
        merged_array.extend(arr2[j:])

    return merged_array
```



```python
array1=[1,3,6]
array2=[2,5,9]
merge(array1,array2)
```

```
[1, 2, 3, 5, 6, 9]
```

2) Analyse the complexity of your function using Big-O notation.

The time complexity of this algorithm can be analyzed as follows:
The loop that merges the two arrays runs exactly n+m times, where n is the length of the first array and m is the length of the second array. so the total time complexity of this loop is O(n+m).

The master method

1) Using the master method analyse the complexity of merge sort.

The merge sort algorithm can be expressed using the following recurrence relation:

$T(n) = 2T(\frac{n}{2}) + O(n)$

Here, n represents the size of the array to be sorted, and $O(n)$ represents the time taken to merge two sorted arrays of size $\frac{n}{2}$.

We can apply the master method to analyze the time complexity of merge sort:

$a = 2$

$b = 2$

$d = 1$

We have : $a = b^d$ so we are in the first case. Therefore, by the master method, we can conclude that the time complexity of merge sort is $\Theta(n \log n)$.

2) Using the master method analyse the complexity of binary search

The binary search algorithm can be expressed using the following recurrence relation:

$$T(n) = T(n/2) + O(1)$$

where $O(1)$ represents the constant time taken to compare the middle element with the search key.

Now, we can apply the master method with $a = 1$, $b = 2$, and $d = 0$: Since $b^d = a = 1$, first case.

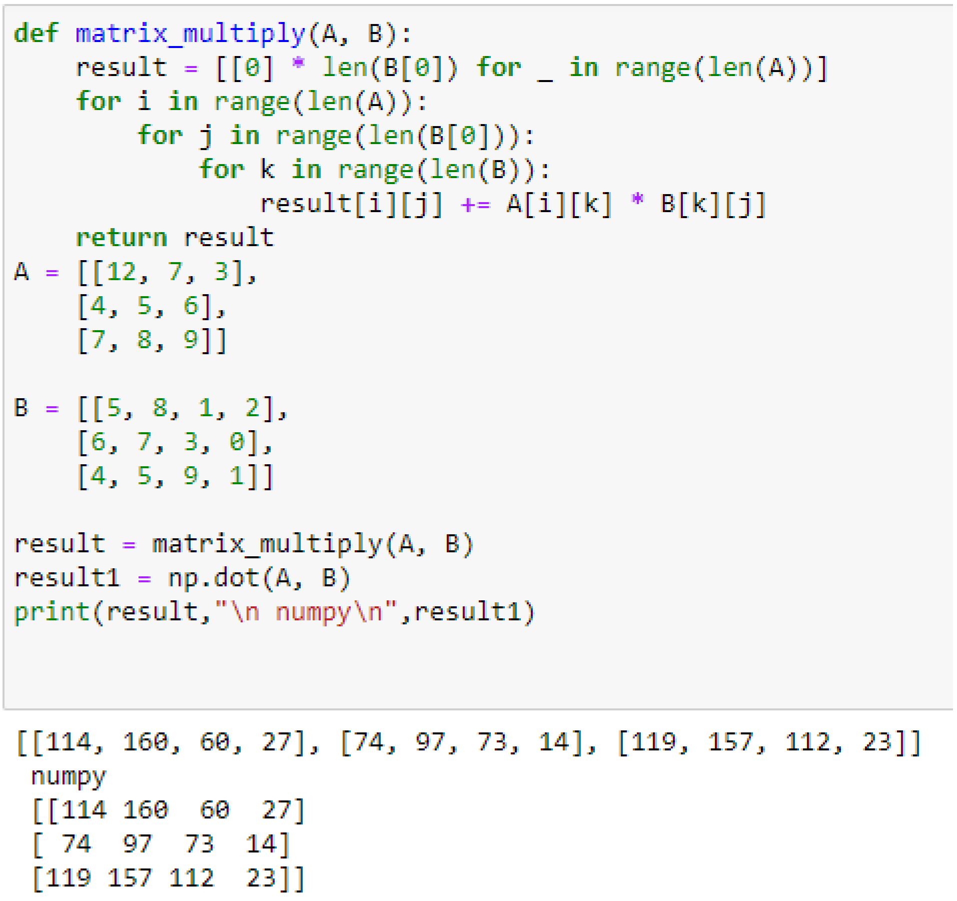
$$\Rightarrow \quad T(n) = \Theta(\log n)$$

Bonus

1) Write a function called merge sort (using a language of your choice) that takes two arrays as parameters and sort those two arrays using the merge sort algorithm.
2) Analyse the complexity of your algorithm without using the master theorem.
3) Prove the 3 cases of the master theorem.
4) Choose an algorithm of your choice and analyse it’s complexity using the Big-O notation.

Matrix multiplication

1) Write a function using python3 that multiply two matrices A,B (without the use of numpy or any external library).



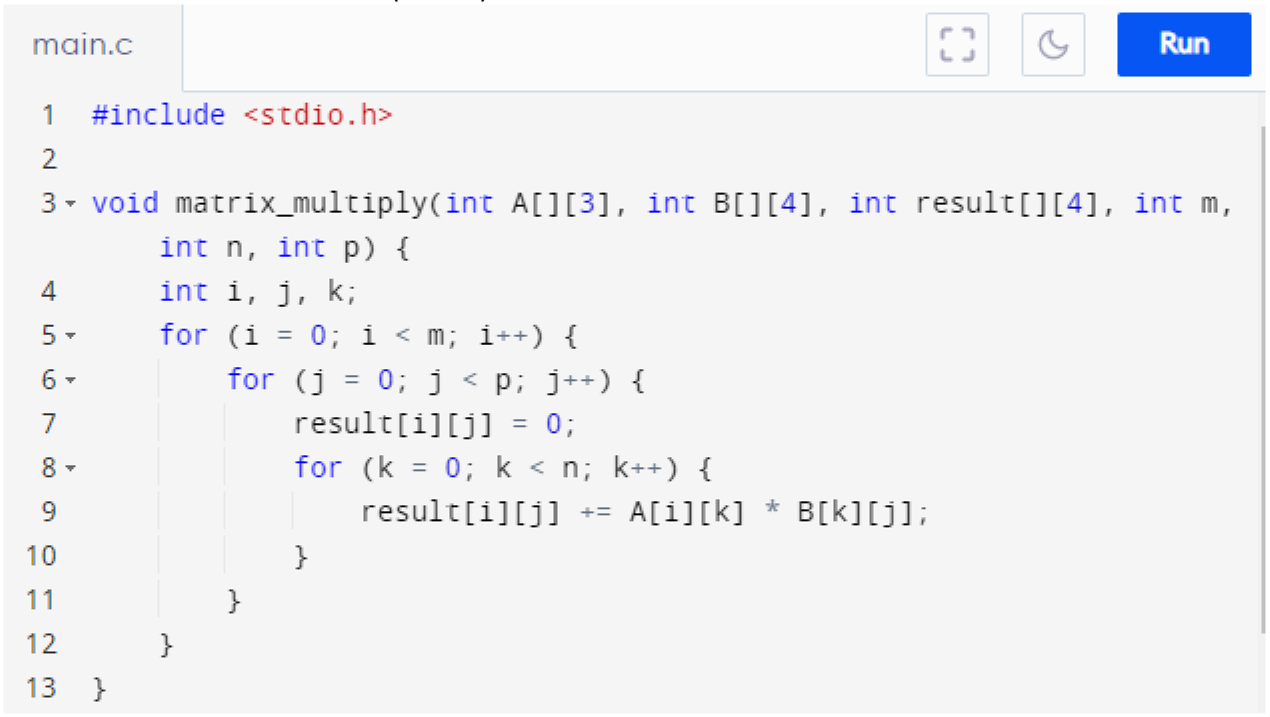
```python
def matrix_multiply(A, B):
    result = [[0] * len(B[0]) for _ in range(len(A))]
    for i in range(len(A)):
        for j in range(len(B[0])):
            for k in range(len(B)):
                result[i][j] += A[i][k] * B[k][j]
    return result
A = [[12, 7, 3],
    [4, 5, 6],
    [7, 8, 9]]

B = [[5, 8, 1, 2],
    [6, 7, 3, 0],
    [4, 5, 9, 1]]

result = matrix_multiply(A, B)
result1 = np.dot(A, B)
print(result,"\n numpy\n",result1)
```

```
[[114, 160, 60, 27], [74, 97, 73, 14], [119, 157, 112, 23]]
 numpy
 [[114 160  60  27]
 [ 74  97  73  14]
 [119 157 112  23]]
```

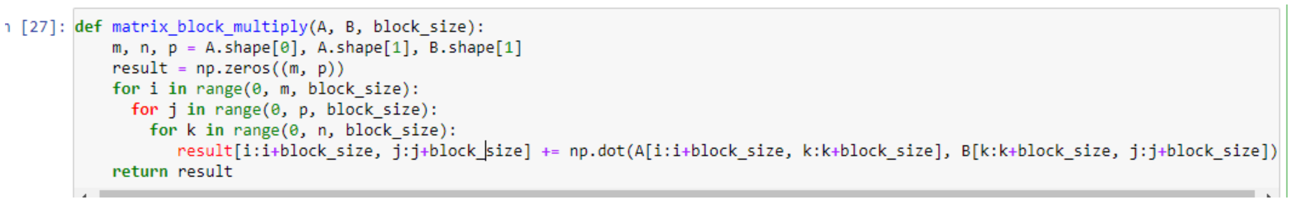
2) What's the complexity of your algorithm (using big-O notation)?
   We have 3 loops, so it's O(n^3)
3) Write the same function in C. (bonus)



```c
#include <stdio.h>

void matrix_multiply(int A[][3], int B[][4], int result[][4], int m,
    int n, int p) {
    int i, j, k;
    for (i = 0; i < m; i++) {
        for (j = 0; j < p; j++) {
            result[i][j] = 0;
            for (k = 0; k < n; k++) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

4) Optimize this multiplication and describe each step of your optimisation.
   Matrix block multiplication

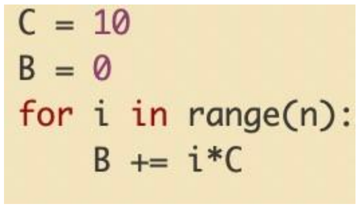


```
n [27]: def matrix_block_multiply(A, B, block_size):
            m, n, p = A.shape[0], A.shape[1], B.shape[1]
            result = np.zeros((m, p))
            for i in range(0, m, block_size):
              for j in range(0, p, block_size):
                for k in range(0, n, block_size):
                   result[i:i+block_size, j:j+block_size] += np.dot(A[i:i+block_size, k:k+block_size], B[k:k+block_size, j:j+block_size])
            return result
```

The optimization of matrix multiplication using block-wise multiplication involves dividing the matrices into smaller submatrices (blocks) and performing the multiplication operation on these blocks. This optimization will increase the chances that the block of numbers will be loaded to the cache memory

Quiz

1) What will be the time complexity for the following fragment of code?



```
C = 10
B = 0
for i in range(n):
    B += i*C
```

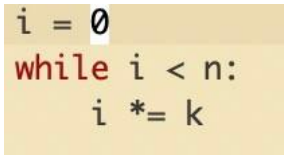
[✓ A) $O(n)$]

B) $O(B)$

C) $O(log_nB)$

D) $O(log_Cn)$

2) What will be the time complexity for the following fragment of code?



```
i = 0
while i < n:
    i *= k
```

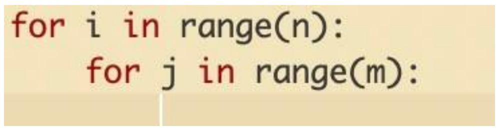
A) $O(n)$

B) $O(k)$

C) $O(log_nK)$

[✓ D) $O(log_kn)$]

3) What will be the time complexity for the following fragment of code?



```
for i in range(n):
    for j in range(m):
```

A) $O(n)$

B) $O(n^2)$

[✓ C) $O(n * m)$]

D) $O(n * \log(n))$