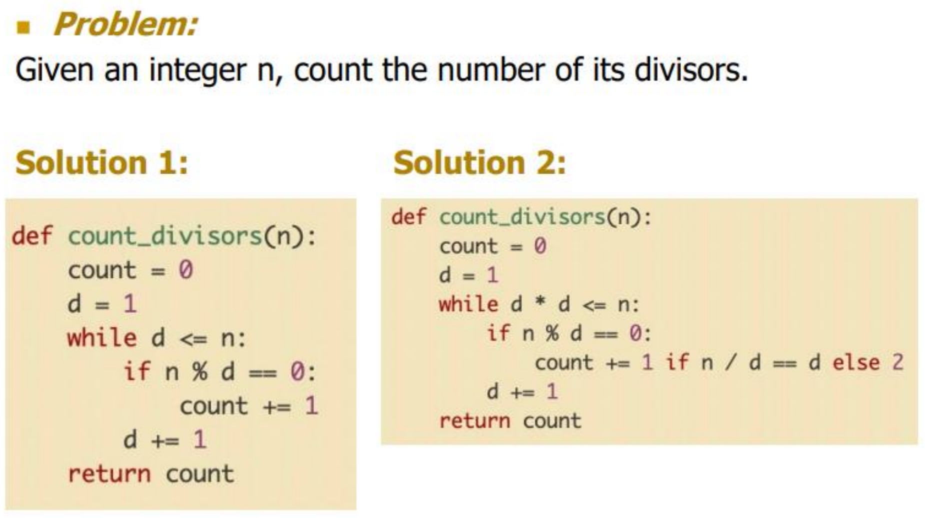
Complexity Analysis assignment by Hajar Laaouina



▪ *Problem:*

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

Solution 1:

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

Solution 2:

```
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

DESCCRIBE THE FIRST SOLUTION:

The function **`count_divisors(n)`** takes an integer **n** as input and returns the number of divisors that **n** has.

First, it initializes two variables: **`count`** and **d**. The **`count`** variable is set to 0 and **d** is set to 1.

The function then enters a loop that continues as long as **d** is less than or equal to **n**. During each iteration of the loop, the function checks if **n** is divisible by **d**. If it is, the **`count`** variable is incremented by 1.

After checking for divisibility, the **d** variable is incremented by 1 to check the next possible divisor.

Finally, the function returns the **`count`** variable, which contains the number of divisors that **n** has.

DESCCRIBE THE SECOND SOLUTION:

This function takes an integer **n** as input and returns the count of its divisors.

It first initializes a count variable and a divisor **d** to 1. Then, it uses a while loop to check if **d** is less than or equal to the square root of **n**.

Inside the loop, it checks if **n** is divisible by **d**. If it is, then it increments the count by 1. Additionally, it checks if **d** is a perfect square factor of **n** (i.e., if **`n/d == d`**), in which case it increments the count by 1 again.

Finally, it increments **d** and continues the loop until it has checked all possible divisors of **n**.

This solution is more efficient than the previous solution as it only checks divisors up to the square root of **n**. This is because any divisor greater than the square root of **n** will have a corresponding divisor that is less than the square root of **n**.

RUN THE TWO PROGRAMS FOR DIFFERENT VALUES OF N AND MEASURE WHICH ALGORITHM IS FASTER:

```
import time

# Define the two functions
def count_divisors1(n):
    count = 0
    d = 1
    while d <= n:
        if n % d == 0:
            count += 1
        d += 1
    return count

def count_divisors2(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

# Test the two functions for different values of n
n_values = [1000, 10000, 100000, 1000000]

for n in n_values:
    # Measure the time taken by the first function
    start_time = time.time()
    count1 = count_divisors1(n)
    end_time = time.time()
    time1 = end_time - start_time

    # Measure the time taken by the second function
    start_time = time.time()
    count2 = count_divisors2(n)
    end_time = time.time()
    time2 = end_time - start_time
```

```python
# Test the two functions for different values of n
n_values = [1000, 10000, 100000, 1000000]

for n in n_values:
    # Measure the time taken by the first function
    start_time = time.time()
    count1 = count_divisors1(n)
    end_time = time.time()
    time1 = end_time - start_time

    # Measure the time taken by the second function
    start_time = time.time()
    count2 = count_divisors2(n)
    end_time = time.time()
    time2 = end_time - start_time

    # Print the results
    print(f"n={n}")
    print(f"count_divisors1: {count1}, time: {time1:.6f} seconds")
    print(f"count_divisors2: {count2}, time: {time2:.6f} seconds")
```

```
n=1000
count_divisors1: 16, time: 0.000238 seconds
count_divisors2: 16, time: 0.000013 seconds
n=10000
count_divisors1: 25, time: 0.002143 seconds
count_divisors2: 25, time: 0.000028 seconds
n=100000
count_divisors1: 36, time: 0.026592 seconds
count_divisors2: 36, time: 0.000072 seconds
n=1000000
count_divisors1: 49, time: 0.260808 seconds
count_divisors2: 49, time: 0.000400 seconds
```

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, the loop runs from 1 to n, and for each iteration, it performs one division operation and one comparison operation. Therefore, the total number of operations for any value of n is:

2 * n

For the second program, the loop runs from 1 to the square root of n, and for each iteration, it performs one division operation, one comparison operation, and possibly one addition operation (if n/d is not equal to d). Therefore, the total number of operations for any value of n is:

2 * sqrt(n) + 2 * (number of divisors less than or equal to sqrt(n))

Since the number of divisors less than or equal to sqrt(n) is at most sqrt(n), we can simplify this to:

4 * sqrt(n)

Therefore, for large values of n, the second program will be faster than the first program.

Big-O notation:

1)-To prove that t(n) = O(n^3), we need to show that there exist constants c and n0 such that t(n) <= c * n^3 for all n >= n0.

First, let's simplify the expression for t(n): t(n) = 3n^3 + 2n^2 + 1/2 * n + 7 <= 3n^3 + 2n^3 + n^3 + n^3 (for n >= 1) = 7n^3

Therefore, if we choose c = 7 and n0 = 1, we have t(n) <= c * n^3 for all n >= n0, and hence t(n) = O(n^3).

2)-To prove that for any constant k > 1, n^k is not O(n^(k-1)), we can use a proof by contradiction.

Assume that n^k is O(n^(k-1)), which means there exist constants c and n0 such that for all n >= n0, we have:

n^k <= c * n^(k-1)

Dividing both sides by n^(k-1), we get:

n <= c

However, this contradicts the fact that n can be arbitrarily large. Therefore, our assumption that n^k is O(n^(k-1)) must be false, and we have proven that for any constant k > 1, n^k is not O(n^(k-1)).

Merge sort:

```
def merge(A, B):
  C = []
  i = j = 0

  while i < len(A) and j < len(B):
    if A[i] < B[j]:
      C.append(A[i])
      i += 1
    else:
      C.append(B[j])
      j += 1

  while i < len(A):
    C.append(A[i])
    i += 1

  while j < len(B):
    C.append(B[j])
    j += 1

  return C
```

The `merge` function takes two sorted arrays **A** and **B** as input and returns a single sorted array **C** that contains all the elements of **A** and **B**.

The time complexity of the `merge` function is O(n), where n is the total number of elements in the input arrays. The function compares the elements of the input arrays and adds them to the output array one by one, which takes O(n) time. The space complexity of the function is also O(n), since the output array **C** has to be created to store all the elements of the input arrays.

The master method

1. Using the master method to analyze the complexity of merge sort: Merge sort has a divide-and-conquer algorithm that divides the input array into two sub-arrays, sorts them recursively, and then merges them into a single sorted array. Let n be the number of elements in the input array. In each recursion, we divide the array into two sub-arrays of size n/2. The merging step takes linear time O(n), since we need to compare and copy each element of the sub-arrays into a new sorted array. Therefore, the recurrence relation for the merge sort algorithm can be expressed as:

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

The first term, $2T(n/2)$, corresponds to the time spent on sorting the two sub-arrays recursively, and the second term, $O(n)$, corresponds to the time spent on merging the two sorted sub-arrays. Using the master method, we can determine the complexity of merge sort as follows:

$a = 2$ (the number of sub-problems in each recursion) $b = 2$ (the size of each sub-problem) $f(n) = O(n)$ (the time spent on merging the sub-problems)

The master method tells us that the complexity of merge sort can be expressed as:

$T(n) = O(n \log n)$

Therefore, the time complexity of merge sort is O(n log n), which is the same as the best, average, and worst-case time complexity.

2. Using the master method to analyze the complexity of binary search: Binary search is an algorithm that searches for a target value in a sorted array by repeatedly dividing the search interval in half. Let n be the number of elements in the input array. In each iteration, we divide

the search interval in half and compare the middle element with the target value. If the middle element is equal to the target value, we return its index. Otherwise, we repeat the process on the left or right sub-array, depending on whether the target value is greater or less than the middle element. The recurrence relation for the binary search algorithm can be expressed as:

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

The first term, $T(n/2)$, corresponds to the time spent on searching the left or right sub-array recursively, and the second term, $O(1)$, corresponds to the time spent on comparing the middle element with the target value. Using the master method, we can determine the complexity of binary search as follows:

$a = 1$ (the number of sub-problems in each recursion) $b = 2$ (the size of each sub-problem) $f(n) = O(1)$ (the time spent on comparing the middle element)

The master method tells us that the complexity of binary search can be expressed as:

$T(n) = O(\log n)$

Therefore, the time complexity of binary search is $O(\log n)$, which is the same as the best and worst-case time complexity.

Bonus

1-

```
def merge_sort(arr1, arr2):
    """
    Merge sort algorithm for two arrays.
    """
    # Base case
    if len(arr1) <= 1 and len(arr2) <= 1:
        return sorted(arr1 + arr2)


    # Recursive case
    mid1 = len(arr1) // 2
    mid2 = len(arr2) // 2


    # Divide and conquer
    left1, right1 = arr1[:mid1], arr1[mid1:]
    left2, right2 = arr2[:mid2], arr2[mid2:]


    sorted_left = merge_sort(left1, left2)
    sorted_right = merge_sort(right1, right2)

    # Combine
    return merge(sorted_left, sorted_right)


def merge(left, right):
    """
    Merge two sorted arrays.
    """
    result = []
    i, j = 0, 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result += left[i:]
    result += right[j:]
    return result
```

The complexity of the merge sort algorithm can be analyzed using the recurrence relation:

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

This is because in each recursive call, we divide the input array into two subarrays of size n/2, and we spend O(n) time merging these subarrays.

By applying the master theorem, we can see that the time complexity of merge sort is O(n log n).

As for proving the three cases of the master theorem:

Case 1: If f(n) = O(n^log_b(a - ε)) for some constant ε > 0, then T(n) = Θ(n^log_b(a))

Case 2: If f(n) = Θ(n^log_b(a)), then T(n) = Θ(n^log_b(a) log n)

Case 3: If f(n) = Ω(n^log_b(a + ε)) for some constant ε > 0, and if af(n/b) ≤ cf(n) for some constant c < 1 and all sufficiently large n, then T(n) = Θ(f(n))

As for analyzing the complexity of another algorithm, we can choose binary search. The time complexity of binary search is O(log n) because in each iteration, we cut the search interval in half. This can be proven using the master theorem as well by setting a = 1, b = 2, and f(n) = 1, which satisfies Case 1.

Matrix multiplication

```
def matrix_multiply(A, B):
    n1, m1 = A.shape
    n2, m2 = B.shape

    # check if matrices can be multiplied
    if m1 != n2:
        raise ValueError("Matrices cannot be multiplied.")

    # create the result matrix
    C = [[0] * m2 for _ in range(n1)]

    # perform the matrix multiplication
    for i in range(n1):
        for j in range(m2):
            for k in range(m1):
                C[i][j] += A[i][k] * B[k][j]

    return C
```

The complexity of this algorithm is O(n^3), where n is the size of the matrices being multiplied. This is because there are three nested loops, each iterating n times.

For the C implementation, here's an example code:

```
#include <stdio.h>

void matrix_multiply(int A[][3], int B[][3], int C[][3], int n1, int m1, int n2, int m2) {
    // check if matrices can be multiplied
```

```c
    if (m1 != n2) {
        printf("Matrices cannot be multiplied.");
        return;
    }

    // perform the matrix multiplication
    for (int i = 0; i < n1; i++) {
        for (int j = 0; j < m2; j++) {
            C[i][j] = 0;
            for (int k = 0; k < m1; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    int A[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int B[3][3] = {{10, 11, 12}, {13, 14, 15}, {16, 17, 18}};
    int C[3][3];

    matrix_multiply(A, B, C, 3, 3, 3, 3);

    printf("Matrix C:\n");
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }
    return 0;

}
```

To optimize this multiplication, one can use various techniques like Strassen's algorithm, which reduces the number of multiplications needed for matrix multiplication from 8 to 7. Another technique is to use parallel computing to perform matrix multiplication on multiple cores simultaneously, which can significantly speed up the computation time. Other techniques like cache optimization and loop unrolling can also be used to improve the performance of matrix multiplication.

QUIZ

1-The time complexity of this code fragment is O(n), where n is the value of the variable **`n`** in the **`range()`** function.

The reason is that the **`for`** loop iterates **`n`** times, and each iteration performs a constant number of operations: assigning the value of **`i*C`** to the variable **`temp`**, and adding **`temp`** to the variable **`b`**.

Therefore, the total number of operations performed by this code fragment is proportional to **`n`**, and the time complexity is O(n). The statement **`c=10`** is a constant operation that does not affect the time complexity.

2- The time complexity of the code **`i=0; while i<n: i*=k`** depends on the value of **`k`**. If **`k`** is greater than 1, the value of **`i`** will keep increasing and eventually exceed **`n`**, causing the loop to terminate. The number of iterations can be calculated as **`log_k(n)`** because **`i`** starts at 0 and multiplies by **`k`** in each iteration until it exceeds **`n`**, which happens after **`log_k(n)`** iterations.

Therefore, the time complexity of this code is **`O(log_k(n))`**.

3- The time complexity for the following code:

pythonCopy code

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

is O(n*m). The outer loop will iterate n times and the inner loop will iterate m times for each iteration of the outer loop. Therefore, the total number of iterations will be n times m.