Student: Brahim MOKNASSI

Class: QFM Master

University Mohamed 6 Polytechnic

QFM Master

Week 1 Assignment

Semester II Academic Year 2022/2023

Subject:	**Numerical Linear Algebra & Parallel Computing**
Date:	28/03/2023
Time of Deadline:	31/03/2023

Problem1: Introduction

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
```

solution:

1-Solution 1: This function uses a while loop to iterate through all the possible divisors of the input integer n. The variable count keeps track of the number of divisors found so far, and d is used as the divisor in each iteration. If n is evenly divisible by d, then the count is incremented by 1. Finally, the function returns the count of divisors found.

2-Solution 2: This function is similar to solution 1, but it only iterates through the divisors up to the square root of n. This is because any divisors greater than the square root of n will have corresponding divisors less than the square root of n. The variable count keeps track of the number of divisors found so far, and d is used as the divisor in each iteration. If n is evenly divisible by d, then the count is incremented by 1. Additionally, if d is a divisor, then $\frac{n}{d}$ is also a divisor, so the count is incremented by 2 if $\frac{n}{d}$ is not equal to d. Finally, the function returns the count of divisors found.

3-To measure the speed of the two algorithms, you can use the time module in Python to time how long it takes to run each function for different values of n. Here's an example:

```
import time

n = 1000000

start_time = time.time()
count_divisors1(n)  # call solution 1
end_time = time.time()
print(f"Solution 1 took {end_time - start_time} seconds to run for n = {n}")

start_time = time.time()
count_divisors2(n)  # call solution 2
end_time = time.time()
print(f"Solution 2 took {end_time - start_time} seconds to run for n = {n}")
```

```
Solution 1 took 0.17098140716552734 seconds to run for n = 1000000
Solution 2 took 0.0 seconds to run for n = 1000000
```

4-To calculate the number of operations executed by each program, we need to count the number of times the while loop is executed for each function. For solution 1, the while loop runs n times,

while for solution 2, the while loop runs $\sqrt{n}$ times. Therefore, we can say that solution 1 has O(n) time complexity, while solution 2 has O($\sqrt{n}$) time complexity.

Problem2: Big-O notation

1) $T(n) = 3n^3 + 2n^2 + \frac{1}{2}n + 7$, Prove that: $T(n) = O(n^3)$.
2) Prove: $\forall k, n^k$ is not $O(n^{k-1})$.

solution:

1) For $n \geq 1$, we have:

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

Thus :

$$T(n) \leq \frac{15}{2}n^3 + 7$$

For $c = \frac{16}{2}$ and for all $n \geq 1$ we have:

$$T(n) \leq \frac{15}{2}n^3 + 7 \leq cn^3$$

Then we got :

$$T(n) = O(n^3)$$

2) By contradiction, we assume that $\forall k, n^k isO(n^k 1)$. Then : $\exists n_0$ such that $\forall n \geq n_0$, $n^k \leq c.n^{k-1}$. Dividing both sides by n^{k1} , we get $n \leq c$ for all $n \geq n_0$. But this is a contradiction, since n can be arbitrarily large and c is a constant., Therefore, we have shown that: $\forall k, n^k$ is not $O(n^{k-1})$

Problem3: 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.

Ex: def merge(A,B):
...
...
return C

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

solution:
1)

```
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
    if i < len(A):
        C.extend(A[i:])
    else:
        C.extend(B[j:])
    return C
```

```
A=np.array([12,3,5])
B=np.array([37,11,12,3,5])
```

```
merge(A,B)
```

```
]: [12, 3, 5, 37, 11, 12, 3, 5]
```

2) The time complexity of this implementation is $O(n)$, where n is the total number of elements in A and B. This is because the while loop iterates through all the elements in both A and B exactly once.

Problem4: The master method

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

solution:

1) The time complexity of merge sort can be analyzed using the master method, which is a method for solving recurrences of the form:

$$T(n) = a \times T(\frac{n}{b}) + f(n)$$

where a is the number of subproblems, $\frac{n}{b}$ is the size of each subproblem, and $f(n)$ is the time complexity of the work done at the current level of the recursion.

For merge sort, we have $a = 2$ because we recursively split the array in two halves, and $f(n) = O(n)$ because we need to merge the two sorted halves. The size of each subproblem is $\frac{n}{2}$, so b = 2. Therefore, the recurrence relation for merge sort is:

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

According to the master method, the time complexity of this recurrence relation is $\mathbf{O}(n \log n)$, which means that merge sort has a worst-case time complexity of $\mathbf{O}(n \log n)$.

2) The time complexity of binary search can also be analyzed using the master method. The recurrence relation for binary search is:

$$T(n) = T(\frac{n}{2}) + O(1)$$

where $T(\frac{n}{2})$ represents the time complexity of the recursive call on a subproblem of size $\frac{n}{2}$, and $O(1)$ represents the time complexity of the work done at the current level of the recursion, which is just one comparison.

According to the master method, the time complexity of this recurrence relation is $O(\log n)$, which means that binary search has a worst-case time complexity of $O(\log n)$.

Problem5: 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.

solution:

1)

```
def merge_sort(arr1, arr2):


    sorted_array = []
    i, j = 0, 0

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

    while i < len(arr1):
        sorted_array.append(arr1[i])
        i += 1

    while j < len(arr2):
        sorted_array.append(arr2[j])
        j += 1

    return sorted_array
```

```
arr1 = [1, 3, 5, 7]
arr2 = [2, 4, 6, 8]
sorted_array = merge_sort(arr1, arr2)
print(sorted_array)
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

2) The time complexity of the merge sort algorithm is $O(n \log n)$, where n is the total number of elements in the input arrays. This can be analyzed as follows:

¿ The divide step takes $O(1)$ time.

¿ The conquer step involves recursively calling the merge_sort function on two halves of the input arrays. The total number of recursive calls is $\log(n)$, since each time we divide the input array in half. At each level of the recursion tree, we need to merge two sorted arrays, which takes $O(n)$ time. Therefore, the total time complexity of the conquer step is $O(n \log n)$.

¿ The merge step is done in linear time $O(n)$.

Overall, the time complexity of merge sort is O(n*log(n)).

3) The three cases of the master theorem are as follows:

Case 1: If a problem of size n is divided into a subproblems of size $\frac{n}{b}$, each subproblem is solved recursively in time $T(\frac{n}{b})$, and the combined solutions are computed in time $f(n)$, then the time complexity of the algorithm can be expressed as $T(n) = a \times T(\frac{n}{b}) + f(n)$, where $a \geq 1$ and $b > 1$. If $f(n) = O(n^c)$ for some constant $c < log_b(a)$, then $T(n) = Theta(n^log_b(a))$.

Case 2: : If a problem of size n is divided into a subproblems of size n/b, each subproblem is solved recursively in time T(n/b), and the combined solutions are computed in time $f(n)$, then the time complexity of the algorithm can be expressed as $T(n) = a \times T(\frac{n}{b}) + f(n)$, where $a \geq 1$ and $b > 1$. If $f(n) = \Theta(n^c \times log^k(n))$ for some constants $c = log_b(a)$ and $k \geq 0$, then $T(n) = \Theta(n^c \times log^(k+1)(n))$.

Case 3: If a problem of size n is divided into a subproblems of size $\frac{n}{b}$, each subproblem is solved recursively in time $T(\frac{n}{b})$, and the combined solutions are computed in time $f(n)$, then the time complexity of the algorithm can be expressed as $T(n) = aT(\frac{n}{b}) + f(n)$, where $a \geq 1$ and $b > 1$. If $f(n) = \Omega(n^c)$ for some constant $c > log_b(a)$, and if $af(\frac{n}{b}) \leq c \times f(n)$ for sufficiently large n, then $T(n) = \Theta(f(n))$.

Problem6: Matrix multiplication

1) Write a function using python3 that mul.ply two matrices A,B (without the use of numpy or any external library).
2) What's the complexity of your algorithm (using big-O nota.on)?

3) Write the same function in C. (bonus)
4) Optimize this multiplication and describe each step of your optimisation.

solution:

1)

```python
def matrix_multiplication(A, B):
    m, n = len(A), len(B[0])
    result = [[0] * n for _ in range(m)]
    for i in range(m):
        for j in range(n):
            for k in range(len(B)):
                result[i][j] += A[i][k] * B[k][j]
    return result
```

```python
A = [[1, 3, 5, 7],[0, 4, 5, 9]]
B = [[2, 4],[ 6, 8],[0, 6],[ 6, 8]]
sorted_array = matrix_multiplication(A, B)
print(sorted_array)
```

```
[[62, 114], [78, 134]]
```

2) The time complexity of this algorithm is $O(n^3)$ where n is the size of the matrices. This is because the function uses three nested loops to multiply the matrices.

3)

```c
#include <stdio.h>

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


    return 0;
}
```

Problem6: Quiz

solution:

1)	2)	3)
A	D	C