Numerical Linear Algebra & parallel computing:

Complexity analysis:

I Introduction:

1) Describing solution 1:

the program initializes a variable named 'count' to zero and the divisor to 1, there is a loop that checks if "n" is divisible by "d" as long as "d" is less than "n" then we increment count by one and we increment the divisor and we repeat the loop until we find "$d \geq n$" then we return count that is the number of the divisors of n.

2) Describing solution 2:

the same for the first solution, just the second one executes for all d such that $d^2 < n$ so the loop executes $\sqrt{n}$ times inside the loop, the program performs one modulo operation and a single division operation for each value of d

3) Comparing the two programs

we can clearly see that the second algorithm is faster than the first one (the program in file.py)

4) the number of operations executed by each programs.

it's showed in the (.py file).

Laghrissi Ilham QFM1.

Big-O Notation.

1) Proving that $T(n) = O(n^3)$

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

So: $T(n) \leqslant 3n^3 + 2n^3 + \frac{1}{2}n^3 + 7n^3$

So, $T(n) \leqslant 12{,}5\, n^3$ so $T(n) = O(n^3)$. otherwise :

$T(n) = n^3\left(3 + \frac{2}{n} + \frac{1}{2}\cdot\frac{1}{n^2} + \frac{7}{n^3}\right)$ $\lim_{n \to +\infty}\left(3 + \frac{2}{n} + \frac{n}{2n^2} + \frac{7}{n^3}\right) = 3$

$\lim_{n \to +\infty} T(n) = \lim_{n \to +\infty} 3n^3$ so $T(n) = O(n^3)$.

Proving that n^k is not $O(n^{k-1})$

Let's suppose that n^k is an $O(n^{k-1})$ so $\exists c = cte$ such that :

$n^k = c \cdot n^{k-1}$ so $\frac{n^k}{n^{k-1}} = c$ so $n = c$ so n is constante.

what is contradiction, so n^k is not $O(n^{k-1})$.

Merge sort :

1)
```
def merge(A, B)
    C = []
    i = 0
    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
    C += A[i:]
    C += B[j:]
    return C.
```

2) Analysing the complexity of merge function using Big-O notation.

my function takes as arguments two sorted arrays, it create a new array which combines the two sequences in sorted order. Let n be the size of the first array A and let m be the size of the second array B so, in all over the algorithm it happens $O(n+m)$ iterations, Hence the time complexity of this algorithm is $O(n+m)$.

The master method:

1) Analysing the complexity of merge sort

If n is the size of the input array and $T(n)$ is the running time we have:

$T(n) = 2 \cdot T(\frac{n}{2}) + O(n)$ by the master method ($T(n) \leq aT(\frac{n}{b}) + O(n^d)$)

a is the number of the subarrays that comes out of the division, in this merge sort function (if the two arrays are not sorted) we divide the original array to 2 parts every time $\Rightarrow a=2$.

$\frac{n}{b}$ is the size of each problem (sub array), $\frac{n}{2}$ is the size of each subarray, so $b=2$ so $a=b=2$.

in the merge step we will have two arrays each of them with a size of $\frac{n}{2}$ so the complexity of it is $O(\frac{n}{2}+\frac{n}{2})$ so it's $O(n)$ that's why we have $T(n) = 2 \cdot T(\frac{n}{2}) + O(n)$

$a=b=2$, $d=1$ the the time complexity of merge-sort is $O(n \log(n))$

2) Analysing the complexity of Binary search:

In this case $a=1$ and $b=2$ so $T(n) = T(\frac{n}{2}) + O(1)$, $d=0$

$a = 1 = b^0$ so $T(n) = O(n^0 (\log(n))^1) = O(\log(n))$.

(the first case of the master method).

Bonus :

1) Merge_sort function : (C Language)

```c
# include < stdio.h>
# include <stdlib.h>
Void merge_sort ( int A[], int n , int B[], int m, int C[]){
        int i=0, j=0, k=0;
       while ( i <= n && j <= m) {
           if ( A[i] <= B[j]) {
               C [k++] = A [i++]; }
           else {
                C[k++] = B [j++]; } }
       while (i < n) {
              C [k++]= A[i++];}
       while  (j < m) {
             C [k++] = B[j++]; }
```

2) Analysing the complexity

the algorithm follows the divide and conquer approach that results in a logarithmic number of levels in the recursion tree, each of them with $O(n)$ and we saw earlier that the merge function takes $O(n)$ so the complexity of the merge_sort algorithm is $O(n \log(n))$

3) the three cases of the master theorem

We know that the work for a level $j \leq C \cdot n^d \cdot \left(\frac{a}{b^d}\right)^j$

and The total work $\leq C \cdot n^d \sum_{j=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^j$

case 1 :

if $a = b^d$ so :

$$cn^d \cdot \sum_{j=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^j = c \cdot n^d \cdot \sum_{j=0}^{\log_b(n)} \left(\frac{b^d}{b^d}\right)^j = c \cdot n^d \sum_{j=0}^{\log_b(n)} 1$$

$$= cn^d(\log_b(n) + 1) = c(n^d \log_b(n)) + cn^d$$

so it's an $O(n^d \log(n))$

Case 2: if $a < b^d \Rightarrow \frac{a}{b^d} < 1$. Let be $q = \frac{a}{b^d}$

$$c.n^d \sum_{j=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^j = c.n^d \sum_{j=0}^{\log_b(n)} q^j$$ (Suite geometrique de raison $q<1$)

$$\leq c n^d . \frac{1}{1-q} = c.n^d.k$$ Let be $k = \frac{1}{1-q}$ = cte

So the complexity is an $O(n^d)$ because $c.k$ = constant

Case 3: if $a > b^d \Rightarrow \frac{a}{b^d} > 1 \Rightarrow q > 1$.

$$c.n^d . \sum_{j=0}^{\log_b(n)} q^j = c.n^d . q^{\log_b(n)} = c.n^d . n^{\log_b(q)}$$

$$= c.n^d . n^{\log_b\left(\frac{a}{b^d}\right)} = c.n^d . n^{\log_b(a) - d\log_b(b)}$$

$$= c . n^{d + \log_b(a) - d}$$

$$= c\ n^{\log_b(a)}$$

So the complexity is $O(n^{\log_b(a)})$.

4) I wrote an algorithm that checks if a number is prime or not:

The algorithme :

```
~~#printf ("Please enter a number");~~
number = int ( input ("Please enter a number :"))
if number == 1 :
    Print (number, "is not a prime number")
elif number > 1 :
    for i in range (2, number) :
        if (number % i)==0 :
            print ( number , "is not a prime number")
            print (i, "is divisor of the number", number).
            break
    else :
        print (number , "is a prime number")
else : # if number <1 "
    print (number, is not prime").
```

Matrix multiplication:

1) Multiplying two matrices A and B:

```
def matrix_multiplication(A,B):
    ligne_A = len(A)
    Col_A = len (A[0])
    ligne_B = len (B)
    Col_B = len (B[0])
    # We have to check if this product is possible or not
    if col_A!= ligne_B :
        print (" Error: the matrices cannot be multiplied")
        return None.
  # if the product is possible, let's initialize the result matrix
    by zeros without np.zeros (numpy).
  C= [[0 for _ in range(col_B)] for _ in range(ligne_A)]
  for i in range (ligne_A) :
      for j in range (col_B) :
          for k in range (col_A):
              C[i][j] += A[i][k] * B[k][j]

  return C.
```

2) the complexity:

if we assume that n is the size of the matrix (square matrix) then the complexity is $O(n^3)$ because I used in this function matrix_multiplication nested loops to compute the product which has time complexity of $O(n^3)$.

3) Multiplication of two matrices in C:

```
# include <stdio.h>
# include <stdlib.h>
int** matrix_multiplication (int** A, int** B, int ligneA,
                             int colA, int ligne_B, int col_B) {
    if (col_A != ligne_B) {
        printf ("the product is impossibl"),
        return Null; }
// ## now if the product is possible, we have to
      allocate memory for the result matrix. //
    int** C = (int**) malloc (ligne_A * sizeof (int*));
    for (int i=0 ; i < ligne_A ; i++) {
        for (int j=0; j < col_B; j++) {
            C[i][j] = 0;
            for (int k=0 ; k < col_A; k++) {
                C[i][j] + = A[i][k] * B[k][j] }
        }
    }
    return C;
}
```

4) optimization of the program:

- we can use directly calloc instead malloc that will initialize directly the memory to Zero by default. to ignore the loop that sets each element of C to zero.
- Replace array indexing by pointer to access to the elements of ma

```
int** C = (int**) malloc (ligne_A * sizeof (int*));
for (int i=0; i < ligne_A ; i++) {
    C[i] = (int*) malloc (col_B * sizeof (int)); }
```

Quiz :

1) A : $\Theta(n)$
2) D : $\Theta(\log_k(n))$
3) C : $o(n * m)$