Lab Assignment #1

Analysis of Algorithm

Name: Bhavyai Gupta

UCID: 30143691

# Response to Question 1

The Big-O time complexity of the algorithm A is O(n2).

Explanation:

Highest order term in the given expression for number of steps 3n2. After discarding the multiplicative constants, the order of growth is quadratic.

# Response to Question 2

Order of functions arranged from most efficient (number 1) to least efficient (number 6) –

1. Constant
2. Logarithmic
3. Linear
4. Quadratic
5. Cubic
6. Exponential

# Response to Question 3

The Big-O time complexity for the code fragment is O(n2).

Explanation:

We ignore the constant-time statement int test = 0;. The outer for loop executes **n** times. The nested for loop executes **n** times for each iteration of the outer loop, making its total number of iterations equal to **n\*n**. Thus, the statement test = test + i \* j; is executed **n2** number of times, making the Big-O running time as **O(n2)**.

# Response to Question 4

The Big-O time complexity for the code fragment is O(n).

Explanation:

We ignore the constant-time statements int test = 0; and return 0; The first for-loop executes **n** number of times. Same is the case for the second for-loop. Since the loops are executed sequentially (one after other), the order of growth remains **n**, hence the time complexity is O(n).

# Response to Question 5

The Big-O time complexity for the code fragment is O(log(n)).

Explanation:

We ignore the constant-time statements int i = n; and int count = 0; and return 0;.

Let us determine how many times our while-loop executes. Constraints on **i** are:

* Initial value of **i** is **n**
* Value of **i** is halved during each iteration
* The loop iterates till **i**>0

So, value of **i** at the time of condition check on each iteration is –

n, n/2, n/22, n/23, n/24, …, n/2k where k+1 is the number of times the while loop executes.

The loop executes as long as n/2k > 0.

* n ≥ 2K (using integer division rules)
* log(n) ≥ k
* log(n)+1 ≥ k+1

This means our code fragment grows logarithmically with the value of **n**. Hence, time complexity is O(log(n)).

# Response to Question 6

Below is a code fragment with time complexity O(n3).

/\*\*

 \* Function to create and print a cube of side n

 \*

 \* @param n the length of the side of the cube

 \*/

void function(int n) {

    int side = n;

    int cube[][][] = new int[side][side][side];

    // loop to initialize the cube with natural numbers

    for (int i = 0; i < cube.length; i++) {

        for (int j = 0; j < cube[i].length; j++) {

            for (int k = 0; k < cube[i][j].length; k++) {

                cube[i][j][k] = (k + 1) + j \* cube[i][j].length + i \* cube[i].length \* cube[i][j].length;

            }

        }

    }

    // loop to print the cube

    for (int i = 0; i < cube.length; i++) {

        for (int j = 0; j < cube[i].length; j++) {

            for (int k = 0; k < cube[i][j].length; k++) {

                System.out.printf("%03d  ", cube[i][j][k]);

            }

            System.out.println();

        }

        System.out.println();

    }

}

# Response to Question 7

In the worst-case scenario, our algorithm performing at O(n2) will run around 72 times, ie, around 49 times, for an input of integer 7. The actual number of operations will depend upon T(n), which is not provided in the question.

# Response to Question 8

The Big-O time complexity for the function **isLeapYear** is O(1).

# Response to Question 9

The Big-O time complexity for the function **chessboardSpace** is O(log(n)).

Explanation:

We ignore the constant-time statements chessboardSpaces = 1; and placedGrains = 1;

Let us determine how many times our while-loop executes. The condition for the while-loop is:

* **placedGrains** < **numberOfGrains**
* **placedGrains** is doubled in every iteration.

So, value of **placedGrains** at the time of condition check on each iteration is –

1, 2, 22, 23, 24, …, 2k where k+1 is the number of times the while loop executes

The loop executes as long as 2k < numberOfGrains.

* numberOfGrains > 2K
* log(numberOfGrains) > k
* log(n)+1 > k+1

This means our code executes almost **log(numberOfGrains)** times. Hence, time complexity is **O(log(n))**.

# Response to Question 10

Calculating the primitive operations of every line -

i = 1;                   // 1 op

sum = 0;                 // 1 op

while (i <= n) {            // 1 op  \* (n+1)

    i = i + 1;              // 2 ops \* (n)

    sum = sum + i;          // 2 ops \* (n)

}

T(n) = 1 + 1 + 1\*(n+1) + 2n + 2n

= 2 + n + 1 + 4n

= 5n + 3

Ignoring the lower order terms and ignoring the multiplicative constant of the highest order term, the Big-O time complexity of the given code is O(n). This is **linear** growth rate.

# Response to Question 11

f(n) = 3nlog(n) - 2n

We must find g(n) such that –

f(n) ≥ cg(n) for n ≥ n0, for a real constant c > 0 and integer constant n0 ≥ 1

Taking **c** as one less than the multiplicative constant of the highest order term in f(n),

c = 3 - 1 = 2

Now, we find n0 such that -

3n0log(n0) - 2n0 = 2 n0log(n0)

* n0log(n0) = 2n0
* log(n0) = 2
* n0 = 100
* f(n) is greater than g(n) = 2nlog(n), for n ≥ 100

So, for Big-Ω notation, we drop multiplicative constants. Hence, Big-Ω is **Ω(nlog(n))**.