QUESTÃO 1

Ruby: Bubble Sort -> O(n²)

Cálculo:

O(n) * O(n) = O(n²), pois possui dois loops aninhados que percorrem todo o array.

```
def bubble_sort(array)
  array_length = array.size
  return array if array_length <= 1

  loop do
    # we need to create a variable that will be checked so that we don't run into an infinite loop
scenario.
    swapped = false

      (array_length-1).times do |i|
      if array[i] > array[i+1]
        array[i], array[i+1] = array[i+1], array[i]
        swapped = true
      end
    end

    break if not swapped
  end

  array
end

unsorted_array = [11,5,7,6,15]
p bubble_sort(unsorted_array)
```

C#: Acessando a primeira posição do vetor -> O(1)

Cálculo:

O acesso à posição desejada se dá de maneira imediata, não importando o tamanho de entrada do vetor, conclui-se que o pior caso é O(1) pois o tempo tomado é de 1 unidade de tempo.

```
using System;
using System.Linq;

public class Program
{
	public static void Main()
	{
			int[] numeros = new int[5];
			numeros[0] = 2;
			numeros[1] = 4;
```

```
            numeros[2] = 15;
            numeros[3] = 11;
            numeros[4] = 2; // meu vetor acaba aqui.

            Console.WriteLine("Eu sou o número da posição 0: " + numeros[0]);
        }
    }
```

Swift: Busca binária -> O(log n)

Cálculo:

A cada iteração do loop, o vetor de tamanho n é dividido em 2. Ou seja, a cada passo, nós temos um tamanho de n, n/2, n/4 e assim por diante, sendo o pior caso até chegar no número 1. Então considerando ‘k’ sendo o número de passos até chegar em 1, 2^k é maior ou igual a n, e pela definição de logaritmo, k = log n na base 2.

```
public fun binarySearch<T: Comparable>(_ a: [T], key: T) -> Int? {
    var lowerBound = 0
    var upperBound = a.count
    while lowerBound < upperBound {
        let midIndex = lowerBound + (upperBound - lowerBound) / 2
        if a [midIndex] == key {
            return midIndex
        } else if a[midIndex] < key {
            lowerBound = midIndex + 1
        } else {
            upperBound = midIndex
        }
    }
    return nil
}
```

Python: Heap Sort -> O(n * log n)

Cálculo:

A altura de uma Heap é de tamanho logn, sendo assim, o método Heapify, no seu pior caso, onde o nó raiz deverá descer toda a altura da árvore, serão feitos logn processos, fazendo sua complexidade ser O(logn).

Visto isso, o pior caso do método heapSort é quando o vetor está em tamanho decrescente, fazendo com que seja necessário fazer n vezes o método heapify. A conta então seria n*logn, resultando na complexidade O(n*logn).

```
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i  + 2

    if l < n and arr[i] < arr[l]:
```

```
            largest = l

    if r < n and arr[largest] < arr[r]:
            largest = r

    if largest != i:
            arr[i], arr[largest] = arr[largest], arr[i]
            heapify(arr, n, largest)

def heapSort(arr):
    n = len(arr)

    for i in range(n, -1, -1):
            heapify(arr, n, i)

    for i in range(n-1, 0, -1):
            arr[i], arr[0] = arr[0], arr[i]
            heapify(arr, i ,0)
```

Kotlin: Calculando tamanho do array -> O(n)

Cálculo:

No seguinte método, é percorrido todo o vetor para definir o seu tamanho. Assim, o pior caso é O(n), pois é necessário percorrer todo n tamanho de entrada do vetor.

```
fun main(args: Array<String>) {
   val arr = arrayOf(2, 4, 6, 8, 10)
   val length = 0

   for (item in arr) {
      length = length + 1
   }
   println("Array size : $length")
}
```

Rust: Busca linear -> O(n)

Cálculo:

No método abaixo, é percorrido todo o vetor até ser achado o valor desejado. Dessa maneira, o pior caso é quando o valor alvo está na última posição do vetor, sendo necessário percorrer todo n tamanho de entrada, resultando na complexidade O(n).

```
fn linear_search(list: Vec<i32>, target: i32) -> Option<usize> {
    for i in 0..list.len() {
            if list[i] == target {
                    return Some(i)
            }
```

```
        }
        return None;
}
```

Scala: Trocando dois valores -> O(1)

Cálculo:

Como não é utilizado nenhuma repetição e as variáveis estão sendo acessadas diretamente, o tempo é constante, sendo O(1)

```scala
object Swap {

  def swap(x: Int, y: Int) = {
    (y,x)
  }

  def main(args: Array[String]): Unit = {
    var (x, y) = (10,6)
    var swapped = swap(x,y)
    x = swapped._1
    y = swapped._2
    println(x, y)
  }

}
```

Java: Quick Sort -> O(n²)

Cálculo:

A complexidade do Quick Sort em java é $O(n^2)$ pois o método Quick Sort pega cada elemento do vetor, que ficam conhecidos como pivôs, e ordena os elementos maiores e menores em relação à ele. O algoritmo repete esse processo até o vetor ficar ordenado, e em seu pior caso, todos os elementos do vetor seriam utilizados como pivôs, chegando na complexidade $O(n^2)$.

```java
public class QuickSort {
  public static void sort(int[] a) {
    sort(a, 0, a.length - 1);
  }

  public static void sort(int[] a, int low, int high) {
    if (low >= high) return;

    int middle = partition(a, low, high);
    sort(a, low, middle - 1);
    sort(a, middle + 1, high);
  }

  private static int partition(int[] a, int low, int high) {
    int middle = low + (high - low) / 2;
```

```
        swap(a, middle, high);
        int storeIndex = low;
        for (int i = low; i < high; i++) {
            if (a[i] < a[high]) {
                swap(a, storeIndex, i);
                storeIndex++;
            }
        }
        swap(a, high, storeIndex);
        return storeIndex;
    }

    private static void swap(int[] a, int i, int j) {
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}
```

C++: Cálculo fatorial -> O(n)

Cálculo:

Utiliza recursividade, chamando a função ‘n’ vezes e retornando o valor fatorial de ‘n’.

```
int factorialRecurrent(unsigned long InputValue)
{
	if (InputValue == 0) {

		return(1);
	}

	else {

		return(InputValue * factorialRecurrent(InputValue - 1));
	}
}
```

Go: Kadanes -> O(n)

Cálculo:

O algoritmo de Kadanes serve para achar o sub-vetor de um vetor cuja soma de seus elementos é a maior possível dentro de todos os outros sub-vetores. Para fazer isso, ele percorre o vetor com um for do começo ao fim utilizando funções de máximo para conseguir obter o sub-vetor com a maior soma. Complexidade = O(n)

```
package main

import (
	"fmt"
)

func Max(x, y int64) int64 {
	if x > y {
		return x
	}
	return y
}

func Kadane(arr []int64) int64 {
	var maxSoFar, maxEnding int64 = 0, 0
	for _, x := range arr {
		maxEnding = Max(0, maxEnding + x)
		maxSoFar = Max(maxSoFar, maxEnding)
	}
	return maxSoFar
}
```

Questão 2

a)

No código em C#, a complexidade é O(n), pois é todo o vetor de tamanho n.
No código em Rust igualmente, utiliza-se um for para percorrer todo o vetor de tamanho n, fazendo a complexidade ser O(n) também.

C#: Busca Linear -> O(n)

```
public static int search(int[] arr, int x)
  {
    int N = arr.Length;
    for (int i = 0; i < N; i++) {
      if (arr[i] == x)
        return i;
    }
    return -1;
  }
```

Rust: Busca linear -> O(n)

```
fn linear_search(list: Vec<i32>, target: i32) -> Option<usize> {
	for i in 0..list.len() {
		if list[i] == target {
			return Some(i)
		}
	}
	return None;
}
```

b)
No código em Ruby, a complexidade é $O(n^2)$, pois para cada elemento do vetor, o próprio vetor é percorrido para realizar as comparações, utilizando dois loops aninhados.
No código em Scala, também é percorrido, para cada elemento do vetor, o vetor inteiro. Pondo dois loops aninhados resultando na complexidade $O(n^2)$.

Ruby: Bubble Sort -> $O(n^2)$

```
def bubble_sort(array)
  array_length = array.size
  return array if array_length <= 1

  loop do
    # we need to create a variable that will be checked so that we don't run into an infinite loop
scenario.
    swapped = false

      (array_length-1).times do |i|
      if array[i] > array[i+1]
        array[i], array[i+1] = array[i+1], array[i]
        swapped = true
      end
    end

    break if not swapped
  end

  array
end

unsorted_array = [11,5,7,6,15]
p bubble_sort(unsorted_array)
```

Scala: Bubble Sort -> $O(n^2)$

```
	def bubbleSort(a:Array[Int]):Array[Int]={
		for(i<- 1 to a.length-1){
			for(j <- (i-1) to 0 by -1){
				if(a(j)>a(j+1)){
					val temp=a(j+1)
					a(j+1)=a(j)
					a(j)=temp
				}
			}
	}
```

c)

No código em Swift, a cada iteração do loop, o vetor de tamanho n é dividido em 2. Ou seja, a cada passo, nós temos um tamanho de n, n/2, n/4 e assim por diante, sendo o pior caso até chegar no número 1. Então considerando ‘k’ sendo o número de passos até chegar em 1, 2^k é maior ou igual a n, e pela definição de logaritmo, k = log n na base 2. (Binary Search).

No código em Python, o algoritmo recebe uma lista ordenada, e para fazer menos comparações, ele pula para posições no array com base em um offset fixo. Se o elemento em uma das posições for menor ou maior, então o código pula para uma posição maior ou menor. Como o tamanho de cada pulo é igual à raiz quadrada do tamanho, o pior caso seria se o código fosse percorrer a lista inteira, o que implica na complexidade sendo O(√n). (Jump Search).

Swift: Busca -> O(log n)

```
public fun binarySearch<T: Comparable>(_ a: [T], key: T) -> Int? {
        var lowerBound = 0
        var upperBound = a.count
        while lowerBound < upperBound {
                let midIndex = lowerBound + (upperBound - lowerBound) / 2
                if a [midIndex] == key {
                        return midIndex
                } else if a[midIndex] < key {
                        lowerBound = midIndex + 1
                } else {
                        upperBound = midIndex
                }
        }
        return nil
}
```

Python: Busca -> O(√n)

```
def jumpSearch( arr , x , n ):

    step = math.sqrt(n)

    prev = 0
    while arr[int(min(step, n)-1)] < x:
        prev = step
        step += math.sqrt(n)
        if prev >= n:
            return -1

    while arr[int(prev)] < x:
        prev += 1

        if prev == min(step, n):
            return -1
```

```
if arr[int(prev)] == x:
    return prev

return -1
```