Ejercicios de Complejidad Algorítmica

1. **Presente paso a paso la complejidad de la siguiente función:**

```
fun intercambiar(x: Int, y: Int): Pair<Int, Int> {
    val aux = x   => O(1)
    x = y         => O(1)
    y = aux       => O(1)
    return x to y => O(1)
}
```

=> O(4) ⟶ O(1)

complejidad : constante
O(1)

2. **Presente la complejidad de la siguiente función**

```
fun max(x: Int, y: Int): Int {
  1 var result: Int = 0                  ⟶ O(1)
  2.1 if (x >= y) {          => O(1)
    2.2 result = x           => O(1)
    }                                    } O(1)
  2.3 else {                 => O(1)
    2.4 result = y           => O(1)
    }
  3 return result                        ⟶ O(1)
}
```

O(1)

complejidad : constante
O(1)

3. **Calcular paso a paso, la complejidad de la siguiente función:**

```
fun fun1(a: IList<Double>): Unit {
O(9) for (i in 0 until 9) {
        a[i] = 0.0 ⟶ O(1)
    }
}
```

O(9·1) ⟶ O(9) ⟶ O(1)

4. **Calcular paso a paso, la complejidad de la siguiente función:**

```
fun fun1(a: IList<Double>): Unit {
    var n = a.size ⟶ O(1)
    for (i in 0 until n) { ⟶ O(n)
        a[i] = 0.0 ⟶ O(n)
    }
}
```

O(n) > O(1) ⟶ O(n)

5. **Calcular paso a paso, la complejidad de la siguiente función:**

```
fun suma(v: IList<Int>): Int {
    var result = 0 ⟶ O(1)
    for (i in 0 until v.size) { ⟶ O(v.size)
        result += v[i] ⟶ O(1)
    }
    return result ⟶ O(1)
}
```

O(v.size)

?

6. Calcular, paso a paso, la complejidad de la siguiente función:

```
fun aSaber(vector: IList<Int>): Int {
    var result = 0 ⟶ O(1)
    var i = 0 ⟶ O(1)
    val n = vector.size ⟶ O(1)
    while (i < n - 1) { ⟶ O(n-1)
        if (v[i] <= v[i+1]) { ⟶ O(1)
            (v[i], v[i+1]) = intercambiar(v[i], v[i+1]) ⟶ O(2)
        }
        i++ ⟶ O(1)
    }
    return result
}  ⟶ O(1)
```

O(4) → O(4(n-1)) → O(4n-4)

? → O(n)

La complejidad del algoritmo es: O(n)

7. Calcular, paso a paso, la complejidad del siguiente algoritmo

```
fun inserción(v: IList<Int>): Unit {
    var min = 0; val n = v.size ⟶ O(1)+O(1) → O(1)
2   for (i in 0 .. n - 2) { ⟶ O(n-1) → O(n)
2.1     min = i ⟶ O(1)
2.2     for (j in i+1 until n) { ⟶ O(n-1)?
2.2.1       if (v[j] < v[min]) ⟶ O(1)
                min = j ⟶ O(1)
            }
        }
2.3     (v[i], v[min]) = intercambiar(v[i], v[min]) ⟶ O(1)
    }
}
```

K: veces que se ejecuta

⟹ O(n²)

Si i=0 → K = n-1 (en el mejor de los casos se ejecuta dos veces; en el peor de los casos se ejecuta n-1 veces)

Si i=(n-2) → K = 2

La complejidad del algoritmo es de: O(n²)

① for (i in a..b) → K = b+1-a; i = {a, ..., b-1, b}

② for (i in a until b) → K = b-a

8. Calcular paso a paso, la complejidad de la siguiente función:

```
fun fun3(n: Int): Int {
1   var i = 0 ⟶ O(1)
2   var acum = 1 ⟶ O(1)
3   while (acum <= n) {
3.1     i++ ⟶ O(1)
3.2     acum *= 2 ⟶ O(1)
    }
4   return i ⟶ O(1)
}
```

O(1)+O(1)+O(√n)+O(1) → O(√n)

③ O(⌈2√n⌉) → O(√n)

La complejidad del algoritmo (peor caso) es:

O(√n)

9. Calcular paso a paso, la complejidad de la siguiente función:

```
fun proc(n: Int): Unit {
1   var i = 1 ⟶ O(1)
2   while (i <= n) { ⟶ O(n)
2.1     var k = n ⟶ O(1)
2.2     while (k >= 0) ⟶ O(n+1)
            k-- ⟶ O(1)
        }
2.3     k = n ⟶ O(1)
2.4     while (k >= 1) { ⟶ O(⌊log₃(n)⌋+1)
            k = k / 3 ⟶ O(1)
        }
2.5     i++ ⟶ O(1)
    }
}
```

O(n) × O(n) → O(n²)

O(log n) × O(log n) → O(log n)

O(n) + O(log n) → O(n)

La complejidad (peor de los casos) del algoritmo es de O(n²)