Innovación en Formación Profesional

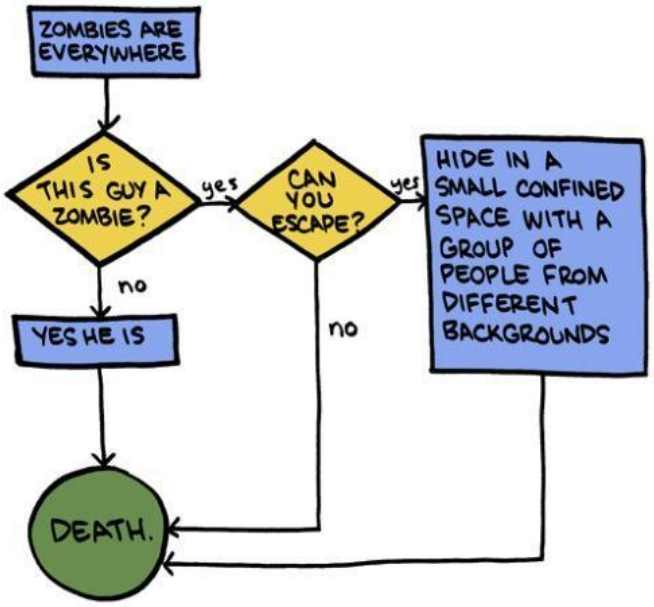
Módulo Profesional 05:
ENTORNOS DE DESARROLLO
Actividad UF1

Nombre del Estudiante: Enrique Verea Reimpell

CICLO FORMATIVO DE GRADO SUPERIOR EN

DESARROLLO DE APLICACIONES MULTIPLATAFORMA

MODALIDAD ONLINE



Desarrollo de la actividad

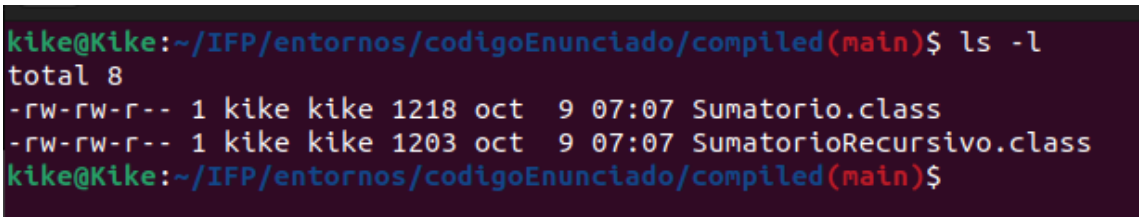
Parte 1: Compilación de un programa informático (5 puntos)

A continuación, dar respuesta a las siguientes preguntas observando el código fuente, intermedio y ejecutable de ambas aplicaciones:

a) **(*1 p.*) ¿cuál de las dos aplicaciones compiladas ocupa más espacio en el disco duro (en bytes)?**

EVIDENCIAS

- Tamaño en bytes de “Sumatorio.java”: 1218
- Tamaño en bytes de “SumatorioRecursivo.java”: 1203



RESPUESTA

1. Respuesta argumentada:

   Sumatorio.class, con 1218 bytes. ocupa más espacio en el disco duro que SumatorioRecursivo.class, que ocupa 1203 bytes.

b) **(*1 p.*) ¿cuál de las dos aplicaciones tiene más líneas de código fuente?**

EVIDENCIAS

- Líneas de “Sumatorio.java”: 17
- Líneas de “SumatorioRecursivo.java”: 13

RESPUESTA

2. Respuesta argumentada:

   Sumatorio.java tiene más líneas de código fuente. SumatorioRecursivo.java extrae la lógica para calcular sumatorios en la función *sumatorio(int.int)* con lo que elimina la duplicación de código que existe en Sumatorio.java, lo que resulta en menos líneas de código.

c) **(*1 p.*) Si desensamblamos el código máquina (bytecodes), ¿cuál de las dos aplicaciones contiene un número mayor de instrucciones?**

EVIDENCIAS

- Líneas de código ensamblador de “Sumatorio.java”: 64
- Líneas de código ensamblador de “SumatorioRecursivo.java”: 44

Sumatorio.java

```
public Sumatorio();
  descriptor: ()V
  flags: (0x0001) ACC_PUBLIC
  Code:
    stack=1, locals=1, args_size=1
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return
    LineNumberTable:
      line 1: 0
```

```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: (0x0009) ACC_PUBLIC, ACC_STATIC
  Code:
    stack=5, locals=6, args_size=1
       0: invokestatic  #7                  // Method java/lang/System.nanoTime:()J
       3: lstore_1
       4: iconst_0
       5: istore_3
       6: iconst_1
       7: istore        4
       9: iload         4
      11: sipush        1000
      14: if_icmpgt     28
      17: iload_3
      18: iload         4
      20: iadd
      21: istore_3
      22: iinc          4, 1
      25: goto          9
      28: getstatic     #13                 // Field java/lang/System.out:Ljava/io/PrintStream;
      31: iload_3
      32: invokedynamic #17,  0             // InvokeDynamic #0:makeConcatWithConstants:(I)Ljava/lang/String;
      37: invokevirtual #21                 // Method java/io/PrintStream.print:(Ljava/lang/String;)V
      40: iconst_0
      41: istore_3
      42: sipush        1001
      45: istore        4
      47: iload         4
      49: sipush        2000
      52: if_icmpgt     66
      55: iload_3
      56: iload         4
      58: iadd
      59: istore_3
      60: iinc          4, 1
      63: goto          47
      66: getstatic     #13                 // Field java/lang/System.out:Ljava/io/PrintStream;
      69: iload_3
      70: invokedynamic #17,  0             // InvokeDynamic #0:makeConcatWithConstants:(I)Ljava/lang/String;
      75: invokevirtual #21                 // Method java/io/PrintStream.print:(Ljava/lang/String;)V
      78: iconst_0
      79: istore_3
      80: sipush        2001
      83: istore        4
      85: iload         4
      87: sipush        3000
      90: if_icmpgt     104
      93: iload_3
      94: iload         4
      96: iadd
      97: istore_3
      98: iinc          4, 1
     101: goto          85
```

```
     104: getstatic     #13                 // Field java/lang/System.out:Ljava/io/PrintStream;
     107: iload_3
     108: invokevirtual #27                 // Method java/io/PrintStream.println:(I)V
     111: invokestatic  #7                  // Method java/lang/System.nanoTime:()J
     114: lstore        4
     116: getstatic     #13                 // Field java/lang/System.out:Ljava/io/PrintStream;
     119: lload         4
     121: lload_1
     122: lsub
     123: invokedynamic #31,  0             // InvokeDynamic #1:makeConcatWithConstants:(J)Ljava/lang/String;
     128: invokevirtual #21                 // Method java/io/PrintStream.print:(Ljava/lang/String;)V
     131: return
   LineNumberTable:
     line 4: 0
     line 6: 4
     line 7: 6
     line 8: 17
     line 7: 22
     line 10: 28
     line 12: 40
     line 13: 42
     line 14: 55
     line 13: 60
     line 16: 66
     line 18: 78
     line 19: 80
     line 20: 93
     line 19: 98
     line 22: 104
     line 24: 111
     line 26: 116
     line 27: 131
   StackMapTable: number_of_entries = 6
     frame_type = 254 /* append */
       offset_delta = 9
       locals = [ long, int, int ]
     frame_type = 250 /* chop */
       offset_delta = 18
     frame_type = 252 /* append */
       offset_delta = 18
       locals = [ int ]
     frame_type = 250 /* chop */
       offset_delta = 18
     frame_type = 252 /* append */
       offset_delta = 18
       locals = [ int ]
     frame_type = 250 /* chop */
       offset_delta = 18
```

SumatorioRecursivo.java

```
public Sumatorio();
  descriptor: ()V
  flags: (0x0001) ACC_PUBLIC
  Code:
    stack=1, locals=1, args_size=1
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return
    LineNumberTable:
      line 1: 0
```

```
public static int sumatorio(int, int);
  descriptor: (II)I
  flags: (0x0009) ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=2, args_size=2
       0: iload_0
       1: iload_1
       2: if_icmpgt     15
       5: iload_0
       6: iload_0
       7: iconst_1
       8: iadd
       9: iload_1
      10: invokestatic  #7                  // Method sumatorio:(II)I
      13: iadd
      14: ireturn
      15: iconst_0
      16: ireturn
    LineNumberTable:
      line 4: 0
      line 5: 5
      line 7: 15
    StackMapTable: number_of_entries = 1
      frame_type = 15 /* same */
```

```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: (0x0009) ACC_PUBLIC, ACC_STATIC
  Code:
    stack=5, locals=5, args_size=1
       0: invokestatic  #13                  // Method java/lang/System.nanoTime:()J
       3: lstore_1
       4: getstatic     #19                  // Field java/lang/System.out:Ljava/io/PrintStream;
       7: iconst_1
       8: sipush        1000
      11: invokestatic  #7                   // Method sumatorio:(II)I
      14: invokedynamic #23,  0              // InvokeDynamic #0:makeConcatWithConstants:(I)Ljava/lang/String;
      19: invokevirtual #27                  // Method java/io/PrintStream.print:(Ljava/lang/String;)V
      22: getstatic     #19                  // Field java/lang/System.out:Ljava/io/PrintStream;
      25: sipush        1001
      28: sipush        2000
      31: invokestatic  #7                   // Method sumatorio:(II)I
      34: invokedynamic #23,  0              // InvokeDynamic #0:makeConcatWithConstants:(I)Ljava/lang/String;
      39: invokevirtual #27                  // Method java/io/PrintStream.print:(Ljava/lang/String;)V
      42: getstatic     #19                  // Field java/lang/System.out:Ljava/io/PrintStream;
      45: sipush        2001
      48: sipush        3000
      51: invokestatic  #7                   // Method sumatorio:(II)I
      54: invokevirtual #33                  // Method java/io/PrintStream.println:(I)V
      57: invokestatic  #13                  // Method java/lang/System.nanoTime:()J
      60: lstore_3
      61: getstatic     #19                  // Field java/lang/System.out:Ljava/io/PrintStream;
      64: lload_3
      65: lload_1
      66: lsub
      67: invokedynamic #37,  0              // InvokeDynamic #1:makeConcatWithConstants:(J)Ljava/lang/String;
      72: invokevirtual #27                  // Method java/io/PrintStream.print:(Ljava/lang/String;)V
      75: return
    LineNumberTable:
      line 12: 0
      line 14: 4
      line 15: 22
      line 16: 42
      line 18: 57
      line 20: 61
      line 21: 75
```

RESPUESTA

3. Respuesta argumentada:

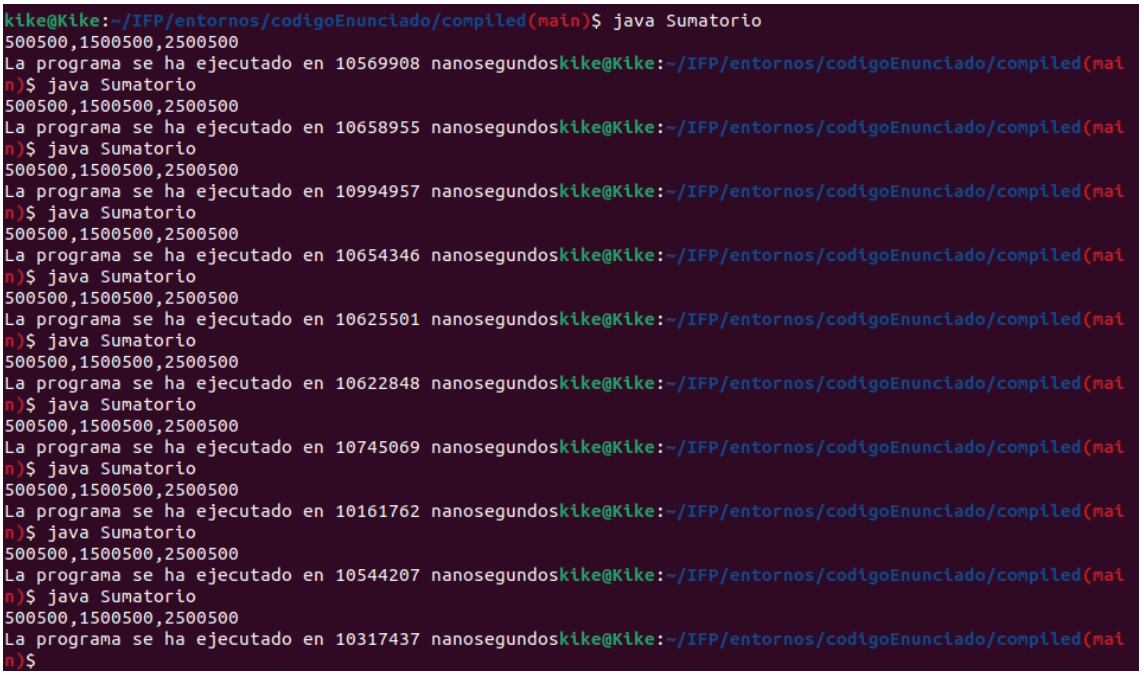
   ‘Sumatorio.java’ tiene un número mayor de instrucciones. Esto se debe a que existe duplicación en su lógica al incluir código por separado para calcular cada sumatorio, con sus respectivos bucles, asignaciones de variables e impresión en consola, lo que genera un conjunto de instrucciones por cada sumatorio (véanse las instrucciones 4-37, 40-75, 78-108), a diferencia de ‘SumatorioRecursivo.java’, que extrae la lógica del sumatorio a la función recursiva *sumatorio(int, int)*, lo que genera un único conjunto de

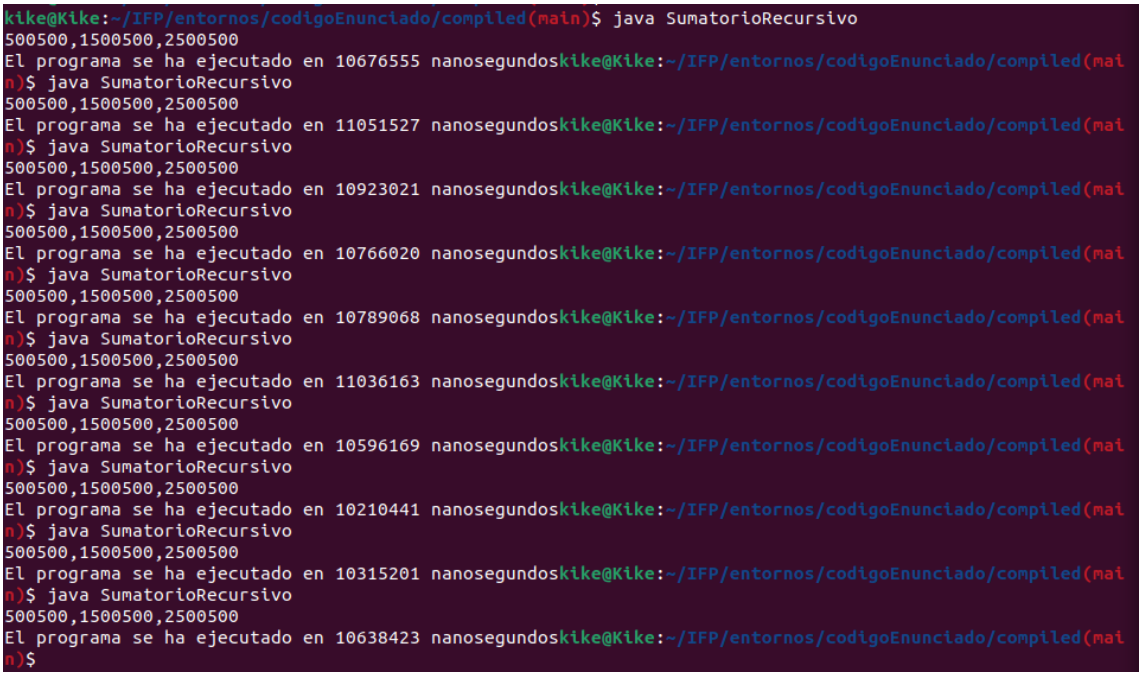
instrucciones para generar sumatorios y una única instrucción para invocar este conjunto de instrucciones por cada sumatorio (las instrucciones 11, 31 y 51 invocan el conjunto de instrucciones de la función *sumatorio(int, int)*).

d) (*1 p.*) Ejecutar cada una de las aplicaciones 10 veces y calcular el promedio de nanosegundos que tardan en ejecutarse, ¿cuál de las dos aplicaciones se ejecuta más rápido?

EVIDENCIAS

- Tiempo promedio de ejecución de “Sumatorio.java”: **12.020.740**
- Tiempo promedio de ejecución de “SumatorioRecursivo.java”: **12.252.199**





RESPUESTA

4. Respuesta argumentada:

   El tiempo promedio de ejecución de Sumatorio.java fue 110.759 nanosegundos más rápido que el tiempo promedio de ejecución de SumatorioRecursivo.java. Aunque es una diferencia pequeña, representando solo un 1,89%, también se puede observar que Sumatorio.java registró el tiempo de ejecución más rápido (10.161.762) y su tiempo de ejecución nunca llegó a superar los 11.000.000 nanosegundos, mientras que SumatorioRecursivo.java registró los 2 tiempos más lentos (11.036.163 y 11.051.527), llegando a superar los 11.000.000 nanosegundos en ambos casos. Se puede llegar a la conclusión de que Sumario.java se ejecuta más rápido que SumarioRecursivo.java.

e) (*1 p.*) Contrastando las respuestas a), b), c) y d) responder a la siguiente pregunta; ¿una aplicación que contenga más líneas de código fuente necesariamente ha de tardar más en ejecutarse que otra que contenga menos líneas?

RESPUESTA

5. Respuesta argumentada:

   Contrastando los resultados obtenidos en los ejercicios anteriores en los que la clase Sumatorio.java obtuvo mejores resultados en tiempo de ejecución que la clase SumatorioRecursivo.java, aún teniendo un ~23% más líneas de código fuente y haber generado un ~30% más de instrucciones de bytecode, se puede argumentar que un mayor número de líneas de código no se traduce en un tiempo de ejecución mayor.

Analizando la ejecución del programa de una manera meramente lógica se puede fácilmente observar que aunque SumatorioRecursivo.java tiene una parte de su lógica abstraída en la función recursiva *sumatorio(int, int)*, las instrucciones generadas por esta parte del código fuente (13 instrucciones) son invocadas 3 veces por el programa, por lo que, en tiempo de ejecución, el número de instrucciones ejecutadas por el procesador no es de 44, número de instrucciones generadas en compilación, sino 70, siendo mayor que las 64 instrucciones ejecutadas por Sumatorio.java. En otras palabras, la función *sumatorio(int, int)*, que es la razón por la que SumatorioRecursivo.java tiene menos líneas de código fuente y genera menos líneas de instrucciones, no representa una ventaja real en instrucciones ejecutadas, sino más bien aumenta ligeramente las mismas.

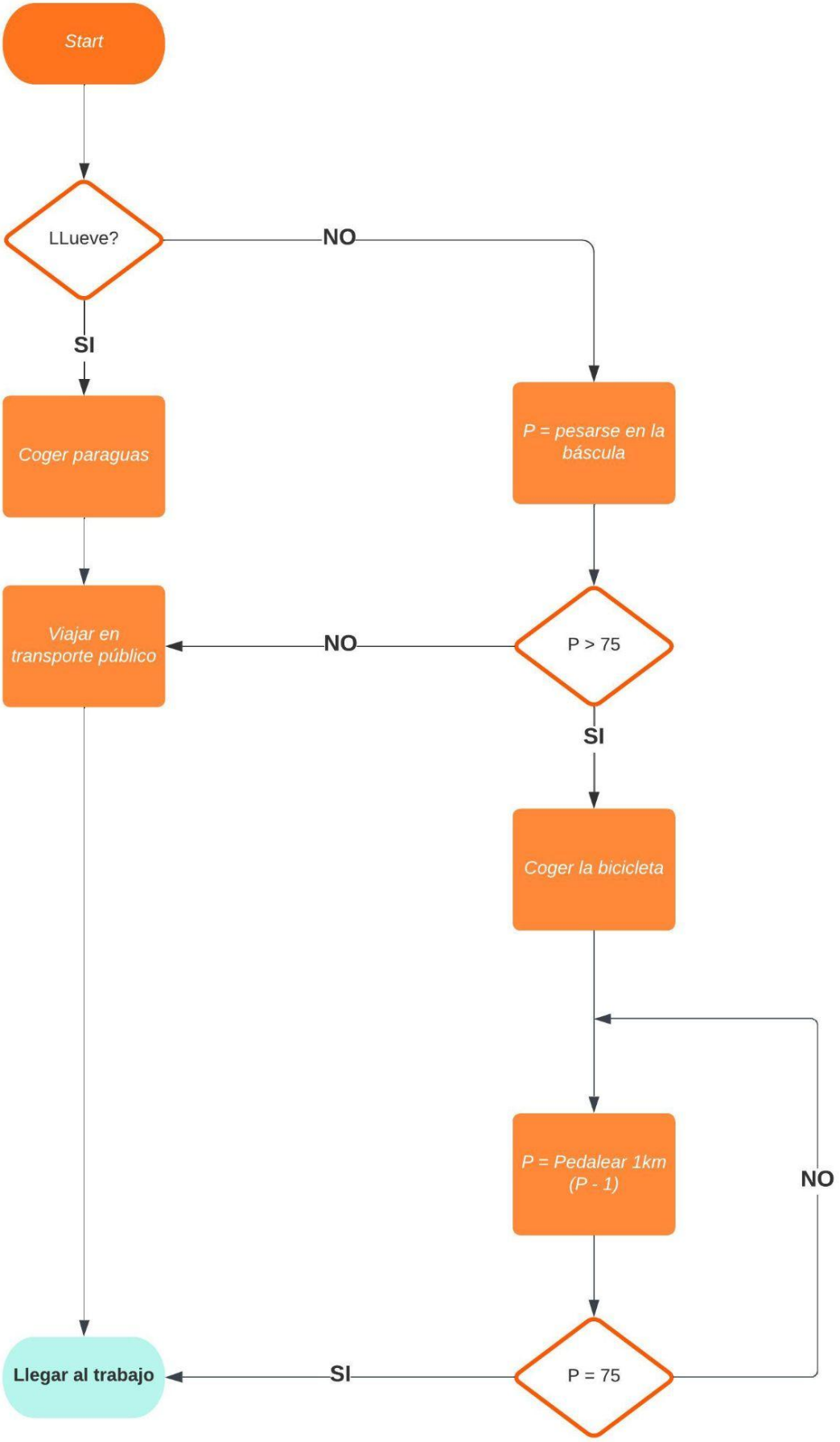
Por otro lado, al ser la función *sumatorio(int, int)* una función recursiva se debe tomar en cuenta que para calcular un sumatorio, la función debe invocarse a sí misma un cantidad *n* de veces, donde *n* = numFinal - numInicial + 1, siendo *n* = 1000 en cada invocación que SumatorioRecursivo.java realiza para calcular un sumatorio y que, multiplicado por 3 sumatorios, se traduce a un total de 3000 invocaciones de esta función, a diferencia de Sumatorio.java, que no invoca ninguna función fuera de las funciones de *System*, que igualmente invoca SumatorioRecursivo.java. Tomando en cuenta que cada invocación a una función tiene un coste en tiempo de ejecución, ésta parece ser la causa de mayor magnitud por la que SumatorioRecursivo.java tiende a ser algo más lento que Sumatorio.java.

En conclusión, un menor número de líneas de código no es un factor determinante en el tiempo de ejecución de una aplicación. Se podría incluso razonar fuera de este análisis realizado mediante los ejercicios anteriores y presentar casos extremos que apoyan esta conclusión, y es que, con el mínimo número de líneas de código fuente (1) se puede crear un programa de con el mayor tiempo de ejecución posible (∞): while(true) {} .

Parte 2: Diseño y conceptualización de un programa informático (5 puntos)

a) (2,5 *p.*) Dibujar el diagrama de flujo del programa informático que representa cómo desplazarse al trabajo en base a los requisitos enunciados anteriormente.

RESPUESTA


Start
LLueve?
NO
SI
Coger paraguas
P = pesarse en la báscula
Viajar en transporte público
NO
P > 75
SI
Coger la bicicleta
P = Pedalear 1km (P - 1)
NO
Llegar al trabajo
SI
P = 75

b) (2,5 *p.*) En base al flujograma anterior, elaborar el pseudocódigo del programa informático.

RESPUESTA

```
SI llueve
    coger el paraguas
    cojer el transporte público
SINO
    P = pesarse en la báscula
    SI P mayor que 75
        coger la bici
        MIENTRAS P mayor que 75 HACER
            pedalear 1km
            P = P - 1
        FIN MIENTRAS
    SINO
        cojer el transporte público
    FIN SI
FIN SI
llegar al trabajo
```