**Ejercicio 1. Dado el siguiente Código de ordenación por el método de la burbuja:**

**public class BubbleSort {**

**/\*\***

**\* This method the BubbleSort method. Returns**

**\* the array in increasing order.**

**\***

**\* @param array The array to be sorted**

**\* @param size The count of total number of elements in array**

**\*\*/**

**public static <T extends Comparable<T>> void BS(T array[], int size) {**

**boolean swap;**

**int last = size - 1;**

**do**

**{**

**swap = false;**

**for (int count = 0; count < last-1; count++)**

**{**

**int comp = array[count].compareTo(array[count + 1]);**

**if (comp > 0)**

**{**

**T temp = array[count];**

**array[count] = array[count + 1];**

**array[count + 1] = temp;**

**swap = true;**

**}**

**}**

**} while (swap);**

**}**

**}**

**Al ejecutar el código con los valores de entrada obtenemos estos resultados:**

**[1 0 0 0 2 0 ] --> [0 0 0 1 2 0 ]**

**[8 7 6 5 4 9 ] --> [4 5 6 7 8 9 ]**

**Rellena la siguiente plantilla aplicando el método científico para depuración:**

|  |  |
| --- | --- |
| **Hipótesis** |  |
| **Predicción** |  |
| **Experimento** |  |
| **Observación** |  |
| **Conclusión** |  |

**Ejercicio 2. Dado el siguiente Código fuente:**

**public class Goldbach {**

**/\*\***

**\* Checks if the number n is prime. Negative numbers and zero are declared**

**\* to be non-prime.**

**\***

**\* @param n**

**\* Number to be tested.**

**\* @return True if n is prime, false if not.**

**\*/**

**public static boolean isPrime(Integer n) {**

**for (int p = 2; p < Math.sqrt(n)-1; p++) {**

**if (n % p == 0) {**

**return false;**

**}**

**}**

**return true;**

**}**

**/\*\***

**\* Checks Goldbach conjecture for n, that is, checks if the even number n**

**\* can be written as the sum of two prime numbers.**

**\***

**\* If n is odd, it returs automatically true.**

**\***

**\* @param n**

**\* Number to be checked.**

**\* @return True if n can be decomposed as sum of prime numbers, false if**

**\* not. Also prints the decomposition if true.**

**\*/**

**public static boolean checkGoldbach(Integer n) {**

**for (int p = 1; p < n / 2; p++) {**

**if (isPrime(p) && isPrime(n - p)) {**

**System.out.println(n + "=" + p + " & " + (n - p));**

**return true;**

**}**

**}**

**return false;**

**}**

**}**

**Al ejecutar el código con los valores de entrada obtenemos estos resultados:**

**13=2 & 11 🡪 CORRECTO!**

**23=4 & 19 🡪 INCORRECTO!**

**Rellena la siguiente plantilla aplicando el método científico para depuración:**

|  |  |
| --- | --- |
| **Hipótesis** | **El algoritmo Goldbach funciona correctamente con números impares** |
| **Predicción** | **Si introducimos un número impar que no se descompone en 2 numeros primos el programa devolvería false** |
| **Experimento** | **Ejecutamos el código con la entrada 23** |
| **Observación** | **El programa nos devuelve true y los valores erróneos 4 y 19. (4 no es primo)** |
| **Conclusión** | **Hipotesis rechazada** |

|  |  |
| --- | --- |
| **Hipótesis** | **El algortimo Goldbach funciona correctamente con números pares** |
| **Predicción** | **Si introducimos un numero par el programa nos debería devovler automáticamente un true y la descomposicion** |
| **Experimento** | **Ejecutamos el código con la entrada 24** |
| **Observación** | **El programa nos devuelve true y los valores 1 y 23** |
| **Conclusión** | **Hipotesis confirmada** |

|  |  |
| --- | --- |
| **Hipótesis** | **La función isPrime() funciona correctamente** |
| **Predicción** | **Si introducimos un número no primo la función nos debería devolver false** |
| **Experimento** | **Ejecutamos isPrime() con las entradas 4 y 16** |
| **Observación** | **El programa nos devuelve true para 4 y false para 16** |
| **Conclusión** | **Hipótesis rechazada** |

|  |  |
| --- | --- |
| **Hipótesis** | **La condición de parada del for en la función isPrime es erronea** |
| **Predicción** | **Si cambiamos “p < Math.sqrt(n)-1;” con “p < n” todo debería funcionar correctamente** |
| **Experimento** | **Cambiar la condición de parada por “p < n” y ejecutamos el programa con la entrada 4** |
| **Observación** | **El programa nos devuelve false para 4** |
| **Conclusión** | **Hipótesis confirmada** |

|  |  |
| --- | --- |
| **Hipótesis** | **La función isPrime() no acepta los valores 0 o negativos** |
| **Predicción** | **Si introducimos 0 o un numero negativo isPrime debería devolver false.** |
| **Experimento** | **Ejecutamos isPrime() con 0 y -1** |
| **Observación** | **El programa devuelve true en ambos** |
| **Conclusión** | **Hipótesis rechazada** |

|  |  |
| --- | --- |
| **Hipótesis** | **No hay una condición if que excluya 0 y números negativos** |
| **Predicción** | **Si creamos una condición if (n<=0){return false;} el programa debería funcionar correctamente** |
| **Experimento** | **Añadiomos la condición if mencionada y ejecutamos isPrime() con 0 y -1** |
| **Observación** | **El programa devuelve false en ambos** |
| **Conclusión** | **Hipotesis confirmada** |

**Ejercicio 3.**

**Dada la siguiente implementación de una pila.**

**public class Stack {**

**private Integer[] arrayStack;**

**private Integer MAX\_SIZE = 5;**

**private Integer top;**

**/\*\***

**\* Create a stack LIFO with finite depth.**

**\*/**

**public Stack() {**

**this.arrayStack = new Integer[MAX\_SIZE];**

**this.top = 1;**

**}**

**/\*\***

**\* Inserts an element in the stack. If the stack is full, it shows a message**

**\* and stops**

**\***

**\* @param element**

**\* Element to be inserted.**

**\*/**

**public void push(Integer element) {**

**if (top >= MAX\_SIZE) {**

**System.out.println("The stack is full");**

**}**

**arrayStack[this.top] = element;**

**top++;**

**}**

**/\*\***

**\* Extracts an element from the stack. If the stack is empty, it shows a**

**\* message and returns -1.**

**\***

**\* @return Extracted element.**

**\*/**

**public Integer pull() {**

**if (top <= 0) {**

**System.out.println("The stack is empty");**

**return 0;**

**}**

**top--;**

**return arrayStack[top - 1];**

**}**

**/\*\***

**\* Returns the number of elements in the stack.**

**\***

**\* @return Number of elements in the stack.**

**\*/**

**public Integer getSize() {**

**return top;**

**}**

**/\*\***

**\* Extract nElements objects of the stack.**

**\***

**\* @param nElements**

**\* Number of elements to be extracted.**

**\* @return Array with the extracted elements.**

**\*/**

**public Integer[] pull(Integer nElements) {**

**Integer[] res = new Integer[nElements];**

**for (int i = 0; i < nElements; i++) {**

**res[i] = pull();**

**}**

**return res;**

**}**

**/\*\***

**\* Copy an stack into a new stack. The new stack has the same elements in**

**\* the same order as the old one.**

**\***

**\* @return Duplicated stack.**

**\*/**

**public Stack copyStack() {**

**Stack newStack = new Stack();**

**for (int i = 0; i < this.getSize(); i++) {**

**newStack.push(arrayStack[i]);**

**}**

**return newStack;**

**}**

**}**

**Al ejecutar el código obtenemos los siguientes resultados.**

**1.**

**Stack stack = new Stack();**

**stack.push(1);**

**stack.push(2);**

**stack.push(3);**

**stack.push(4); Correcto!!**

**2.**

**Stack stack = new Stack();**

**stack.push(12);**

**stack.push(2);**

**stack.push(43);**

**stack.push(95);**

**Integer v[] = stack.pull(stack.getSize());**

**Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Array index out of range: -1**

**at stack.Stack.pull(Stack.java:44)**

**at stack.Stack.pull(Stack.java:67)**

**at stack.Stack.main(Stack.java:97)**

**Siendo esta ejecución INCORRECTA**

**3.**

**Stack stack = new Stack();**

**stack.push(1);**

**stack.push(2);**

**stack.push(3);**

**stack.push(4);**

**Stack nueva = stack.copyStack();**

**The stack is full**

**Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Array index out of range: 5**

**at stack.Stack.push(Stack.java:30)**

**at stack.Stack.copyStack(Stack.java:86)**

**at stack.Stack.main(Stack.java:102)**

**Siendo esta ejecución INCORRECTA**

**4.**

**stack.push(1);**

**stack.push(2);**

**stack.push(3);**

**stack.push(4);**

**System.out.println("Elemento = " + stack.pull());**

**Elemento = 3**

**Siendo esta ejecución INCORRECTA**

**Aplique el método científico para su depuración rellenando la plantilla correspondiente.**

**Viendo la cantidad de errores que hay, prepare una batería de test lo suficientemente completa para validar el correcto funcionamiento de la pila.**

|  |  |
| --- | --- |
| **Hipótesis** | **El programa funciona correctamente al ejecutar la entrada 1** |
| **Predicción** | **Si introducimos la entrada 1. La pila debería funcionar correctamente** |
| **Experimento** | **Ejecutamos el código con entrada 1.** |
| **Observación** | **El programa funciona correctamente** |
| **Conclusión** | **Hipótesis confirmada** |

|  |  |
| --- | --- |
| **Hipótesis** | **El programa funciona correctamente al ejecutar la entrada 2** |
| **Predicción** | **Si introducimos la entrada 2. La pila debería funcionar correctamente** |
| **Experimento** | **Ejecutamos el código con entrada 2.** |
| **Observación** | **Programa devuelve ArrayIndexOutOfBounds** |
| **Conclusión** | **Hipótesis rechazada** |

|  |  |
| --- | --- |
| **Hipótesis** | **El método getSize() de la pila es correcto** |
| **Predicción** | **Si imprimimos el valor de getSize() nos debería dar el valor 4** |
| **Experimento** | **Ejecutamos el código con un println de arraysize()** |
| **Observación** | **getSize() = 5** |
| **Conclusión** | **Hipótesis rechazada** |

|  |  |
| --- | --- |
| **Hipótesis** | **El Integer top (usado como contador) está mal inicializado** |
| **Predicción** | **Si en el constructor cambiamos el valor inicial de top a 0 getSize() debería funcionar correctamente** |
| **Experimento** | **Cambiamos el valor inicial top = 0 e imprimimos getSize() en la entrada 2** |
| **Observación** | **getSize() = 4** |
| **Conclusión** | **Hipótesis confirmada** |

|  |  |
| --- | --- |
| **Hipótesis** | **Después de este cambio el programa debería funcionar correctamente** |
| **Predicción** | **Si introducimos la entrada 2. La pila debería funcionar correctamente** |
| **Experimento** | **Ejecutamos el código con entrada 2.** |
| **Observación** | **Programa devuelve error** |
| **Conclusión** | **Hipótesis rechazada** |

|  |  |
| --- | --- |
| **Hipótesis** | **El método pull() devuelve el valor en la cima de la pila** |
| **Predicción** | **Con la entrada 2 el método pull() debería devolver el valor en la cima, es decir 95** |
| **Experimento** | **Ejecutamos el código con entrada 2 imprimiento el resultado de stack.pull()** |
| **Observación** | **El valor devuelto es 43, el segundo elemento de la pila** |
| **Conclusión** | **Hipotesis rechazada** |

|  |  |
| --- | --- |
| **Hipótesis** | **El método pull() está mal codificado, resta 2 veces el contador top** |
| **Predicción** | **Si quitamos una de las restas el programa debería funcionar correctamente** |
| **Experimento** | **Cambiamos return** arrayStack[top-1]; a **return** arrayStack[top]; **y ejecutamos la entrada 2.** |
| **Observación** | **La cima devuelta es la correcta y el programa funciona correctamente** |
| **Conclusión** | **Hipotesis confirmada** |

|  |  |
| --- | --- |
| **Hipótesis** | **El programa funciona correctamente al ejecutar la entrada 3** |
| **Predicción** | **Si introducimos la entrada 3. La pila debería funcionar correctamente** |
| **Experimento** | **Ejecutamos el código con entrada 3.** |
| **Observación** | **El programa funciona correctamente** |
| **Conclusión** | **Hipótesis confirmada** |

|  |  |
| --- | --- |
| **Hipótesis** | **El programa funciona correctamente al ejecutar la entrada 4** |
| **Predicción** | **Si introducimos la entrada 4. La pila debería funcionar correctamente** |
| **Experimento** | **Ejecutamos el código con entrada 4.** |
| **Observación** | **El programa funciona correctamente** |
| **Conclusión** | **Hipótesis confirmada** |