Analisis de algoritmos para el software TicoBingo

Ivan Rene Ramirez Castro
Pontificia Universidad Javeriana
23 de noviembre de 2022

Resumen

El presente trabajo aborda de una manera didáctica el análisis de algoritmos para el software TicoBingo, el cual es un juego de bingo en el que se pueden jugar hasta 10 cartones al mismo tiempo. El software es desarrollado en el lenguaje de programación Java, utilizando hilos para la ejecución de los cartones. El análisis de algoritmos se realiza mediante el calculo de la complejidad temporal y espacial de los algoritmos utilizados en el software. Se realiza una comparación entre los algoritmos utilizados en el software y una implementación de los mismos tratando de minimizar la complejidad temporal y espacial. Se concluye que la implementación de los algoritmos en el software TicoBingo es óptima, ya que en promedios de tiempo son muy similares a los algoritmos optimizados creados para este análisis.

KEYWORDS: *Algoritmos, Complejidad temporal, Complejidad espacial, Bingo, Java, Hilo*

1. Introducción

En el mundo de la informática, el análisis de algoritmos es una de las herramientas más importantes para el desarrollo de software. El análisis de algoritmos permite conocer la complejidad temporal y espacial de los algoritmos utilizados en un software, lo cual permite conocer la eficiencia de los mismos. El análisis de algoritmos se realiza mediante el cálculo de la complejidad temporal y espacial de los algoritmos utilizados en el software. La complejidad temporal de un algoritmo se define como el número de operaciones básicas que se realizan para resolver un problema. La complejidad espacial de un algoritmo se define como el número de variables que se utilizan para resolver un problema.

Para nuestro caso, el software TicoBingo tiene algurítmos que se ejecutan en un tiempo muy corto, algoritmos de validaciones de victoria, creacion de cartones, entre otros.

Nuestro objetivo es realizar un análisis de estos algoritmos y compararlos con una implementación de los mismos tratando de minimizar la complejidad temporal y espacial.

1.1. TicoBingo

El juego TicoBingo es un juego de bingo en el que se pueden jugar hasta 10 cartones al mismo tiempo. El software es desarrollado en el lenguaje de programación Java, utilizando hilos para la ejecución de los cartones. El juego se inicia con la creación de los cartones, los cuales son creados de manera aleatoria teniendo en cuenta que no se repitan los números en un mismo cartón. Cada cartón es un hilo separado que llenará su propio cartón y luego se duerme, pero cuando se genera una bolita desde el programa principal (tómbola), se despiertan todos los hilos al mismo tiempo para que busquen en su cartón si ese número está presente y lo marcará (Colorea) en el cartón de la vista. Luego de marcar la casilla con el número de la tómbola que acaba de salir, cada hilo debe revisar si ha ganado la partida, y si gana, el juego termina y el jugador de ese cartón gana, pero si no gana ningún cartón los hilos se duermen para esperar que salga otra bolita de la tómbola.

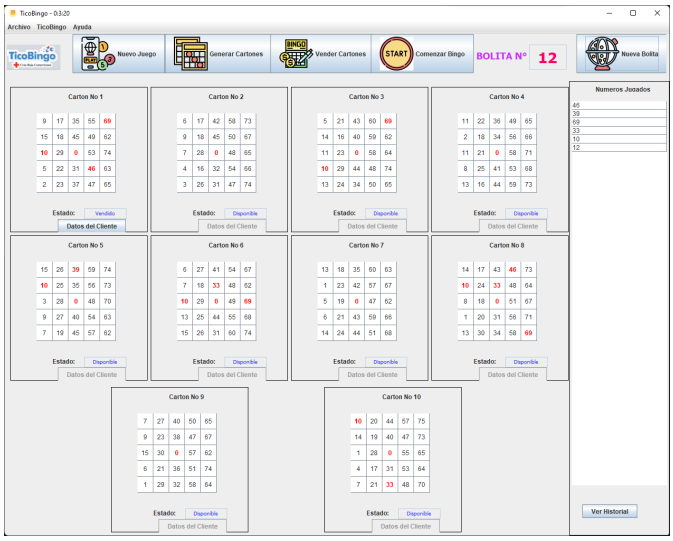

Figura 1: Interfaz del juego TicoBingo

1.2. Tablero de bingo

El tablero de bingo es una matriz de 5x5, en la cual se colocan los números aleatorios del 1 al 15, en la primera columna, los números del 16 al 30 en la segunda columna, los números del 31 al 45 en la tercera columna, los

números del 46 al 60 en la cuarta columna y los números del 61 al 75 en la quinta columna. En el centro de la matriz se coloca el número 0, ya que esta casilla no se utiliza en el juego. Un numero solo puede aparecer una unica vez en la matriz.

Figura 2: Tablero de bingo

1.3. Victoria en el juego

Para ganar el juego, el jugador debe marcar 5 casillas en una misma fila, columna, diagonal o conseguir las 4 esquinas.

2. Marco teórico

En un mundo donde la tecnología avanza a pasos agigantados, es necesario que los programas que se desarrollen sean eficientes, es decir, que se ejecuten en el menor tiempo posible. Para esto, es necesario que los algoritmos que se utilicen en los programas sean eficientes, es decir, que tengan una complejidad temporal y espacial baja.

2.1. Algoritmos

Una buena analogía para entender un algoritmo es la de una caja negra. Una caja negra es un objeto que no se puede ver por dentro, pero se sabe que tiene una entrada y una salida. La entrada es la información que se le da a la caja negra, y la salida es la información que se obtiene de la caja negra. La caja negra realiza una serie de operaciones para obtener la salida a partir de la entrada. En el caso de los algoritmos, la entrada es el problema que se quiere resolver, y la salida es la solución del problema. El algoritmo realiza una serie de operaciones para obtener la solución a partir del problema.

Pueden existir cientos de algoritmos para resolver un mismo problema, pero no todos los algoritmos son eficientes. Un algoritmo es eficiente si tiene una complejidad temporal y espacial baja. [1]

2.2. Complejidad temporal

La complejidad temporal es el número de operaciones que realiza un algoritmo para completar su tarea (considerando que cada operación dura el mismo tiempo). El algoritmo que realiza la tarea en el menor número de operaciones se considera el más eficiente en términos de complejidad temporal. Sin embargo, la complejidad espacial y temporal se ve afectada por factores como el sistema operativo y el hardware, pero no los incluiremos en discusión. [2]

Una de las notaciones más utilizadas para medir la complejidad temporal de un algoritmo es la notación Big O. La notación Big O es una notación asintótica que nos permite medir la complejidad temporal de un algoritmo. La notación Big O nos permite medir la complejidad temporal de un algoritmo en el peor de los casos.

A partir de la notación Big O, podemos evaluar si los algoritmos que se utilizan en TicoBingo son eficientes o no.

El estado del arte de la complejidad temporal de los algoritmos se utilizan benchmarks, que son pruebas que se realizan para medir el tiempo de ejecución de un algoritmo a nivel de procesador. Sin embargo, los benchmarks no son una buena forma de medir la complejidad temporal de un algoritmo, ya que dependen de la configuración del hardware, del sistema operativo y del lenguaje de programación que se utilice.

2.3. Complejidad espacial

La complejidad espacial es la cantidad de espacio en memoria que un algoritmo emplea al ejecutarse. En otras palabras, cómo el algoritmo ocupa espacio en memoria con la cantidad de elementos de entrada que debe procesar. [3]

La notación Big O también se utiliza para medir la complejidad espacial de un algoritmo. La notación Big O nos permite medir la complejidad espacial de un algoritmo en el peor de los casos, pero se deben tener en cuenta los siguientes aspectos:

- La cantidad de memoria requerida por el código del algoritmo.
- La cantidad de memoria requerida para almacenar los datos de entrada.

- La cantidad de memoria requerida para los datos de salida (algoritmos como los de ordenación suelen reorganizar los datos de entrada y por ello no necesitan memoria extra para la salida).

- La cantidad de memoria requerida en cuanto a espacio de trabajo del algoritmo para realizar los cálculos y asignaciones (tanto para variables como cualquier espacio necesario en la pila para almacenar llamadas a subrutinas, este espacio es particularmente significativo para algoritmos que utilizan técnicas recursivas).

[4]

3. Desarrollo del programa

El programa TicoBingo se desarrolló en el lenguaje de programación Java, para este análisis se tomaron los algoritmos que se utilizaron en el programa, se transcribieron a lenguaje de python donde se realizaron las pruebas de complejidad temporal y espacial, y la comparación con algoritmos que deben ser mas eficientes.

3.1. Algoritmos utilizados en el programa

- **verificarGanador**: Valida si el tablero actual es ganador, si lo es, se agrega a la lista de ganadores del estado.

- **estaRepetido**: Verifica si un numero esta repetido en el tablero.

- **generarNumeroAleatorio**: Genera un numero aleatorio entre minimo y maximo.

- **generarTableroAleatorio**: Genera un tablero aleatorio.

3.2. Análisis de complejidad

3.2.1. verificarGanador

El algoritmo verificarGanador se encarga de verificar si el tablero actual es ganador, si lo es, se agrega a la lista de ganadores del estado. El algoritmo verificarGanador se muestra a continuación: Ver Codigo 1

3.2.2. estaRepetido

El algoritmo estaRepetido se encarga de verificar si un numero esta repetido en el tablero. El algoritmo estaRepetido se muestra a continuación: Ver Codigo 2

3.2.3. generarNumeroAleatorio

El algoritmo generarNumeroAleatorio se encarga de generar un numero aleatorio entre minimo y maximo. El algoritmo generarNumeroAleatorio se muestra a continuación: Ver Codigo 3

3.2.4. generarTableroAleatorio

El algoritmo generarTableroAleatorio se encarga de generar un tablero aleatorio. El algoritmo generarTableroAleatorio se muestra a continuación: Ver Codigo 4

4. Conclusiones

4.1. Funcion verificarGanador

Ahora analizaremos la funcion verificarGanador, a continuacion se muestra una tabla con el analisis de complejidad temporal y espacial de la funcion verificarGanador: Ver Tabla 1

4.2. Datos de entrada

- Un diccionario con 5 claves, donde 3 son enteros, 1 es una lista de listas y 1 es una lista

4.3. Datos de salida

- Un diccionario con 6 claves, donde 3 son enteros, 1 es una lista de listas, 1 es una lista y 1 es None o una lista

4.4. Analisis

- Podemos concluir que la complejidad temporal de la funcion verificarGanador no es O(n) ya que no se recorre ninguna lista con un n desconocido

- Por lo tanto la complejidad temporal de la funcion verificarGanador es O(1)

4.5. Funcion estaRepetido

Ahora analizaremos la funcion estaRepetido, a continuacion se muestra una tabla con el analisis de complejidad temporal y espacial de la funcion estaRepetido: Ver Tabla 2

4.6. Datos de entrada

- Un entero

4.7. Datos de salida

- Un booleano

4.8. Analisis

- Podemos concluir que la complejidad temporal de la funcion estaRepetido es O(1), al ser compuesta por 5 operaciones estaticas
- Podemos concluir que la complejidad espacial de la funcion estaRepetido es O(1), por no crear espacio de memoria variables

4.9. Funcion generarNumeroAleatorio

Ahora analizaremos la funcion generarNumeroAleatorio, a continuacion se muestra una tabla con el analisis de complejidad temporal y espacial de la funcion generarNumeroAleatorio: Ver Tabla 3

4.10. Datos de entrada

- Dos enteros

4.11. Datos de salida

- Un entero

4.12. Analisis

- Podemos concluir que la complejidad temporal de la funcion generarNumeroAleatorio es O(1), al ser compuesta por 1 operacion estatica
- Podemos concluir que la complejidad espacial de la funcion generarNumeroAleatorio es O(1), por no crear espacio de memoria variables

4.13. Funcion generarTableroAleatorio

Ahora analizaremos la funcion generarTableroAleatorio, a continuacion se muestra una tabla con el analisis de complejidad temporal y espacial de la funcion generarTableroAleatorio: Ver Tabla 4

4.14. Datos de entrada

- Una matriz de 5x5

4.15. Datos de salida

- Una matriz de 5x5

4.16. Analisis

- Podemos concluir que la complejidad temporal de la funcion generarTableroAleatorio es O(80 + ? + ?) = O(?), ya que no tenemos certeza de la complejidad de las funciones estaRepetido y generarNumeroAleatorio ya que es una funcion que nosotros no podemos controlar, en el peor de los casos la complejidad de la funcion estaRepetido es O(n) y la complejidad de la funcion generarNumeroAleatorio es O(1), por lo tanto la complejidad temporal de la funcion generarTableroAleatorio es O(80 + n + 1) = O(n)
- La complejidad espacial de la funcion generarTableroAleatorio es O(47) = O(1), ya que la complejidad espacial de los datos de entrada es O(1) y la complejidad espacial de los datos de salida es O(1)

Referencias

[1] B. Matos, "Análisis de la complejidad del algoritmo," *Alura Cursos*, 2020.

[2] J. Rivera, "Introducción a la complejidad temporal de los algoritmos," *FreeCodeCamp*, 2021.

[3] M. Arias, "Complejidad espacial," *Platzi*, 2021.

[4] G. L. Steele and R. G. White, "Debunking the 'expensive procedure call' myth, or, procedure call implementations considered harmful, or, lambda: the ultimate goto," *MIT AI Lab*, 1977.

5. Anexos

5.1. Anexo 1: Codigo fuente verificarGanador()

```
1
2  # Transcripci n del c digo de Java a Python
3  def verificarGanador(datos: any):
4          """
5          Valida si el tablero actual es ganador, si lo es, se agrega a la lista de ganadores del
               estado
6          """
7          # Un jugador puede ganar llenando cualquier fila, columna, diagonales o cuatro esquinas
               solamente, los numeros que se encuentran en el centro no cuentan
8          # los numeros que han salido estan en la lista de numeros
9          # id: complejidad temporal, complejidad espacial(si aplica) osea si se esta usando memoria
                extra
10         ganador = False # 1: O(1), O(1)
11         # verificar filas
12         for i in range(5): # 2: O(5), O(1)
13                 numerosEncontradosFila = [False] * 5 # 3: O(5), O(5)
14                 for j in range(5): # 4: O(5), O(1)
15                         for k in range(datos["indexNumeros"]): # 5: O(5), O(1)
16                                 if datos["tablero"][i][j] == datos["numeros"][k]: # 6: O(1), O(0)
17                                         numerosEncontradosFila[j] = True # 7: O(1), O(1)
18                 if numerosEncontradosFila[0] and numerosEncontradosFila[1] and
                       numerosEncontradosFila[2] and numerosEncontradosFila[3] and
                       numerosEncontradosFila[4]: # 8: O(1), O(0)
19                         ganador = True # 9: O(1), O(1)
20                         break # 10: O(1), O(0)
21         # verificar columnas
22         if not ganador: # 11: O(1), O(0)
23                 for i in range(5):  # 12: O(5), O(1)
24                         numerosEncontradosColumna = [False] * 5 # 13: O(5), O(5)
25                         for j in range(5): # 14: O(5), O(1)
26                                 for k in range(datos["indexNumeros"]): # 15: O(5), O(1)
27                                         if datos["tablero"][j][i] == datos["numeros"][k]: # 16: O
                                               (1), O(0)
28                                                 numerosEncontradosColumna[j] = True # 17: O(1), O
                                                       (1)
29                         if numerosEncontradosColumna[0] and numerosEncontradosColumna[1] and
                               numerosEncontradosColumna[2] and numerosEncontradosColumna[3] and
                               numerosEncontradosColumna[4]: # 18: O(1), O(0)
30                                 ganador = True # 19: O(1), O(1)
31                                 break # 20: O(1), O(0)
32         # verificar diagonales
33         if not ganador: # 21: O(1), O(0)
34                 numerosEncontradosDiagonal1 = [False] * 5 # 22: O(5), O(5)
35                 numerosEncontradosDiagonal2 = [False] * 5 # 23: O(5), O(5)
36                 for i in range(5): # 24: O(5), O(1)
37                         for k in range(datos["indexNumeros"]): # 25: O(5), O(1)
38                                 if datos["tablero"][i][i] == datos["numeros"][k]: # 26: O(1), O(0)
39                                         numerosEncontradosDiagonal1[i] = True # 27: O(1), O(1)
40                                 if datos["tablero"][i][4 - i] == datos["numeros"][k]: # 28: O(1),
                                       O(0)
41                                         numerosEncontradosDiagonal2[i] = True # 29: O(1), O(1)
42                 if numerosEncontradosDiagonal1[0] and numerosEncontradosDiagonal1[1] and
                       numerosEncontradosDiagonal1[2] and numerosEncontradosDiagonal1[3] and
                       numerosEncontradosDiagonal1[4]: # 30: O(1), O(0)
43                         ganador = True # 31: O(1), O(1)
44                 if numerosEncontradosDiagonal2[0] and numerosEncontradosDiagonal2[1] and
                       numerosEncontradosDiagonal2[2] and numerosEncontradosDiagonal2[3] and
                       numerosEncontradosDiagonal2[4]: # 32: O(1), O(0)
45                         ganador = True # 33: O(1), O(1)
46         # verificar esquinas
47         if not ganador: # 34: O(1), O(0)
48                 numerosEncontradosEsquinas = [False] * 4 # 35: O(4), O(4)
49                 for i in range(4): # 36: O(4), O(1)
50                         for k in range(datos["indexNumeros"]): # 37: O(5), O(1)
51                                 if datos["tablero"][0][0] == datos["numeros"][k]: # 38: O(1), O(0)
52                                         numerosEncontradosEsquinas[0] = True # 39: O(1), O(1)
```

```
                                if datos["tablero"][0][4] == datos["numeros"][k]: # 40: O(1), O(0)
                                        numerosEncontradosEsquinas[1] = True # 41: O(1), O(1)
                                if datos["tablero"][4][0] == datos["numeros"][k]: # 42: O(1), O(0)
                                        numerosEncontradosEsquinas[2] = True # 43: O(1), O(1)
                                if datos["tablero"][4][4] == datos["numeros"][k]: # 44: O(1), O(0)
                                        numerosEncontradosEsquinas[3] = True # 45: O(1), O(1)
                if numerosEncontradosEsquinas[0] and numerosEncontradosEsquinas[1] and
                    numerosEncontradosEsquinas[2] and numerosEncontradosEsquinas[3]: # 46: O(1), O
                    (0)
                        ganador = True # 47: O(1), O(1)
        if ganador: # 48: O(1), O(0)
                if datos["ganadores"] != None: # 49: O(1), O(0)
                        # Agregar a la lista de ganadores
                        ganadores = datos["ganadores"] # 50: O(1), O(1)
                        # verificar que este tablero no este en la lista de ganadores
                        if not datos["id"] in ganadores: # 51: O(1), O(0)
                                ganadores.append(datos["id"]) # 52: O(1), O(1)
                        datos["ganadores"] = ganadores # 53: O(1), O(1)
                else: # 54: O(1), O(0)
                        # Crear la lista de ganadores
                        ganadores = [] # 55: O(1), O(1)
                        ganadores.append(datos["id"]) # 56: O(1), O(1)
                        datos["ganadores"] = ganadores # 57: O(1), O(1)

# Test
datos = {
        "id": 1,
        "tablero": [
                [4, 16, 43, 54, 65],
                [8, 17, 44, 55, 66],
                [9, 18, 45, 56, 67],
                [10, 19, 46, 57, 68],
                [11, 20, 47, 58, 69]
        ],
        "numeros": [4, 16, 43, 54, 65, 8, 17, 44, 55, 66, 9, 18, 45, 56, 67, 10, 19, 46, 57, 68,
            11, 20, 47, 58, 69],
        "indexNumeros": 25,
        "ganadores": None
}
# en milisegundos
verificarGanador(datos)
print(datos)
```

Listing 1: Algoritmo verificarGanador

5.2. Anexo 2: Codigo fuente estaRepetido()

```
# Metodo privado para verificar si un numero esta repetido en el tablero
def estaRepetido(numero: int) -> bool:
        # Recorre el arreglo de numeros
        for i in range(5): # 1: O(5), O(1)
                # Recorre el arreglo de numeros
                for j in range(5): # 2: O(5), O(1)
                        # Verifica si el numero esta repetido
                        if tablero[i][j] == numero: # 3: O(1), O(0)
                                return True # 4: O(1), O(1)
        return False # 5: O(1), O(1)
```

Listing 2: Algoritmo estaRepetido

5.3. Anexo 3: Codigo fuente generarNumeroAleatorio()

```
import random
# Metodo privado para generar un numero aleatorio
def generarNumeroAleatorio(minimo: int, maximo: int) -> int:
        return random.randint(minimo, maximo) # 1: O(1), O(1)
```

Listing 3: Algoritmo generarNumeroAleatorio

5.4. Anexo 4: Codigo fuente generarTableroAleatorio()

```
# Metodo privado para generar un tablero
def generarTableroAleatorio(data: any):
        numero: int # 1: O(1), O(1)
        columna: int # 2: O(1), O(1)
        fila: int # 3: O(1), O(1)
        numeros: list[int] = [0] * 25 # 4: O(25), O(25)
        # Genera los numeros aleatorios
        for i in range(25): # 5: O(25), O(1)
                # Genera un numero aleatorio
                numero = generarNumeroAleatorio(1, 75) # 6: O(funcion), O(1)
                # Verifica que el numero no este repetido
                while estaRepetido(numero): # 7: O(?), O(0)
                        numero = generarNumeroAleatorio(1, 75) # 8: O(funcion), O(1)
                # Agrega el numero al arreglo
                numeros[i] = numero # 9: O(1), O(1)
        # Asigna los numeros al tablero
        for i in range(25): # 10: O(25), O(1)
                # Obtiene el numero
                numero = numeros[i] # 11: O(1), O(1)
                # Obtiene la columna
                columna = i % 5 # 12: O(1), O(1)
                # Obtiene la fila
                fila = i // 5 # 13: O(1), O(1)
                # Asigna el numero al tablero
                data[fila][columna] = numero # 14: O(1), O(1)
        # Asigna los numeros a las columnas
        for i in range(5): # 15: O(5), O(1)
                # Obtiene la columna
                columna = i # 16: O(1), O(1)
                # Obtiene el numero minimo
                minimo = (columna * 15) + 1 # 17: O(1), O(1)
                # Obtiene el numero maximo
                maximo = minimo + 14 # 18: O(1), O(1)
                # Asigna los numeros a la columna
                for j in range(5): # 19: O(5), O(1)
                        # Obtiene el numero
                        numero = data[j][columna] # 20: O(1), O(1)
                        # Verifica que el numero este en el rango
                        if numero < minimo or numero > maximo: # 21: O(1), O(0)
                                # Genera un numero aleatorio
                                numero = generarNumeroAleatorio(minimo, maximo) # 22: O(funcion),
                                    O(1)
                                # Verifica que el numero no este repetido
                                while estaRepetido(numero): # 23: O(?), O(0)
                                        numero = generarNumeroAleatorio(minimo, maximo) # 24: O(
                                            funcion), O(1)
                                # Asigna el numero al tablero
                                data[j][columna] = numero # 25: O(1), O(1)
        # Asigna el numero 0 a la casilla central
        data[2][2] = 0 # 26: O(1), O(1)
        # this.imprimirTablero();
        # refreshPanel()
```

Listing 4: Algoritmo generarTableroAleatorio

Linea	Operacion	Valor	Tipo	Espacio
1	=	1	bool	1
2	for	5	int	1
3	=	5	bool	5
4	for	5	int	1
5	for	5	int	1
6	==	1	bool	1
7	=	1	bool	1
8	and	1	bool	1
9	=	1	bool	1
10	break	1	None	0
11	not	1	bool	1
12	for	5	int	1
13	=	5	bool	5
14	for	5	int	1
15	for	5	int	1
16	==	1	bool	1
17	=	1	bool	1
18	and	1	bool	1
19	=	1	bool	1
20	break	1	None	0
21	not	1	bool	1
22	=	5	bool	5
23	=	5	bool	5
24	for	5	int	1
25	for	5	int	1
26	==	1	bool	1
27	=	1	bool	1
28	==	1	bool	1
29	=	1	bool	1
30	and	1	bool	1
31	=	1	bool	1
32	and	1	bool	1
33	=	1	bool	1
34	not	1	bool	1
35	=	4	bool	4
36	for	4	int	1
37	for	5	int	1
38	==	1	bool	1
39	=	1	bool	1
40	==	1	bool	1
41	=	1	bool	1
42	==	1	bool	1
43	=	1	bool	1
44	==	1	bool	1
45	=	1	bool	1
46	and	1	bool	1
47	=	1	bool	1
48	if	1	None	0
49	!=	1	bool	1
50	=	1	list	1
51	not	1	bool	1
52	append	1	None	0
53	=	1	list	1
54	else	1	None	0
55	=	1	list	1
56	append	1	None	0
57	=	1	list	1
Total		115		70

Cuadro 1: Complejidad temporal de la funcion verificarGanador

Linea	Operacion	Valor	Tipo	Espacio
1	for	5	int	1
2	for	5	int	1
3	==	1	bool	1
4	=	1	bool	1
5	=	1	bool	1
Total		13		5

Cuadro 2: Complejidad temporal de la funcion estaRepetido

Linea	Operacion	Valor	Tipo	Espacio
1	=	1	int	1
Total		1		1

Cuadro 3: Complejidad temporal de la funcion generarNumeroAleatorio

Linea	Operacion	Valor	Tipo	Espacio
1	=	1	int	1
2	=	1	int	1
3	=	1	int	1
4	=	1	list[int]	25
5	for	25		1
6	=	1	int	1
7	while	?		0
8	=	1	int	1
9	=	1	int	1
10	for	25		1
11	=	1	int	1
12	=	1	int	1
13	=	1	int	1
14	=	1	int	1
15	for	5		1
16	=	1	int	1
17	=	1	int	1
18	=	1	int	1
19	for	5		1
20	=	1	int	1
21	if	1		0
22	=	1	int	1
23	while	?		0
24	=	1	int	1
25	=	1	int	1
26	=	1	int	1
Total		80 + ? + ?		47

Cuadro 4: Complejidad temporal de la funcion generarTableroAleatorio