UNIVERSIDAD SAN FRANCISCO DE QUITO

COLEGIO: CIENCIAS E INGENIERÍAS

ESTRUCTURAS DE DATOS

Unit testing: evaluación de algoritmos y tiempos de ejecución

Christian Eduardo Santamaría López (00215605)

Edwin Xavier Jaramillo Sandoval (00320842)

Juan Diego Vaca Maya (00205778)

Johana Belén Duchi Tipán (00321980)

Profesor: Alejandro Proaño

13 de diciembre 2022

Tema: Unit testing: Evaluación de algoritmos y tiempos de ejecución.

Introducción

Los algoritmos son parte fundamental dentro de la programación, estos comprenden todas las operaciones que un programador diseña para que un programa las ejecute y logre cumplir con el propósito de la aplicación, en ocasiones, el código que da vida a los algoritmos puede ser sumamente extenso, como dentro de aplicaciones complejas, es justamente en estos casos que probar la funcionalidad de ciertos segmentos de código puede ser complicado y llevar demasiado tiempo.

Por esta razón, se debe buscar alternativas más eficientes de hacerlo, para determinar la funcionalidad y la eficiencia de un segmento de código. En el análisis de algoritmos, una de las misiones básicas que debe realizar un programador, cuando resuelve un problema es atender no solo la necesidad sino también resolver el problema de la manera más óptima; de ahí surge la interrogante de ¿cómo hacer un código más eficiente?, una de las formas es analizando el tiempo que se demora en ejecutarse, otra qué estructuras son las más adecuadas al problema en cuestión y, además, encontrar la vulnerabilidad en el código para evitar ingresos de datos erróneos, respuestas incoherentes, fuga de memoria, entre otros.

Objetivos

Objetivo General

Desarrollar un programa que reciba código fuente, lo examine y registre los resultados de esas pruebas, a través del lenguaje de programación Python, para obtener un código mucho más eficiente.

Objetivo específico

1. Evaluar la correctitud y eficiencia de propuestas de algoritmos como resolución de los problemas planteados en los deberes enviados a lo largo del presente curso, mediante el uso de pruebas unitarias (Unit Testing), para comprobar el funcionamiento del código.
2. Analizar el proceso de importación dado un directorio mediante el uso de librerías propias de python, para poder enviarlo a las pruebas unitarias.

3. Crear un registro de las pruebas unitarias, mediante estructuras de datos para comparar algoritmos y obtener el mejor en términos de tiempo.

Motivación

Se pretende que, al finalizar este proyecto, la idea sea clara en cuanto a cómo funcionan los Unit Tests, utilizados en diferentes empresas para la elaboración conjunta de software, qué estructuras de datos tienen un mejor performance para determinadas tareas, qué casos pueden mejorar o empeorar el performance de un algoritmo en base a la estructura utilizada, y, finalmente, qué estructura de datos funciona mejor para guardar registros según criterios previamente definidos

Componentes principales

1. Elaborar un programa que reciba una función (considerando un conocimiento previo de los parámetros de entrada y qué debería retornar), se le aplique una determinada cantidad de pruebas, sencillas y robustas, a fin de determinar si la implementación ha sido correcta, considerando todos los posibles casos.
2. Determinar los tiempos de ejecución de los algoritmos evaluados, sus resultados, la estructura de datos utilizada y guardarlos con la finalidad de presentar al usuario las pruebas que se han realizado y, con esa información, evalúe qué algoritmo resulta mejor.

Análisis previo de herramientas necesarias

1. Para el desarrollo del proyecto se necesitará aplicar Unit Testing, un método de chequeo de software, en el que unidades individuales de código fuente son evaluados para determinar si están aptos para ser utilizados. El objetivo es diseñar e implementar los criterios/resultados que son útiles para verificar si la unidad de código puesta a prueba es correcta. Cualquier fallo en el algoritmo evaluado será reportado dentro de un resumen al final de la inspección. Para esto utilizaremos la librería de Python unittest, la cual brinda las herramientas necesarias para una correcta implementación de nuestras ideas.
2. Otro punto importante es cómo “importar” y pasar como argumento los algoritmos a

evaluar. Existen varias formas de hacerlo, sin embargo, una de las más sencillas es buscar y ejecutar un script de Python que contenga únicamente la función que se quiere testear. Una librería que ayuda en buena manera es runpy, ya que cuenta con funciones de localización y ejecución módulos de Python.

3. Dado que se busca tener un registro de las pruebas realizadas, dividiremos los tests de acuerdo al nombre del algoritmo (esto es posible ya que de antemano se conocen los nombres de las funciones), en estas divisiones buscaremos ordenar los registros de la forma más óptima posible dentro de un fichero de texto. Esta última parte se decidirá durante la elaboración del proyecto, debemos comparar la eficiencia del uso y ordenamiento de los registros en los diferentes tipos de datos aprendidos en la clase; nuestras conclusiones se verán reflejadas en el programa final y se expondrá su debida justificación.

Análisis posterior a la aplicación de las herramientas

1. **Unit Test**

Para comprobar y examinar el correcto funcionamiento del código se eligió Unit Test porque permite fragmentar el código en pequeñas unidades que ayudan a determinar posibles errores o vulnerabilidades en el código (Geek for geeks, 2022).

Por lo tanto, para hacer un unit test efectivo, se necesita utilizar el método deductivo, que permitirá obtener los criterios más útiles para el análisis.

Para realizar estas pruebas es importante analizar el contexto del programa, que requerimientos se tiene y así elegir la prueba que se acople mejor a las necesidades del programador. Existen 4 tipos de pruebas: primera generación, segunda generación, parametrizadas e impulsadas por propiedades.

Las primeras consisten en generar excepciones y validar respuestas, para ello se utiliza el módulo assert que nos indica si la prueba falló más no en dónde ni cómo falló, por este motivo, aunque es una forma sencilla de desarrollar no es recomendable usarla debido a que si se realizan estas pruebas se requiere entender cuál fue el problema con el código enviado.

Debido a este motivo surgen las pruebas de segunda generación, que usan unittest.TestCase con módulos como assertTrue() y assertEqual(), assertRaices(), que

además de comprobar el código, proporcionan un informe en el caso de que falle en alguna de las pruebas. Sin embargo, probar con un número en específico aparte de ser tedioso, no es suficiente para detectar fallos, y así surge las pruebas parametrizadas que consisten en crear un conjunto finito de valores. A pesar de ser un buen método de prueba, aún se puede construir algo más riguroso, como las pruebas impulsadas por propiedades usando la hypothesis, given y list que generan un conjunto finito pseudoaleatorio que se comparará con una propiedad invariable independientemente de los datos prueba, por ejemplo, el unit test de Fibonacci utiliza la fórmula de Binet que permite encontrar el n término de la secuencia, independientemente de la entrada que se le dé, otro ejemplo del uso en nuestro proyecto es el test de factorial que compara usando la definición de que el factorial de un número dividido por el factorial de un número menos uno nos da el valor de la entrada.

En el proyecto se realizaron pruebas unitarias de segunda generación, pruebas parametrizadas y pruebas impulsadas por propiedad, distribuidas de la siguiente manera:

- Pruebas de segunda generación: Test_MergeArrays, Test_MergeLL, Test_Reverse_LL, Test_Sum.
- Pruebas parametrizadas: Test_MergeArrays, Test_MergeLL, Test_Reverse_LL.
- Pruebas impulsadas por propiedades: Test_Factorial, Test_Fibonacci, Test_Insertion_Sort, Test_Merge_Sort, Test_Quick_Sort.

2. Importar archivos

Parte fundamental del proyecto consistió en importar archivos desde la carpeta nombrar ‘FilestoTest’ al main de la aplicación. Esto debido a que dentro de dicha carpeta se encontraban el código fuente al que se buscaba realizar un Unit Test. Para realizar esto se utilizó el módulo os, especializado en la lectura y escritura de archivos (Python Software Foundation, 2022b). Dentro del módulo mencionado, la función más utilizada fue open ().

La función open () se encarga abrir el archivo y retornar un objeto tipo archivo en la aplicación que se está ejecutando, para su manipulación posterior. Un ejemplo de la

aplicación de esta función se observa en la línea 101 del archivo main.py del presente proyecto en GitHub. La función open() recibe varios argumentos, siendo los esenciales: path y mode. El path especifica la dirección del archivo en el directorio de la máquina, y es precisamente por este argumento que, antes de ejecutar esta línea de código, se le pide al usuario que el código que busca testear esté específicamente en la carpeta 'FiletoTest' dentro del mismo directorio del archivo 'main.py'.

El segundo argumento de la función, mode, permite especificar cuál es el objetivo del usuario con dicho archivo. En caso de utilizar 'r' como argumento, entonces el archivo se abre únicamente para lectura. En cambio, en caso de utilizar 'w', es posible escribir en el archivo (Python Software Foundation, 2022a).

Posteriormente a la apertura del archivo, en nuestro código se envía dicho archivo al código fuente correspondiente al Unit Test que se haya escogido. En el caso de la línea 101, se envía el contenido del archivo a la función Test_Sum.main() (línea 102), que se encarga de ejecutar el Unit Test para dicha función en específico. Cabe recalcar que el usuario previamente escogió qué función es la que desea poner a prueba en el menú que inicia desde la línea 93, por lo que esta lógica se repite para cada uno de los casos definidos.

Por otro lado, también se utiliza la función read() en el manejo de archivos de nuestro programa. A diferencia de la función open (), esta es más simple. Una vez que se abre el archivo, se puede utilizar la extensión. read() para leer el contenido del archivo abierto. Un ejemplo se puede observar en la línea 185 del main.py. La función únicamente toma un argumento, que es el número de bytes que se desean leer. En caso de que se quiera leer la totalidad del archivo, el argumento es -1, que es lo que se puede observar en dicha línea.

Por último, dentro del manejo de archivos de nuestra aplicación, se utiliza el método close(), como su nombre lo indica, este archivo se encarga de cerrar el archivo, dejando de permitir que este se pueda leer o escribir. A pesar de que Python automáticamente cierra un archivo una vez que la variable utilizada para abrirlo se reescriba, se considera que es una buena práctica de programación cerrarlos (Tutorials Point, 2022).

3. **Registro**

Dado que se buscaba tener un registro de las pruebas realizadas, se dividió los tests de acuerdo con el nombre del algoritmo. En estas divisiones se definió una función que se encarga de guardar estos archivos. Para esto se utilizó algunos métodos como lo son: TestLoader(), la cual es una clase de Python que carga los casos de prueba creados localmente como un archivo. Es decir, crea conjuntos de tests a partir de las clases divididas previamente. Esto nos ayuda a generar una lista de los errores que se consideran no fatales durante la carga de los tests.

De igual manera se utilizó LoadTestsFromModule(), el cual devuelve todos los casos de test contenidos en el módulo dado. Este método busca en module clases derivadas y crea una instancia de la clase para cada método de test definido. Por último, TextTestRunner() el cual es una implementación básica del test runner que produce resultados en una corriente (Python Software Foundation, 2022).

En donde Test Runner nos brinda una interfaz ejecutable para la ejecución de pruebas y entrega los resultados al usuario. En nuestro caso nos devuelve un código estándar para notificar los resultados de la ejecución de la prueba. (Python Software Foundation, 2022).

Estructuras de datos utilizadas

1. **Código a testear**
    - Arrays - Insertion_Sort, Merge_Sort, Merge_arrs, Quick_Sort
    - Linked List - Merge_Linked_List
2. **Unit Test**
    - Arrays - Test_MergeArrays
    - Linked List – Test_MergeLL, Test_Reverse_LL
3. **Main**

    Se usó en la muestra de los registros.
    - Diccionarios (Hash Tables) - def by_author_files(files)
    - Arrays - def getfiles(dir), def good_files, def find_index(scores), def get_scores(files), def regOpts(files, basepath)

Análisis de complejidad

Nota: Si se suman las complejidades al final nos importa el peor de los casos por lo que puede simplificarse, ejemplo: $O(n+2)$, la complejidad de la constante se puede obviar dándonos como resultado una complejidad de $O(n)$.

1. **Código a testear**

   Tabla No. 1

   Análisis complejidad (código a testear)

   Fuente:https://github.com/ChrisEduS/cmp-3002-fall22-csantamarial/tree/main/Final%20Proyect/DataStructuresFinalProject/FilestoTest

   Descripción: Cálculo de complejidad de tiempo y espacio del código que va a ser testeado.

Código	Complejidad de tiempo	Complejidad de espacio
Factorial (recursión)	$O(n)$	$O(n)$
Fibonacci (recursión)	$O(2^n)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(1)$
Merge Linked Lists*	$O((n+m)\log(n+m))$	$O(n+m)$
Merge Sort	$O(nlogn)$	$O(n)$
Merge Arrays*	$O((n+m)\log(n+m))$	$O(n+m)$
Quick Sort	$O(n^2)$	$O(n)$
Suma	$O(n)$	$O(1)$

   *Merge linked list y Merge arrays usan dos listas/arreglos ordenados, los unen y ordenan por eso se suma la complejidad de la una lista más la complejidad de la otra lista.

2. **Main**

   Llamar una función dentro de otra tiene una complejidad de $O(1)$ y como estamos buscando el peor de los casos al simplificar el cálculo esta no marcará representativamente.

Tabla No. 2

Análisis complejidad (main)

Código	Justificación		Complejidad de tiempo	Complejidad de espacio
by author files	sort() for() reversed()	$O(n^2)$ $O(n^2)$ $O(n)$	$O(n + 2n^2)$ $= O(n^2)$	$O(1)$

get files	for()
	if ()
	append()
		$O(n^2)$
		$O(n)$
		$O(1)$
			$O(n + n^2)$
			$= O(n^2)$
				$O(1)$

good files	for()
	if ()
	append()
		$O(n^2)$
		$O(n)$
		$O(1)$
			$O(n + n^2)$
			$= O(n^2)$
				$O(1)$

find index	for()
	if ()
	append()
		$O(n^2)$
		$O(n)$
		$O(1)$
			$O(n + n^2)$
			$= O(n^2)$
				$O(1)$

get scores	for()	$O(n^2)$	$O(n^2)$	$O(1)$
def menu	while() if ()	$O(\log n)$ $O(4n)$	$O(n)$	$O(1)$
def dispCases	for()	$O(n^2)$	$O(n^2)$	$O(1)$
def newTestw	while() if ()	$O(\log n)$ $O(10n)$	$O(n)$	$O(1)$
def orderofreg	Solo tiene print	$O(1)$	$O(1)$	$O(1)$

def regOpts	for()
	while()
	if ()
	sorted()
	reverse()
		$O(3n^2)$
		$O(\log n)$
		$O(4n)$
		$O(2\left(\frac{n^2-1}{2}\right))$
		$O(n)$
			$O(3n^2 + \log n + 4n + n^2 - 1 + n) = O(n^2)$	$O(1)$

defpRegistersW	while()	$O(\log n)$	$O(n)$	$O(1)$

	if () $O(19n)$		

Fuente:https://github.com/ChrisEduS/cmp-3002-fall22-csantamarial/blob/main/Final%20Proyect/DataStructuresFinalProject/main.py

Descripción: Cálculo de complejidad de tiempo y espacio del main.

Evidencia de la implementación

https://github.com/ChrisEduS/cmp-3002-fall22 csantamarial/tree/main/Final%20Proyect

Conclusiones y resultados de aprendizaje

1. Se puede concluir que, las pruebas de testeo forman parte de las mejores prácticas porque nos brindan una primera cara del análisis de eficiencia y vulnerabilidad de un código, ya que se centra en funciones específicas del mismo y con ello resalta los riesgos del algoritmo, tales como valores erróneos o cálculos incorrectos. Estas pruebas son de vital importancia porque se puede corregir desviaciones del código, además, permite comprender el funcionamiento del programa, sin embargo, no detectan todos los errores en un programa. Se recomienda usar pruebas impulsadas por propiedades antes que de segunda/primera generación ya que involucra características probadas e invariables.
2. Con respeto al manejo de archivos dentro del proyecto, consideramos que la parte más sensible es la forma en la que estos son manejados internamente en la aplicación, la opción de mostrar por fecha de creación, de nuevo a antiguo, o de antiguo a nuevo, debe considerar que cada una de estas va a tener una dirección distinta, por lo que se debe prestar suma atención al realizar la acción de lectura. Además, se debe tener cuidado de que todos los archivos funcionen correctamente, y sobre todo mantener el orden de la aplicación y la minuciosidad durante el desarrollo de la misma.
3. Para guardar los registros se tuvo que implementar la misma función aplicada en cada clase que se quería testear, unittest nos brindaba esta opción la cual era mucho mejor que otras que se tuvieron en mente como guardar en archivos texto usando otras librerías. Se recomienda tener presente los parámetros que se deben enviar a la función y a cada método, así como la dirección de memoria con la que estamos trabajando.

Referencias

Geeksforgeeks.org. (2022). *Pruebas unitarias | Pruebas de software*. Recuperado de: https://www.geeksforgeeks.org/unit-testing-software-testing/

Python Software Foundation. (2022a). *Funciones Built-in.* Retrieved from https://docs.python.org/es/3.10/library/functions.html#open

Python Software Foundation. (2022b). os — *Interfaces misceláneas del sistema operativo.* Recuperado de: https://docs.python.org/es/3.10/library/os.html

Python Software Foundation. (2022). *unittest — Unit testing framework.* Retrieved from https://docs.python.org/3/library/unittest.html

Tutorials Point. (2022). *Python File close() Method.* Retrieved from https://www.tutorialspoint.com/python/file_close.htm