# Assignment 1 – Michael Floerchinger

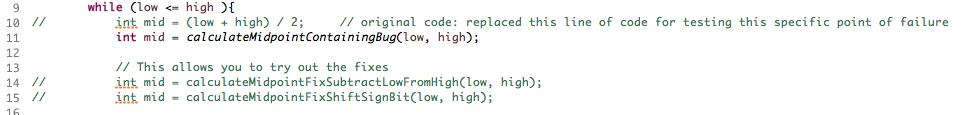
Assignment code:

<https://github.com/sonikp/BinarySearch>

# Task 1: Identify Error

Identified error in the provided code at line 6, where the code calculates the mid point value by adding the low + high value together. In instances where each of the values is close to Integer.MAX\_VALUE, the total sum is greater than Integer.MAX\_VALUE, and therefore the sequence of number rolls over to a negative value.

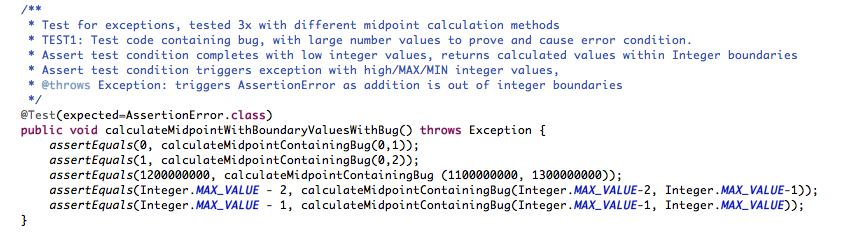
int mid = (low + high) / 2;



***ScreenShot: Code with bug as it has been provided***

Test: This is identified in the JUNIT Test, BinarySearchTests.java, test called:

calculateMidpointWithBoundaryValuesWithBug()

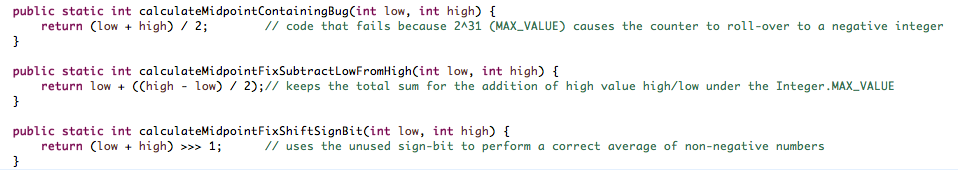


***ScreenShot: JUNIT test code which causes an error to occur in the buggy code***

The first two tests add two numbers where the total is below Integer.MAX\_VALUE

The third test adds 1100000000 – 1300000000, where the sum before being divided by /2 is greater than Integer.MAX\_VALUE, causing the test to fail.

These tests were run aimed directly at the failing and remediated code, which calculates the midpoint value.



***ScreenShot: Three methods created to test the mid-point calculations with specific JUNIT tests***

# Task 2: Correct the code (i)

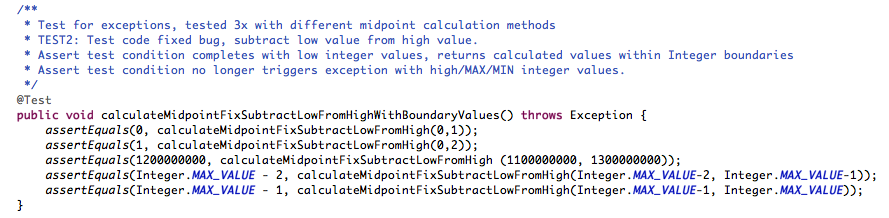
The remediated code aims to ensure that during the operation of finding the midpoint value, the sum of the two LOW and HIGH value integers is performed in a way that the total sum of the two values does not exceed Integer.MAX\_VALUE. This is achieved using two possible solutions:

int mid = low + ((high - low) / 2);

Here the high is subtracted from the low value before the total is then re-added to the low value. From this, the maximum value of any part of this equation never exceeds the Integer.MAX\_VALUE limit.

Test: This is test in the JUNIT Test, BinarySearchTests.java, test called:

calculateMidpointFixSubtractLowFromHighWithBoundaryValues()



***ScreenShot: JUNIT tests for testing first fix***

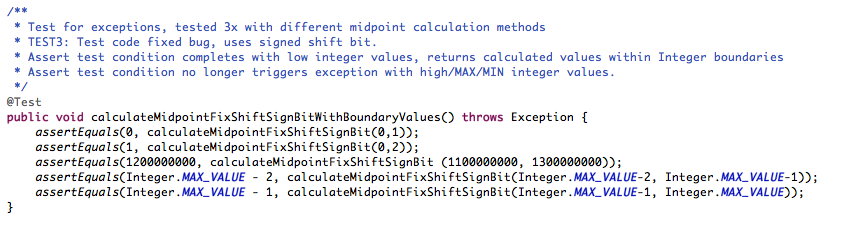
# Task 2: Correct the code (ii)

The second remediation solution is to use the unused sign-bit to perform a bit shift of the numbers therefore representing the /2 function by shifting the numbers to lose the least-significant-bit and thereby enabling the calculation without rotating the sum into a negative number.

int mid = (low + high) >>> 1

Test: This is test in the JUNIT Test, BinarySearchTests.java, test called:

calculateMidpointFixShiftSignBitWithBoundaryValues()



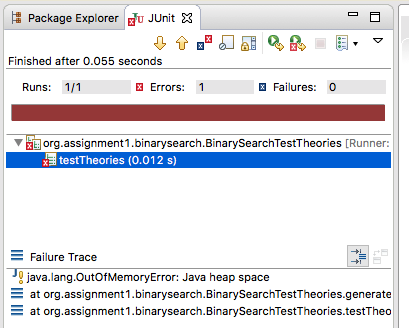
***ScreenShot: JUNIT tests for testing second bit-shift fix***

# Task 3: Experimenting with Heap Size

In order to perform some of these tests, it was required to reach the limits of the Integer.MAX\_VALUE for array size. This was achieved by generating a JUNIT test that created an array, where the arraySize was = to Integer.MAX\_VALUE.

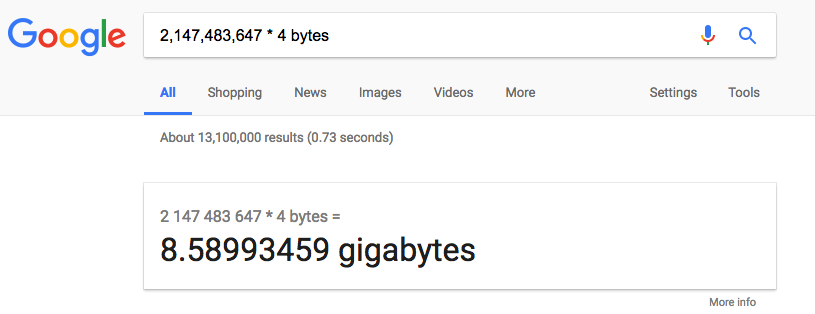
The JUNIT BinarySearchTestTheories.java was developed to create random array sizes up to Integer.MAX\_VALUE, and to populate the array with random values ranging from 0 – Integer.MAX\_VALUE. Also a key value was generated, either one that was contained in the randomArray, or a key value that was not.

At first when the tests executed for arraySizes that were close to the limits of Integer.MAX\_VALUE, eclipse would return Java Heap space java.lang.OutOfMemoryError’s.



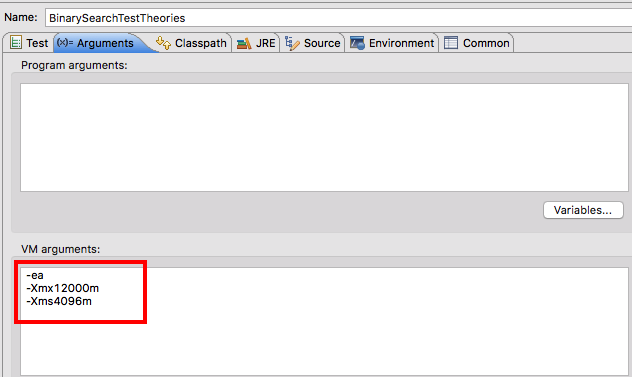
***ScreenShot: JUNIT test out of memory error***

To resolve the heap memory errors, it was necessary to calculate the required space and to set that within the eclipse console. An integer is 4 bytes in size, so the required space for an array of size Integer.MAX\_VALUE is:



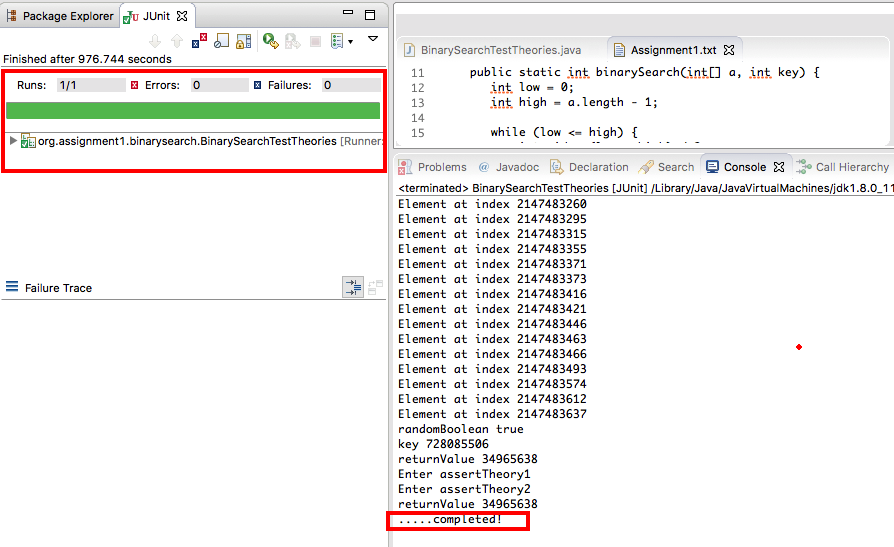
***ScreenShot: Calculating how much memory an Integer.MAX\_VALUE array would use***

Applying changes and memory tuning to the configuration settings, identified a setting that would no longer result in the heap memory errors:



***ScreenShot: Tuning memory settings***

Tests Run successfully with the updated memory allocation.



***ScreenShot: Successfully tuned memory settings***