Given:

$g(x) = \cos(x) - x^3$ {has root on (0, 1)}

Root $r \sim 0.865474$

Find answer within 4 decimal places; Satisfy $|x_c - r| < 0.5 \times 10^{-4}$

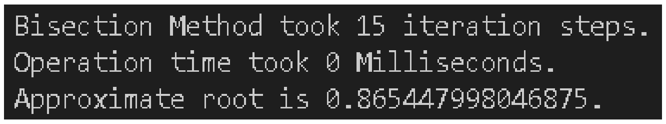
Problem Statement 1: Use the Bisection method with a given internal $a_0 = 0$ and $b_0 = 1$.

Algorithm Description: Given 2 initial internals, I will split it into 2 intervals via their midpoint and update the internals and midpoint accordingly based on what interval the function changes sign. The point is to help pin down the value of root by using smaller intervals and close enough approximation of the solution via the midpoint, which will start converging when comparing with previous midpoints such that it is within 4 decimal places and only then will the midpoint be returned as the solution. Lastly, I will keep track of how good the algorithm is based on how many iterations of this method were conducted and in how much time the program took to run by keeping track of system time before and after the program execution for performance purposes.

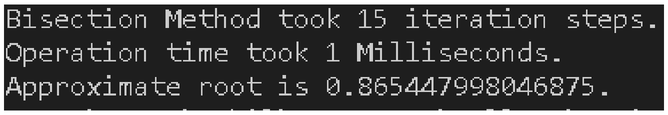
Pseudocode: (I have 0 experience, I just wrote it in readable python-ish format)

```
Function bisection(internal1, internal2, PreviousSolution, count):
    newMidpoint = (internal1 + internal2) / 2

    If abs(newMidpoint - previousSolution) < tolerance AND count <
1:
        print("Bisection method took " + count + iterations.")
        return newMidpoint

    If g(internal1) * g(newMidpoint) < 0:
        internal2 = newMidpoint //root on left interval
    Else:
        internal1 = newMidpoint //root on right interval

    Return bisection(internal1, internal2, newMidpoint, count + 1)
End

Function g(x):
    Return cos(x) - x^3
End
```

Results: (p11.java)



```
Bisection Method took 15 iteration steps.
Operation time took 0 Milliseconds.
Approximate root is 0.865447998046875.
```



```
Bisection Method took 15 iteration steps.
Operation time took 1 Milliseconds.
Approximate root is 0.865447998046875.
```

Observations & Analysis

The root estimated by *p11.java* is approximately 0.865448 which is within 4 decimal points of the approximately correct root of 0.865474. As for the program efficiency, I would say it is sufficient as it only took no more than a millisecond to conduct 15 iterations of 15 floating point operations at most (last iteration could be just a few less operations returning on first if statement) on each iteration which is about 225,000 operations per second. As for the algorithm, I can attempt to approximate runtime as the point at which the division of the interval results in a midpoint being within 4 decimal places of the prior midpoint as each pair of intervals of each iteration will be approximately equally sized where for simplicity we can use the smaller internal and midpoint for calculations.

First Iteration midpoint: $(a+b) / 2$
Second Iteration: $(a + (a+b)/2) / 2 = (3a + b) / 4$
Third Iteration: $(a + (3a + b)/4) / 2 = (7a + b) / 8$
Fourth Iteration: $(a + (7a + b)/8) / 2 = (15a + b) / 16$
…
Formula for n iterations: $(2^i - 1)a + b) / 2$

We want n such that difference on nth and n-1 th iteration is less than 0.00005:

$$(2^i - 1)a + b) / 2 - (2^{(i-1)} - 1)a + b) / (2^{(i-1)}) <= 0.00005$$
$$(a2^i - a + b - a2^i + 2a + 2b) <= 2^i * 0.00005$$
$$(a + 3b) <= 2^i * 0.00005$$
$$\log_2 (a + 3b) <= \log_2 2^i + \log_2 (0.00005)$$
$$\log_2 (a + 3b) <= i -14.2877$$
$$i >= 14.2877 + \log_2 (a + 3b)$$

This means that given two internals a and b, the algorithm will be executed in about $O(\log_2(a+3b))$ runtime with other factors being not as necessary as they don't scale as much for large values compared to 2 and 3 in the Big-O expression. If my analysis was accurate enough, any $O(\log(m+n))$ or $O(\log(n))$ algorithm is a good algorithm. I am guessing there are ways to write algorithms using bitwise operators for probably faster results and other more ways unknown to me that could result in faster runtimes. This program is scalable to any predefined internal point generally and runs with any java configured environment.
