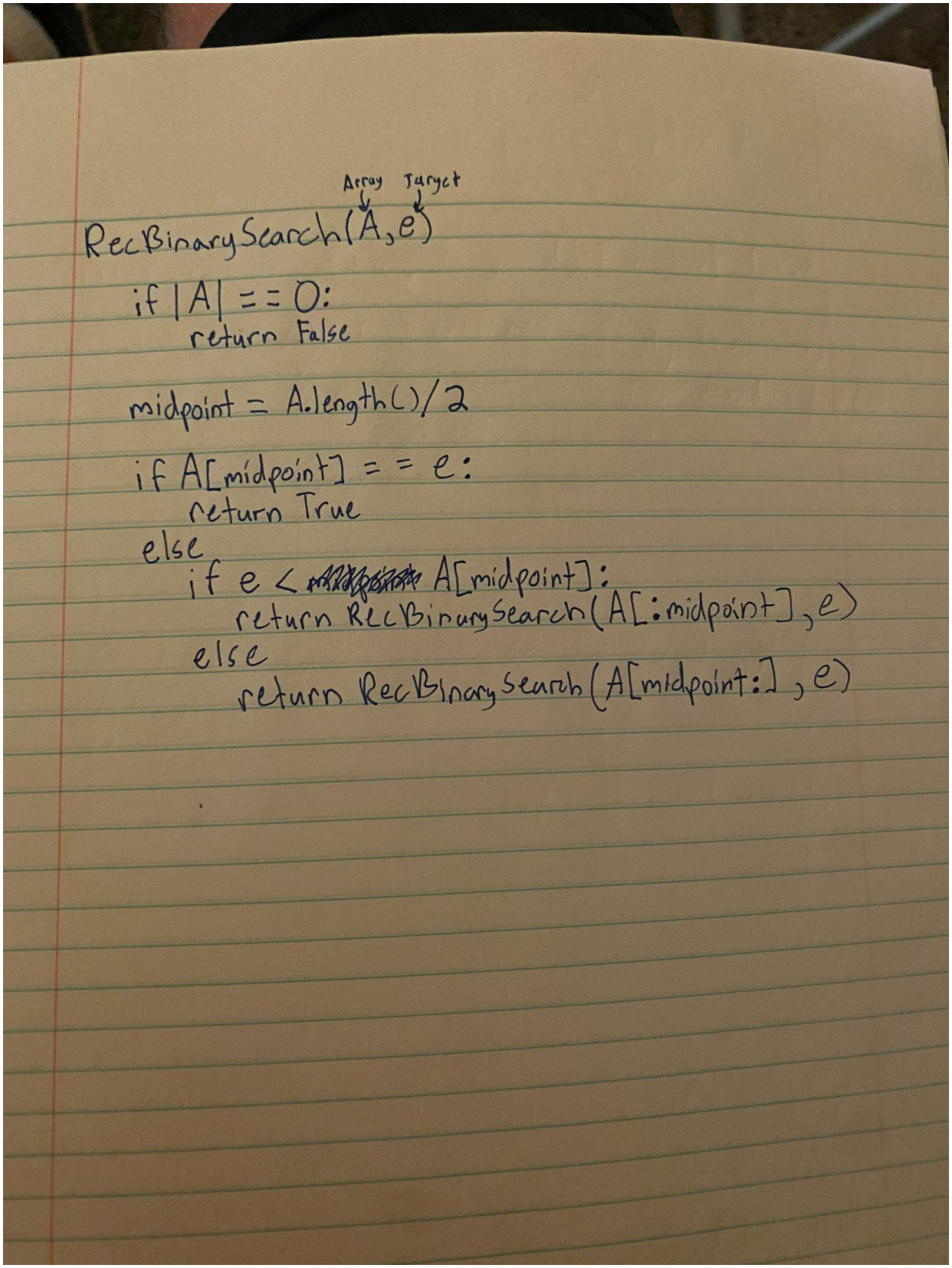
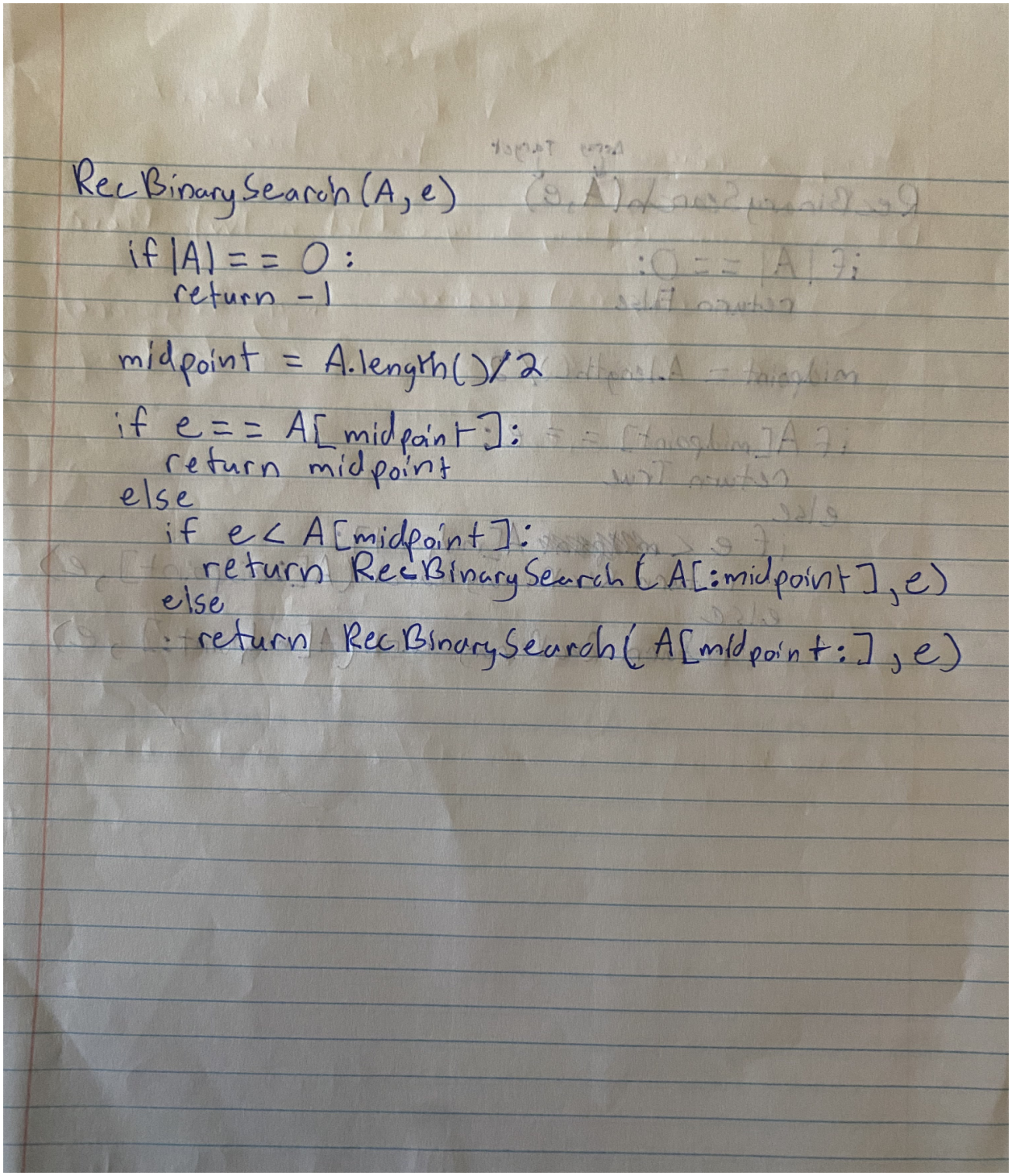
Recursive Binary Search From Class:



```
                    Array  Target
                      ↓     ↓
RecBinarySearch(A, e)
    if |A| == 0:
        return False

    midpoint = A.length()/2

    if A[midpoint] == e:
        return True
    else
        if e < A[midpoint]:
            return RecBinarySearch(A[:midpoint], e)
        else
            return RecBinarySearch(A[midpoint:], e)
```

The implementation above from class does not utilize a functional approach, meaning that instead of returning the index of the target if it is in the array, the algorithm instead returns a Boolean confirming if the target is or is not in the array. While this is a valid approach, to identify all bugs we must adjust the above pseudo-code so that it works functionally. Below is the adjusted pseudo code:



```
RecBinarySearch(A, e)
    if |A| == 0:
        return -1
    midpoint = A.length()/2
    if e == A[midpoint]:
        return midpoint
    else
        if e < A[midpoint]:
            return RecBinarySearch(A[:midpoint], e)
        else
            return RecBinarySearch(A[midpoint:], e)
```

Bugs:

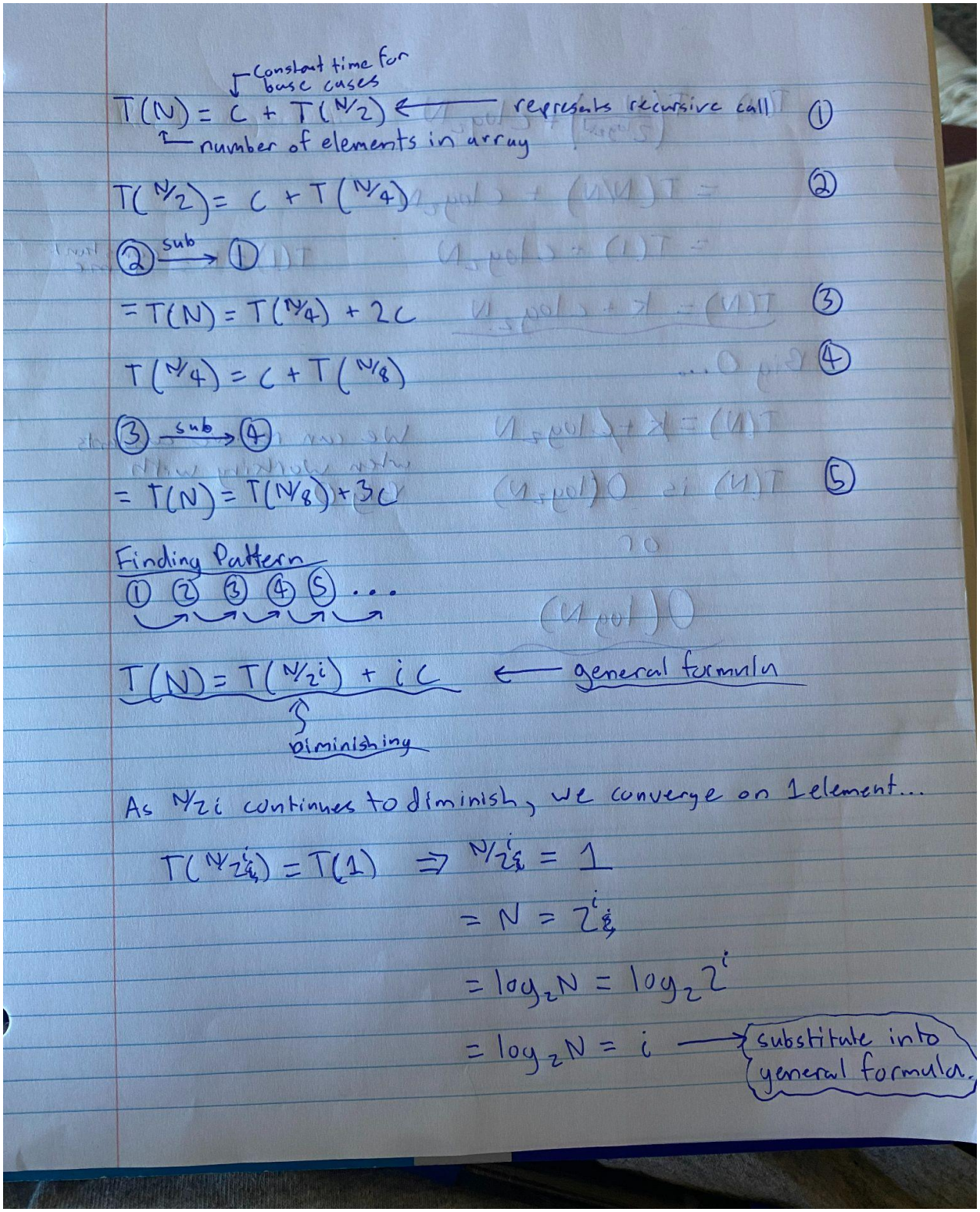
1) The first bug we identified was that the Recursive Binary Search would not work for a null integer array input. Because we use a method to get the length of the array A, we will get an error attempting to find the length of A when it is null.
2) The second bug we identified was in the returning of the proper index when the target variable was present in the array. In the recursive step of our original pseudo code, we returned a new array that was half the size of the previous step. Because we were returning a new array our midpoint would be different every time the method was recursively called. For example, if our array was A = {1,2,3,4,5,6,7,8,9,10} and our target variable was 4, our original pseudo code would return 0, instead of 3. By tracking the size of the array and midpoint as we search for the target variable we see why.**1** Size 10, midpoint = 5, A[5] = 6.**2** Size 5, midpoint 2, A[2] = 3.**3** Size 2, midpoint = 1, A[1] = 5.**4** Size 1, midpoint = 0, **A[0] = 4**. Return 0. In our fixed pseudo code we address this issue, only referencing the original array when looking to return the target if it is found in the array.
3) The third bug we identified was minor, and came in the step when we got the midpoint through division. Because we were using floor division in our original pseudo code, sometimes the midpoint would not be completely accurate. This had the potential to mess up the output depending on the input. This could be fixed through implementing a different midpoint scheme, or adjusting for the fact that division will always floor the result.

Fixed Pseudo Code

```
RecBinarySearch(A, e, l, r)
    if A == null:
        return -1

    if l > r:
        return -1

    midpoint = l + (r - l) / 2

    if e == A[midpoint]:
        return midpoint
    else if e < A[midpoint]:
        return RecBinarySearch(A, e, left, midpoint - 1)
    else:
        return RecBinarySearch(A, e, midpoint + 1, r)
```

Time Complexity:

The time complexity of the recursive binary search is O(log(n)):



$$T(N) = T\left(\frac{N}{2^{\log_2 N}}\right) + c\log_2 N$$

$$= T(N/N) + c\log_2 N$$

$$= T(1) + c\log_2 N \qquad T(1) = K = \text{constant time}$$

$$\underline{T(N) = K + c\log_2 N}$$

Big O...

$T(N) = \cancel{K} + \cancel{c}\log_2 N$ — We can remove constants when working with Big O.

$T(N)$ is $O(\log_2 N)$

or

$$\underline{O(\log N)}$$