2. Test whether a recursive, iterative or linked-type binary search is faster by testing it on arrays of size 1 million, 10 million, and 100 million with arrays that are filled with random numbers going from smallest to largest.

Attacking this problem, I decided to make the algorithms for the recursive and iterative search first. I have seen it before, so it was not hard to understand when looking it up. The recursion algorithm has 4 parameters to pass, the array, start of the array, end of the array, and the element we are searching for. Since we are going to use the function to adjust the side that will be searched, we need them in the parameters. We first get the middle index of the array which we use the start + end and we divide that answer by 2. From there we check if the array at that index is our element. If so, we return the index. If not, we go down to the if statements and we check to see what side of the array we will check next. If the element is less than the mid then we call the function again, but we change the end index of the array to be the mid-1. If its greater than we make the start mid+1. We run this function till we get the array[mid] to equal the element. If it does not exist, then the function returns -1.

The iterative solution is nearly identical to the recursive, but we have less parameters and we only call the function once. We must create the start and end variables in the function to change where we are going to check in the array. From there a while loop will keep going while start is less than or equal to end. We get the mid index and do the same as the recursion by checking the if array[mid] is greater than or less than the element and changing the start and end variables accordingly.

Linked binary search was tricky. I easily made the classes to handle the linked list but how to get the mid value in the list was not easy. After a bit of searching I got a neat solution. We have a function that works the same as the others. It checks the mid object data to see if it equals the element and it has if statements with greater than and less than the element to see which side of the list to check next. To find the mid a function with two pointers were used to go through the array along with 2 other pointers to keep track of the side of the array we are checking. A fast-moving pointer and a slow pointer that goes through the array till the end. The fast pointer moves twice as fast as the slow. When the fast pointer reaches the end the slow should be at the middle of the array since its moving half as fast. From there the slow pointer is returned and checked trough the if statements. From there the two pointers holding the position in the array are adjusted to the side that we need to check next.

Finally, the final part of the program was a sort to make our random array in ascending order. I tried a bubble sort from another question, but it takes too long and caused some issues with the array size. From there I decided to use quick sort which I read was fast. From there I just needed to code the quick sort algorithm which splits the array into two sides and swaps values from the array that are less than a pivot point (middle value) and moves the values less than the pivot to the left and the values greater than the pivot to the right. From there the array is divided again into 2 and the program sorts that again. Toke a sort (bubble) which toke a minute or 2 and completed it in less than 30 sec.