Binary search is a way of searching through a list of numbers that is way faster than normal native search. By native search, I mean looping through the list and then checking if the individual items on each iteration match the target. Binary search takes advantage of the fact that the list of numbers is sorted. If we were to have a function that helps us search for a number in a list it would look like this.

def native\_search(l, target):

    # example l = [1, 3, 10, 12]

    for i in range(len(l)):

        if l[i] == target:

            return i

    return -1

Our function takes in two arguments, the first one is the list “l” and the second one is the target. We loop through and if we get our target, we return the index, if we don’t get the target, it means our target is not in the list and we return -1. <br />

Binary search on the other hand is basically divide and conquer. If we get the middle of the list and then the value there doesn’t match the target, we can then compare if it is lower or if it is higher. If it’s higher than the target then we search again on the right side of the middle, if it is lower, we search on the left side. Defining our binary search function would look like this.

def binary\_search(l, target, low=None, high=None):

    if low is None:

        low = 0

    if high is None:

        high = len(l) - 1

    if high < low:

        return -1

    midpoint = (low + high) // 2

We add two new arguments that were not in the native search own. This would help us get our midpoint. If low is none, make it zero, because the lowest index our list can have is zero (0) and if high is none, make it len(l) – 1. If we have a five item list, our length is five (5) but the last index is four (4) hence the minus one (1). Then we calculate our midpoint by floor dividing the sum of low and high by 2 so we get a whole number we can use as index.

if l[midpoint] == target:

        return midpoint

    elif target < l[midpoint]:

        return binary\_search(l, target, low, midpoint-1)

    else:

        # target > l[midpoint]

        return binary\_search(l, target, midpoint+1, high)

If the item at that midpoint is equal to the target, then we can return that index. Else, if the item at that midpoint is higher than the target, then we return the binary search function. This means we are going to call the function again but this time, our high and low won’t be none, instead the low would still be low which at this point is zero (0) and then the high would be our midpoint minus one (1). Remember, high and low are indexes we are using to check for the target and it midpoint minus one because we already know the midpoint doesn’t match so we are reducing the boundaries. <br />

If the item at the midpoint is lower than the target, then we call the function again but this time our high would still be high and then our low would be midpoint plus one (1). So let’s say we have a list <code>l = [1, 2, 3, 4, 5]</code>, and our target is 4. When we run binary search on this list, the first time it runs, our midpoint would be 2 (5//2 = 2) and item at index position 2 is 3. Now 3 is less than 4 so our function would run again but this time, our new low would be 3 (2+1) and our high would still be 4 (5(len(l)-1). The midpoint this time would be 3 (7//2 = 3). Item at index position 3 is equal to our midpoint so we can return that midpoint (3). If at any point, our low becomes greater than our high or vice-versa, it means that the target is not in the list so we can return -1 just like we did for the native search.

def binary\_search(l, target, low=None, high=None):

    if low is None:

        low = 0

    if high is None:

        high = len(l) - 1

    if high < low:

        return -1

    midpoint = (low + high) // 2

    if l[midpoint] == target:

        return midpoint

    elif target < l[midpoint]:

        return binary\_search(l, target, low, midpoint-1)

    else:

        # target > l[midpoint]

        return binary\_search(l, target, midpoint+1, high)

If we run the two functions on a list, we would get the same index as shown below

if \_\_name\_\_ == "\_\_main\_\_":

    l = [1, 3, 5, 10, 12]

    target = 10

    print(native\_search(l, target))

    print(binary\_search(l, target))

<img src=><br />

If we also give it a target that is not in the list, we would have the same result of minus one from both functions. <br />

However, the aim is to prove that binary search is a lot faster than native search and to do this, we are going to create a large list of numbers and then sort the numbers so we can call the functions on the list. We are also going to time them. To create the large list of numbers, we would make use of a set and then convert it to a sorted list.

length = 10000

    # build a sorted list of length 10000

    sorted\_list = set()

    while len(sorted\_list) < length:

        sorted\_list.add(random.randint(-3\*length, 3\*length))

    sorted\_list = sorted(list(sorted\_list))

After creating the list, we then use the python built-in time object so we can get the time. The aim is we want to get the time when the function starts running and when it ends then we subtract start time from end time.

start = time.time()

    for target in sorted\_list:

        naive\_search(sorted\_list, target)

    end = time.time()

    print("Naive search time: ", (end - start)/length, "seconds")

    start = time.time()

    for target in sorted\_list:

        binary\_search(sorted\_list, target)

    end = time.time()

    print("Binary search time: ", (end - start)/length, "seconds")

This is going to basically make each 10,000 item in the list a target and run it each time so we can see how long it takes.

<img src=><br />

As we can see from above, the native search took about 0.000323 seconds and the binary took 5.7969e-06 which is 0.0000057967 seconds.