Ternary Search Efficiency

The goal of this project is to practice recursion, searching, and comparing the efficiency of algorithms using noisy real run-time data. This is an individual project, your submission should be your own, and certainly for the mini-report your data and graphs should be unique given the specific environment of your tests, and your answers should be in your own words. You can still get help from all of the usual suspects, and talk to your classmates about algorithms, general concepts, debugging advice, etc.

Deliverables: your solution file named `user.py` **and** a separate file named `user.xxx` containing your time data and graphs. Follow the instructions for the 2nd file carefully, you may only submit 1 extra file in addition to your .py file. Don't forget to use good coding standard including appropriate comments. In addition, you should use actual doc string descriptions for each of your functions (comments in triple quotes on the line immediately after your function definition).

Your task

Make 2 functions implementing 2 different versions of a recursive ternary search, and a main function that times each method as described below.

Ternary Search

Ternary search will be similar to binary search except instead of subdividing the list into 2 halves in each function iteration, you will subdivide it into 3 thirds. We went over in class on Wed a function that implements binary search in the same manner as the first ternary search function described below, and the book contains code for binary search implemented in the same manner as the second ternary search function. You can (and should) use these binary search functions as the start to your ternary search functions.

In both ternary search implementations, you will perform the same basic recursive algorithm, with the 3 parts as follows:

1. smaller problem that looks the same as the original problem

- search appropriate 1/3 of the list for the item.

2. base case

- base case 1: if the function is called on a list (or a section of a list) with 2 or fewer items, just check all items in list, if search item is found return True, otherwise return False.
- base case 2: if the search item is equal to either the item 1/3 of the way through the list, or the item 2/3 of the way through the list, return True, otherwise continue to recursion step.

3. recursive call on smaller version of the problem

- if the search item is less than the item 1/3 of the way through the list, recurse on the 1st 3rd of the list. If it's between the item 1/3 of the way and the item 2/3 of the way through the list, recurse on the middle 3rd of the list. If it's greater than the item 2/3 of the way through the list, recurse on the last 3rd of the list.

The difference between the 2 implementations will be how you pass the 1/3 of the list as an argument in the recursive call.

Ternary Search function 1

The first implementation will use the same method that we went over in class on Wed for a binary search. This ternary search function will take only 2 parameters, a list and an item to search for. It will search the entire given list for that item. This means that to make the recursive call, you will actually pull out the appropriate third of the original list and pass that smaller list as the first argument to the function. The best way to do this is to use a slice, e.g. for binary search we used alist[:midIndex] for the 1st half of the list, and alist[midIndex+1:] for the second half of the list. Note that the item at midIndex was **not** included in either list since it has already been checked. In the ternary search you should exclude both the items at the 1/3 index and 2/3 index from whatever third of the list you pass as an argument.

Ternary Search function 2

The second implementation will use the method discussed in the book to more efficiently perform binary search. This ternary search function will take 4 parameters, again a list and an item to search for, but now also the start and end indices defining the section of the list to search. This means that the first 2 arguments of the recursive calls never change, it is always the same list and search item passed through. What does change in each call is either the start index gets bigger or the end index gets smaller (always exactly one of those 2, never both, and never neither). The problem thus moves towards the base case by shrinking the size of the section of the list that will be searched (size of list section = end-start+1), instead of shrinking the list itself. Be careful when calculating your 1/3 and 2/3 indices that you find the values 1/3 and 2/3 of the way between the actual start and end values.

For example, if your list *SearchList* contains 100 items, and the current start index is 60 and the end index is 74, then the size of the current list section is 15 (remember it includes both ends), and your 2 indices should split the **section** of the list into 3rds of size 5 each. However, *SearchList* still has 100 items, and your indices should indicate which items to check in *SearchList,* so your actual 1/3 index is 65, and your 2/3 index is 70.

Main Function - testing and timing 2 methods for ternary search

In your main function you will measure the actual time it takes each of your 2 ternary search functions to search for the same item in the same list. Input the list length and search item as command line arguments, with the list length first and the search item second (don't forget to convert these to type *int* before attempting to use them). Then create an in-order list of numbers with the given length, and output the time it takes your 2 methods in searching that one list for the given search term. Each step is described in more detail below.

Create an in-order list of the given length

You can quickly create a sorted list of length N using the following line of code:

```
alist = [x for x in range(N)]
```

or if you want to make it more interesting, you could make a list of N ascending even numbers with

```
alist = [2*x for x in range(N)]
```

You can, in fact, put any expression as the first term in the brackets. The above line of code is the equivalent of

```
alist = []
for x in range(N):
    nextItem = 2*x
    alist.append(nextItem)
```

So anything you could set nextItem equal to would also be valid in the one-line list creation syntax. This special syntax is called a "list comprehension" and executes much much faster than the equivalent several-line loop, for reasons beyond the scope of this course. For your purposes in this HW, you will need to create very large lists and thus just need to know how to use the one-line syntax to make a basic list of a given length.

Measure the time to search the list

You will use the time() method from the Python time module to calculate the difference between the time just before calling one of your functions and the time just after the function returns the final result. To do this, first add the line "import time" to the top of your file, and then use code something like the below to determine how long one of your functions takes to complete. This assumes you have already read and converted your command line arguments (N and toSearch), and made an ordered list of length N (alist).

```
timeBefore = time.time()
found = ternaryA(alist, toSearch)
timeAfter = time.time()
print("item found?  ", found)
print("time to search: ", timeAfter - timeBefore)
```

Make sure you get the time immediately before and after calling each function. The code that creates the list itself takes longer to run than your searches, so you really don't want to include list creation in your search time measure!

You will time your functions on 5 list lengths, in a worst-case scenario, which is searching for an item that is not in the list. You could use something like -1 for your search item. The 5 test lengths will be 5, 10, 15, and 25 million. This will hopefully give you some appreciation of just how fast computers are executing instructions, you will likely see your searches complete in well under 1 second even for a list with 25 million items, unless you are doing a bunch of other stuff at the same time. A 1GHz processor executes a billion instructions per second, and that's a slow computer these days!

Remember using actual time to compare algorithms is dangerous because it depends on the exact computer you are using, what background applications are running, and many other things. To mitigate the potential noise in the time values, you should
0. Test both of your functions at once in your main() function, using just the one list and search term. So a single run of your program should create a list of the given length, then execute the above test code twice using that list, once for each function, and print the found results and times for both. This will make any comparison of the 2 methods as fair as possible.
1. Run all of your tests in a row, as quickly as possible on the same computer.
2. Perform 3 test runs for each required list length and average each function's run-time values across all 3 runs to get your "actual" time for that function with that list length.

Your final code submission should allow me to run your program with

python3 you.py *length number*

and for each of the 2 implementation methods when searching for *number* in a list of the given *length*, it will output the found result (True or False or something to that effect) and the time it took to perform that search.

Report your results

You need to somehow combine the following 4 things into a single file. You can do it all in something like excel or word or powerpoint, or you can do some parts electronically and some by hand (if you graph by hand make sure it's precise, using graph paper or some other method of making straight lines). You just have to combine all of the parts into a single file in the end **that I can open on a lab mac**. You can take pictures of parts you do on paper and copy/paste or import or whatever to get all of the pictures into one document, and you can take screenshots of different parts you do electronically, like a graph made in excel. If using a mac, there's a tool named "grab" in the utilities folder that will allow you to capture the whole screen, a window, or any selection of the screen you choose. Just ask if you have questions about how to combine your answers!

Record your exact times (you can round reasonably, but keep at least a couple of significant digits, i.e. a couple of digits that aren't just 0's) for each of the 3 individual runs on each of the 5 test list lengths for both of the 2 functions (that's 30 values). Organize these times into a table or some other readable order.

Then average each set of 3 identical tests (e.g. list length 15 million for function 1), so you have 10 final values, 5 for each ternary search implementation.

Make 2 graphs, one for each implementation, where the x axis is the list length (5 to 25 million), and the y axis is the average time to run that function on that list length.

Finally, based on the graphs, state whether the efficiency of each implementation method is approximately log(N), N, or N^2, and explain your reasoning.