Algorithms and data structures coursework

1. **Description**

My interview question is *‘Compare linear and binary search by finding a number from a randomly generated list of numbers and get a computer to sort it.’* This problem involves creating an algorithm using pseudocode and programming to generate a list of random numbers; sort the list and find the number guessed by the computer. The requirements are to create code that generates a random list of numbers between 1 and 100; the applicant must then sort the list of integers in ascending order, before finding the number. This will use the list data structure, iteration and sorting.

The requirements for the applicant include having a basic understanding of lists and how they are created; how to sort lists in ascending order and how to use linear and binary search algorithms. Some constraints are the list of numbers generated must be unordered and within the range of 1 and 100. The program must not continue until the applicant sorts the list in order from smallest to largest. Another constraint is both solutions must be iterative and not recursive. This is because recursion may use less lines of code, but it is harder to read.

An example of input would be an integer or number, such as *5*, *23*, *39*, but you can only do one number at a time. An example of output would be ‘*Found at index*’ or ‘*Number not found*.’ There is a built-in function to describe the location if the number is found within the list.

1. **Proposed Solutions**

My basic solution covers basic data types and algorithms: lists and sorting. Lists are a mutable data structure, which means the values inside of it can be changed. In addition, lists can be of any length, which means they have the capacity to store a sequence of numbers within the range of 1 and 100. The reason this is useful is when sorting an unordered list, the list must be modified to make it ordered. This solution uses the sorting algorithm ‘linear search’. Which is both efficient and inefficient. The efficiency of this sorting algorithm depends on the length of the list because as the time taken increases the number of items in the list increases. Therefore, meaning it has a linear time complexity, O(N) and space complexity of O(1) because the solution is iterative.

Similarly, my optimised solution covers three of the topics listed and an extra topic. These include lists from basic data types, sorting and binary search from algorithms. Likewise, this solution covers lists for the same reasons as above. This solution uses iterations instead of recursion; creating a function that calls itself. This solution covers does not include recursion because it might use less lines of code when compared to iterations, but wastes memory by calling functions itself. Therefore, making it time consuming and wastes resources. The features in both my solutions are used for the same reason, sorting is used in my optimised solution to allow binary search to work. This is because binary search requires a sorted list to work, as it divides the list into two and removes half of the list after each iteration. Meaning the length of the lists would be unequal if the lists were not sorted and result in the integer in the list never being found.

Binary search is a more optimised solution, compared to linear search searches. Which searches through all items within the list. Instead, binary search divides the list into two and remove half of the list after each iteration, thus making it more efficient by reducing running time and space complexity. Therefore, resulting in an increased time complexity of O(1), constant and space complexity of O(log2N), logarithmic and quadratic algorithm.

Test cases: Basic Solution/Linear Search

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input** | **Time** | **Explanation** | **Result** | **Complexity** | **Explanation** |
|  |  | There are 2 elements in the list. So, 2/2 = 1 and 1+1 = 2.  Therefore, meaning the program only needs to make 2 comparisons. |  | O(N) – constant  n/2 = 2/2 = 1  Complexity = 1 | Linear search makes n/2 comparisons to check for the target element and there are 2 elements in the list. So 2/2 = 1. Only one comparison needs to be made. |
|  |  | There are 50 elements in the list. So, n/2 = 50/2 = 25  Therefore, the program needs to make 25 comparisons. |  | O(N) = 50/2 = 25  The complexity is 25. | The program needs to make 25 comparisons. |
|  |  | There are 100 elements in the list. 100/2 = 50. So, the program needs to make 50 comparisons. |  | O(N) = 100/2 = 50  The complexity is 50. | The program needs to make 50 comparisons. |
|  |  | There are 500 elements. So, 500/2 = 250. Meaning the program needs to make 250 comparisons. |  | O(N) = 500/2 = 250  The complexity is 250. | The program needs to make 250 comparisons. |
|  |  | There are 1000 elements. So, 1000/2 = 500. Meaning the program needs to make 500 comparisons. |  | O(N) = 1000/2 = 500  The complexity is 500. | The program needs to make 500 comparisons. |

<https://www.freecodecamp.org/news/time-complexity-of-algorithms/>

Test cases for basic solution/linear search

|  |  |  |  |
| --- | --- | --- | --- |
| Test case | Input | Output | Explanation |
| 1 | 1 | Not found | This simple test case checks to see if the program displays the correct output. |
| 2 | 10, 12 | Too many parameters | This intermediate test case checks for robustness. This is done by providing the program with an erroneous input. To check if the program would display an error or not. |
| 3 | -100 | Cannot accept negative numbers. | This simple test case checks to see if the program handles negative numbers as an input. I did this because my program is not supposed to handle negative numbers. |
| 4 | 12345 | Unacceptable range. | This complex test case checks to see if the program can accept large integers without the code being modified. |
| 5 | 1.2 | Cannot accept floating point numbers. | This intermediate test case checks to see if the program can handle decimal number as an input. I did this because the program is not supposed to be able to handle decimal numbers. |
| 6 | + | Cannot accept operators. | This simple test case checks to see if operators are accepted as inputs. |
| 7 | @ | Can only accept integers | This simple test case checks to see if symbols are accepted as inputs. |

Complexity and running time of optimised solution: Binary search

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input** | **Time** | **Explanation** | **Result** | **Complexity** | **Explanation** |
|  |  |  |  |  |  |
|  |  | The purpose of this test is to see how long it would take for my program to find an item in a list of 10 numbers. |  | n/4 = 10/4 = 2.5 | This is a quadratic algorithm. meaning there are two for loop. |
|  |  | The reason for this test is to see how long it would take for my program to find a number in a list of 100 numbers. |  | n/4 = 100/4 = 25 | Quadratic algorithms consist of two for loops. |
|  |  | I did this to measure how long it would take my program to find a number in a list of 1000 numbers. |  | N/4 = 1000/4 = 250 | This is a quadratic algorithm. Meaning there are two for loops. |
|  |  | I did this test to see how long it would take my program to find a number in a list of 10,000 numbers. |  | n/4 = 10000/4 = 2500 | Quadratic algorithms consist of two for loops. |
|  |  | I did this test to see how long it would take my program to find a number in a list of 100,000 numbers. |  | n/4 = 100000/4 = 25000 | This is a quadratic algorithm. Meaning there are two for loops. This has a logarithmic complexity because it works in the opposite way to indices. |
|  |  | I did this test to see how long it would take to find a number in a list of 200,000 numbers. |  | n/4 = 200000/4 = 50000 | Quadratic algorithms consist of two for loops because it requires two conditions to be met consecutively. |

Test cases for optimised solution/binary search

|  |  |  |  |
| --- | --- | --- | --- |
| Test case | Input | Output | Explanation |
| 1 | 1 | Number not found | This simple test case checks to see if the program can handle small inputs. |
| 2 | 10, 12 | Too many parameters | This intermediate test case checks for robustness. It does this by providing the program with an erroneous input. To check if the program will display an error. |
| 3 | -100 | Cannot accept negative numbers. | This simple test case checks to see if the program handles negative numbers as an input. I did this because the program is not supposed to cope with negative numbers. |
| 4 | 12345 | Unacceptable range. | This complex test case checks to see if the program can accept large integers without the code being modified. |
| 5 | 1.2 | Cannot accept floating point numbers. | This intermediate test case checks to see if the program handles decimal numbers. I did this because the program is not supposed to be able to handle decimal numbers. |
| 6 | + | Cannot accept operators | This complex test case checks to see if operators are accepted as inputs. |
| 7 | @ | Can only accept integers. | This simple test case checks to see if symbols are accepted as inputs. |

**Appendices**

Code 1 – Linear search

1. # Basic solution
2. # Iterative
3. **import** time
4. **import** random  # Imports time module for measuring running and execution time.

7. # import random # Imports random module to generate a random list
8. # import numpy
10. # import numpy as np

13. **def** test\_run():  # Defines a function called test\_run
14. t1 = time.time()  # Assigns the calculated time to a variable called t1.
15. **print**("The running time after the first execution is: ", t1)  # Displays a message aaying the running time.
16. t2 = time.time()
17. **print**("The time taken for execution and running is ", t2 - t1, " seconds")

20. # Linear search
21. **def** linearsearch(arr, n, x):  # Defines a function called linear search and provides it with parameters.
22. **for** i **in** range(0, n):  # Loops through the list, array, from the first element to the last element.
23. **if** (arr[i] == x):  # Checks to see if the list is equal to x.
24. **return** i  # The value thought of by the computer is returned, if the list is equal to x.
25. **return** -1  # Otherwise minus 1 is returned.

28. # x = [2, 4, 6, 8, 10]
29. arr = []
31. **for** i **in** range(0, 1000):
32. x = random.randint(1, 1000)
33. arr.append(x)
34. # arr = [1, 2, 3, 4, 10, 40] # Creates a list and assigns values to it.
35. arr.sort()
36. x = 10  # Creates a variable called x and assigns an integer to it.
37. n = len(arr)  # Creates a variable called n and assugbs the length of the list to it.
38. **for** num **in** arr:
39. **print**(num)
40. **print**(arr)
41. start = time.time()
42. result = linearsearch(arr, n, x)  # Creates a variable called result and assigns the function created above to it.
43. end = time.time()
44. **print**(result)  # Displays the list thought of by the program.

47. **if** result != -1:  # Checks to see if the value is not in the list.
48. **print**(
49. 'Number found')  # if the value is not in the list found is returned to tell the user their number is in the list.
50. **else**:  # Moves the program to the next condition if the previous condition is false.
51. **print**(
52. 'Number not found')  # Display not found, if the value though of is not equal to the value thought of by the program.
53. **print**("The time taken is", end - start)

Code 2 – Binary search

*# Sophisticated solution  
  
#def BinarySearch(arr, low, high, x):  
# if high >= low:  
# mid = low + (high - low) // 2  
   
# if arr[mid] == x:  
# return mid  
   
# elif arr[mid] > x:  
# return mid  
   
# elif arr[mid] > x:  
# retun BinarySearch(arr, low, mid-1, x)  
   
# else:  
# return BinarySearch(arr, mid + 1, high, x)  
   
#else:  
# return -1  
   
   
#arr = [1, 2, 3, 4, 10, 40]  
#x = 4  
  
#result = BinarySearch(arr, 0, len(arr)-1, x)  
#if result == -1:  
# print("Not found")  
#else:  
# print("Found")  
  
  
#def BinarySearch(arr, low, high, x):  
# if high >= low:  
# mid = low + (high - low) // 2  
   
# if arr[mid] == x:  
# return mid   
   
# elif arr[mid] > x:  
# return BinarySearch(arr, low, mid-1, x)  
   
# else:  
# return BinarySearch(arr, mid + 1, high, x)  
   
# else:  
# return -1  
   
   
#arr = [1, 2, 3, 4, 10, 40]  
#x = 10  
  
#result = BinarySearch(arr, 0, len(arr)-1, x)  
#if result == -1:  
# print("Not found")  
  
#elif result > -1:  
# print("My value is greater than your value")  
  
#elif result < -1:  
# print("My value is less than your value")  
  
#else:  
# print("Found")***import** time  
**import** sys  
  
  
**def** test\_run(): *# Defines a function called test\_run* t1 = time.time() *# Assigns the calculated time to a variable called t1.* print(**"The running time after the first execution is: "**, t1) *# Displays a message saying the running time.* t2 = time.time()  
 print(**"The time taken for execution and running is "**, t2 - t1,**" seconds"**)  
  
  
  
**def** binarysearch(arr, low, high, x): *# Creates a function called BinarySearch and initialises the parameters stored within it.* **while** low <= high: *# Iterates over the list until the condition: the value in the list is lower than the next value is false.* mid = low + (high -1)//2 *# Defines a variable called mid to split the list in half after each iteration.* **if** arr[mid] == x: *# Checks if the middle of the list is equal to the value stored in the variable x.* **return** mid *# If the above condition is true, the value in the middle of the list is returned.* **elif** arr[mid] < x: *# If the above condition is false, the code checks if the middle value is less than x.* low = mid - 1 *# If the above condition is true, the middle value subtract 1 is returned.* mid = low + (high -1)//2   
   
 **else**: *# Tells the code what to do if both conditions are false.* high = mid - 1 *# Returns the values to the right of the list.* **return** -1 *# Returns -1  
   
  
  
# sorting the list***def** selection\_sort():  
 **for** i **in** range(len(arr)):  
 low = i  
 **for** high **in** range(i+1, len(lst)):  
 **if** arr[high] < arr[mid]:  
 low = high  
  
x = 10 *# Creates a variable called x.  
  
# print (np.random.randint(1, 500, 5))***'''  
def test\_run(): # Defines a function called test\_run  
 t1 = time.time() # Assigns the calculated time to a variable called t1.  
 print("The running time after the first execution is: ", t1) # Displays a message aaying the running time.  
 t2 = time.time()  
 print("The time taken for execution and running is ", t2 - t1," seconds")  
  
start = time.time()  
def BinarySearch(arr, x):  
 low = 0  
 high = len(arr)-1  
 mid = int((low + high)/2)  
 while low <= high:  
 mid = int((low + high)/2)  
 if arr[mid] == x:  
 return mid  
 if arr[mid] > x:  
 high = mid - 1  
 else:  
 low = mid + 1  
 return -1  
'''***#arr = [2, 1]  
#arr = [5, 4, 3, 2, 1]  
#x = 4*arr = [2, 3, 4, 10, 40] *# Creates a list*start = time.time()  
result = binarysearch(arr, 0, len(arr)-1, x) *# Assigns the function to a variable called result.*end = time.time()  
**if** result != -1: *# Checks to see if the list is not equal to -1.* print(**"Element is present at index %d"**) % result *# Displays the current element.***else**:  
 print(**"Element is not present in array"**) *# Otherwise, says the element is not in the list*print(**"The time taken is"**, end - start)