Week: 06 Date: 3/11/2020

|  |
| --- |
| 15-110 Recitation Week 6 |

**Reminders for Students**

* Hw 3 due Monday 3/15 @ Noon EDT
* How was Quiz2?
* Recitation feedback form: <https://forms.gle/WKrrbawKktmRu1xp9>

**Overview**

* Big-O
* Linear Search vs. Binary Search
* Dictionaries (mostWins)
* Bonus: Recursion (code writing interleave)

|  |
| --- |
| Problems |

# **BIG-O EXERCISE**

Calculate the Big-O for the following examples:

|  |  |
| --- | --- |
| Returning the last character in a string | Answer |
| An algorithm that checks if there are duplicates in a list by looking at an element in the list, *x*, and then checking the rest of the list for *x*, and repeating this process for each element in the list.  | Answer |
| def powersOfTwo(n): m = 1 while m <= n:  print(m) m \*= 2 | Answer |
| def foo(L): if L == []: return 0 else: L.append(L[0])  n = L.find(10) L.pop(0) return n | Answer |

#

# **LINEAR SEARCH VS BINARY SEARCH**

**Run a visual trace on the list [2, 4, 6, 7, 10, 11] and find 1 and 7 using both linear and binary search.**

Discuss the actual number of *steps* that happen during each. A *step* is when you move to a new element of the list for comparison (Linear - each element iterated through/checked is a step, Binary - each time the function body runs is a step)

Linear Search:

 Find 1: answer

 Find 7: answer

Binary Search:

Find 1: answer

Find 7: answer

During binary search, how many elements (maximum) are checked?

Answer

Discuss the complexities (Big O) of linear search versus binary search and explain when we use one over the other and in what situations.

Answer

**Now briefly review the important pieces of the recursive function (as shown in lecture) that we use to implement the binary search algorithm:**

def binarysearch(L, item):

 #base case 1

if len(L) == 0:

 return False

 #find the middle value

 mid = len(L)//2

 #base case 2

if L[mid] == item:

 return True

#Recurse on everything less than mid

 elif L[mid] > item:

 return binarysearch(L[:mid],item)

 #Recurse on everything greater than mid

else:

 return binarysearch(L[mid+1:],item)

Explanation for Key Points in Algorithm:

* Calculation of mid index – consider how integer division splits the list in half
* Base case return values
	+ When the list is empty, there is nothing left to search through so we know we didn’t find the value we were looking for
	+ If L[mid] == value, then the value is in our list and we return True
* Consider what the recursive calls/return values mean:
	+ Since we are given that the input list is sorted, based on the comparison of L[mid] and value, we know whether the value we are searching for lies in the bottom half or the top half of the list, and slice the list accordingly.
	+ Thus, we only need to continue performing binary search on the half of the list containing the element
	+ Note that the middle element isn’t included in the list slice passed into the recursive call.

# **CODE WRITING: mostWins**

**Given a list of wins by CMU, Pitt, OSU, PennState’s, and another unspecified number of football teams, return the team with the most wins. There will be no ties.**

mostWins(["OSU", "PennState", "PennState", "CMU", "OSU", "OSU", "Pitt"])

 -> "OSU"

mostWins(["PennState", "PennState", "MIT", "Stanford", "UF"])

 -> "PennState"

First, work through an O(n^2) solution in the starter file under **mostWinsSlow!**

(Hint: use a list method to count the number of times a team appears in the list)

def mostWinsSlow(L):

 ...

Now you have a valid solution, we can solve this problem even quicker. To do this, we use dictionaries!

Dictionary Review:

* Key-Value Pairs: Dictionaries store information in key-value pairs - you can access specific values in the dictionary by looking up the key! It’s similar to how you use an integer index to look for a specific value in a list, except dictionary keys can be more complex IMMUTABLE types like strings.
* We can also use FOR ITERABLE loops when working with dictionaries! We can loop directly over the keys in a dictionary by writing for k in d where k will be each key occurring in dictionary d. To access the value corresponding to every key k, we write d[k].
* Why do we use dictionaries? You’ll learn more about this later, but we can search for a specific key in a dictionary in constant O(1) time! Once we know the key, we can get the corresponding value in constant time as well.

Basic Dictionary Code:

#create empty dictionary

d = dict() #or d={}

#add keys/values to dictionary

d["Jasmine"] = 32 # d -> {"Jasmine":32}

d["Darryl"] = 15 # d -> {"Jasmine":32, "Darryl":15}

#update or change values

d["Darryl"] = d["Darryl"] + 1 # d -> {"Jasmine":32, "Darryl":16}

d["Jasmine"] = 12 # d -> {"Jasmine":12, "Darryl":16}

#loop through dictionary

for key in d:

 print("Key = ", key)

 print("Value = ", d[key])

Now that you know more about dictionaries, work through an O(n) solution in the starter file under **mostWinsFast!** (Hint: store the counts of each team in a dictionary, and then look through all the teams (keys) in the dictionary to see which one has the greatest associated value)

def mostWinsFast(L):

 ...

What is the runtime of this function mostWinsFast?

Answer

# **BONUS (IF TIME): RECURSIVE INTERLEAVE**

Below we’ve written an iterative function interleave, which takes in two input lists of the same size and combines them into one list, alternating elements from the first list and the second list. For example, interleave([1,2,3], [4,5,6]) should return [1,4,2,5,3,6].

Now, try to write this function recursively.

Iterative interleave:

def interleave(lst1, lst2):

 result = []

 for i in range(len(lst1)):

 result = result + [lst1[i]] + [lst2[i]]

 return result

Recursive interleave:

Code