Week 8: Interview Preparation (Discussions)

Introduction

When applying for jobs in the Tech industry, it is very likely that you will have to do some kind of technical/coding exam! In this worksheet we will work through some practice questions and get you aware of some common types of questions/solutions.

Many companies test the same topics - normally questions on data structures / algorithms. Some popular topics can be seen below:

- Arrays / Sorting Algorithms
- Linked Lists
- Hash Tables
- Trees / Graphs
- Search Algorithms
- Time Complexity
- Binary Numbers
- Recursion / Dynamic Programming

This week we will touch on some of these topics and discuss solutions to a few problems - next week you will implement some solutions in code! **It is important to build your skills on both discussions and coding**; some coding tests will be during the actual interview so you will only discuss solutions and some you will be required to type your solutions out!

All solutions will be released next week so don't worry if you are unsure about the best answer.

TASK 1: Time Complexity

In an interview you may be asked to look at some code and give the time complexity. The time complexity is a function which describes the runtime of a function *in relation to the size of the input* - this is normally about observing the loops in the code.

Also, time complexity will come into most interview questions that you have, even if the question is not specifically on this topic. Often there are multiple solutions to the question they might be asking, so you will have to consider the time complexities of these and choose the best one. It is also impressive to the interviewers if you can state the time complexity of your algorithm!

TO DO:

1. Consider the following function which turns a list into a counter dictionary: what is the time complexity in O notation?

   ```
   def ListToDict(List):
       Dict = dict()
       for X in List:
           Value = Dict.get(X,0)
           if Value == 0:
               Dict[X] = 1
           else:
               Dict[X] += 1
       return Dict
   ```

   ANSWER: $O(n)$

2. Consider the following function which sums the elements in an array - what is the time complexity?

   ```
   def SumArray(Arr):
       Sum = 0
       n, m = Arr.shape()
       for i in range(n):
           for j in range(m):
               Sum += Arr[i,j]
       return Sum
   ```

   ANSWER: $O(nm)$

3. In what kind of situation would you have a runtime of order $O(n+m)$?
   ANSWER: When you have two separate loops (one after the other, not inside each other) - one of size n and one size m?

4. Merge sort has runtime $O(n \log n)$ - can you explain why?
   ANSWER: With merge sort you keep splitting the list in halves until each separate list has length 1. So, the runtime depends on how many times you can divide the length of the list n by 2. So, we need to find x such that $2^x = n$ which is $x = \log_2 n$. This n in the $O(n \log n)$ comes from the fact that we have to iterate through each element doing comparisons and swaps. So we have n comparisons for $\log n$ iterations, giving us $n \log n$.

TASK 2: Hash Tables

A very common data structure that is used in interviews is the hash table. However, it is often hidden and not immediately obvious to use this data structure (but is normally the answer they are looking for!).

We briefly touched on hash tables within Python, also known as dictionaries, in Week 2 and 3. Dictionaries consist of pairs of 'keys' and 'values', in Python that looks like this:

```
Dictionary = {Key1: Value1, Key2: Value2, ..., KeyN: ValueN}
```

They have a number of different uses, however one common use (often needed to solve interview questions) is to view the dictionary as a counter for a list - an example can be seen below. **Think about using hash tables in your algorithm solutions when the ordering of the elements does not matter.**

If we had the list:

```
L = [1,2,3,4,3,2,1]
```

We can store a counter for the elements in a dictionary that looks like:

```
D = {1: 2, 2: 2, 3: 2, 4: 1}
```

Also, consider using a hashtable / dictionary when you have pairs of elements and need to access one element given the other in a pair. For the example above we could call D[4] and it will return 1, rather than having to find the position of where the 4 element is and accessing that indexed position.

TO DO:

1. Discuss how you would write a function that inputs a string of text and returns the number of **unique** words. You may wish to discuss your solution as if you would implement it in Python. Things to consider are:

   (a) Upper case and lower case letters

   (b) Punctuation

   (c) How to split the string up into separate words

   (d) Counting the unique words

   ANSWER:
   a) Make all the characters either upper case or lower case (Python has an upper() and lower() function).
   b) You would have to remove all punctuation apart from spaces (you can do this using ASCII encoding of characters (any character that isn't in the number range for A-Z and any character that isn't a space should be removed).
   c) You can split the string up into separate words by searching for spaces (using String.split(' ') in Python) - this could also be done in the stage where removing

punctuation.
d) Once you have a list of all the separate words (without spaces and punctuation), then you could transform it into a counter dictionary. This will create a list of words and their frequencies - all you need to do now is get the length of the dictionary and you have the number of unique words.

TASK 3: Strings

Many interview questions will involve the use of strings, these questions may sometimes seem easy but the trick is to get the most efficient answer.

TO DO:

1. Discuss the best way the check if a string is a palindrome (i.e the word can be mirrored - 'hannah' is an example). If you don't wish to participate in the discussion please feel free to try implement the function yourself!
   ANSWER: The most efficient way is to have a pointer at the beginning and the end of the word and move them both in towards the middle one step at a time checking that the letters match. If the pointers cross over the middle then you can stop. This is the most efficient method as you only traverse through the word once, other methods may involve going through all the letters twice or more.

2. Discuss how you would write a function that would check if a word has a **permutation** that is a palindrome. The function should input a string and return a Boolean. For example, 'accerar' would return true because it can be rearranged to say 'racecar', which is a palindrome. *Hint: if we are considering permutations then order doesn't matter - only the counts of the letters matter here.*
   ANSWER: This one is different to the answer above as the word can be reordered. Some people may want to find every permutation of the letters and do a palindrome check but this is extremely inefficient. The best way to do it is to count the letters - we must have an even number for every count with the exception of one odd count (for the middle letter). Note that the odd count doesn't have to just be a count of 1 ('ababa') is a palindrome.

TASK 4: Linked Lists

Another very common topic are linked lists, where you either have a singly or doubly linked list.

Linked lists differ to a normal list as each element in the list has a 'next' pointer which will point to the following element in the list. To get to the n^{th} element in a linked list you have to start at the beginning (sometimes referred to as the 'head') and then keep following the next pointers until you reach the specified element.

You might get questions about how to apply sorting algorithms to linked lists.

TO DO:

1. Discuss how you might implement a linked list in Python.
   ANSWER: The best way to do this would be by making a class, let's call this the element class, which would have the properties of 'value' and 'next'. We can build up the list by creating multiple instances of the element class and linking them together with the next property.

2. Discuss how you would find the k^{th} from last element in a singly linked list. Try think of the most optimum way of doing this.
   ANSWER: Some people might think to go through all elements in the list to find the lengths of the list, n, then go back to the beginning and go back along to the $(n-k)^{th}$ element. This means we would have $2n-k$ steps. The best way to do this is to have to pointers starting at the beginning: for the first k steps only move one pointer, let's call this the second one. Then, keep stepping both pointers along one until the second one has reached the end - the first pointer should be $-k$ steps away from the second one which is at the end. This would only take n steps.