Q: How might you determine the efficiency of an algorithm
Determined by the Big O notation

Q: Why is it important?
So that we can have the code as smooth as possible. Speed is important.
Planning for growth / scaling up

Q: What categories of complexities are described in Big O Notation
- Time complexity (How long code will take to execute as data increases)
- Space complexity (How much space is needed and how efficiently it is used. Possibly just look for them to understand the fact that there is often a trade off between time and space complexity.)

More space = less time and the other way around

https://github.com/mariusbanea/web-developers-toolkit/blob/master/algorithms/big-o-notation/big-o-cheatsheet.pdf

Q: what is O(1)
Constant Time

Examples: setting a variable once, performing a calculation, determining if an integer is even or odd.

Basically a flat line on a graph. Easy to plan for.

Q: what is O(n)
Linear Time

Examples:basic non-nested for loops, finding smallest or largest item in an unsorted array.

Basically a 45 degree line on graph, still easy to plan for, scaling probably won't run away from you.

Q: what is O(n^2)
N squared

- basic nested for loops, bubble sort, insertion sort.

Starts to get harder to plan for as data sets increase dramatically.

Q: what is O(log n)
Log N (Logarithmic Time)

- binary search - when you can easily eliminate large sections of data on each iteration.

Means that scaling large data sets won't dramatically increase the output time.

cheatseet for this is here https://github.com/mariusbanea/web-developers-

toolkit/blob/master/algorithms/big-o-notation/big-o-cheatsheet.pdf

big-o-notation-in-plain-english is here https://github.com/mariusbanea/web-developers-toolkit/blob/master/algorithms/big-o-notation/big-o-notation-in-plain-english.pdf (especially page 1 bottom and page 2)

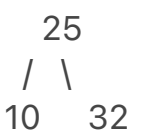
big-o-notation-table-for-interviews is here: https://github.com/mariusbanea/web-developers-toolkit/blob/master/algorithms/big-o-notation/big-o-notation-table-for-interviews.pdf

Q : Describe 3 differences between a linked list and an array
Linked list
- Non-contiguous space requirement. Ex: Data doesn't HAVE to be in same space on disk.
- Can be more efficient for memory allocation. Easier to resize.
- Can be less efficient for caching (not contiguous)
- No "find by index" feature, so accessing individual elements can be slower
- Easier to add elements (insert nodes) to various positions within list.

Array
- Must be in same contiguous disk space.
- Harder to resize. Also, increasing length may require moving the array in memory to a larger space.
- Better caching - all addresses are contiguous
- Fast access for reading when searching by index
- Harder to insert new elements to front and within the array as you then have to shift existing elements after the new element is added.

Q: the difference between a Binary Tree and a Binary Search Tree
BST: All left descendants are less than or equal to node, and all right descendants are greater than node.
https://github.com/mariusbanea/web-developers-toolkit/blob/master/algorithms/data-structures-binary-trees/binary-search-tree-graph.jpg

Q: describe the 3 types of BST traversal:



In-order (from the smallest number to the biggest) (ex: 10, 25, 32)
Visit left branch, then current node, then right branch. Visits all nodes in ascending order on a BST.

Pre-order (looks like the "<" sign) (node first, branches last) (ex: 25, 10, 32)
Visits current node first, then left branch, then right branch. Root node is always first node visited. Useful for directory structures, organisation charts

Post-order (looks like the ">" sign) (branches first, node last) (ex: 10, 32, 25)
Visits left branch, then right branch, then current node. Root node is always last visited. Useful for language processors

Q: Algorithms for searching through arrays and trees

- Linear search (https://www.youtube.com/watch?v=-PuqKbu9K3U) - look through an array one-by-one until you find what you are looking for
- Binary search (https://www.youtube.com/watch?v=iP897Z5Nerk) - narrow it down by always guessing in the middle of the range. So you would start at 50 (half-way between 0 and 100), then if you were too high go to 25 (half-way between 0 and 50), then if you were too low go to 37 (half-way between 25 and 50), and so on.
- Depth-first search - traverse as far as you can down the tree before back-tracking (ex: check if a person related to a past has the right to be a king).
- Breadth-first search - search works across the rows of a tree (the top row will be handled first, then the second row, and so on) (ex: check if multiple people from the same generation are part of the royal family)

Q: how the arrays sorting methods work

(ALL CAPITALS TEXT has the keywords to be remembered by)

Merge sort works like this :
- DIVIDE THE UNSORTED LIST INTO N SUBLISTS, each containing 1 element (a list of 1 element is considered sorted).
- repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.
(example: https://www.youtube.com/watch?v=XaqR3G_NVoo)

Bubble sort works like this:
- REPEATEDLY SWAPPING the adjacent elements if they are in wrong order.
(example: https://www.youtube.com/watch?v=lyZQPjUT5B4)

Quick sort works like this :
- DIVIDE AND CONQUER algorithm. It creates two empty arrays to hold elements less than the pivot value and elements greater than the pivot value, and then recursively sort the sub arrays. There are two basic operations in the algorithm, swapping items in place and partitioning a section of the array.
(example: https://www.youtube.com/watch?v=ywWBy6J5gz8)

Selection sort works like this:
- REPEATEDLY FINDING THE MINIMUM ELEMENT (considering ascending order) from unsorted part and putting it at the beginning
(example: https://www.youtube.com/watch?v=Ns4TPTC8whw)

Insertion sort works like this :
- begin at the left-most element of the array and invoke Insert to INSERT EACH ELEMENT ENCOUNTERED INTO ITS CORRECT POSITION.
- The ordered sequence into which the element is inserted is stored at the beginning of the array in the set of indices already examined.
(example: https://www.youtube.com/watch?v=ROalU379l3U&t=97s)

Heap sort works like this :

- involves preparing the list by first TURNING IT INTO A MAX HEAP (a complete Binary Tree represented as an array)
- then repeatedly swaps the first value of the list with the last value, decreasing the range of values considered in the heap operation by one, and sifting the new first value into its position in the heap.
(example: https://www.youtube.com/watch?v=Xw2D9aJRBY4)

https://github.com/mariusbanea/web-developers-toolkit/blob/master/algorithms/algorithms-sorting/algorithms-arrays-sorting.gif

Q: What are the arrays sorting methods by speed
The slowest is selection, then bubble, then insertion, then shell, then merge, then heap, then quick, then quick3

Memory tricks to remember the speed order:
- by sentence:
-—> The SELECTED, BUBBLE can't INSERT into the SHELL while MERGING the HEAP QUICKly
- by abbreviation:
-—> Se.Bu.In.Sh.Me.He.Qs
- or find a rhythm to rap it :-)

Q: What is faster than a bubble sort?
Insert, Shell, Heap, Merge, Quick and Quick3

https://github.com/mariusbanea/web-developers-toolkit/blob/master/algorithms/algorithms-sorting/algorithms-arrays-sorting.gif

==== possible code challenges ====

- Create an algorithm for counting how many times a word appears in a document.

```
const lib = {}

'hello hey hello hey'.split(' ').forEach(word => {
  lib[word] = lib[word] + 1 || 1
})
console.log(lib)
```

For this answer, the candidate should be directed to the choice of data structure if they need some hints.

The best choice of data structure for this will be a Hashmaps. First, we will go through the document hashing each word in a hashmap and as we do so, we will increment the count of each word. Hashmaps are key-value pairs. In this case, the word will be the key to hash and its value will be the count of the word. This is O(1) operation. Then once all the words are hashed, we will do a lookup of the word that we want and find and from the word, we can get its count to find out how many times the word is used in the document. This lookup will be a O(1) operation. Hashmaps will be the right choice due to its O(1) insert and lookup.

- Develop an algorithm to find the Fibonacci series of a given number.

```
function fib(n) {
  if (n < 2) {return n} // base case

  return fib(n - 1) + fib(n - 2);
}
console.log(fib(7))
```

Using recursion would be an elegant solution from a readability point of view since it will require 2-3 lines of code.

You will have a base case to stop the recursion – this could be:

if (n<=2)
return 1

Then you will recursively call the function to add the previous 2 numbers – this will be something like:
return fib(n-1) + fib(n-2)

Define what is a Fibonacci sequence is (The Fibonacci sequence is one where a number is found by adding up the two numbers before it. Starting with 0 and 1, the sequence goes 0, 1, 1, 2, 3, 5, 8, 13...). What is the 7th value in the Fibonacci sequence? Can you explain how you came up with your solution?

- What is the runtime of the Fibonacci recursive algorithm? what can you tell me about this runtime?

Exponential O(2^n). This is not a desirable runtime if we were to find the Fibonacci sequence for a large number because the runtime will grow exponentially. Although the code looks very elegant, it would be better to implement this using iterative approach or using memorization to cache the results of each recursive calls to be used without calling the function again.

- You are given a dataset. How would you create a balanced binary search tree from it?

const data = [1,2,3,4,5]

// 3
// 2 4
// 1 5

A balanced binary search tree can be defined as one where the height of the left and right subtree can't differ by more than one. Which means that there should be almost equal dataset on each sub tree of the left and right child of the root. In order to do this, find out if the data is sorted. If it is not, then you

can use one of the efficient sorting algorithms such as quicksort or merge sort to sort the data. Once the data is sorted, then we will find the middle of the dataset and use that as the root. This way we can say that there will be almost equal amount of data on each side of the middle element.

Then we will use the same strategy to find the middle of the left hand side to find the left child and the middle of the right hand side to find the right child. And we will continue doing this on each halves recursively until there is no more element left.

- Design an algorithm to find the nth element in a linked list in constant time.

Linked list do not allow random access. Therefore it is not possible to find the nth element in constant time. The only way you can have a O(1) access is if you are finding the 1st element. Otherwise finding an element in a linked list will require O(n) run time.

- A binary search tree has 20 nodes and is relatively balanced. The root is considered to be at level 0. How many nodes are there at level 4?

Answer:
level 0 = 1 node
level 1 = 2 nodes
level 2 = 4 nodes
level 3 = 8 does
level 4 = only 5 nodes left to make a total of 20 (these are their values: 1 2 4 8 16)

- You are given a dataset containing positive and negative integers. Write an algorithm to find the largest sum in a continuous sequence.

```
let maxSubarray = function(arr) {
    let maxEndingHere = 0;
```

```
    let maxSoFar = 0;
    for (let i = 0; i < arr.length; i++) {
        let item = arr[i];
        maxEndingHere = Math.max(0, maxEndingHere + item);
        maxSoFar = Math.max(maxSoFar, maxEndingHere);
        console.log(item, maxEndingHere, maxSoFar);
    }
    return maxSoFar;
};
```

- Write a function to remove duplicates in a linked list.

```
let list = new LinkedList;

let currentNode1 = list.head;
while (currentNode1) {

   let currentNode2 = currentNode1.next;
   while (currentNode2) {
     if (currentNode2.value === currentNode1.value) {
       list.remove(currentNode2);
     }
     currentNode2 = currentNode2.next;
   }

   currentNode1 = currentNode1.next;
}
```

The common solution is O(n^2). Check every node data against every other node data and if find duplicates, delete that node.

Solve this in O(n) if they have done it in O(n^2). The O(n) will require different approach:
- If it is a sorted list then check the current node with the next one and delete the duplicates
- If the list is not sorted, create a hash table
- Iterate through the list - if the data is present in the hash table, delete that node otherwise continue adding the node value into the hash table - run time is O(n), assuming hash table insertion is O(1)

- Write an algorithm that takes an alphanumeric phone number and returns its actual phone number. For example if you type 1-800-Flowers it should return 1-800-3569377. You can ignore spaces and any punctuation). What is the runtime of your algorithm?

```
const numberKeys = {
  'abc': 1,
  'def': 2,
  'ghi': 3,
}

'1-800-Flowers'.split('').forEach((char, idx) => {

Object.keys(numberKeys).forEach(el => {

    number[el]
  })

})

console.log(Object.keys(numberKeys))
```

- Given the following expression, write an algorithm to check if the formula has correctly implemented parenthesis (check to see if there are any lone parenthesis. That every open parenthesis has a closed on) - Check if the formula has any unmatched parenthesis.

Test cases:
x*(x+z) + x/(y-z) + d should return true
t – (s-k + x should return false

If the student is very good and found the solution before the allocated time, you can ask them to think about this - How would you change this to account for other parenthesis such as {}, [].

Test cases:
x*(x+z) + [x/(y-z) + d] should return true
t – (s-k) } + x should return false

```
let maxSubarray = function(arr) {
    let maxEndingHere = 0;
    let maxSoFar = 0;
    for (let i=0; i<arr.length; i++) {
        let item = arr[i];
        maxEndingHere = Math.max(0, maxEndingHere + item);
        maxSoFar = Math.max(maxSoFar, maxEndingHere);
        console.log(item, maxEndingHere, maxSoFar);
    }
    return maxSoFar;
};
```

- write an O(n^2) / O(1) / O(n) / O(log n) algorithm (find details here https://github.com/mariusbanea/web-developers-toolkit/blob/master/algorithms/big-o-notation/o-1-example.js, https://github.com/mariusbanea/web-developers-toolkit/blob/master/algorithms/big-o-notation/o-2-n-example.js, https://github.com/mariusbanea/web-developers-toolkit/blob/master/algorithms/big-o-notation/o-log-n-example.js, https://github.com/mariusbanea/web-developers-toolkit/blob/master/algorithms/big-o-notation/o-n-example.js, https://github.com/mariusbanea/web-developers-toolkit/blob/master/algorithms/big-o-notation/o-n-k-example.js)

- give a binary search tree example in JS (https://github.com/mariusbanea/web-developers-toolkit/blob/master/algorithms/data-structures-binary-trees/binary-search-tree-example.js)

- give an example of a hash map (https://github.com/mariusbanea/web-

developers-toolkit/blob/master/algorithms/data-structures-hash-maps/hash-map-example.js)

- create a linked list in JS https://github.com/mariusbanea/web-developers-toolkit/blob/master/algorithms/data-structures-linked-lists/linked-list-example-es6.js (line 1 to 150 or so)

- Explain how you would write an algorithm to find the middle element of a linked list
Parse the linked list element by element from the beginning to the end and count the total number of elements. After that parse the list again and stop at the element with the index = counter/2