Asymptotic Notation: JavaScript

Learn about the asymptotic notation of JavaScript data structures and functions.

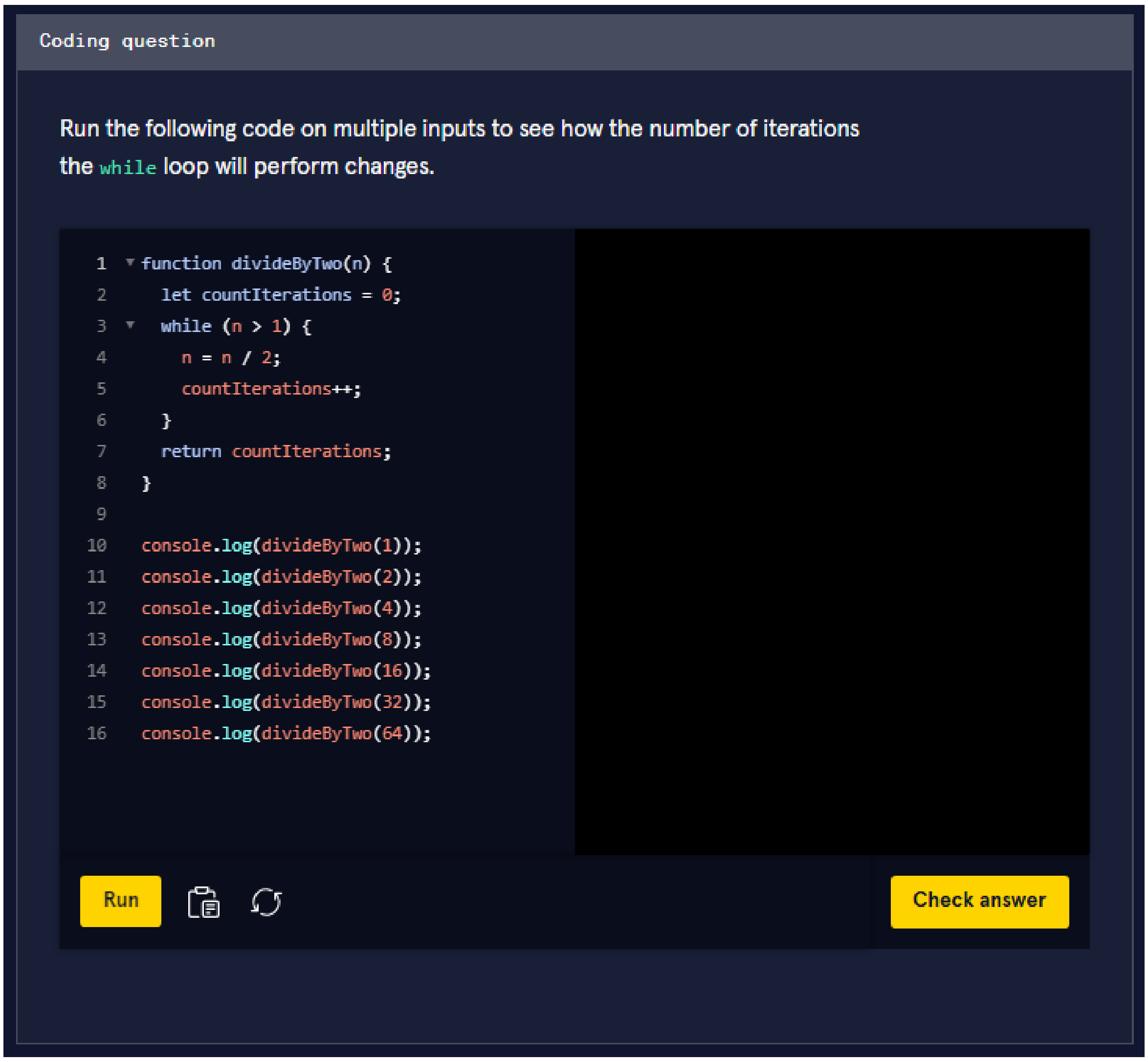
Analyzing Runtimes

Now that you've started learning how to use asymptotic notation to measure the runtime of a function, let's practice with JavaScript!

When analyzing the runtime of a function, it's necessary to check the number of iterations the loop will perform based on the size of the input.

The divideByTwo() function below takes in a positive integer of size n, and returns the number of times n must be divided by 2 until it reaches 1.

We can analyze the runtime of this function by counting the number of iterations the while loop will perform based on the size of the input (n).



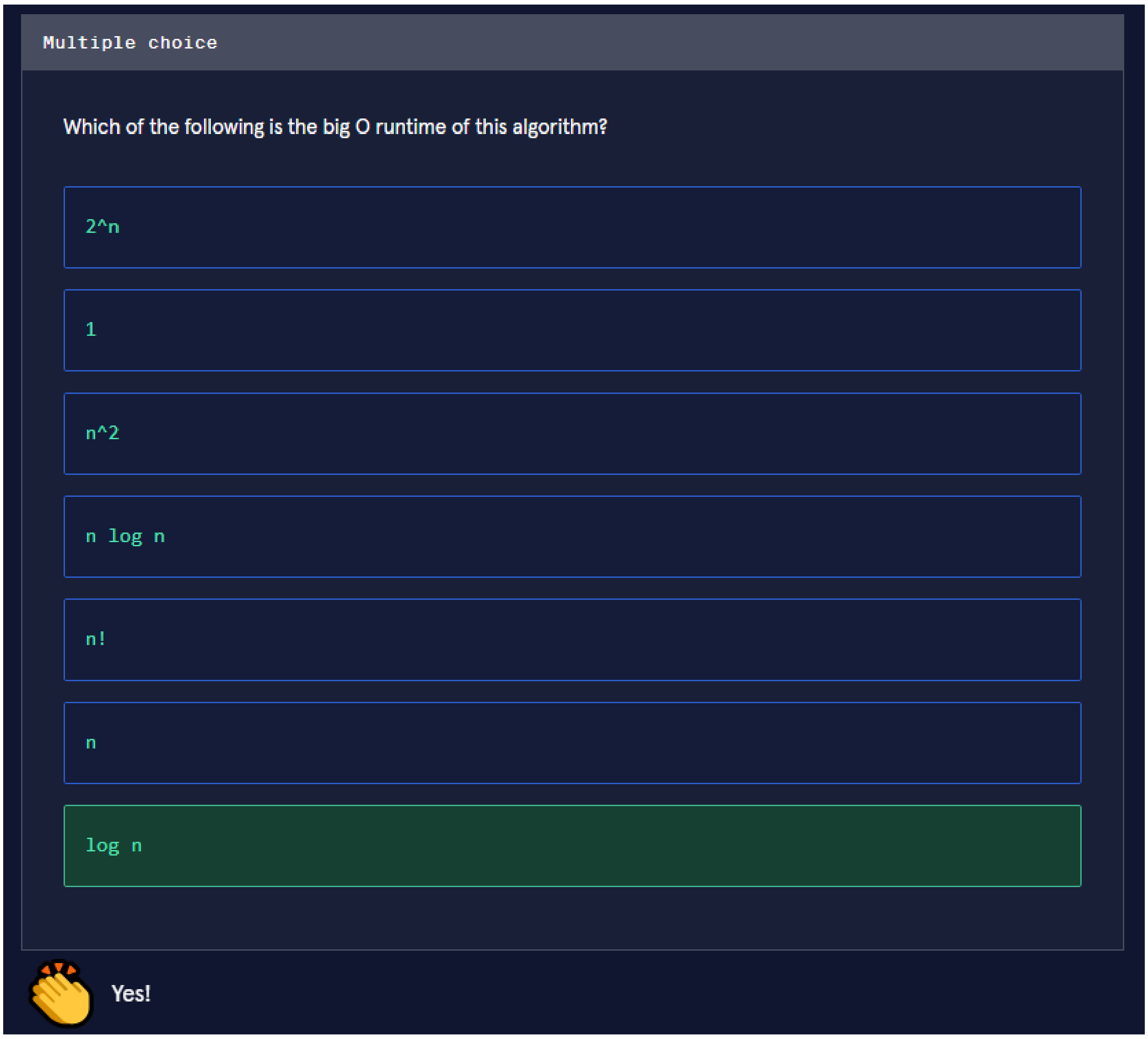
Run the following code on multiple inputs to see how the number of iterations the `while` loop will perform changes.

```javascript
function divideByTwo(n) {
  let countIterations = 0;
  while (n > 1) {
    n = n / 2;
    countIterations++;
  }
  return countIterations;
}

console.log(divideByTwo(1));
console.log(divideByTwo(2));
console.log(divideByTwo(4));
console.log(divideByTwo(8));
console.log(divideByTwo(16));
console.log(divideByTwo(32));
console.log(divideByTwo(64));
```

Do you notice a pattern forming? With n being divided by 2 each iteration, we can use that to establish a big O runtime.

Multiple choice



Multiple choice

Which of the following is the big O runtime of this algorithm?

- 2^n
- 1
- n^2
- n log n
- n!
- n
- [✓ log n]

Yes!

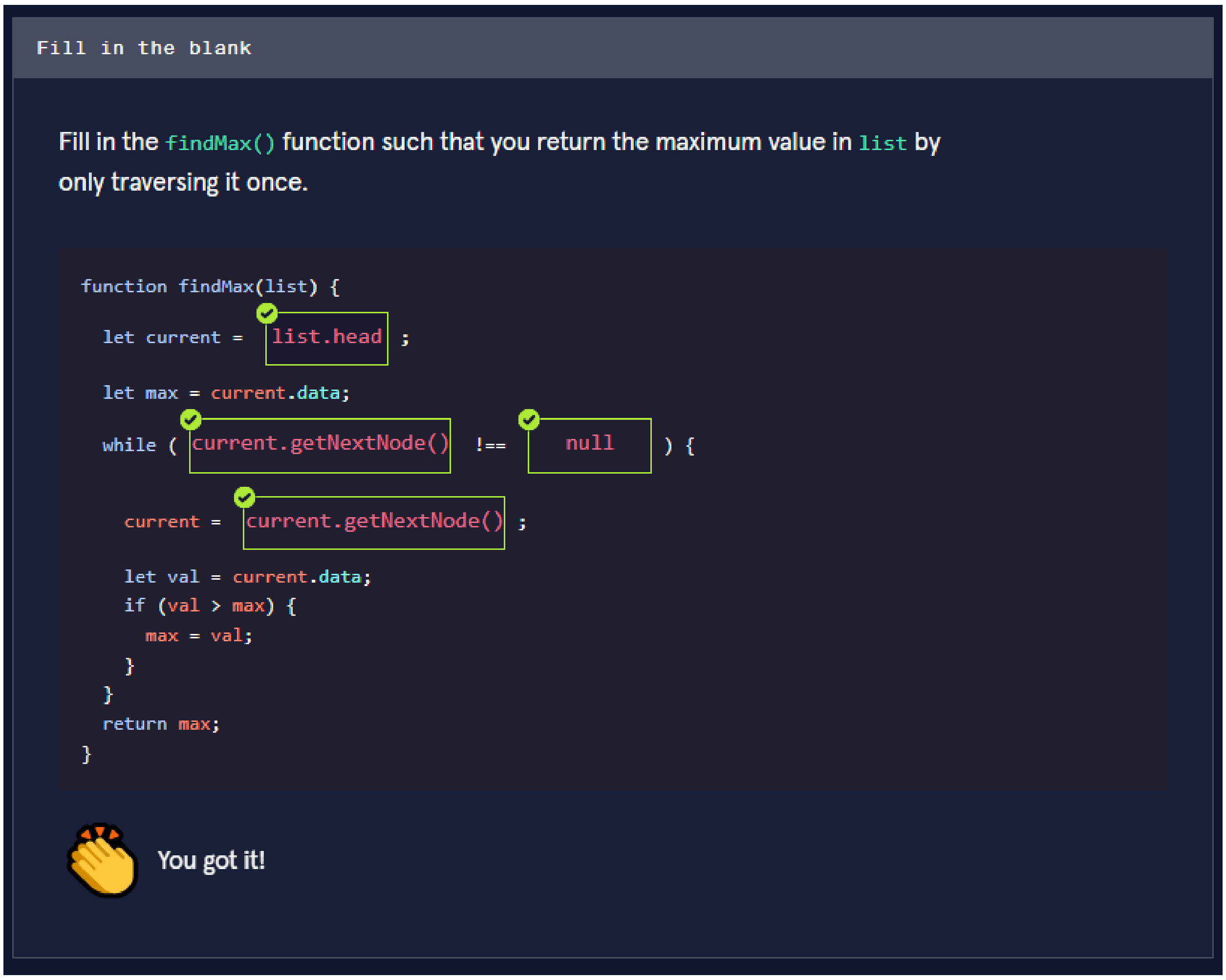
divideByTwo() has a big O runtime of log n because the function divides n by two every iteration, and terminates when n is 1. countIterations counts how many times the while loop runs, and you can see in the output that it is $\log_2(n)$. Since we drop constants for asymptotic notation, the big O runtime is just log n.

Finding the Maximum Value in a Linked List

Now that we can analyze the runtime of a function, let's take a look at the runtime of data structures.

We often search through data structures to find a specific value. In this exercise, you will complete a function that finds the maximum value of a linked list, and you will also analyze the runtime of your function.

The function, findMax(), takes in list as an input. The function should return the maximum value in the linked list.

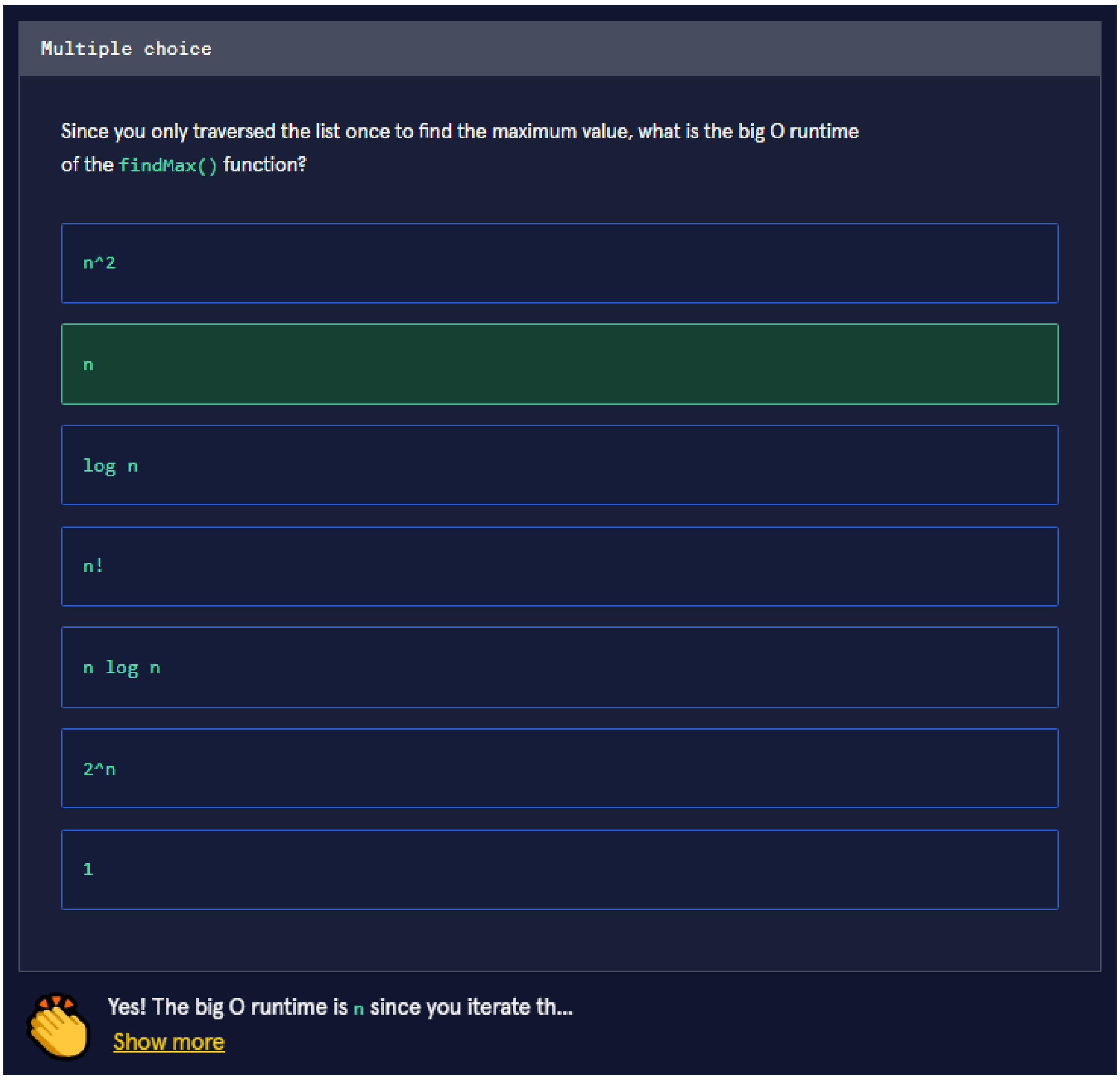


Fill in the blank

Fill in the `findMax()` function such that you return the maximum value in `list` by only traversing it once.

```
function findMax(list) {
  let current = list.head ;
  let max = current.data;
  while ( current.getNextNode() !== null ) {
    current = current.getNextNode() ;
    let val = current.data;
    if (val > max) {
      max = val;
    }
  }
  return max;
}
```

You got it!


Multiple choice

Since you only traversed the list once to find the maximum value, what is the big O runtime of the `findMax()` function?

- `n^2`
- [✓ `n`]
- `log n`
- `n!`
- `n log n`
- `2^n`
- `1`

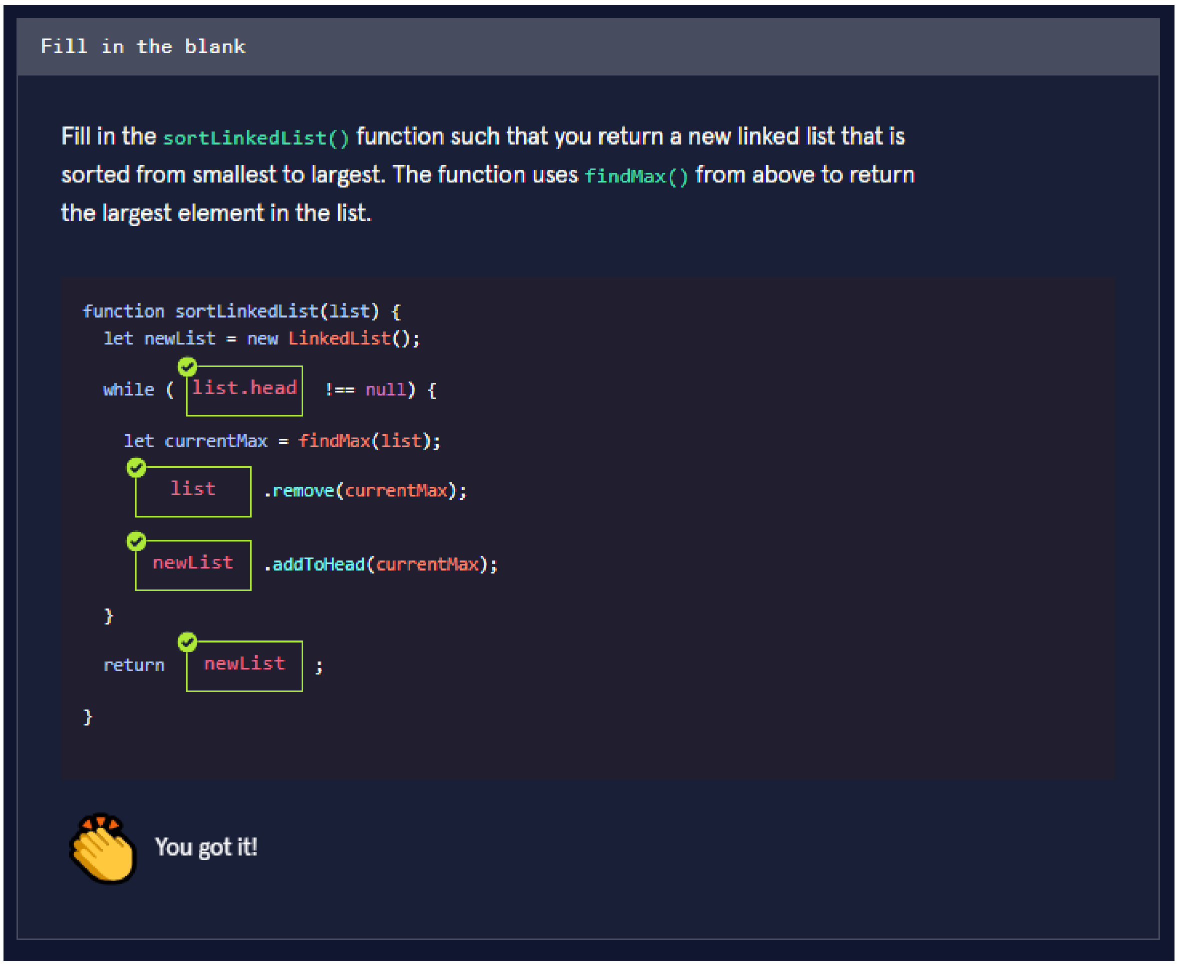
Yes! The big O runtime is `n` since you iterate th...
Show more

Sorting a Linked List

We also often sort data structures in order to organize the values stored in them. In this exercise, you will sort a linked list from the smallest value to the largest value.

There are many ways to sort a linked list, but one way is as follows:

1. Instantiate a new linked list
2. Find the maximum value of the linked list input
3. Insert the maximum to the beginning of the new linked list
4. Remove the maximum value from the linked list input
5. Repeat steps 2-4 until the linked list input is empty
6. Return the new linked list

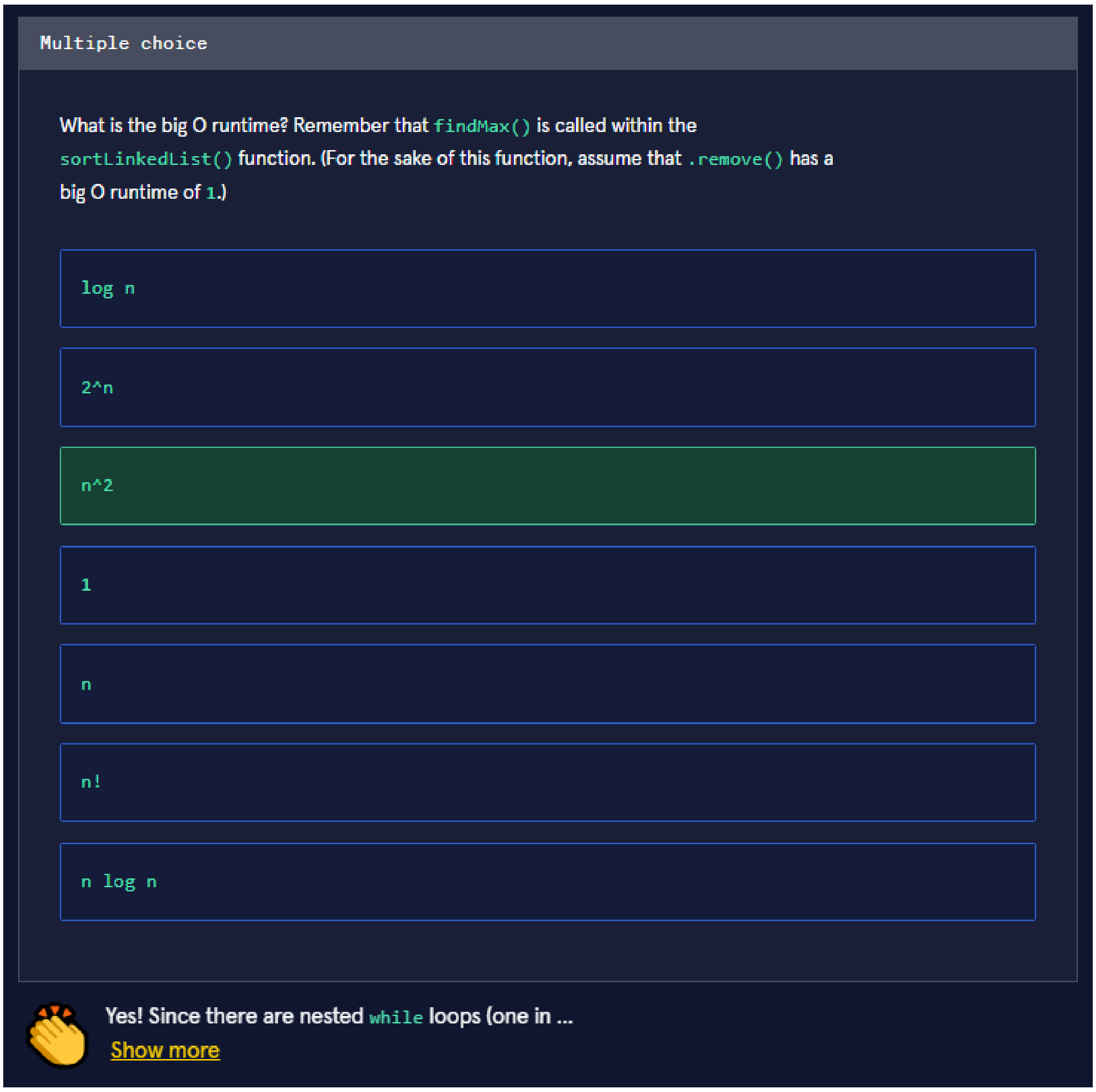


Fill in the blank

Fill in the `sortLinkedList()` function such that you return a new linked list that is sorted from smallest to largest. The function uses `findMax()` from above to return the largest element in the list.

```
function sortLinkedList(list) {
  let newList = new LinkedList();
  while ( list.head  !== null) {
    let currentMax = findMax(list);
      list  .remove(currentMax);
      newList  .addToHead(currentMax);
  }
  return  newList  ;
}
```

You got it!


Multiple choice

What is the big O runtime? Remember that `findMax()` is called within the `sortLinkedList()` function. (For the sake of this function, assume that `.remove()` has a big O runtime of `1`.)

- `log n`
- `2^n`
- [✓ `n^2`]
- `1`
- `n`
- `n!`
- `n log n`

Yes! Since there are nested `while` loops (one in ...

Show more

Stack Runtimes vs Queue Runtimes

In addition to analyzing the runtimes of various data structures, it is also important to compare the runtimes of different data structures.

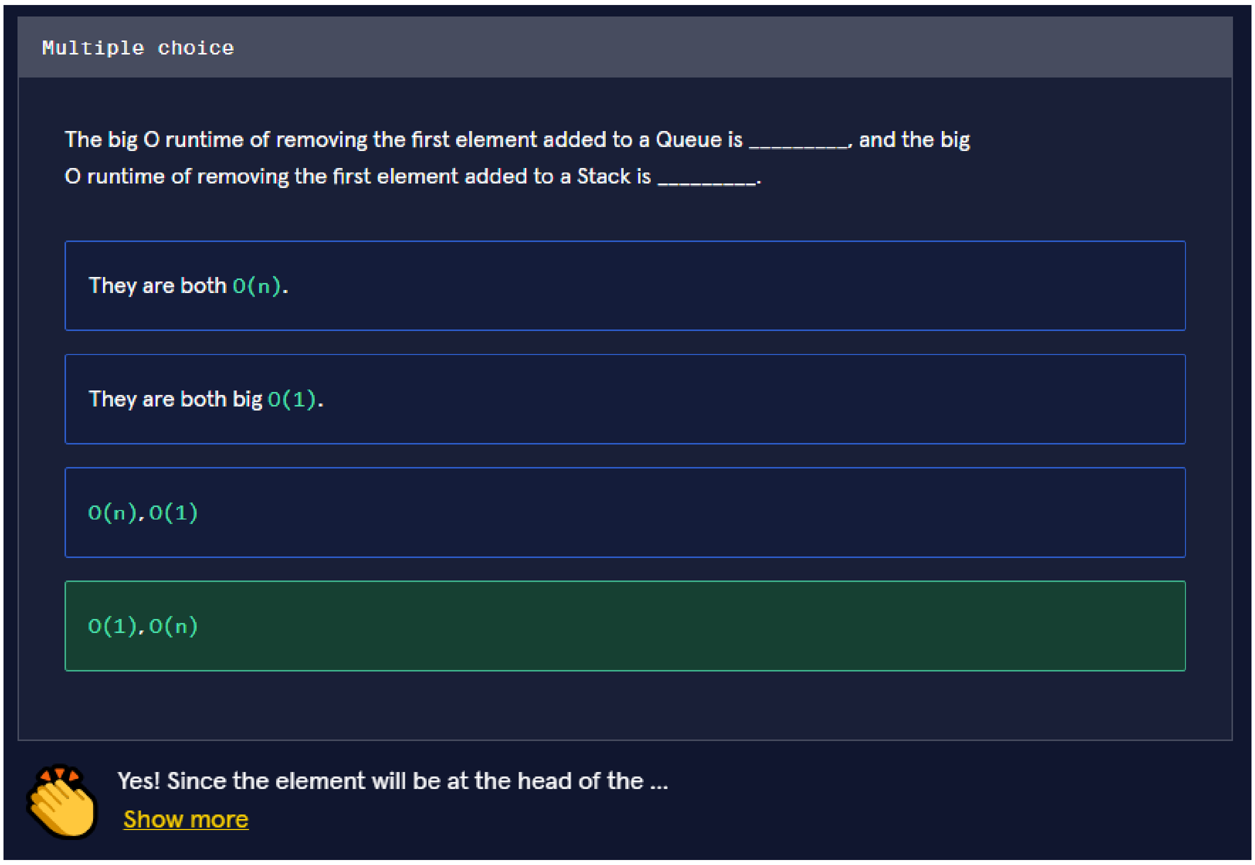
We will compare the runtimes of retrieving the first value added to a queue to the runtime of retrieving the first value added to a stack.

Removing the First Value Added to a Queue

A queue is a FIFO (first in, first out) data structure, which means that the first element added to it, will always be the first element removed from it. Removing this element does not require you to iterate through the queue.

Removing the First Value Added to a Stack

On the other hand, a stack is a FILO (first in, last out) data structure. This means that the first element added will be the last element removed. Removing this element will require you to iterate through the stack, all the way to the bottom.


Multiple choice

The big O runtime of removing the first element added to a Queue is ________, and the big O runtime of removing the first element added to a Stack is ________.

They are both `O(n)`.

They are both big `O(1)`.

`O(n)`, `O(1)`

[✓ `O(1)`, `O(n)`]

Yes! Since the element will be at the head of the ...
Show more

While finding the first value added to a queue has a better big O runtime than doing so in a stack, consider finding the *last* value added. In a queue, we will have to iterate through the entire queue to retrieve the element at the end. This will be a big O runtime of O(n). On the other hand, the last value added to a stack is the value at the top of the stack, so removing it will just be a big O runtime of O(1).

Hash Map Runtimes vs Linked List Runtimes

Similarly, let's compare the runtimes of searching for a particular element in a linked list and in a hash map.

Retrieving an Element from a Linked List

To find an element in a linked list, we will have to search through the entire list to see if the element is there. Refer to the findMax() function we looked at above for an example. Iterating through the list means that this process has a big O runtime of O(n).

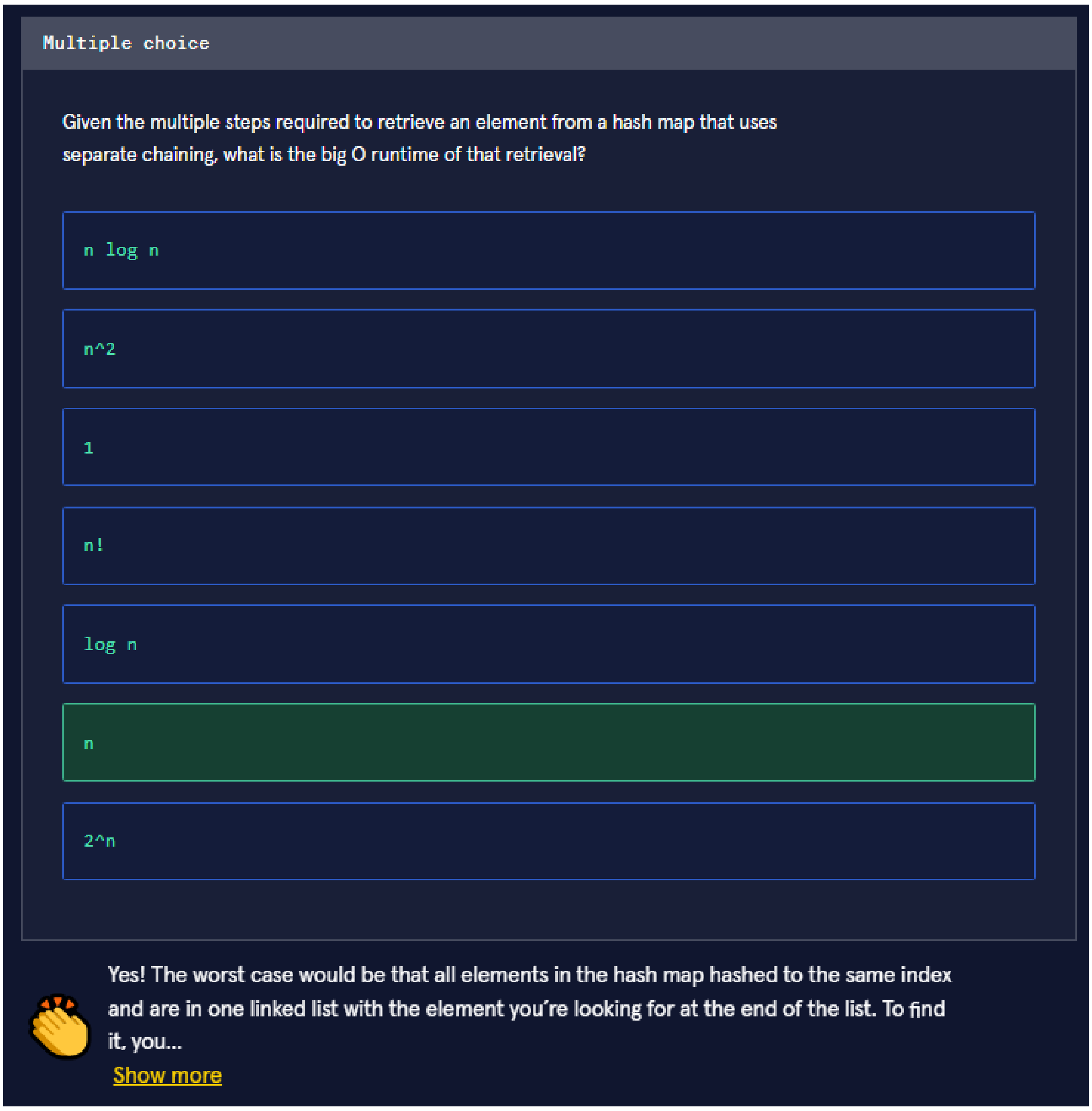
Retrieving an Element from a Hash Map

Retrieving an element from a hash map is more efficient, due to its structure. Hash maps store information using key-value pairs, which means that every value is linked to a unique key. In order to find the value from the key, it uses the hash function, which has a big O runtime of O(1). If you don't have to search through the entire data structure, retrieving an element from a hash map is faster than retrieving an element from a linked list.

However, there is the possibility that the element you are looking for is not at the spot that you expect it to be. This happens when two keys have the same hash. There are a few ways hash maps resolve this issue, including separate chaining and open addressing.

Separate Chaining

One way to solve hash map collisions is to create a linked list at the array index where the collision occurred. All elements that hash to the same index will be in that list. This means that to find an element in a hash map that uses separate chaining, you must first find the correct index, and then search through the list at that index (if there is more than one element).


Multiple choice

Given the multiple steps required to retrieve an element from a hash map that uses separate chaining, what is the big O runtime of that retrieval?

- n log n
- n^2
- 1
- n!
- log n
- [✓ n]
- 2^n

Yes! The worst case would be that all elements in the hash map hashed to the same index and are in one linked list with the element you're looking for at the end of the list. To find it, you...

Show more

Open Addressing

Another way to solve hash map collisions is to simply move down the array until you find an open index, and place the element there. This is a type of open addressing that is called linear probing. When retrieving an element from a hash map that uses linear probing, the worst case would be if the element hashes to the first index, but is actually at the last index. Since you would have to search through the entire array, the big O runtime for retrieving an element from this kind of hash map is O(n).