**MERGE SORT: CONCEPTUAL**

**What Is A Merge Sort?**

Merge sort is a sorting algorithm created by John von Neumann in 1945. Merge sort’s “killer app” was the strategy that breaks the list-to-be-sorted into smaller parts, sometimes called a *divide-and-conquer algorithm*.

In a divide-and-conquer algorithm, the data is continually broken down into smaller elements until sorting them becomes really simple.

Merge sort was the first of many sorts that use this strategy, and is still in use today in many different applications.

**How To Merge Sort:**

Merge sorting takes two steps: splitting the data into “runs” or smaller components, and the re-combining those runs into sorted lists (the “merge”).

When splitting the data, we divide the input to our sort in half. We then recursively call the sort on each of those halves, which cuts the halves into quarters. This process continues until all of the lists contain only a single element. Then we begin merging.

When merging two single-element lists, we check if the first element is smaller or larger than the other. Then we return the two-element list with the smaller element followed by the larger element.

**Merging**

When merging larger pre-sorted lists, we build the list similarly to how we did with single-element lists.

Let’s call the two lists left and right. Bothleft and right are already sorted. We want to combine them (to *merge* them) into a larger sorted list, let’s call it both. To accomplish this we’ll need to iterate through both with two indices, left\_index and right\_index.

At first left\_index and right\_index both point to the start of their respective lists. left\_index points to the smallest element of left (its first element) and right\_index points to the smallest element of right.

Compare the elements at left\_index and right\_index. The smaller of these two elements should be the first element of both because it’s the smallest of both! It’s the smallest of the two smallest values.

Let’s say that smallest value was in left. We continue by incrementing left\_index to point to the next-smallest value in left. Then we compare the 2nd smallest value in left against the smallest value of right. Whichever is smaller of these two is now the 2nd smallest value of both.

This process of “look at the two next-smallest elements of each list and add the smaller one to our resulting list” continues on for as long as both lists have elements to compare. Once one list is exhausted, say every element from left has been added to the result, then we know that all the elements of the other list, right, should go at the end of the resulting list (they’re larger than every element we’ve added so far).

**Merge Sort Performance**

Merge sort was unique for its time in that the best, worst, and average time complexity are all the same: Θ(N\*log(N)). This means an almost-sorted list will take the same amount of time as a completely out-of-order list. This is acceptable because the worst-case scenario, where a sort could stand to take the most time, is as fast as a sorting algorithm can be.

Some sorts attempt to improve upon the merge sort by first inspecting the input and looking for “runs” that are already pre-sorted. [Timsort](https://en.wikipedia.org/wiki/Timsort" \t "_blank) is one such algorithm that attempts to use pre-sorted data in a list to the sorting algorithm’s advantage. If the data is already sorted, Timsort runs in Θ(N) time.

Merge sort also requires space. Each separation requires a temporary array, and so a merge sort would require enough space to save the whole of the input a second time. This means the worst-case space complexity of merge sort is O(N).

**MERGE SORT: JAVASCRIPT**

**Introduction**

In this lesson, you will learn how to implement the merge sort algorithm in JavaScript. The algorithm consists of two distinct steps:

1. *Splitting the input array* – The algorithm recursively splits the input array until each element is in its own array. This portion of the algorithm is represented in the top half of the image to the right.
2. *Merging sorted arrays* – The algorithm compares and combines the elements of arrays until the input array is sorted. This is shown in the bottom half of the image.

**Splitting: Base Case**

In this implementation of merge sort, you will build a recursive function, called mergeSort(), that splits the input array until each element is in its own array.

So, if the input array is:

[3, 5, 2]

splitting these elements into their own arrays will look like:

[3]  
[5]  
[2]

The base case of this recursive function is when the input array has only one element in it. Below is a pseudocode implementation of the base case:

function mergeSort(arr)

if the length of arr equals 1

return arr

**Instructions**

**1.**

In mergeSort(), add a line that saves the length of startArray to a constant called length.

Checkpoint 2 Passed

**2.**

Add an if statement that checks if length is equal to 1. If it is, return startArray.

Checkpoint 3 Passed

ANSWER: index.js

const mergeSort = (startArray) => {

  const length = startArray.length;

  if (startArray.length === 1) {

    return startArray;

  }

}

const inputArr = [3];

console.log(mergeSort(inputArr));

module.exports = {

  mergeSort

};

**Splitting: Recursive Case**

The recursive case of our mergeSort() function requires that we first split the input array into a leftArray and rightArray:

function mergeSort(arr)

if the length of arr equals 1

return arr

midIndex = the floor integer of (left + right) / 2

leftArr = arr from 0 to midIndex

rightArr = arr from midIndex to end

In the example above, the leftArray is equal to the input arr from 0 to the middle index. The right array is from the middle index to the end.

Next, we pass the left and right arrays into the mergeSort() function:

function mergeSort(arr)

if the length of arr equals 1

return arr

midIndex = the floor integer of (left + right) / 2

leftArray = arr from 0 to midIndex

rightArray = arr from midIndex to end

mergeSort(leftArray)

mergeSort(rightArray)

This is our recursive call.

**Instructions**

**1.**

Create a constant variable called mid and set it equal to the floor of length / 2.

Checkpoint 2 Passed

**2.**

Now you are going to create the left and right arrays.

Create a variable called leftArray and set it equal to a new array of elements from startArray, from 0 to mid.

Create a variable called rightArray and set it equal to the elements of startArray from mid to length.

Checkpoint 3 Passed

**3.**

On two separate lines, pass leftArray into mergeSort() and rightArray into mergeSort().

We added a console.log() statement to the if block, so you should see each element of the input array ([3, 5, 2, 90, 4, 7]) printed to the console when you run your code.

Checkpoint 4 Passed

Answer: index.js

const mergeSort = (startArray) => {

  const length = startArray.length;

  if (length === 1) {

    console.log(startArray);

    return startArray;

  }

const mid = Math.floor(length / 2);

const leftArray = startArray.slice(0, mid);

const rightArray = startArray.slice(mid, length);

  mergeSort(leftArray);

  mergeSort(rightArray);

}

const inputArr = [3, 5, 2, 90, 4, 7];

console.log(mergeSort(inputArr));

module.exports = {

  mergeSort

};

**Call merge() from mergeSort()**

At this point, we have a function that recursively splits the input array until each element is in a single-element array. The final step is to call the function that is responsible for merging the leftArray and rightArray.

function mergeSort(arr)

if the length of arr equals 1

return arr

midIndex = the floor integer of (left + right) / 2

leftArray = arr from 0 to midIndex

rightArray = arr from midIndex to end

return merge(mergeSort(leftArray), mergeSort(rightArray))

In the last line of the pseudocode, we call a function named merge(). In the next exercise, you will implement a merge() function that combines the sorted leftArray and rightArray halves into a larger sorted array.

**Instructions**

**1.**

We added a merge() function to **index.js** that prints leftArray and rightArray to the console.

In mergeSort() return a call to merge(). Pass mergeSort(leftArray) and mergeSort(rightArray) as the first and second arguments.

Checkpoint 2 Passed

Answer: index.js

const mergeSort = (startArray) => {

  const length = startArray.length;

  if (length === 1) {

    return startArray;

  }

  const mid = Math.floor(length / 2);

  const leftArray = startArray.slice(0, mid);

  const rightArray = startArray.slice(mid, length);

  return merge(mergeSort(leftArray), mergeSort(rightArray));

}

const merge = (leftArray, rightArray) => {

  console.log(leftArray);

  console.log(rightArray);

}

const inputArr = [3, 5, 2, 90, 4, 7];

console.log(mergeSort(inputArr));

module.exports = {

  mergeSort

};

**Merging**

Now, let’s turn our attention to the merge() function. First, let’s think about its arguments and what it returns:

* Arguments: two sorted lists as inputs (leftArray and rightArray)
* Returns: a sorted list with the elements of leftArray and rightArray combined. We will call this new array sortedArray

Let’s break the implementation of this function into three parts:

* Create a while loop that continues while there are still elements in leftArray and rightArray.

function merge(leftArray, rightArray)

sortedArray = []

while leftArray and rightArray have a length greater than 0

// Do something

return sortedArray

* Create conditions that adds an element to sortedArray with each loop.

function merge(leftArr, rightArr)

sortedArray = []

while leftArray and rightArray have a length greater than 0

if leftArray[0] is less than rightArray[0]

push leftArray[0] onto sortedArray

remove leftArray[0] from leftArray

else

push rightArray[0] onto sortedArray

remove rightArray[0] from rightArray

This code will add the smaller number, between leftArray[0] and rightArray[0] to the new array. Then, it will remove that number from the array.

* Return sortedArray, with leftArray and rightArray concatenated.

function merge(leftArr, rightArr)

sortedArray = []

while leftArray and rightArray have a length greater than 0

if leftArray[0] is less than rightArray[0]

push leftArray[0] onto sortedArray

remove leftArray[0] from leftArray

else

push rightArray[0] onto sortedArray

remove rightArray[0] from rightArray

return sortedArray with leftArray and rightArray concatenated

Because the while loop continues until either leftArray or rightArray is empty, you need to concatenate whatever is left in the other array to the sorted array. In JavaScript, it’s easiest to implement this by concatenating both arrays, because the empty array will not alter the original.

### Instructions

**1.**

You will implement the entire merge() function in this checkpoint. Follow the steps below to do so:

1. Implement a while loop that continues until leftArray and rightArray are empty.
2. Add an if statement that checks if leftArray[0] is less than rightArray[0]. If it is, append that number to sortedArray and remove it from leftArray
3. Add an else statement that appends rightArray[0] to sortedArray and then removes that number from rightArray.
4. Once the while loop has completed, return the sortedArray with leftArray and rightArray concatenated.

Checkpoint

Answer: index.js

const mergeSort = (startArray) => {

  const length = startArray.length;

  if (length === 1) {

    return startArray;

  }

  const mid = Math.floor(length / 2);

  const leftArray = startArray.slice(0, mid);

  const rightArray = startArray.slice(mid, length);

  return merge(mergeSort(leftArray), mergeSort(rightArray))

}

const merge = (leftArray, rightArray) => {

  const sortedArray = [];

  while (leftArray.length > 0 && rightArray.length > 0) {

    if (leftArray[0] < rightArray[0]) {

      sortedArray.push(leftArray[0]);

      leftArray.shift();

    } else {

      sortedArray.push(rightArray[0]);

      rightArray.shift();

    }

  }

  return sortedArray.concat(leftArray).concat(rightArray);

}

const inputArr = [3, 5, 2, 90, 4, 7];

console.log(mergeSort(inputArr));

module.exports = {

  mergeSort

};

**Review**

Nice work building your own mergeSort() function. An important point to remember about merge sort is that the algorithm is broken into two parts: splitting and merging.

Regardless of the order or length (including odd or even lengths) of an input array, the merge sort algorithm will always split the elements into their own arrays first, and then combine them into a sorted array. The fact that the same steps are taken regardless of the input (order and length) results in an average, best, and worst case complexity all equal to the same value, O(n log n).

This time complexity makes merge sort one of the most efficient and popular sorting algorithms. Take a look at merge sort compared to a few others on [toptal.com](https://www.toptal.com/developers/sorting-algorithms).

**Instructions**

In **index.js**, we included the solution code from the last exercise. If you’re up for the challenge, see if you can combine the two lines in your if and else blocks into one line to make your code look a little cleaner.

Answer: index.js

const mergeSort = (startArray) => {

  const length = startArray.length;

  if (length === 1) {

    return startArray;

  }

  const mid = Math.floor(length / 2);

  const leftArray = startArray.slice(0, mid);

  const rightArray = startArray.slice(mid, length);

  return merge(mergeSort(leftArray), mergeSort(rightArray))

}

const merge = (leftArray, rightArray) => {

  const sortedArray = [];

  while (leftArray.length > 0 && rightArray.length > 0) {

    if (leftArray[0] < rightArray[0]) {

      sortedArray.push(leftArray[0]);

      leftArray.shift();

    } else {

      sortedArray.push(rightArray[0]);

      rightArray.shift();

    }

  }

  return sortedArray.concat(leftArray).concat(rightArray);

}

const inputArr = [3, 5, 2, 90, 4, 7];

console.log(mergeSort(inputArr));

module.exports = {

  mergeSort

};

**QUICKSORT: CONCEPTUAL**

**Introduction to Quicksort**

Quicksort is an efficient recursive algorithm for sorting arrays or lists of values. The algorithm is a *comparison* sort, where values are ordered by a comparison operation such as > or <.

Quicksort uses a *divide and conquer* strategy, breaking the problem into smaller sub-problems until the solution is so clear there’s nothing to solve.

The problem: many values in the array which are out of order.

The solution: break the array into sub-arrays containing **at most** one element. One element is sorted by default!

We choose a single *pivot* element from the list. Every other element is compared with the pivot, which *partitions* the array into three groups.

1. A sub-array of elements **smaller than** the pivot.
2. The pivot itself.
3. A sub-array of elements **greater than** the pivot.

The process is repeated on the sub-arrays until they contain zero or one element. Elements in the “smaller than” group **are never compared** with elements in the “greater than” group. If the smaller and greater groupings are roughly equal, this cuts the problem in half with each partition step!

[6,5,2,1,9,3,8,7]  
6 # The pivot  
[5, 2, 1, 3] # lesser than 6  
[9, 8, 7] # greater than 6  
   
   
[5,2,1,3]  # these values  
# will never be compared with   
[9,8,7] # these values

Depending on the implementation, the sub-arrays of one element each are recombined into a new array with sorted ordering, or values within the original array are swapped in-place, producing a sorted mutation of the original array.

**Quicksort Runtime**

The key to Quicksort’s runtime efficiency is the division of the array. The array is partitioned according to comparisons with the pivot element, so which pivot is the optimal choice to produce sub-arrays of roughly equal length?

The graphic displays two data sets which always use the *first* element as the pivot. Notice how many more steps are required when the quicksort algorithm is run on an already sorted input. The partition step of the algorithm hardly divides the array at all!

The worst case occurs when we have an imbalanced partition like when the first element is continually chosen in a sorted data-set.

One popular strategy is to select a random element as the pivot for each step. The benefit is that *no particular data set* can be chosen ahead of time to make the algorithm perform poorly.

Another popular strategy is to take the first, middle, and last elements of the array and choose the median element as the pivot. The benefit is that the division of the array tends to be more uniform.

Quicksort is an unusual algorithm in that the worst case runtime is O(N^2), but the average case is O(N \* logN).

We typically only discuss the worst case when talking about an algorithm’s runtime, but for Quicksort it’s so uncommon that we generally refer to it as O(N \* logN).

**Quicksort Review**

Quicksort is an efficient algorithm for sorting values in a list. A single element, the pivot, is chosen from the list. All the remaining values are partitioned into two sub-lists containing the values smaller than and greater than the pivot element.

Ideally, this process of dividing the array will produce sub-lists of nearly equal length, otherwise, the runtime of the algorithm suffers.

When the dividing step returns sub-lists that have one or less elements, each sub-list is sorted. The sub-lists are recombined, or swaps are made in the original array, to produce a sorted list of values.

For a recursive version of quicksort, what would be the base case when the algorithm stops recursing?

Correct! A single element array is sorted. These elements are then either recombined, or if the sort was “in-place”, they’ve already been swapped into the correct location.

When the array passed in as an argument is no longer than one element.

**QUICKSORT: JAVASCRIPT**

**Introduction**

Quicksort is an efficient sorting algorithm that is based on the divide and conquer strategy. Like merge sort, the input array is partitioned into smaller parts and then combined after the elements have been rearranged. Unlike merge sort, which requires additional memory for auxiliary arrays, quicksort is space-saving because it deploys in-place sorting.

As runtime performance goes, quicksort requires more comparisons for sorting a larger input than mergesort. Like bubble sort, quicksort has a worst case runtime of O(N^2). This can happen when quicksort’s input data set comprises:

* pre-sorted numbers,
* backward-sorted numbers, or
* all similar elements along with a poorly chosen pivot element that produces a partition of zero or one element.

On average, like merge sort, the runtime of quicksort is O(N \* log N) if partition sizes are roughly equal.

The basic idea of the quicksort algorithm is to:

* split the initial unsorted data set into a left partition and a right partition
* sort each partition recursively until there is only one element left
* return the sorted array

We use a pivot element to divide our unsorted array into two parts. The elements in these parts must meet these conditions after partitioning:

* all elements in the left partition must be less than or equal to the pivot element
* all elements in the right partition must be greater than or equal to the pivot element

Determining the pivot index is done through a procedure called partitioning. Our algorithm uses an array to store the data set and stipulates the boundary of the data set with left and right pointers. The pseudocode for our quicksort algorithm is as follows:

If there is more than one element left in the array:

Find the pivot index through partitioning

If the left pointer is less than the pivot index:

Call quicksort() on the portion of the array between the left pointer and the pivot.

If the pivot index is less than the right pointer:

Call quicksort() on the portion of the array between the pivot index and the right pointer.

Return the sorted array

The partitioning process will be explained in later exercises.

### Instructions

**1.**

The code in **script.js** creates a small set of five random numbers and sorts it using our quicksort algorithm. Run the script several times to see the sorted results.

Move to the next exercise when you are ready to continue.

Checkpoint 2 Passed

Answer: scritp.js

const { quicksort, partition } = require('./quicksort');

const randomize = () => Math.floor(Math.random() \* 40);

let numbers = [];

for (let i = 0; i < 5; i++) {

  numbers.push(randomize());

}

console.log('Before quicksort:', numbers);

const sorted = quicksort(numbers);

console.log('After  quicksort:', sorted);

quickSort.js

const swap = require('./swap');

const quicksort = (array, leftBound = 0, rightBound = array.length - 1) => {

  if (leftBound < rightBound) {

    const pivotIndex = partition(array, leftBound, rightBound);

    quicksort(array, leftBound, pivotIndex - 1);

    quicksort(array, pivotIndex, rightBound);

  }

  return array;

}

const partition = (array, leftIndex, rightIndex) => {

  const pivot = array[Math.floor((rightIndex + leftIndex) / 2)];

  while (leftIndex <= rightIndex) {

    while (array[leftIndex] < pivot) {

      leftIndex++;

    }

    while (array[rightIndex] > pivot) {

      rightIndex--;

    }

    if (leftIndex <= rightIndex) {

      swap(array, leftIndex, rightIndex);

      leftIndex++;

      rightIndex--;

    }

  }

  return leftIndex;

}

module.exports = {

  quicksort,

  partition,

};

Swap.js

const swap = (arr, indexOne, indexTwo) => {

  const temp = arr[indexTwo];

  arr[indexTwo] = arr[indexOne];

  arr[indexOne] = temp;

}

module.exports = swap;

**Partitioning Part I - The Pivot Element**

Partitioning is the crux of the quicksort algorithm. Without it, we wouldn’t know how to split our unsorted array into useful partitions.

This procedure utilizes two internal indices, leftIndex and rightIndex that move in opposite directions. These indices are used for:

* computing the pivot element
* comparing the elements located at each index with the pivot element
* determining the pivot index, the desired location of the pivot element in the set after elements have been swapped, if any

The basic idea of partitioning is as follows:

* Start with the middle element
* While you haven’t looked through the whole array (leftIndex is still < rightIndex)
  + move leftIndex up until you find something greater than the pivot
  + move rightIndex down until you find something less than the pivot
  + swap those elements, and move the indices in by one step so to continue checking if swaps are necessary
* return the last left element index

An initial pivot element can be arbitrarily chosen in the beginning of the partitioning process to be one of the following by default:

* first element of the array
* last element of the array
* middle element of the array
* random element of the array

The final location of the pivot element will be determined at the end of the partitioning process.

In some quicksort implementations, the first or last element is commonly picked as the pivot element. In our JavaScript implementation, we will use the middle element instead because it provides a better average runtime. To do this, we will need both leftIndex and rightIndex.

pivot = the average of the sum of leftIndex and rightIndex rounded down

### Instructions

**1.**

In **quicksort.js**, an empty arrow function, partition() has been defined for you. Add three parameters: array, leftIndex, and rightIndex to it.

Checkpoint 2 Passed

**2.**

Inside partition():

* define a const variable, pivot, to be the pivot element
* assign pivot to be the middle element of the input array
* round pivot down to produce a whole number.
* temporarily return pivot for now

answer: quicksort.js

const swap = require('./swap');

/\* Define partition() here \*/

const partition = (array, leftIndex, rightIndex) => {

  const pivot = array[Math.floor((rightIndex + leftIndex) / 2)];

  return pivot;

}

module.exports = {

  partition

};

Swap.js

const swap = (arr, indexOne, indexTwo) => {

  const temp = arr[indexTwo];

  arr[indexTwo] = arr[indexOne];

  arr[indexOne] = temp;

}

module.exports = swap;

**Partitioning Part II - The Left and Right Indices**

The leftIndex and rightIndex of a set or subset are going to set the bounds of the partition. For the first iteration, both indices mark the entire span of the original data set. In the following illustrations, L and R represent leftIndex and rightIndex respectively.

[ 3, 1, 4, 2, 8, 9 ]

L P R

The pivot element for this set will be 4 as it is located near the halfpoint of the data set and indicated by P.

Next, we want to compare the element at leftIndex with the pivot element, 4. As long as it is less than the pivot, meaning that it is in the correct half of the partition, we want to move the leftIndex forward one step to the right.

3 < 4, move L forward

[ 3, 1, 4, 2, 8, 9 ]

L P R

1 < 4, move L forward

[ 3, 1, 4, 2, 8, 9 ]

L R

P

4 = 4, stop

We stop leftIndex at position 2 because the element at index 2 (4) is not less than the pivot element 4. Next, we switch focus to the rightIndex and compare the element at rightIndex with the pivot element, 4. As long as it is greater than the pivot, we want to move the rightIndex backward one step to the left.

[ 3, 1, 4, 2, 8, 9 ]

L R

P

9 > 4, move R backward

[ 3, 1, 4, 2, 8, 9 ]

L R

P

8 > 4, move R backward

[ 3, 1, 4, 2, 8, 9 ]

L R

P

2 < 4, stop

We stop the rightIndex at position 3 because the element at 3 (2) is not greater than the pivot element 4. This tells us that 2 does not belong in its current position because it should be on the left of the pivot element 4. In this case, we need to swap the elements at leftIndex and rightIndex.

We will handle swapping of index elements in the next exercise.

**Instructions**

**1.**

Open up **quicksort.js**. In the function, partition(), below the declaration of pivot, write a while loop that will execute as long as leftIndex is less than or equal to rightIndex.

Checkpoint 2 Passed

**2.**

Inside this while loop, write another while loop that increments leftIndex as long as the element at leftIndex is less than pivot.

Checkpoint 3 Passed

**3.**

Below the inner while loop, write a while loop that decrements rightIndex as long as the element at rightIndex is greater than pivot.

Checkpoint 4 Passed

Answer: quicksort.js

const swap = require('./swap');

const partition = (array, leftIndex, rightIndex) => {

  const pivot = array[Math.floor((rightIndex + leftIndex) / 2)];

  while (leftIndex <= rightIndex) {

    while (array[leftIndex] < pivot) {

      leftIndex++;

    }

    while (array[rightIndex] > pivot) {

      rightIndex--;

    }

  }

}

module.exports = {

  partition

};

Swap.js

const swap = (arr, indexOne, indexTwo) => {

  const temp = arr[indexTwo];

  arr[indexTwo] = arr[indexOne];

  arr[indexOne] = temp;

}

module.exports = swap;

script.js

const { quicksort, partition } = require('./quicksort');

const randomize = () => Math.floor(Math.random() \* 40);

let numbers = [];

for (let i = 0; i < 5; i++) {

  numbers.push(randomize());

}

console.log('Before quicksort:', numbers);

const sorted = quicksort(numbers);

console.log('After  quicksort:', sorted);

**Partitioning Part III - Swapping**

Recall that our leftIndex and rightIndex were at 2 and 3 respectively. They cannot move any further because their respective elements are greater than or less than the pivot element. When this happens, we need to swap those elements so that they will end up at the correct side of the partition.

[ 3, 1, 4, 2, 8, 9 ]

L R

P

swap 4 and 2

[ 3, 1, 2, 4, 8, 9 ]

L R

P

After we swap them, we move L forward and R backward.

Move L forward and R backward

[ 3, 1, 2, 4, 8, 9 ]

R L

P

We return to our outer while loop condition to check if leftIndex (3) is less than or equal to rightIndex (2). In this case, 3 > 2, so we exit the while loop.

At this juncture, the elements that are less than the pivot are to the left of it and the elements that are greater than the pivot are to the right of it. We can stop partitioning and return the leftIndex which points to the pivot element 4. Hence, our pivot index is 3 which is also the leftIndex.

### Instructions

**1.**

At this juncture, the leftIndex should be pointing at an element that is not less than the pivot element and the rightIndex should be pointing at an element that is not greater than the pivot element. Since these elements are on the wrong side of the partition because they do not meet the heap condition, they need to be swapped.

Inside the outer while loop, after the previous two loops, determine if the leftIndex and rightIndex can be swapped by checking if leftIndex is still less than or equal to rightIndex.

If it is, do the following:

* swap the elements at both indices using the swap(arrayToSwap, indexOne, indexTwo) helper function.
* increment leftIndex
* decrement rightIndex

Checkpoint 2 Passed

**2.**

Outside the outer while loop, before exiting partition(), return leftIndex.

Checkpoint 3 Passed

Answer: quicksort.js

const swap = require('./swap');

const partition = (array, leftIndex, rightIndex) => {

  const pivot = array[Math.floor((rightIndex + leftIndex) / 2)];

  while (leftIndex <= rightIndex) {

    while (array[leftIndex] < pivot) {

      leftIndex++;

    }

    while (array[rightIndex] > pivot) {

      rightIndex--;

    }

    if (leftIndex <= rightIndex) {

      swap(array, leftIndex, rightIndex);

      leftIndex++;

      rightIndex--;

    }

  }

  return leftIndex;

}

module.exports = {

  partition

};

Swap.js

const swap = (arr, indexOne, indexTwo) => {

  const temp = arr[indexTwo];

  arr[indexTwo] = arr[indexOne];

  arr[indexOne] = temp;

}

module.exports = swap;

**Recursive Quicksort**

Now that we have finished implementing partition(), let’s implement the quicksort() function, which is recursive. This function takes in three parameters:

* Input array
* Left pointer
* Right pointer

The base case for this function is when the array has one element, meaning that it is sorted. As a result, the array is returned. Our JavaScript implementation does in-place sorting, hence, the array size does not change. A one-element array is symbolized by both left and right pointers pointing to the same element.

Our quicksort() function will start by calling the partition() function with the input array bounded by the left and right pointers as long as the left pointer is less than the right pointer.

The recursive steps are executed after partitioning:

* Call quicksort() to process only the left partition bounded by the left pointer and (pivot index - 1) to exclude the pivot element from the left partition
* Call quicksort() to process only the right partition bounded by the pivot index and right pointer

Continuing from the example in the last exercise, recall that we returned a pivot index, P, that points to pivot element 4 at index 3 as pointed to by L.

[ 3, 1, 2, 4, 8, 9 ]

R L

P

Recall that the initial left pointer, which we will call leftBound is 0 and the initial right pointer, rightBound, is 4.

Recursively call quicksort() with the array [ 3, 1, 2, 4, 8, 9 ], left pointer 0 and right pointer 2 for the left partition [ 3, 1, 2 ] which excludes the pivot index, 3.

Similarly, we will recursively call quicksort() with the array [ 3, 1, 2, 4, 8, 9 ], left pointer 3 and right pointer 5 for the right partition [ 4, 8, 9 ] which includes the pivot index, 3.

### Instructions

**1.**

In **quicksort.js**, an empty quicksort() arrow function has been defined for you.

* Add 3 parameters: array, leftBound and rightBound to it.
* Assign leftBound to have a default value of 0
* Assign rightBound to have a default value of array.length-1

Checkpoint 2 Passed

**2.**

Recall that quicksort will stop recursing when there is only one element left. When this happens, we return the sorted array. Let’s implement the recursive case where the array has more than one element.

Inside quicksort(), write an if statement where the input array has more than one element. Since we will always pass the same array and do in-place swaps, the array .length property won’t change between calls. To tell if the array has more than one element, check if rightBound is greater than leftBound.

Inside the if block, do the following:

* create a const variable, pivotIndex and assign it to the return value of calling partition() with the input array, leftBound and rightBound.

Outside of the if block, return the input array.

Checkpoint 3 Passed

**3.**

Once we have determined the location of the pivot element, we can now call quicksort() to recursively sort the left array partition excluding the pivot element.

Inside your if statement, below the declaration of pivotIndex, call quicksort() with the input array, leftBound and pivotIndex - 1 as parameters.

Checkpoint 4 Passed

**4.**

We will also call quicksort() to recursively sort the elements in the right array partition inclusive of the pivot element.

Inside the same if statement, call quicksort() with the input array, pivotIndex and rightBound as parameters.

Checkpoint 5 Passed

ANSWER: quicksort.js

const swap = require('./swap');

const quicksort = (array, leftBound = 0, rightBound = array.length - 1) => {

  if (rightBound > leftBound) {

   const pivotIndex = partition(array, leftBound, rightBound);

   quicksort(array, leftBound, pivotIndex - 1);

   quicksort(array, pivotIndex, rightBound);

  }

  return array;

}

const partition = (array, leftIndex, rightIndex) => {

  const pivot = array[Math.floor((rightIndex + leftIndex) / 2)];

  while (leftIndex <= rightIndex) {

    while (array[leftIndex] < pivot) {

      leftIndex++;

    }

    while (array[rightIndex] > pivot) {

      rightIndex--;

    }

    if (leftIndex <= rightIndex) {

      swap(array, leftIndex, rightIndex);

      leftIndex++;

      rightIndex--;

    }

  }

  return leftIndex;

}

module.exports = {

  partition,

  quicksort

};

Swap.js

const swap = (arr, indexOne, indexTwo) => {

  const temp = arr[indexTwo];

  arr[indexTwo] = arr[indexOne];

  arr[indexOne] = temp;

}

module.exports = swap;

**Logging**

Let’s put our implementation of the quicksort algorithm into practice. In order to understand what is going on internally inside each call to quicksort() and partition(), we have inserted log statements at various steps to illustrate the following events:

* a recursive call is about to occur
* partitioning is taking place
* leftIndex and rightIndex are incremented
* swapping has taken place

### Instructions

**1.**

Open **script.js** and run it. Each run will produce a randomized input array. Study the output statements to get a better understanding of the quicksort implementation. Open **quicksort.js** to study the implementation of the log statements.

Checkpoint 2 Passed

Answer: script.js

const { quicksort, partition } = require('./quicksort');

const randomize = () => Math.floor(Math.random() \* 40);

let numbers = [];

for (let i = 0; i < 5; i++) {

  numbers.push(randomize());

}

console.log('Before quicksort:', numbers);

const sorted = quicksort(numbers);

console.log('After  quicksort:', sorted);

quicksort.js

const swap = require('./swap');

const quicksort = (array, leftBound = 0, rightBound = array.length - 1) => {

  if (leftBound < rightBound) {

    console.log('. Calling partition', array, `with leftBound ${leftBound} and rightBound ${rightBound}`);

    const pivotIndex = partition(array, leftBound, rightBound);

    console.log(`. Returning pivotIndex = ${pivotIndex}`);

    console.log(`\nCalling quicksort for left partition with leftBound ${leftBound} and (pivotIndex-1) ${pivotIndex - 1}`);

    quicksort(array, leftBound, pivotIndex - 1);

    console.log(`\nCalling quicksort for right partition with pivotIndex ${pivotIndex} and rightBound ${rightBound}`);

    quicksort(array, pivotIndex, rightBound);

  }

  return array;

}

const partition = (array, leftIndex, rightIndex) => {

  const pivot = array[Math.floor((rightIndex + leftIndex) / 2)];

  console.log(`.. Partitioning with pivot ${pivot} leftIndex ${leftIndex} rightIndex ${rightIndex}`);

  while (leftIndex <= rightIndex) {

    while (array[leftIndex] < pivot) {

      leftIndex++;

      console.log(`.. ${array[leftIndex-1]} < ${pivot} : Incremented leftIndex => ${leftIndex}`);

    }

    while (array[rightIndex] > pivot) {

      rightIndex--;

      console.log(`.. ${array[rightIndex+1]} > ${pivot} : Decremented rightIndex => ${rightIndex}`);

    }

    if (leftIndex <= rightIndex) {

      const string = `${leftIndex} <= ${rightIndex}`;

      swap(array, leftIndex, rightIndex);

      console.log(`.. ${string} : Swapped leftIndex ${leftIndex} with rightIndex ${rightIndex}`, array);

      leftIndex++;

      rightIndex--;

      console.log(`......... : Incremented leftIndex => ${leftIndex} Decremented rightIndex => ${rightIndex}`);

    }

  }

  return leftIndex;

}

module.exports = {

  quicksort,

  partition

};

Swap.js

const swap = (arr, indexOne, indexTwo) => {

  const temp = arr[indexTwo];

  arr[indexTwo] = arr[indexOne];

  arr[indexOne] = temp;

}

module.exports = swap;

**Review**

* Quicksort is a divide-and-conquer algorithm that splits an unsorted data set into two partitions recursively and sorts the partitioned arrays in-place until there is only one element left in a partition.
* To determine the elements that belong in a partition, we need a pivot element, pivot, that is sandwiched between the two partitions and its location called the pivotIndex.
* We implemented the partition() function which groups and swaps the elements either to the left or right of the pivot element and returns the leftIndex that is the same as the pivotIndex.
* We implemented the quicksort() function that first calls partition() to determine the pivotIndex then recursively calls itself to sort the smaller partitions until there is only one element left in the partition.

### Instructions

**1.**

Open up script.js and run it. The size of the input array is 1,000,000 pre-sorted backwards. You should find the results sorted correctly. The results of running the script should look like this:

Before quicksort number @ index 0 = 1000000

Before quicksort number @ index 250000 = 750000

Before quicksort number @ index 500000 = 500000

Before quicksort number @ index 750000 = 250000

Before quicksort number @ index 999999 = 1

---

After quicksort number @ index 0 = 1

After quicksort number @ index 249999 = 250000

After quicksort number @ index 499999 = 500000

After quicksort number @ index 749999 = 750000

After quicksort number @ index 999999 = 1000000

Answer: script.js

const { quicksort, partition } = require('./quicksort');

let numbers = [];

let max = 1000000;

for (let i = max; i > 0; i--) {

  numbers.push(i);

}

console.log(`Before  quicksort number @ index      0 = ${numbers[0]}`);

console.log(`Before  quicksort number @ index ${max/4} = ${numbers[max/4]}`);

console.log(`Before  quicksort number @ index ${max/2} = ${numbers[max/2]}`);

console.log(`Before  quicksort number @ index ${3\*max/4} = ${numbers[3\*max/4]}`);

console.log(`Before  quicksort number @ index ${max-1} = ${numbers[max - 1]}`);

const sorted = quicksort(numbers);

console.log(`---`);

console.log(`After   quicksort number @ index      0 = ${sorted[0]}`);

console.log(`After   quicksort number @ index ${max/4-1} = ${sorted[max/4-1]}`);

console.log(`After   quicksort number @ index ${max/2-1} = ${sorted[max/2-1]}`);

console.log(`After   quicksort number @ index ${3\*max/4-1} = ${sorted[3\*max/4-1]}`);

console.log(`After   quicksort number @ index ${max-1} = ${sorted[max - 1]}`);

quicksort.js

const swap = require('./swap');

const quicksort = (array, leftBound = 0, rightBound = array.length - 1) => {

  if (leftBound < rightBound) {

    const pivotIndex = partition(array, leftBound, rightBound);

    quicksort(array, leftBound, pivotIndex - 1);

    quicksort(array, pivotIndex, rightBound);

  }

  return array;

}

const partition = (array, leftIndex, rightIndex) => {

  const pivot = array[Math.floor((rightIndex + leftIndex) / 2)];

  while (leftIndex <= rightIndex) {

    while (array[leftIndex] < pivot) {

      leftIndex++;

    }

    while (array[rightIndex] > pivot) {

      rightIndex--;

    }

    if (leftIndex <= rightIndex) {

      swap(array, leftIndex, rightIndex);

      leftIndex++;

      rightIndex--;

    }

  }

  return leftIndex;

}

module.exports = {

  quicksort,

  partition

};

Swap.js

const swap = (arr, indexOne, indexTwo) => {

  const temp = arr[indexTwo];

  arr[indexTwo] = arr[indexOne];

  arr[indexOne] = temp;

}

module.exports = swap;

# Introduction: Search & Graph Search Algorithms

# Binary Search and Search Trees

## Binary Search

When given a sorted array of data, binary search is a way of searching through that data to find an element in **O(log n)** time using a divide and conquer approach. It doesn’t require you to look through the entire array in a linear way, which would have a linear big O runtime of **O(n)**.

## Binary Search Trees

Binary search trees are a type of tree data structure with the added condition that each element to the left of a node must be less than that parent node, and each element to the right of a node must be greater than that parent node. Each left and right subtree is also itself a binary search tree, which makes searching for elements more efficient.

**BINARY SEARCH: CONCEPTUAL**

**Learn Binary Search**

With a sorted data-set, we can take advantage of the ordering to make a sort which is more efficient than going element by element.

Let’s say you were looking up the word “Telescope” in the dictionary. You wouldn’t flip through the “A” words and “B” words, page by page, until you got to the page you wanted because you know “T” is near the end of the alphabet.

You might flip it open near the end and see “R” words. Maybe then you jump ahead and land at “V” words. You would then go slightly backward to find the “T” words.

At each point, you knew to look forward or backward based on the ordering of the alphabet. We can use this intuition for an algorithm called *binary search*.

Binary search requires a sorted data-set. We then take the following steps:

1. **Check the middle value of the dataset.**
   * If this value matches our target we can return the index.
2. If the middle value is **less than our target**
   * Start at step 1 using the **right half** of the list.
3. If the middle value is **greater than our target**
   * Start at step 1 using the **left half** of the list.

We eventually run out of values in the list, or find the target value.

**Time Complexity of Binary Search**

How efficient is binary search?

In each iteration, we are **cutting the list in half.** The time complexity is O(log N).

A sorted list of 64 elements will take **at most** log2(64) = 6 comparisons.

In the worst case:

* Comparison 1: We look at the middle of all 64 elements
* Comparison 2: If the middle is not equal to our search value, we would look at 32 elements
* Comparison 3: If the new middle is not equal to our search value, we would look at 16 elements
* Comparison 4: If the new middle is not equal to our search value, we would look at 8 elements
* Comparison 5: If the new middle is not equal to our search value, we would look at 4 elements
* Comparison 6: If the new middle is not equal to our search value, we would look at 2 elements

When there’s 2 elements, the search value is either one or the other, and thus, there is at most 6 comparisons in a sorted list of size 64.

**BINARY SEARCH: JAVASCRIPT**

**Iterative Binary Search**

In this lesson, you will implement an iterative binary search function in JavaScript.

The function will:

* Accept an array of numbers and a value as arguments
* Return the index of the value if it is present in the array
* Return null if a value is not in the array

You will test your function by inputting the array shown in the gif to the right as an argument. By the end of this lesson, the following JavaScript code will print 6 to the console.

const searchable = [1, 2, 7, 8, 22, 28, 41, 58, 67, 71, 94];  
const target = 41;  
   
console.log(binarySearch(searchable, target)) // 6

In the code above, also shown visually to the right, we use the binarySearch() function to find the index in searchable that is equal to 41. The index is 6.

**Finding the Middle Index**

A key step in each binary search iteration is to find the middle value of the current list context. In practice, we do this by tracking the first and last indices, then finding their average.

The first index we check will always be the middle value of the original list. Because of this, we start by setting the following first (left) and last (right) indices. Below, we show a pseudocode example of how to set these variables.

function binarySearch (arr, target)

left = 0

right = length of arr

. . .

We could call a JavaScript implementation of this function with the following code:

const searchable = [1, 2, 7, 8, 22, 28, 41, 58, 67, 71, 94];  
const target = 41;  
   
console.log(binarySearch(searchable, target))

Because we pass in an array of length 11, the right variable is set to 11.

Next, we calculate the middle index of the array:

function binarySearch (arr, target)

left = 0

right = length of arr

indexToCheck = the floor integer of (left + right) / 2

. . .

The above function will calculate the middle index of the array by calculating the average of right and left and rounding it to the floor integer. Given left is zero and right is 11:

Floor(11 + 0) /2 = 5

So, the first index the function checks is 5.

Now you know how to calculate the first indexToCheck. In the next exercise, you will implement an approach to check whether the value at that index is equal to the target.

**Instructions**

**1.**

Create a let named left and set it equal to 0.

Create a let named right and set it equal to the length of arr.

Checkpoint 2 Passed

**2.**

Create a const called indexToCheck and set it equal to the floor integer of the average of left and right.

Return indexToCheck. You should see 5 printed to the console.

Answer: index,js

const binarySearch = (arr, target) => {

  // Add left and right variables below

  let left = 0;

  let right = arr.length;

  // Add indexToCheck calculation below

  const indexToCheck = Math.floor((left + right) / 2);

  return indexToCheck;

}

  console.log(Math.floor(5));

const searchable = [1, 2, 7, 8, 22, 28, 41, 58, 67, 71, 94];

const target = 28;

console.log(binarySearch(searchable, target));

module.exports = {binarySearch};

**Checking the Middle Index**

Let’s consider how to implement an approach to check whether the value at indexToCheck is equal to the target value. Below, we use pseudocode to display two additional steps that will check if the target value is found.

function binarySearch (arr, target)

left = 0

right = length of arr

indexToCheck = the floor integer of (left + right) / 2

checking = value of arr at indexToCheck

if checking is the target

return indexToCheck

In the example above, we set a variable called checking to the value in arr at the position indexToCheck. Then, we return the index if it is equal to the target value.

**Instructions**

**1.**

Create a constant called checking and set it equal to the value at indexToCheck.

Checkpoint 2 Passed

**2.**

Create an if statement that returns indexToCheck if checking is equal to target. If your solution is implemented correctly, the script should log 5 to the console.

Checkpoint 3 Passed

Answer: index.js

const binarySearch = (arr, target) => {

  let left = 0;

  let right = arr.length;

  const indexToCheck = Math.floor((left + right) / 2);

  // 1. Create a constant called checking

  const checking = arr[indexToCheck]

  // 2. Create a conditional below

  if (checking === target) {

    return indexToCheck;

  }

  return null;

}

const searchable = [1, 2, 7, 8, 22, 28, 41, 58, 67, 71, 94];

const target = 28;

console.log(binarySearch(searchable, target));

module.exports = {binarySearch};

**Iterative Checking**

At this point, you have a function that checks the middle index of an input array and returns the index *if* the value equals target. Let’s consider how to extend the function to iteratively check sublists when the middle value is not equal to target.

Remember back to our algorithm, the function continues to execute until the left and right indices converge or the target is found. In practice, we can implement this with the following while condition.

while right is greater than left

indexToCheck = the floor integer of (left + right) / 2

checking = value of arr at indexToCheck

if checking is the target

then return indexToCheck

Unfortunately, the above code will execute infinitely because our right and left variables do not converge from one iteration to the next. To address this issue, in addition to checking if the current value is the target value, we need to adjust the right or left index with each iteration.

while right is greater than left

indexToCheck = the floor integer of (left + right) / 2

checking = value of arr at indexToCheck

if checking is the target

then return indexToCheck

if target is greater than checking

then set left to indexToCheck + 1

else

set right to indexToCheck

In the above code, we set the left or right index to a new value based on whether target is greater than or less than checking. The above while loop will continue to execute until the left index is greater than the right index.

In the checkpoints below, you will add conditions that change the left or right index based on whether checking is greater than or less than target. With each iteration, the distance from left to right will halve.

**Instructions**

**1.**

**WARNING: Refresh the page if you create an infinite loop while implementing the solution.**

Add a condition to the while loop so it continues to execute while right is greater than left.

Checkpoint 2 Passed

**2.**

Add else if and else blocks that set new values to left or right based on a comparison of target and checking.

Checkpoint 3 Passed

**3.**

Change target, at the bottom of **index.js** to another value in the searchable array to see if your loop returns the correct index.

Checkpoint

Answer: index.js

const binarySearch = (arr, target) => {

  let left = 0;

  let right = arr.length;

  while (right > left) {

    const indexToCheck = Math.floor((left + right) / 2);

    const checking = arr[indexToCheck];

    console.log(`indexToCheck equals: ${indexToCheck}`)

    if (checking === target) {

      return indexToCheck;

    } else if (target > checking) {

      left = indexToCheck + 1;

    } else {

      right = indexToCheck;

    }

  }

  return null;

}

const searchable = [1, 2, 7, 8, 22, 28, 41, 58, 67, 71, 94];

const target = 2;

const targetIndex = binarySearch(searchable, target);

console.log(`The target index is ${targetIndex}.`);

module.exports = {binarySearch};

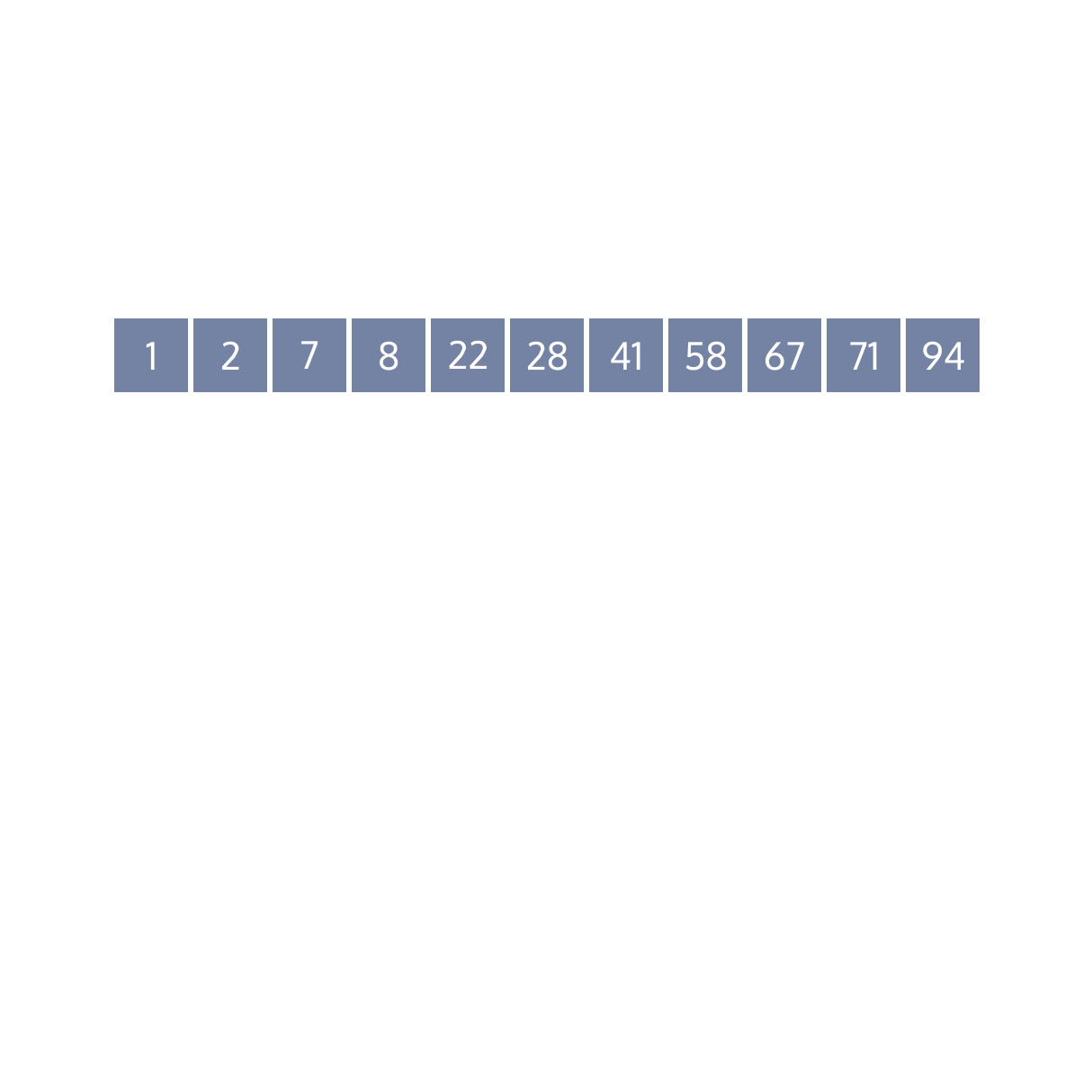
**Review**

* Initialize the left and right indices as 0 and the length of the array.
* Create a while loop that continues to execute until the left index equals the right index.
* Get the value at the index that falls in the middle of left and right.
* Return the index if the value is equal to target.
* Set left equal to the current index plus one if the target is greater than the value.
* Set right equal to the current index minus one if the target is less than the value.

While the benefits of binary search are significant compared to linear search, it’s important to remember that the function will only work on sorted lists.

### Instructions

The gif below is a visual representation of the binary search solution that you implemented. Compare this gif to the code in index.js. You should see that indexToCheck is equal to the same numbers checked in the gif: 5, 8, 7, 6.



### Answer:

const binarySearch = (arr, target) => {

  let left = 0;

  let right = arr.length;

  while (right > left) {

    const indexToCheck = Math.floor((left + right) / 2);

    const checking = arr[indexToCheck];

    console.log(indexToCheck);

    if (checking === target) {

      return indexToCheck;

    } else if (checking < target) {

      left = indexToCheck + 1;

    } else {

      right = indexToCheck;

    }

  }

  return null;

}

const searchable = [1, 2, 7, 8, 22, 28, 41, 58, 67, 71, 94];

const target = 41;

targetIndex = binarySearch(searchable, target);

console.log(`The target index is ${targetIndex}.`);

module.exports = binarySearch;

### 

**Introduction**

A binary tree is an efficient data structure for fast data storage and retrieval due to its O(log N) runtime. It is a specialized tree data structure that is made up of a root node, and at most two child branches or subtrees. Each child node is itself a binary tree.

Each node has the following properties:

* data
* a depth value, where depth of 1 indicates the top level of the tree and a depth greater than 1 is a level somewhere lower in the tree
* a left pointer that points to a left child which is itself a binary tree, and must have a data lesser than the root node’s data
* a right pointer that points to a right child which is itself a binary tree, and must have a data greater than the root node’s data

**Instructions**

**1.**

We have provided an empty BinaryTree class in **BinaryTree.js**.

* Define a constructor that takes two parameters: value and depth where value is the data contained within a binary tree and depth indicates the level of the tree
* Assign a default depth of 1 to depth in the parameter list
* Declare an instance property, value, and assign this to the parameter value.
* Declare another instance property, depth, and assign this to the parameter depth.

Checkpoint 2 Passed

**2.**

Define left and right instance properties to represent pointers to the left and right binary tree nodes and assign each one to null.

Checkpoint 3 Passed

**3.**

Open up **script.js**. Instantiate a BinaryTree class with an initial value of 15 and assign it to a const variable bt.

Display the content of the binary tree, bt.

Answer: BinaryTree.js

class BinaryTree {

 constructor(value, depth = 1) {

  this.value = value;

  this.depth = depth;

  this.left = null;

  this.right = null;

 }

};

module.exports = BinaryTree;

script.js

const BinaryTree = require('./BinaryTree');

// create a BinaryTree here

const bt = new BinaryTree(15);

console.log(bt);

**Inserting a Value**

When inserting a new value into a binary tree, we compare it with the root node’s value:

If the new value is less than the root node's value

If a left child node doesn't exist

Create a new BinaryTree with the new value at a greater depth and assign it to the left pointer

Else

Recursively call .insert() on the left child node

Else

If a right child node doesn't exist

Create a new BinaryTree with the new value at a greater depth and assign it to a right pointer

Else

Recursively call .insert() on the right child node

Let’s illustrate the insertion procedure with a tree whose root node has the data 100.

Insert 50

50 < 100, left child node doesn't exist, create a left child node

100

/

50

Insert 125

125 > 100, right child node doesn't exist, create a right child node

100

/ \

50 125

Insert 75

75 < 100, left child node of 50 exists, recursive insert at left child

75 > 50, right child node doesn't exist, create a right child node

100

/ \

50 125

\

75

Insert 25

25 < 100, left child node of 50 exists, recursive insert at left child

25 < 50, left child node doesn't exist, create a left child node

100

/ \

50 125

/ \

25 75

**Instructions**

**1.**

Define a method, .insert(), with a parameter, value, below the constructor.

Checkpoint 2 Passed

**2.**

We want to know where to place the target value. If the target value is less than the root node’s value, we will need to place it in a left child node. Before doing so, we need to check if a left child node already exists. If so, we will call the .insert() method for the left child node. If not, we will add another level to our binary tree by creating a left binary tree with the target value and a new depth.

* Write an outer if else statement block.
  + The if statement should check if the target value is less than the root node’s value.
  + The else statement should be left blank for now.
* Inside the outer if statement block, write an inner if else statement block.
  + The if statement should check if a left child node exists
    - If it exists, call its .insert() method passing on value
    - If it doesn’t exist, instantiate a BinaryTree with the target value and a new depth and assign it to the left pointer
  + The else statement should be left blank for now.

Checkpoint 3 Passed

**3.**

Alternatively, if the target value is not less than the root node’s value, then we will place it in the right child node. Before doing so, we need to check if a right child node already exists. If so, we will call the .insert() method of the right child node. If not, we will add another level to our binary tree by creating a right binary tree with the target value and a new depth.

* Inside the outer else statement block from the previous step, write an if statement that checks if a right child node exists
  + If it exists, call its .insert() method passing on value
  + If it doesn’t exist, instantiate a BinaryTree with the target value and a new depth and assign it to the right pointer

Checkpoint 4 Passed

**4.**

Open up **script.js**. A default BinaryTree has been created for you with an initial value of 100. Insert the following values to the tree in this order:

* 50
* 125
* 75
* 25

Display the content of this tree and study the terminal output. The result should match the tree illustration in the lesson.

Answer: BinaryTree.js

class BinaryTree {

  constructor(value, depth = 1) {

    this.value = value;

    this.depth = depth;

    this.left = null;

    this.right = null;

  }

  insert(value) {

    if (value < this.value) {

      if (!this.left) {

        this.left = new BinaryTree(value, this.depth + 1);

      } else {

        this.left.insert(value);

      }

    } else {

      if (!this.right) {

        this.right = new BinaryTree(value, this.depth + 1);

      } else {

        this.right.insert(value);

      }

    }

  }

};

module.exports = BinaryTree;

script.js

const BinaryTree = require('./BinaryTree');

// create a BinaryTree here

const bt = new BinaryTree(100);

// insert values to the BinaryTree here

bt.insert(50);

bt.insert(125);

bt.insert(75);

bt.insert(25);

// display the BinaryTree here

console.log(bt);

**Retrieve a Node by Value**

Recall that a binary search tree provides a fast and efficient way to store and retrieve values. Like with .insert(), the procedure to retrieve a tree node by its value is recursive. We want to traverse the correct branch of the tree by comparing the target value to the current node’s value.

The base case for our recursive method is that when the values match, we return the current node. The recursive step is to call itself from an existing left or right child node with the value.

If target value is the same as the current node value

Return the current node

Else

If target value is less than the root node's value and there is a left child node

Recursively search from the left child node

Else if there is a right child node

Recursively search from the right child node

Given the following tree:

100

/ \

50 125

/ \

25 75

To retrieve 75, the algorithm would proceed as follows:

At root node, 75 < 100 and there is a left child

100

/ \

==> 50 125

/ \

25 75

At the node 50, 75 > 50 and there is a right child

100

/ \

50 125

/ \

25 75 <==

Node 75 = 75, return this node

**Instructions**

**1.**

Define a new method, .getNodeByValue(), below the .insert() method that takes one parameter, value.

Checkpoint 2 Passed

**2.**

The first thing we do is to compare the target value with the root node’s value. If they are the same, return the node. This is the base case.

Checkpoint 3 Passed

**3.**

If the target value cannot be found in the root node, we want to navigate further down the binary tree. We start with the left child node if it exists and recursively search in the left subtree. To traverse the left tree, we need to make sure the target value is less than the root node’s value.

Write an else if statement that checks:

* if the left child node exists, and
* if the target value is less than this.value

Inside the else if block, return with a call to .getNodeByValue() method of the left child node.

Checkpoint 4 Passed

**4.**

Next, we want to implement the recursive step for the right child node if it exists.

Write an else if block that:

* checks if the right child node exists, and
* return with a call to .getNodeByValue() method for the right child node if it exists

Checkpoint 5 Passed

**5.**

If the target value does not exist in the binary tree, we should return null.

Create an else block that returns null.

Checkpoint 6 Passed

**6.**

Open up **script.js**. Search for the value 75 in the BinaryTree object already created for you and display the contents of the returned tree node.

Search for a non-existent value of 55 in the same BinaryTree object and display the returned value.

Answer: BinaryTree.js

class BinaryTree {

  constructor(value, depth = 1) {

    this.value = value;

    this.depth = depth;

    this.left = null;

    this.right = null;

  }

  insert(value) {

    if (value < this.value) {

      if (!this.left) {

        this.left = new BinaryTree(value, this.depth + 1);

      } else {

        this.left.insert(value);

      }

    } else {

      if (!this.right) {

        this.right = new BinaryTree(value, this.depth + 1);

      } else {

        this.right.insert(value);

      }

    }

  }

  getNodeByValue(value) {

    if (this.value === value) {

      return this;

    } else if ((this.left) && (value < this.value)) {

        return this.left.getNodeByValue(value);

    } else if (this.right) {

        return this.right.getNodeByValue(value);

    } else {

      return null;

    }

  }

};

module.exports = BinaryTree;

script.js

const BinaryTree = require('./BinaryTree');

// create a BinaryTree

const bt = new BinaryTree(100);

// insert values to the BinaryTree

bt.insert(50);

bt.insert(125);

bt.insert(75);

bt.insert(25);

// search for value 75 in BinaryTree

let node = bt.getNodeByValue(75);

console.log(node);

// search for a non-existent value in BinaryTree

node = bt.getNodeByValue(55);

console.log(node);

**Traversing a Binary Tree**

There are two main ways of traversing a binary tree: breadth-first and depth-first. With breadth-first traversal, we begin traversing at the top of the tree’s root node, displaying its data and continuing the process with the left child node and the right child node. Descend a level and repeat this step until we finish displaying all the child nodes at the deepest level from left to right.

With depth-first traversal, we always traverse down each left-side branch of a tree fully before proceeding down the right branch. However, there are three traversal options:

* *Preorder* is when we perform an action on the current node first, followed by its left child node and its right child node
* *Inorder* is when we perform an action on the left child node first, followed by the current node and the right child node
* *Postorder* is when we perform an action on the left child node first, followed by the right child node and then the current node

For this lesson, we will implement the inorder option. The advantage of this option is that the data is displayed in a sorted order from the smallest to the biggest.

To illustrate, let’s say we have a binary tree that looks like this:

15

/------+-----\

12 20

/ \ / \

10 13 18 22

/ \ / \ / \ / \

8 11 12 14 16 19 21 25

We begin by traversing the left subtree at each level, which brings us to 8, 10 and 11 reside. The inorder traversal would be:

8, 10, 11

We ascend one level up to visit root node 12 before we descend back to the bottom where the right subtree of 12, 13, and14` resides. Inorder traversal continues with:

12, 12, 13, 14

We again ascend one level up to visit root node 15 before we traverse the right subtree starting at the bottom level again. We continue with the bottom leftmost subtree where 16, 18 and 19 reside. The inorder traversal continues with:

15, 16, 18, 19

We ascend one level up to visit root node 20 before we descend back to the bottom where the rightmost subtree of 21, 22 and 25 resides.

Traversal finishes with:

20, 21, 22, 25

The entire traversal becomes:

8, 10, 11, 12, 12, 13, 14, 15, 16, 18, 19, 20, 21, 22, 25

Notice that all the values displayed are sorted in ascending order.

**Instructions**

**1.**

Define the method, .depthFirstTraversal() below .getNodeByValue(). It takes no parameters.

Checkpoint 2 Passed

**2.**

Using inorder traversal, we will traverse the left child node followed by the root node and the right child node. Inside .depthFirstTraversal():

* Check to see if the left child node exists
* If it does, call this method on the left child node

Checkpoint 3 Passed

**3.**

We have just traversed the left subtree. Now we want to look at the root node. We can simply print out the data it contains and the tree level of the root node. Log the value of the root node to the console.

Checkpoint 4 Passed

**4.**

The next step would be to traverse the right subtree. Like in the step before last, we want to:

* Check to see if the right child node exists
* If it does, call this method on the right child node

Checkpoint 5 Passed

**5.**

Open up **script.js** and run it to display a depth-first traversal of a binary tree example described above.

Answer: BinaryTree.js

class BinaryTree {

  constructor(value, depth = 1) {

    this.value = value;

    this.depth = depth;

    this.left = null;

    this.right = null;

  }

  insert(value) {

    if (value < this.value) {

      if (!this.left) {

        this.left = new BinaryTree(value, this.depth + 1);

      } else {

        this.left.insert(value);

      }

    } else {

      if (!this.right) {

        this.right = new BinaryTree(value, this.depth + 1);

      } else {

        this.right.insert(value);

      }

    }

  }

  getNodeByValue(value) {

    if (this.value === value) {

      return this;

    } else if ((this.left) && (value < this.value)) {

        return this.left.getNodeByValue(value);

    } else if (this.right) {

        return this.right.getNodeByValue(value);

    } else {

      return null;

    }

  }

  depthFirstTraversal() {

    if (this.left) {

      this.left.depthFirstTraversal();

    }

    console.log(`Depth=${this.depth}, Value=${this.value}`);

    if (this.right) {

      this.right.depthFirstTraversal();

    }

  }

};

module.exports = BinaryTree;

scritp.js

const BinaryTree = require('./BinaryTree');

const bt = new BinaryTree(15);

let numbers = [ 12, 20, 10, 13, 18, 22, 8, 11, 12, 14, 16, 19, 21, 25 ];

for (let i = 0; i < numbers.length; i++) {

  bt.insert(numbers[i]);

  console.log(`Insert ${numbers[i]} to binary tree`);

}

console.log('Depth First Traversal');

bt.depthFirstTraversal();

**Review**

* a BinaryTree class containing value, left and right child nodes and a depth
* an .insert() method to add and place a value at the correct location in the binary tree
* a .getNodeByValue() method to retrieve a child node by its value or null
* a .depthFirstTraversal() method to traverse the binary tree using the inorder traversal option, and

script.js

const BinaryTree = require('./BinaryTree');

const bt = new BinaryTree(15);

let numbers = [ 12, 20, 10, 13, 18, 22, 8, 11, 12, 14, 16, 19, 21, 25 ];

for (let i = 0; i < numbers.length; i++) {

  bt.insert(numbers[i]);

  console.log(`Insert ${numbers[i]} to binary tree`);

}

console.log('Depth First Traversal');

bt.depthFirstTraversal();

BinaryTree.js

class BinaryTree {

  constructor(value, depth = 1) {

    this.value = value;

    this.depth = depth;

    this.left = null;

    this.right = null;

  }

  insert(value) {

    if (value < this.value) {

      if (!this.left) {

        this.left = new BinaryTree(value, this.depth + 1);

      } else {

        this.left.insert(value);

      }

    } else {

      if (!this.right) {

        this.right = new BinaryTree(value, this.depth + 1);

      } else {

        this.right.insert(value);

      }

    }

  }

  getNodeByValue(value) {

    if (this.value === value) {

      return this;

    } else if ((this.left) && (value < this.value)) {

        return this.left.getNodeByValue(value);

    } else if (this.right) {

        return this.right.getNodeByValue(value);

    } else {

      return null;

    }

  }

  depthFirstTraversal() {

    if (this.left) {

      this.left.depthFirstTraversal();

    }

    console.log(`Depth=${this.depth}, Value=${this.value}`);

    if (this.right) {

      this.right.depthFirstTraversal();

    }

  }

};

module.exports = BinaryTree;

Which of these depth-traversal methods returns a sorted data list in a binary tree?

Inorder

The following code is a JavaScript snippet to insert a value into a binary tree. Fill in the code with the correct **if** statements for adding a value in the left subtree.

if (value < this.value) {  
  if (!this.left) {  
    this.left = new BinaryTree(value, this.depth + 1, 'left');  
  } else {  
    this.left.insert(value);  
    }  
  // more statements  
}

# Graph Traversals

### Depth-First Search

Simply put, a depth-first graph search continues down a path until it reaches the end. It is helpful for determining if a path exists between two vertices. DFS has many applications, including topological sorting and detecting cycles, but one of the more interesting real-world applications is that it can be used to solve problems that have a singular correct answer (a path between the start state and end state), such as a sudoku exercise.

### Breadth-First Search

On the other hand, a breadth-first graph search approach checks the values of all neighboring vertices before moving into another level of depth. This is an incredibly inefficient way to find just any path between two vertices, but it’s an excellent way to identify the shortest path between them. Because of this, BFS is helpful for figuring out directions from one place to another.

### Dijkstra’s Algorithm

Dijkstra’s algorithm is a method for finding the shortest distance from a given point to every other point in a weighted graph. The algorithm works by keeping track of all the distances and updating the distances as it conducts a breadth-first search. A common application of this algorithm is to find the quickest route from one destination to another.

**Graph Search Conceptual Introduction**

There are two common approaches to using a graph search to progress through a graph:

* depth-first search, known as DFS follows each possible path to its end
* breadth-first search, known as BFS broadens its search from the point of origin to an ever-expanding circle of neighbouring vertices

To enable searching, we add vertices to a list, visited. This list is pretty important because it keeps the search from visiting the same vertex multiple times! This is particularly vital for cyclical graphs where you might otherwise end up in an infinite loop.

So how do you calculate the runtime for graph search algorithms?

In an upper bound scenario, we would be looking at every vertex and every edge. Because of this, the big O runtime for both depth-first search and breadth-first search is O(vertices + edges).

**Depth-First Search (DFS) Conceptual**

Imagine you’re in a car, on the road with your friend, “D.” D is on a mission to get to your destination by process of elimination. D won’t stop and ask for directions. D just sticks to a chosen path until you reach the end.

At that point, if the end wasn’t actually your destination, D brings you back to the last point when there was an intersection and tries another path.

Like your friend D, depth-first search algorithms check the values along a path of vertices before moving to another path.

While this isn’t exactly ideal when you want to find the shortest path between two points, DFS can be very helpful for determining *if* a path even exists.

In order to accomplish this path-finding feat, DFS implementations use either a stack data structure or, more commonly, recursion to keep track of the path the search is on and the current vertex.

In a stack implementation, the most recently added vertex is popped off the stack when the search has reached the end of the path. Meanwhile, in a recursive implementation, the DFS function is recursively called for each connected vertex.

**Breadth-First Search (BFS) Conceptual**

You’re back in a car, but this time, your friend “B” is navigating. Unlike D, B is a bit hesitant about whether you’ve gone the right way and keeps checking in to see if you are on the best path. At each intersection, B tries out each possible route one by one, but only for a block in each direction to see if you’ve found your destination.

Like B, breadth-first search, known as *BFS*, checks the values of all neighboring vertices before moving into another level of depth.

This is an incredibly inefficient way to find just *any* path between two points, but it’s an excellent way to identify the *shortest path* between two vertices. Because of this, BFS is helpful for figuring out directions from one place to another.

Unlike DFS, BFS graph search implementations use a queue data structure to keep track of the current vertex and vertices that still have unvisited neighbors. In BFS graph search a vertex is dequeued when all neighboring vertices have been visited.

**Graph Search Traversal Order**

What if you don’t need to find a path, but you *do* need to get a list of all the values in a graph?

Well, it turns out that in addition to path-finding, depth-first search is pretty adept at organizing vertices (or vertex values) with a clear order of visitation from beginning to end.

There are three main traversal orders that you’ll come across for graph traversal:

* *Preorder*, in which each vertex is added to the “visited” list and added to the output list BEFORE getting added to the stack
* *Postorder*, in which each vertex is added to the “visited” list and added to the output list AFTER it is popped off the stack
* Reverse Post-Order (also known as *Topological Sort*), which returns an output list that is exactly the reverse of the post-order list

Take a look at the directed graph structure we have depicted here. Let’s say that we want a list of all vertex values, starting with “Lasers”, in the order that they are added to the stack.

A pre-order DFS traversal would come in handy and we might end up with the following list. (We’ll assume here that this algorithm prefers visiting things in alphabetical order if there is a choice.):

["Lasers", "Lava", "Snakes", "Spikes", "Piranhas"]

Now, let’s say we want the same values, but with each value only added to the list once its vertex has been popped from the stack. In this case, our post-order DFS traversal would result in a list that looked like:

["Spikes", "Snakes", "Lava", "Piranhas", "Lasers"]

You may notice that the post-order list is not the reverse of the pre-order list. A reverse post-order list would still begin with “Lasers”, but then begin to differ:

["Lasers", "Piranhas", "Lava", "Snakes", "Spikes"]

What happens if there are unvisited vertices that are not reachable from the current path? The search would visit them in (here alphabetical) order after exploring the current path.

**Graph Search Conceptual Review**

You’ve learned a bunch about graph searches and how to use them effectively:

* You can use a graph search algorithm to traverse the entirety of a graph data structure to locate a specific value
* Vertices in a graph search include a “visited” list to keep track of whether or not each vertex has been checked
* Depth-first search (DFS) and breadth-first search (BFS) are two common approaches to graph search
* The runtime for graph search algorithms is O(vertices + edges)
* DFS, which employs either recursion or a stack data structure, is useful for determining whether a path exists between two points
* BFS, which generally relies on a queue data structure, is helpful in finding the shortest path between two points
* There are three common traversal orders which you can apply with DFS to generate a list of all values in a graph: pre-order, post-order, and reverse post-order

**GRAPH TRAVERSAL: JAVASCRIPT**

Traversals are incredibly useful when you are trying to find a particular value or a particular path in a graph. We’ll first explore the depth-first traversal function for traversing through a directed graph. To recap, depth-first traversals iterate down each vertex, one neighbor at a time, before going back up and looking at the neighbor’s connected vertices. In this exercise, we will focus on traversing down the full length of one path and logging each vertex’s data value.

For simplicity, we’ll implement the traversal iterator as a separate function instead of as a method on the Graph class. In other implementations, the iterator can be seen as a class method.

We have also set up a sample graph in **testGraph.js** for you to test the traversals against. Feel free to take a look at the file to familiarize yourself with the structure of the graph.

### Instructions

**1.**

Let’s start by setting up our traversal function. Since it will be used to traverse a graph, we can expect for the graph to be provided in the form of the starting vertex.

We’ve provided an empty depthFirstTraversal() function. Add a vertex parameter, start, and print out the parameter’s data property so we can see which vertex we are on.

Checkpoint 2 Passed

**2.**

Since we are only focusing on the full length of one path, the next vertex we want to traverse is the starting vertex’s first neighbor. We cannot always assume that all vertices will have outgoing edges, so we will need to handle this case.

After the start vertex’s data property is printed, add an if statement with a condition that checks if the start vertex has edges to traverse.

Inside the if block, create a variable, neighbor and set it to the first neighbor, which can be found among the start vertex’s list of edges. Go ahead and print out the neighbor’s data to verify that we grabbed the right neighbor.

Checkpoint 3 Passed

**3.**

Now that we have the first neighbor, we want to go down this neighbor’s first edge, and then traverse down that vertex’s first edge, and so on. To do that, we can leverage recursion to take care of the downward traversal by passing the neighbor vertex as the new starting vertex in the function call.

Call the depthFirstTraversal() function if the start vertex has neighbors left to traverse. Make sure to pass in the neighbor vertex to the recursive call so that we can go down the path and iterate through the neighbor’s first connected vertex.

Since we’re making a recursive call through the neighbors, we can remove the call to console.log that prints out the neighbor.data.

Checkpoint 4 Passed

**4.**

If there was a cycle, or if the neighbor’s first connected vertex is the neighbor, we would be stuck in an infinite loop, iterating between the same neighbors. To account for this, we can add an array to keep track of all the vertices that we have visited. We should pass it in every recursive call, to make sure we never visit the same vertex more than once.

Add a second parameter to the function called visitedVertices. By default, it should be set to an array that contains the start argument.

Then, add an if statement to check if it does not include the neighbor vertex. We shouldn’t visit a vertex more than once, so make the recursive call if the neighbor is not included in the visitedVertices.

Checkpoint 5 Passed

**5.**

Whenever we make another call to traverse down the paths, we are visiting a new vertex so we should update the list of visitedVertices to reflect that.

Right before we make the recursive call to the depthFirstTraversal() function, add the neighbor vertex to the array of visitedVertices.

The visitedVertices is now changed, so our recursive call should be aware of that. Pass the updated visitedVertices as the second argument to the call to depthFirstTraversal().

Answer: depthTraversal.js

const testGraph = require('./testGraph.js');

const depthFirstTraversal = (start, visitedVertices = [start]) => {

  console.log(start.data)

  if (start.edges.length) {

    const neighbor = start.edges[0].end;

    if (!visitedVertices.includes(neighbor)) {

      visitedVertices.push(neighbor);

      depthFirstTraversal(neighbor, visitedVertices);

    }

  }

};

depthFirstTraversal(testGraph.vertices[0]);te

testGraph.js

const { Graph } = require('./Graph.js');

const simpleGraph = new Graph(true, false);

const startNode = simpleGraph.addVertex('v0.0.0');

const v1 = simpleGraph.addVertex('v1.0.0');

const v2 = simpleGraph.addVertex('v2.0.0');

const v11 = simpleGraph.addVertex('v1.1.0');

const v12 = simpleGraph.addVertex('v1.2.0');

const v21 = simpleGraph.addVertex('v2.1.0');

const v111 = simpleGraph.addVertex('v1.1.1');

const v112 = simpleGraph.addVertex('v1.1.2');

const v121 = simpleGraph.addVertex('v1.2.1');

const v211 = simpleGraph.addVertex('v2.1.1');

simpleGraph.addEdge(startNode, v1);

simpleGraph.addEdge(startNode, v2);

simpleGraph.addEdge(v1, v11);

simpleGraph.addEdge(v1, v12);

simpleGraph.addEdge(v2, v21);

simpleGraph.addEdge(v11, v111);

simpleGraph.addEdge(v11, v112);

simpleGraph.addEdge(v12, v121);

simpleGraph.addEdge(v21, v211);

module.exports = simpleGraph;

**Depth-First Traversal (All paths)**

We’ve gotten the hang of traversing down one path, but we want to traverse down all the paths (not just the first possible path). We will modify our existing implementation to iterate down all the other paths by using a .forEach() loop to iterate through all of the start vertex’s edges.

We won’t have to worry about iterating through all the neighbors before going down the neighbor’s first connected vertex. This is because the recursive call occurs before the next iteration of the for loop.

### Instructions

**1.**

To traverse down all paths, we no longer need the if statement to check if there are edges to traverse. Instead, we will use an iterator to go through all of the vertex’s edges. If there are no edges, then the edges array would be empty and nothing would happen.

Replace the if statement with a .forEach() iterator that iterates through the start vertex’s list of edges.

We will also want to set the neighbor to the end vertex of the edge parameter. The end vertex is given as a parameter by the .forEach() iterator.

Checkpoint 2 Passed

**2.**

Great, we have completed the depth-first traversal! It iterates down each path until it hits a dead end, continues down the next path at the neighboring vertex until it hits a dead end, and so on.

To see this in action, we have provided you with a sample graph called testGraph and passed it in our call to depthFirstTraversal(). You should see the traversal move in the following order: v0.0.0, v1.0.0, v1.1.0, v1.1.1, v1.1.2, v1.2.0, v1.2.1, v2.0.0, v2.1.0, and v2.1.1.

Checkpoint 3 Passed

Answer:

const testGraph = require('./testGraph.js');

const depthFirstTraversal = (start, visitedVertices = [start]) => {

  console.log(start.data);

  start.edges.forEach(edge => {

    const neighbor = edge.end;

    if (!visitedVertices.includes(neighbor)) {

      visitedVertices.push(neighbor);

      depthFirstTraversal(neighbor, visitedVertices);

    }

  });

};

depthFirstTraversal(testGraph.vertices[0]);

treeGraph.js

const { Graph } = require('./Graph.js');

const simpleGraph = new Graph(true, false);

const startNode = simpleGraph.addVertex('v0.0.0');

const v1 = simpleGraph.addVertex('v1.0.0');

const v2 = simpleGraph.addVertex('v2.0.0');

const v11 = simpleGraph.addVertex('v1.1.0');

const v12 = simpleGraph.addVertex('v1.2.0');

const v21 = simpleGraph.addVertex('v2.1.0');

const v111 = simpleGraph.addVertex('v1.1.1');

const v112 = simpleGraph.addVertex('v1.1.2');

const v121 = simpleGraph.addVertex('v1.2.1');

const v211 = simpleGraph.addVertex('v2.1.1');

simpleGraph.addEdge(startNode, v1);

simpleGraph.addEdge(startNode, v2);

simpleGraph.addEdge(v1, v11);

simpleGraph.addEdge(v1, v12);

simpleGraph.addEdge(v2, v21);

simpleGraph.addEdge(v11, v111);

simpleGraph.addEdge(v11, v112);

simpleGraph.addEdge(v12, v121);

simpleGraph.addEdge(v21, v211);

module.exports = simpleGraph;

**Depth-First Traversal (Callbacks)**

Our current implementation of the depth-first traversal simply prints out the vertices of the graph as they are traversed. This would be useful in scenarios where we want to see the order that the traversal occurs in. For example, if the graph was an instruction list, we need the exact order that the steps will occur to determine which dependencies need to be resolved first.

However, there may be other instances where we want to do something other than printing out the traversal order. For example, if we just need to determine if a path exists, like seeing if a maze is solvable, we just need a true or false value. We can do this by opening up a callback parameter for the user.

**Instructions**

**1.**

Since we want to open up another parameter as a callback, add another parameter to depthFirstTraversal() called callback as the second parameter.

In the recursive call to depthFirstTraversal(), add the callback argument.

We want to avoid making the callback the third parameter to simplify depthFirstTraversal() for the user. This means they won’t be forced to supply the visitedVertices parameter if they also want to override the default callback argument.

Checkpoint 2 Passed

**2.**

Now let’s put the callback to work. Replace the call to console.log() with a call to callback() and pass in the start vertex.

To test the new callback, add a function as a second argument to depthFirstTraversal() at the bottom of **depthFirstTraversal.js**. This function should accept a vertex as an argument and print out the data property. Since we passed in a print function as our callback, we should still see the order that the vertices are traversed in.

This wraps up our implementation of depth-first traversal! If you’re feeling up for it, here are some challenge tasks to tackle:

* This is currently a pre-order traversal. How would you modify the implementation to be a post-order traversal?
* How would you modify the implementation to use a queue instead of recursion?

Checkpoint 3 Passed

Answer: depthTraversal.js

const testGraph = require('./testGraph.js');

const depthFirstTraversal = (start, callback, visitedVertices = [start]) => {

  callback(start);

  start.edges.forEach((edge) => {

    const neighbor = edge.end;

    if (!visitedVertices.includes(neighbor)) {

      visitedVertices.push(neighbor);

      depthFirstTraversal(neighbor, callback, visitedVertices);

    }

  });

};

depthFirstTraversal(testGraph.vertices[0], (vertex) => { console.log(vertex.data) });

treeGraph.js

const { Graph } = require('./Graph.js');

const simpleGraph = new Graph(true, false);

const startNode = simpleGraph.addVertex('v0.0.0');

const v1 = simpleGraph.addVertex('v1.0.0');

const v2 = simpleGraph.addVertex('v2.0.0');

const v11 = simpleGraph.addVertex('v1.1.0');

const v12 = simpleGraph.addVertex('v1.2.0');

const v21 = simpleGraph.addVertex('v2.1.0');

const v111 = simpleGraph.addVertex('v1.1.1');

const v112 = simpleGraph.addVertex('v1.1.2');

const v121 = simpleGraph.addVertex('v1.2.1');

const v211 = simpleGraph.addVertex('v2.1.1');

simpleGraph.addEdge(startNode, v1);

simpleGraph.addEdge(startNode, v2);

simpleGraph.addEdge(v1, v11);

simpleGraph.addEdge(v1, v12);

simpleGraph.addEdge(v2, v21);

simpleGraph.addEdge(v11, v111);

simpleGraph.addEdge(v11, v112);

simpleGraph.addEdge(v12, v121);

simpleGraph.addEdge(v21, v211);

module.exports = simpleGraph;

**Breadth-First Traversal (First layer)**

Now it’s time to focus on breadth-first traversal! Just as a reminder, breadth-first iterates through the whole graph in layers by going down one layer, which comprises the start vertex’s direct neighbors. Then it proceeds down to the next layer which consists of all the vertices that are neighbors of the vertices in the previous layer.

For this exercise, let’s focus on traversing down one layer. We will take a similar approach as we did with the depth-first traversal by keeping an array of visitedVertices to prevent us from iterating through the same vertices.

However, we will iterate through all of the direct neighbor vertices instead of iterating down the neighbor’s first edge. We will also use a queue to traverse through the graph instead of recursion to explore the different ways we can implement the traversals.

### Instructions

**1.**

Let’s start by creating our breadthFirstTraversal() function. We should expect for the graph to be provided in the form of the starting vertex.

Add the start vertex as a parameter to breadthFirstTraversal() .

Checkpoint 2 Passed

**2.**

Next, we will go down one layer and traverse all of the start vertex’s neighbor. Set up a .forEach() iterator to iterate through all of its edges.

Each edge contains the neighboring vertex in its end property, which will be our neighbor. Create a const variable, neighbor, and set it to the end property of each edge.

Checkpoint 3 Passed

**3.**

Great! Now we should set up our list of visitedVertices so we can mark the neighbor vertex as “visited”. We won’t need to provide the visitedVertices as an argument since we are using a queue to traverse through the graph instead of recursion.

Inside the function and before the forEach loop, create a const variable, visitedVertices. Set it to an array with the start vertex as the first element.

Checkpoint 4 Passed

**4.**

Almost there! Before we can mark the neighbor as “visited”, we need to check that the visitedVertices does not already include the neighbor vertex. Otherwise, we could end up “visiting” the vertices multiple times by adding duplicates of the vertex in our visitedVertices list.

After the neighbor vertex is declared, add an if statement that checks that the neighbor is not included in the visitedVertices. If the neighbor is not, then we can mark it as “visited” by adding it to the list of visitedVertices.

Checkpoint 5 Passed

**5.**

We’ve now successfully iterated through all of the start vertex’s neighbors. To check that we are “visiting” the vertices in the correct order, print out the visitedVertices right before the end of the function.

When we run the function with our test graph, we should see the vertices in the following order: v0.0.0, v1.0.0, and v2.0.0.

Checkpoint 6 Passed

Answer:

const testGraph = require('./testGraph.js');

const breadthFirstTraversal = (start) => {

  const visitedVertices = [start];

  start.edges.forEach(edge => {

    const neighbor = edge.end;

    if (!visitedVertices.includes(neighbor)) {

      visitedVertices.push(neighbor)

    }

  });

  console.log(visitedVertices)

};

breadthFirstTraversal(testGraph.vertices[0]);

treeGraph.js

const { Graph } = require('./Graph.js');

const simpleGraph = new Graph(true, false);

const startNode = simpleGraph.addVertex('v0.0.0');

const v1 = simpleGraph.addVertex('v1.0.0');

const v2 = simpleGraph.addVertex('v2.0.0');

const v11 = simpleGraph.addVertex('v1.1.0');

const v12 = simpleGraph.addVertex('v1.2.0');

const v21 = simpleGraph.addVertex('v2.1.0');

const v111 = simpleGraph.addVertex('v1.1.1');

const v112 = simpleGraph.addVertex('v1.1.2');

const v121 = simpleGraph.addVertex('v1.2.1');

const v211 = simpleGraph.addVertex('v2.1.1');

simpleGraph.addEdge(startNode, v1);

simpleGraph.addEdge(startNode, v2);

simpleGraph.addEdge(v1, v11);

simpleGraph.addEdge(v1, v12);

simpleGraph.addEdge(v2, v21);

simpleGraph.addEdge(v11, v111);

simpleGraph.addEdge(v11, v112);

simpleGraph.addEdge(v12, v121);

simpleGraph.addEdge(v21, v211);

module.exports = simpleGraph;

**Breadth-First Traversal (All layers)**

So far, we can iterate down one layer, but we have yet to iterate down the remaining layers. In order to do so, we will introduce a queue that will keep track of all of the vertices to visit.

As we iterate through the neighbors, we will add its connected vertices to the end of the queue, pull off the next neighbor from the queue, add its connected vertices, and so on. This way allows us to maintain the visiting order; we will visit the vertices across the same layer while queueing up the next layer. When there are no vertices left in the current layer, the vertices of the next layer are already queued up, so we move down and iterate across the next layer.

We will use our implementation of the Queue data structure that was covered in a previous course. It is located in **Queue.js**. Go ahead and take a quick look to refresh your memory of the data structure and the available methods.

**Instructions**

**1.**

We will create our queue with the start vertex as the first connected vertex to iterate through.

Right after we create our list of visitedVertices, create a const variable, visitQueue. Instantiate a new Queue and assign it to the visitQueue.

Then .enqueue() the start vertex to the queue.

Checkpoint 2 Passed

**2.**

When we are looking at a vertex from visitQueue, we want to dequeue it so that we don’t look at it again. The visitedVertices array ensures that it does not get enqueued into visitQueue again.

Before the .forEach() iterator, .dequeue() the next vertex from the visitQueue and assign it to the current variable with const. Go ahead and print out the data property of the current vertex so we can see which vertex we’re looking at.

Checkpoint 3 Passed

**3.**

The queue holds all of the vertices that we have yet to iterate through. This means we want to continue iterating through these vertices as long as there are vertices left in the queue.

After we enqueue the start vertex, add in a while loop that continues to run as long as the visitQueue is not empty. Make sure that it includes dequeuing the next vertex and the forEach() iterator, since we also want to update visitedVertices if there are still vertices in the queue.

Checkpoint 4 Passed

**4.**

Next, we want to iterate through the current vertex’s neighbors and enqueue them, not just the start vertex’s neighbors.

Update the .forEach() iterator to iterate through the current vertex’s edges instead. Then, inside the .forEach() iterator, if the visitedVertices does not include the neighbor, we should enqueue the neighbor to the visitQueue.

Checkpoint 5 Passed

**5.**

That’s it! When we run the graph traversal with the testGraph, we should see the vertices printed out in the following order: v0.0.0, v1.0.0, v2.0.0, v1.1.0, v1.2.0, v2.1.0, v1.1.1, v1.1.2, v1.2.1, and v2.1.1.

If you’re feeling up for a challenge, take a moment to consider the following:

* How would you modify this to take a recursive approach?
* How would you add in a callback to expand the utility of the function?

Checkpoint 6 Passed

Answer: const testGraph = require('./testGraph.js');

const Queue = require('./Queue.js');

const breadthFirstTraversal = (start) => {

  const visitedVertices = [start];

  const visitQueue = new Queue();

  visitQueue.enqueue(start);

  while (!visitQueue.isEmpty()) {

    const current = visitQueue.dequeue();

    console.log(current.data);

    current.edges.forEach((edge) => {

      const neighbor = edge.end;

      if (!visitedVertices.includes(neighbor)) {

        visitedVertices.push(neighbor);

        visitQueue.enqueue(neighbor);

      }

    })

  }

};

breadthFirstTraversal(testGraph.vertices[0]);

testGraph.js

const { Graph } = require('./Graph.js');

const simpleGraph = new Graph(true, false);

const startNode = simpleGraph.addVertex('v0.0.0');

const v1 = simpleGraph.addVertex('v1.0.0');

const v2 = simpleGraph.addVertex('v2.0.0');

const v11 = simpleGraph.addVertex('v1.1.0');

const v12 = simpleGraph.addVertex('v1.2.0');

const v21 = simpleGraph.addVertex('v2.1.0');

const v111 = simpleGraph.addVertex('v1.1.1');

const v112 = simpleGraph.addVertex('v1.1.2');

const v121 = simpleGraph.addVertex('v1.2.1');

const v211 = simpleGraph.addVertex('v2.1.1');

simpleGraph.addEdge(startNode, v1);

simpleGraph.addEdge(startNode, v2);

simpleGraph.addEdge(v1, v11);

simpleGraph.addEdge(v1, v12);

simpleGraph.addEdge(v2, v21);

simpleGraph.addEdge(v11, v111);

simpleGraph.addEdge(v11, v112);

simpleGraph.addEdge(v12, v121);

simpleGraph.addEdge(v21, v211);

module.exports = simpleGraph

Queue.js

const LinkedList = require('./LinkedList');

class Queue {

  constructor(maxSize = Infinity) {

    this.queue = new LinkedList();

    this.maxSize = maxSize;

    this.size = 0;

  }

  hasRoom() {

    return this.size < this.maxSize;

  }

  isEmpty() {

    return this.size === 0;

  }

  enqueue(data) {

    if (this.hasRoom()) {

      this.queue.addToTail(data);

      this.size++;

    } else {

      throw new Error('Queue is full!');

    }

  }

  dequeue() {

    if (!this.isEmpty()) {

      const data = this.queue.removeHead();

      this.size--;

      return data;

    } else {

      throw new Error('Queue is empty!');

    }

  }

}

module.exports = Queue;