

What is the big O complexity in JavaScript?



The letter O is used because the growth rate of a function is also called its order. Big O Notation characterizes functions according to their growth rates. It describes the implementation of an algorithm in terms of the size of the input. The time complexity of this code is $O(n)$. 22 Nov 2022

What is the time complexity of set in JavaScript?



$O(1)$

Sets are internally implemented as hash tables or search trees that give us a constant **$O(1)$ or $O(\log N)$** lookup time in searching an element. Although different browsers or JavaScript engines may implement Set differently, it is always guaranteed to give us time complexity better than $O(N)$. 26 Sept 2022



Why time complexity is used?

Time complexity is very **useful measure in algorithm analysis**. It is the time needed for the completion of an algorithm. To estimate the time complexity, we need to consider the cost of each fundamental instruction and the number of times the instruction is executed.

Example 1: Addition of two scalar variables. 5 Dec 2024


What is time complexity in JS functions?

Big O Notation — Time Complexity in Javascript

- $O(1)$: Run time is independent of the input size of the problem.
- $O(n)$: The problem requires a small amount of processing time for each element in the input. ...
- $O(n^2)$: The problem process all the pairs of the element.

What are the 4 types of algorithm?

AI Overview

Four commonly recognized algorithm types are Brute-Force, Divide and Conquer, Greedy, and Dynamic Programming. 

Runtime

To fully understand algorithms we must understand how to evaluate the time an algorithm needs to do its job, the runtime.

Exploring the runtime of algorithms is important because using an inefficient algorithm could make our program slow or even unworkable.

By understanding algorithm runtime we can choose the right algorithm for our need, and we can make our programs run faster and handle larger amounts of data effectively.

Time Complexity

To evaluate and compare different algorithms, instead of looking at the actual runtime for an algorithm, it makes more sense to use something called time complexity.

Time complexity is more abstract than actual runtime, and does not consider factors such as programming language or hardware.

Time complexity is the number of operations needed to run an algorithm on large amounts of data. And the number of operations can be considered as time because the computer uses some time for each operation.


For example, in the algorithm that finds the lowest value in an array, each value in the array must be compared one time. Every such comparison can be considered an operation, and each operation takes a certain amount of time. So the total time the algorithm needs to find the lowest value depends on the number of values in the array.

The time it takes to find the lowest value is therefore linear with the number of values. 100 values results in 100 comparisons, and 5000 values results in 5000 comparisons.

The relationship between time and the number of values in the array is linear, and can be displayed in a graph like this:

Time Complexity - Point Wise Description

1. **Definition:** Time complexity represents the number of operations an algorithm performs based on input size, rather than actual runtime.
2. **Independence from Hardware:** It does not depend on factors like programming language, hardware, or system efficiency.
3. **Operations as Time Measure:** Each operation takes time, so the total time is estimated based on the number of operations.
4. **Example - Finding Minimum in an Array:**
 - Each element is compared once.
 - If there are n elements, there will be n comparisons.
 - Time complexity = $O(n)$ (linear complexity).
5. **Graph Representation:** A linear algorithm results in a straight-line graph where doubling input size doubles execution time.

Best, worst, and average case analysis are methods used to evaluate the performance of algorithms by examining their time complexity under different input scenarios: best case (minimum steps), worst case (maximum steps), and average case (average steps). 

Algorithms

JavaScript algorithms are a set of programming instructions, known as inputs and outputs, that allow a data operation to function precisely at every execution.

Array indexing

Arrays in JavaScript are ordered collections of values, where each value is associated with an index. These indices start from 0 for the first element, 1 for the second, and so on. Array indexing allows direct access to specific elements within the array using their corresponding index.

Array traversal

Array traversal in JavaScript involves accessing each element within an array, typically to perform operations on them. Several methods facilitate this process:

Array traversal

Array traversal is a fundamental concept in Data Structures and Algorithms (DSA) that every developer should master. In this

Linear search

Linear search, also known as sequential search, is a simple algorithm for finding an element within a list. It involves checking each element in the list one by one until a match is found or the end of the list is reached.

JavaScript



```
function linearSearch(arr, target) {  
  for (let i = 0; i < arr.length; i++) {  
    if (arr[i] === target) {  
      return i; // Return the index if the target is found  
    }  
  }  
  return -1; // Return -1 if the target is not found  
}
```

In this function:

- `arr` is the input array or list.
- `target` is the value being searched for.
- The `for` loop iterates through each element of the array.
- Inside the loop, it checks if the current element `arr[i]` is equal to the `target`.
- If a match is found, the function immediately returns the index `i` of the element.
- If the loop completes without finding a match, it means the `target` is not in the array, and the function returns `-1`.

Time Complexity in Linear search

The time complexity of linear search is $O(n)$ in the worst case, where n is the number of elements in the list. This is because, in the worst case, it may have to examine every element in the list. However, in the best case, where the target element is the first element in the list, the time complexity is $O(1)$.

Concatenating arrays

How to concatenate array elements in JavaScript?



The `concat()` method of Array instances is used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array. 11 Feb 2025

Selection sort

Selection Sort **sorts an array of n values**. On average, about n^2 elements are compared to find the lowest value in each loop. And Selection Sort must run the loop to find the lowest value approximately n times.

Selection sort

Selection Sort - Time Complexity (Simplified)

1. **Sorting Process:** Selection Sort repeatedly finds the smallest element and moves it to its correct position.
2. **Comparisons:** The algorithm runs $(n - 1)$ times, comparing n , $(n - 1)$, $(n - 2)$, ... elements, leading to $O(n^2)$ comparisons.
3. **Swaps:** It performs $O(n)$ swaps, since each element is moved at most once.
4. **Overall Complexity:** The number of operations simplifies to $O(n^2)$ (quadratic time complexity), making it inefficient for large datasets.

Bubble sort

The Bubble Sort algorithm loops through every value in the array, comparing it to the value next to it. So for an array of n values, there must be n such comparisons in one loop. And after one loop, the array is looped through again and again n times.

Introduction to 2D array

A two-dimensional array, also known as a 2D array, is a collection of data elements arranged in a grid-like structure with rows and columns. Each element in the array is referred to as a cell and can be accessed by its row and column indices/indexes. 17 Jan 2023

2D array operations

What are the operations on arrays in DSA?

The basic operations in the Arrays are **insertion, deletion, searching, display, traverse, and update**. These operations are usually performed to either modify the data in the array or to report the status of the array. Following are the basic operations supported by an array.

2D array operations

What is a 2D array in DSA?



A two dimensional array is **a data structure that contains a collection of cells laid out in a two dimensional grid**, similar to a table with rows and columns although the values are still stored linearly in memory.

2D array operations

Modifying 2D array elements

Insertion and Deletion

Adding Rows/Columns:

- `push()` adds to the end.
- `unshift()` adds to the beginning.
- `splice()` offers more control for adding at specific indices.

Removing Rows/Columns:

- `pop()` removes from the end.
- `shift()` removes from the beginning.
- `splice()` can remove elements from any position.

Modifying 2D array elements

To modify an element within a 2D array in JavaScript, you specify the element's position using its row and column indices. The syntax `array2D[rowIndex][columnIndex] = newValue` assigns `newValue` to the element at the specified location.

Modifying 2D array elements

Modifying Elements in 2d Arrays

To modify an element in a two-dimensional array, use the syntax `arrayName[i][j] = value`. This will set the value stored at index `i`, subindex `j` to the desired value. For example, `arr[1][3] = 5` would set the fourth value in the second element of the array `arr` to 5. 5 May 2024

Introduction to singly linked list

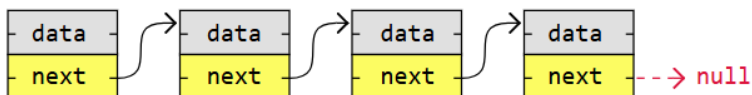
Introduction. A singly linked list is a basic type of data structure that includes a series of connected nodes. Each node stores a piece of data and a link to the next node in the sequence. 21 Feb 2025

Introduction to singly linked list

A **Linked List** is, as the word implies, a list where the nodes are linked together. Each node contains data and a pointer. The way they are linked together is that each node points to where in the memory the next node is placed.

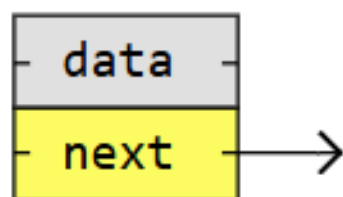
Linked Lists

A linked list consists of nodes with some sort of data, and a pointer, or link, to the next node.

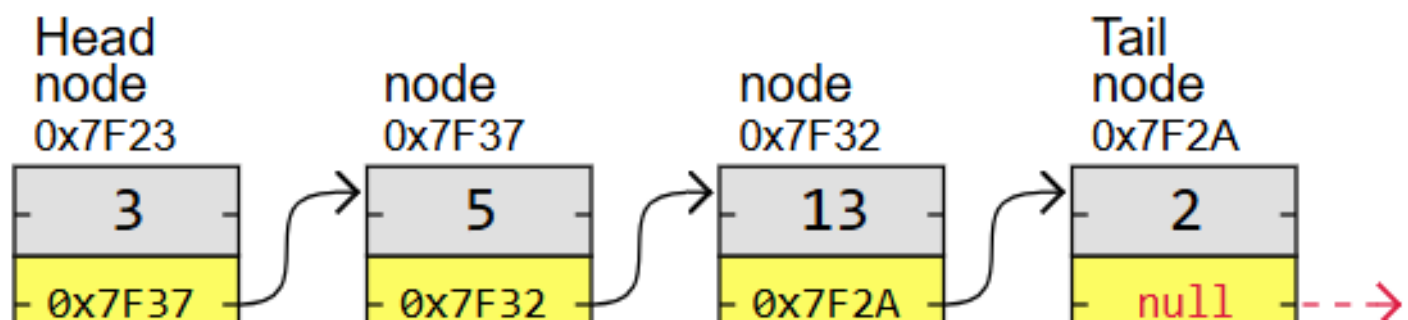


Diff Array VS Linked Lists

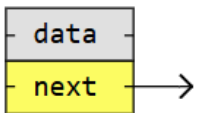
	Arrays	Linked Lists
<i>An existing data structure in the programming language</i>	Yes	No
<i>Fixed size in memory</i>	Yes	No
<i>Elements, or nodes, are stored right after each other in memory (contiguously)</i>	Yes	No
<i>Memory usage is low (each node only contains data, no links to other nodes)</i>	Yes	No
<i>Elements, or nodes, can be accessed directly (random access)</i>	Yes	No
<i>Elements, or nodes, can be inserted or deleted in constant time, no shifting operations in memory needed.</i>	No	Yes



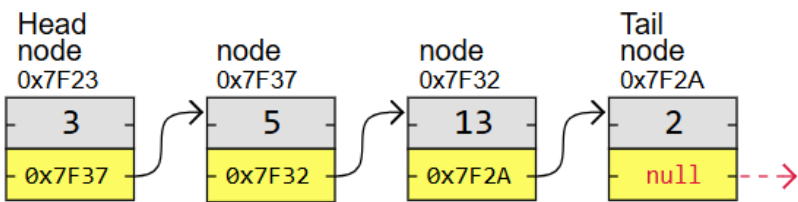
ve put the same four nodes from the previous example together u



Linked list traversal



If we put the same four nodes from the previous example together using this new visualization, it looks like this:



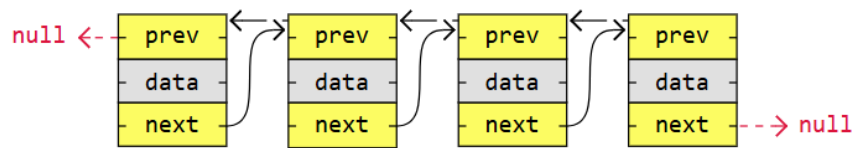
As you can see, the first node in a linked list is called the "Head", and the last node is called the "Tail".

Types of Linked Lists

There are three basic forms of linked lists:

1. Singly linked lists
2. Doubly linked lists
3. Circular linked lists

A **doubly linked list** has nodes with addresses to both the previous and the next node, like in the image below, and therefore takes up more memory. But doubly linked lists are good if you want to be able to move both up and down in the list.

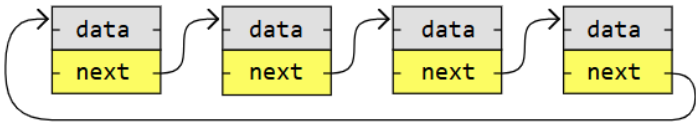


A **circular linked list** is like a singly or doubly linked list with the first node, the "head", and the last node, the "tail", connected.

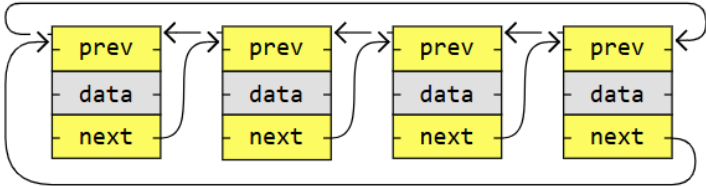
In singly or doubly linked lists, we can find the start and end of a list by just checking if the links are `null`. But for circular linked lists, more complex code is needed to explicitly check for start and end nodes in certain applications.

Circular linked lists are good for lists you need to cycle through continuously.

The image below is an example of a singly circular linked list:



The image below is an example of a doubly circular linked list:



Linked List Implementations

Below are basic implementations of:

1. Singly linked list
2. Doubly linked list
3. Circular singly linked list
4. Circular doubly linked list

Linked List Operations

Basic things we can do with linked lists are:

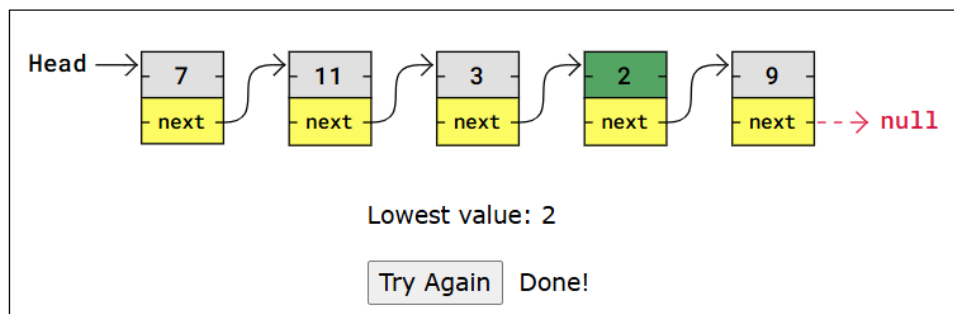
1. Traversal
2. Remove a node
3. Insert a node
4. Sort

Find The Lowest Value in a Linked List

Let's find the lowest value in a singly linked list by traversing it and checking each value.

Finding the lowest value in a linked list is very similar to how we found the lowest value in an array, except that we need to follow the next link to get to the next node.

This is how finding the lowest value in a linked list works in principle:

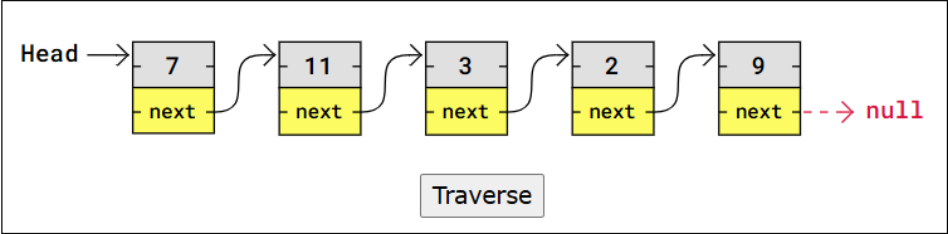


Traversal of a Linked List

Traversing a linked list means to go through the linked list by following the links from one node to the next.

Traversal of linked lists is typically done to search for a specific node, and read or modify the node's content, remove the node, or insert a node right before or after that node.

To traverse a singly linked list, we start with the first node in the list, the head node, and follow that node's next link, and the next node's next link and so on, until the next address is null, like in the animation below:



Delete a Node in a Linked List

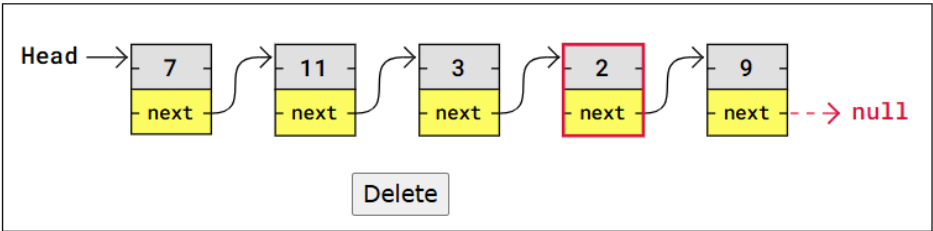
In this case we have the link (or pointer or address) to a node that we want to delete.

It is important to connect the nodes on each side of the node before deleting it, so that the linked list is not broken.

So before deleting the node, we need to get the next pointer from the previous node, and connect the previous node to the new next node before deleting the node in between.

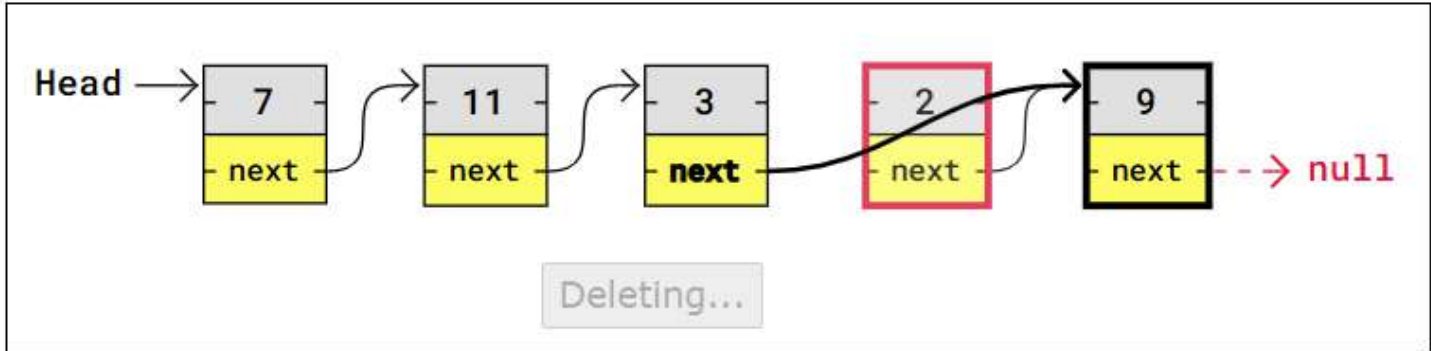
In a singly linked list, like we have here, to get the next pointer from the previous node we actually need traverse the list from the start, because there is no way to go backwards from the node we want to delete.

The simulation below shows the node we want to delete, and how the list must be traversed first to connect the list properly before deleting the node without breaking the linked list.



Also, it is a good idea to first connect next pointer to the node after the node we want to delete, before we delete it. This is to avoid a 'dangling' pointer, a pointer that points to nothing, even if it is just for a brief moment.

In the code below, the algorithm to delete a node is moved into a function called `deleteSpecificNode`.



Reverse a linked list

Reverse a linked list in dsa js

Algorithm:

1. Step 1: Create an empty stack. ...
2. Step 2: Traverse the linked list using a temporary variable `temp` till it reaches null. ...
3. Step 3: Set variable `temp` back to the head of the linked list. ...
4. Step 4: Return the head as the new head of the reversed linked list.

Deleting a node in a singly linked list without the head pointer requires a different approach since standard traversal is not possible. The key is to copy the data from the next node to the node to be deleted and then bypass the next node.

JavaScript



```
class Node {  
  constructor(data) {  
    this.data = data;  
    this.next = null;  
  }  
}  
  
function deleteNodeWithoutHead(node) {  
  if (!node || !node.next) {  
    return; // Nothing to delete or node is the last element  
  }  
  
  node.data = node.next.data; // Copy data from next node  
  node.next = node.next.next; // Bypass the next node  
}
```

This method works because it shifts the problem from deleting the current node to deleting the next node, which is achievable by manipulating the `next` pointer of the current node.

Linear Search


Linear search iterates through each element of an array sequentially until the target element is found or the end of the array is reached.

JavaScript

```
function linearSearch(arr, target) {  
  for (let i = 0; i < arr.length; i++) {  
    if (arr[i] === target) {  
      return i; // Return the index if found  
    }  
  }  
  return -1; // Return -1 if not found  
}  
  
// Example usage:  
const array = [5, 2, 9, 1, 5, 6];  
const target = 9;  
const result = linearSearch(array, target);  
console.log(result); // Output: 2
```

- **Time Complexity:** $O(n)$ in the worst case, where n is the number of elements in the array.
- **Space Complexity:** $O(1)$

Binary Search

Binary search is more efficient but requires the array to be sorted. It works by repeatedly dividing the search interval in half. If the middle element is the target, the index is returned. If the target is less than the middle element, the search continues in the left half; otherwise, in the right half. 

JavaScript



```
function binarySearch(arr, target) {  
  let left = 0;  
  let right = arr.length - 1;  
  
  while (left <= right) {  
    const mid = Math.floor((left + right) / 2);  
  
    if (arr[mid] === target) {  
      return mid; // Return the index if found  
    } else if (arr[mid] < target) {  
      left = mid + 1; // Search right half  
    } else {  
      right = mid - 1; // Search left half  
    }  
  }  
  return -1; // Return -1 if not found  
}  
  
// Example usage:  
const sortedArray = [1, 2, 5, 6, 9];  
const target = 6;  
const result = binarySearch(sortedArray, target);  
console.log(result); // Output: 3
```

- **Time Complexity:** $O(\log n)$, where n is the number of elements in the array.
- **Space Complexity:** $O(1)$

Two pointer technique in dsa js

The two-pointer technique is an algorithm design pattern where two pointers are used to iterate through a data structure, often an array or string, in a coordinated manner. It's particularly useful for solving problems that involve searching for pairs, merging sorted lists, or finding sub-sequences that satisfy certain conditions.

What is the two pointer technique in JavaScript?

Two pointers is really an easy and effective technique which is **typically used for searching pairs in a sorted array**. Given a sorted array A (sorted in ascending order), having N integers, find if there exists any pair of elements $(A[i], A[j])$ such that their sum is equal to X. 16 Sept 2024

Kadane algorithm in dsa js

Kadane's Algorithm is a dynamic programming algorithm used to solve the maximum subarray problem. The maximum subarray problem is the task of finding the contiguous subarray within a one-dimensional array of numbers that has the largest sum. The subarray can be of any length, including the entire array. 3 May 2024

What is the Kadane's algorithm in DSA?



What is Kadane's Algorithm? Kadane's Algorithm is a linear time algorithm used to find the maximum subarray sum in a given array. A subarray is defined as a contiguous subset of elements within the array.

Does Kadane work for negative numbers?



Kadane's Algorithm is efficient and requires only a single scan through the array. The algorithm is simple to understand and implement. **It works well for arrays with both positive and negative numbers**. 27 Feb 2023

Recursion in data structures and algorithms (DSA) is a method where a function calls itself within its definition. It's a powerful technique for solving problems that can be broken down into smaller, self-similar subproblems. A recursive function continues to call itself until it reaches a base case, which is a condition that stops the recursion and returns a value directly, preventing an infinite loop.

Key Components of Recursion

Base Case:

This is the condition that terminates the recursion. It's essential to have a base case to prevent the function from calling itself indefinitely.

Recursive Step:

This is where the function calls itself with a modified input, moving closer to the base case.

Example: Factorial Calculation

Calculating the factorial of a number is a classic example of recursion. The factorial of a non-negative integer n , denoted as $n!$, is the product of all positive integers less than or equal to n .

JavaScript



```
function factorial(n) {  
  // Base case: if n is 0, return 1  
  if (n === 0) {  
    return 1;  
  }  
  // Recursive step: n * factorial(n-1)  
  return n * factorial(n - 1);  
}
```

Introduction of recursion in dsa js

JavaScript

```
function factorial(n) {  
  // Base case: if n is 0, return 1  
  if (n === 0) {  
    return 1;  
  }  
  // Recursive step: n * factorial(n-1)  
  return n * factorial(n - 1);  
}  
  
console.log(factorial(5)); // Output: 120
```

How it Works

- When `factorial(5)` is called, it checks if `n` is 0. Since it's not, it proceeds to the recursive step.
- It returns `5 * factorial(4)`.
- `factorial(4)` is called, and it returns `4 * factorial(3)`, and so on.
- This continues until `factorial(0)` is called, which returns 1 (base case).
- The results are then returned back up the call stack: `1`, `1*1`, `2*1`, `3*2*1`, `4*3*2*1`, `5*4*3*2*1`, resulting in 120.

Advantages of Recursion

Elegance and Readability:

Recursion can make code cleaner and easier to understand for problems with inherent recursive structures.

Problem Decomposition:

It simplifies complex problems by breaking them into smaller, manageable subproblems.

Disadvantages of Recursion

Stack Overflow:

If the base case is not defined correctly or the recursion depth is too large, it can lead to a stack overflow error.


Performance Overhead:

Recursive calls can be slower than iterative solutions due to the overhead of function calls.

Use Cases

Recursion is commonly used in:

- Tree and graph traversals
- Divide-and-conquer algorithms (e.g., merge sort, quicksort)
- Mathematical functions (e.g., Fibonacci sequence)

Recursive algorithms in JavaScript involve a function calling itself to solve smaller instances of the same problem. This technique is particularly useful for tasks that can be broken down into simpler, repetitive subproblems. 

Structure of a Recursive Function

A recursive function typically consists of two parts:

- **Base Case:** A condition that stops the recursion, preventing an infinite loop.
- **Recursive Step:** The function calls itself with a modified input, moving towards the base case.

Types of Recursion

- **Direct Recursion:** A function calls itself directly.
- **Indirect Recursion:** Function A calls function B, which in turn calls function A.
- **Tail Recursion:** The recursive call is the last operation in the function (can be optimized in some environments).
- **Binary Recursion:** The function calls itself twice in each step.

Recursion

Recursion is a technique where a function calls itself within its definition. It's like a set of Russian dolls, where each doll contains a smaller version of itself. A recursive function typically has two parts:

- **Base case:** A condition that stops the recursion.
- **Recursive step:** The function calls itself with a modified input, moving towards the base case.

JavaScript



```
function factorialRecursive(n) {  
  if (n === 0) {  
    return 1; // Base case  
  } else {  
    return n * factorialRecursive(n - 1); // Recursive step  
  }  
}
```

Iteration

Iteration uses loops (like `for` or `while`) to repeat a block of code until a certain condition is met. It's like following a set of instructions step by step. [🔗](#)

JavaScript



```
function factorialIterative(n) {  
  let result = 1;  
  for (let i = 1; i <= n; i++) {  
    result *= i;  
  }  
  return result;  
}
```

Recursive vs. iterative in dsa js

Key Differences

Feature	Recursion	Iteration
Mechanism	Function calls itself	Loops (for, while)
Structure	Base case and recursive step	Loop condition and body
Memory Usage	Higher due to function call stack	Lower, more memory-efficient
Performance	Can be slower due to function call overhead	Generally faster
Readability	Can be more elegant for certain problems	Can be more straightforward for simple repetitions
Problem types	Naturally suited for problems that can be broken down into similar subproblems (e.g., tree traversal, divide and conquer)	Suitable for repetitive tasks with a clear end condition
Time Complexity	Depends on the number of times the function calls itself	Depends on the number of iterations
Space Complexity	Higher due to function call stack	Lower

What is binary search algorithm in Javascript?

The Binary Search algorithm is **used to search for any element in a sorted array**. If the element is found, it returns the element's index. If not, it returns -1. 7 Jun 2021

What are the 7 steps of a binary search?

Binary Search Algorithm


- Step 1: set $\text{start} = \text{lower_bound}$, $\text{end} = \text{upper_bound}$, $\text{pos} = -1$.
- Step 2: repeat steps 3 and 4 while $\text{start} \leq \text{end}$.
- Step 3: set $\text{mid} = (\text{start} + \text{end})/2$ or $\text{start} + (\text{end} - \text{start})/2$.
- Step 4: if $a[\text{mid}] = \text{target}$.
- set $\text{pos} = \text{mid}$.
- print pos .
- go to step 6.
- else if $a[\text{mid}] > \text{target}$.

What is binary search tree DSA?

A Binary Search Tree is **a Binary Tree where every node's left child has a lower value, and every node's right child has a higher value**. A clear advantage with Binary Search Trees is that operations like search, delete, and insert are fast and done without having to shift values in memory.

What is binary search in DSA?

AI Overview

In the context of Data Structures and Algorithms (DSA), binary search is a highly efficient algorithm used to find the position of a target value within a sorted array by repeatedly dividing the search interval in half. 

What is the time complexity of binary search in Javascript?

Time Complexity: $O(\log N)$. 10 Jan 2025

In JavaScript, Set and Map are built-in data structures introduced in ES6 (ECMAScript 2015) that offer efficient ways to handle collections of data.

Set

A Set is a collection that stores unique values of any data type. It ensures that no duplicate values are present within the set. Sets are useful when you need to keep track of unique items, such as IDs, without the overhead of manually checking for duplicates.

JavaScript



```
const mySet = new Set();

mySet.add(1);
mySet.add(2);
mySet.add(2); // Duplicate value, will not be added
mySet.add('hello');
mySet.add({ a: 1 });

console.log(mySet); // Output: Set(4) {1, 2, "hello", {a: 1}}
console.log(mySet.has(2)); // Output: true
console.log(mySet.size); // Output: 4
mySet.delete(2)
console.log(mySet) // Output: Set(3) {1, "hello", {a: 1}}
```

Map

A Map is a collection that stores key-value pairs, where both keys and values can be of any data type. Unlike objects, where keys are limited to strings and symbols, Maps allow you to use any value (including objects and functions) as a key. Maps maintain the insertion order of elements. [↗](#)

JavaScript



```
const myMap = new Map();

myMap.set('name', 'John');
myMap.set(1, 'Number one');
myMap.set({ a: 1 }, 'Object key');

console.log(myMap); // Output: Map(3) {"name" => "John", 1 => "Number one", {a:
console.log(myMap.get('name')); // Output: John
console.log(myMap.has(1)); // Output: true
console.log(myMap.size); // Output: 3
myMap.delete(1)
console.log(myMap) // Output: Map(2) {"name" => "John", {...} => "Object key"}
```

Most important methods set & Map in dsa js

Set Methods

- `add(value)` : Adds a new element with the given `value` to the `Set` . If the `value` is already present, it does not add it again (ensuring uniqueness).
- `delete(value)` : Removes the element associated with the given `value` from the `Set` . Returns `true` if the element was successfully removed, and `false` otherwise.
- `has(value)` : Checks if an element with the specified `value` exists in the `Set` . Returns `true` if the element is present, and `false` otherwise.
- `clear()` : Removes all elements from the `Set` .
- `size` : Returns the number of elements currently present in the `Set` .

Map Methods

set(key, value):

Adds a new key-value pair to the **Map** or updates the value if the key already exists.

get(key):

Retrieves the value associated with the specified **key** from the **Map**. Returns **undefined** if the key is not found.

delete(key):

Removes the key-value pair associated with the given **key** from the **Map**. Returns **true** if the element was successfully removed, and **false** otherwise.

has(key):

Checks if a key-value pair with the specified **key** exists in the **Map**. Returns **true** if the key is present, and **false** otherwise.


clear():

Removes all key-value pairs from the **Map**.


size:

Returns the number of key-value pairs currently present in the **Map**.


keys():

Returns an iterator that yields the keys for each element in the map in insertion order. 

values():

Returns an iterator that yields the values for each element in the map in insertion order. 

entries():

Returns an iterator that yields **[key, value]** pairs for each element in the map in insertion order. 

Most important methods set & Map in dsa js



size:

Returns the number of key-value pairs currently present in the **Map**.

keys():

Returns an iterator that yields the keys for each element in the map in insertion order. [🔗](#)

values():

Returns an iterator that yields the values for each element in the map in insertion order. [🔗](#)

entries():

Returns an iterator that yields **[key, value]** pairs for each element in the map in insertion order.

forEach(callbackFn):

Executes a provided function once for each key-value pair in the Map, in insertion order. [🔗](#)

What are the methods of map in JavaScript?

JavaScript Map Methods


- The new Map() Method. You can create a map by passing an array to the new Map() constructor: ...
- Map.get() You get the value of a key in a map with the get() method. ...
- Map.size. The size property returns the number of elements in a map: ...
- Map.delete() ...
- Map.clear() ...
- Map.forEach() ...
- Map.entries() ...
- Map.keys()

Which is better, map or set?

In addition, Sets are typically used to store a collection of unique elements, while Maps are used to store data in a more structured way. **Sets are also more efficient when it comes to searching for elements, as they can be searched in constant time, while Maps require linear time.** 5 May 2024

WeakSet and **WeakMap** are specialized collections in JavaScript that, unlike **Set** and **Map**, hold objects weakly. This means they do not prevent garbage collection of the objects they contain. If an object stored in a **WeakSet** or as a key in a **WeakMap** is no longer referenced elsewhere, it can be automatically removed from the collection, freeing up memory.

WeakSet

A **WeakSet** is a collection of objects where each object may occur only once. It is similar to a **Set**, but with the "weak" referencing behavior. 

Key Characteristics:

- It can only store objects, not primitive values.
- It does not have a **size** property, and you cannot iterate over its elements directly.
- It is useful for tracking objects' existence without preventing their garbage collection.

Common Use Cases:

- Managing collections of objects, like tracking event handlers or unique object references.
- Marking objects as "seen" or "processed" without keeping them alive indefinitely.

WeakMap

A **WeakMap** is a collection of key-value pairs, where the keys must be objects, and the values can be of any type. Like **WeakSet**, it holds its keys weakly. [↗](#)

Key Characteristics: [↗](#)

- Keys must be objects.
- It does not have methods like **keys()**, **values()**, or **forEach()** for iteration.
- It is suitable for associating data with objects without preventing garbage collection.

Understanding WeakSet, MapSet in dsa js

WeakSet: Holds only objects as values. The values in a WeakSet can be checked for existence, but there is no associated data. **WeakMap:** Holds key-value pairs, where the keys must be objects and the values can be any data type. Each key-value pair represents an association between an object and some data. 7 Jun 2024

A stack is a linear data structure adhering to the Last-In, First-Out (LIFO) principle. It operates like a stack of plates, where the last plate placed on top is the first one removed. In JavaScript, stacks can be implemented using arrays or linked lists. [🔗](#)

Basic Stack Operations:

- **Push:** Adds an element to the top of the stack.
- **Pop:** Removes and returns the top element from the stack.
- **Peek:** Returns the top element of the stack without removing it.
- **isEmpty:** Checks if the stack is empty.
- **size:** Returns the number of elements in the stack.

Applications of Stacks:

- **Function call management:** Stacks manage function calls and their execution context.
- **Undo/redo functionality:** Used in applications for implementing undo and redo features.
- **Expression evaluation:** Evaluating arithmetic expressions.
- **Depth-first search (DFS):** Used in graph traversal algorithms.
- **Backtracking algorithms:** Solving problems by exploring all possible solutions.
- **Browser history:** Managing the history of visited web pages.

A queue is a fundamental data structure that follows the First-In, First-Out (FIFO) principle. It operates like a real-world queue, where the first element added is the first one to be removed. In JavaScript, queues can be implemented using arrays or linked lists. [↗](#)

Basic Queue Operations

- **Enqueue (add):** Adds an element to the rear (end) of the queue.
- **Dequeue (remove):** Removes and returns the element from the front of the queue. [↗](#)
- **Peek (front):** Returns the element at the front of the queue without removing it.
- **isEmpty:** Checks if the queue is empty.
- **size:** Returns the number of elements in the queue.