2024前端高频— 数据结构与算法篇

1.两数之和-给定一个数组 nums 和一个目标值 target，在该数组中找出和为目标值的两个数

在JavaScript中，你可以通过遍历数组并使用一个对象（在JavaScript中通常使用对象作为哈希表）来存储已经遍历过的数字及其索引，以找出和为特定目标值的两个数。以下是一个可能的实现：

```
function twoSum(nums, target) {
    // 创建一个空对象来存储数组中的数字及其索引
    const numMap = {}

    // 遍历数组
    for (let i = 0
        const num = nums[i]
        const complement = target - num

        // 检查补数是否已经在哈希表中
        if (complement in numMap) {
            // 如果在，返回结果
            return [numMap[complement], i]
        }

        // 如果不在，将当前数字及其索引添加到哈希表中
        numMap[num] = i
    }

    // 如果没有找到符合条件的两个数，抛出一个错误
    throw new Error('No two sum solution')
}
```

```
// 示例
const nums = [2, 7, 11, 15]
const target = 9

try {
    const result = twoSum(nums, target)
    console.log(result)
} catch (error) {
    console.error(error.message)
}
```

在这个实现中，我们首先创建了一个空对象 `numMap`。然后，我们遍历数组 `nums` 中的每个元素。对于每个元素，我们计算目标值与当前元素的差值（补数）。然后，我们检查这个补数是否已经在 `numMap` 中。如果在，说明我们已经找到了两个数的和为目标值，我们返回这两个数的索引。如果不在，我们将当前元素及其索引添加到 `numMap` 中，以便后续的检查。

如果遍历完整个数组都没有找到符合条件的两个数，我们抛出一个错误。在实际应用中，你可能希望返回一个特殊的值（如 `null` 或 `undefined`），而不是抛出错误，这取决于你的具体需求。

2.三数之和-给定一个数组nums，判断 nums 中是否存在三个元素a，b，c，使得 a + b + c = target，找出所有满足条件且不重复的三元组合

你可以使用两个嵌套的循环来遍历数组，然后对于每一对元素，使用第三个循环来查找第三个元素，使得三者的和等于目标值。但是，这种方法的时间复杂度为O(n^3)，对于大数组来说可能效率不高。

一个更高效的方法是使用哈希表（在JavaScript中，我们通常使用对象作为哈希表）来存储数组中的元素，使得我们可以在O(1)的时间内查找某个元素是否存在。然后，我们可以使用两个指针来遍历数组，同时更新哈希表。这种方法的时间复杂度为O(n^2)。

以下是一个使用这种方法的JavaScript代码示例：

```
function threeSum(nums, target) {
    const result = []
```

```
    nums.sort((a, b) => a - b)
    const map = {}

    for (let i = 0
        // 跳过重复元素
        if (i > 0 && nums[i] === nums[i - 1]) continue

        let left = i + 1
        let right = nums.length - 1

        while (left < right) {
            const sum = nums[i] + nums[left] + nums[right]

            if (sum === target) {
                result.push([nums[i], nums[left], nums[righ
t]])

                // 跳过重复元素
                while (left < right && nums[left] === nums[le
ft + 1]) left++
                while (left < right && nums[right] === nums[r
ight - 1]) right--

                left++
                right--
            } else if (sum < target) {
                left++
            } else {
                right--
            }
        }
    }

    return result
```

```
}
```

这个函数的输入是一个数组`nums`和一个目标值`target`，输出是一个二维数组，其中每个子数组都是一个满足条件的三元组合。这个函数首先对数组进行排序，然后使用两个指针`left`和`right`来遍历数组，同时更新哈希表。当找到一个满足条件的三元组合时，将其添加到结果数组中，并跳过所有重复的元素。最后，返回结果数组。

3.输入一个字符串，找到第一个不重复字符的下标

要找到字符串中第一个不重复字符的下标，你可以遍历字符串并使用一个对象（在JavaScript中作为哈希表使用）来记录每个字符出现的次数。然后再次遍历字符串，找到第一个出现次数为1的字符并返回其下标。以下是实现这一功能的JavaScript代码示例：

```
function firstNonRepeatingChar(str) {
    // 使用对象来记录每个字符出现的次数
    const charCount = {}
    for (let i = 0
        const char = str[i]
        // 如果字符已经存在于对象中，则增加计数
        if (charCount[char]) {
            charCount[char]++
        } else {
            // 否则，初始化计数为1
            charCount[char] = 1
        }
    }

    // 再次遍历字符串，找到第一个出现次数为1的字符
    for (let i = 0
        const char = str[i]
        if (charCount[char] === 1) {
            return i
        }
    }
```

```
    // 如果没有找到不重复字符，则返回-1
    return -1
}

// 示例用法
const inputString = "abcabcbb"
const index = firstNonRepeatingChar(inputString)
console.log(index)
```

在这个函数中，`charCount`对象用于存储每个字符及其出现的次数。第一次遍历字符串时，我们更新这个对象的计数。第二次遍历字符串时，我们检查每个字符的计数，并返回第一个计数为1的字符的下标。如果遍历完整个字符串都没有找到这样的字符，则返回-1表示没有找到。

4.输入一个字符串，打印出该字符串中，所有字符的排列组合

在JavaScript中，要打印出字符串中所有字符的排列组合，可以使用递归的方法。以下是一个简单的实现示例：

```
function permute(str, memo = [], result = []) {
    // 如果memo的长度等于str的长度，说明找到了一个完整的排列，将其添加到结果数组中
    if (memo.length === str.length) {
        result.push(memo.join(''))
        return
    }

    // 遍历字符串中的每个字符
    for (let i = 0
        // 如果当前字符已经在memo中，则跳过以避免重复
        if (memo.includes(str[i])) continue

        // 将当前字符添加到memo中，并递归调用permute函数继续处理剩余字符
        memo.push(str[i])
```

```
            permute(str, memo, result)

            // 回溯，将当前字符从memo中移除，以便尝试其他排列
            memo.pop()
        }

        return result
}

// 示例用法
const inputString = "abc"
const permutations = permute(inputString)

// 打印所有排列组合
permutations.forEach(permutation => {
    console.log(permutation)
})
```

这个函数`permute`接受一个字符串`str`作为输入，并返回该字符串所有字符的排列组合。`memo`数组用于跟踪当前排列中已经包含的字符，而`result`数组则用于存储所有找到的排列。

在函数内部，我们首先检查`memo`数组的长度是否等于输入字符串的长度。如果是，说明我们已经找到了一个完整的排列，可以将其添加到结果数组中。然后，我们遍历输入字符串中的每个字符，对于每个字符，如果它尚未在`memo`中出现，我们就将其添加到`memo`中，并递归调用`permute`函数处理剩余的字符。在递归调用返回后，我们通过将字符从`memo`中移除来执行回溯，以便尝试其他可能的排列。

最后，我们调用`permute`函数并遍历返回的排列数组，打印出每个排列。

请注意，对于较长的字符串，排列的数量会非常大（n的阶乘，其中n是字符串的长度），这可能会导致性能问题和内存问题。因此，对于较长的字符串，这种方法可能不是非常实用。在实际应用中，您可能需要考虑一些优化策略或者限制排列的数量。

5.冒泡排序

冒泡排序是一种简单的排序算法，它重复地遍历要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。遍历数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。

以下是用 JavaScript 实现冒泡排序的代码：

```
function bubbleSort(arr) {
    var len = arr.length
    for (var i = 0
        for (var j = 0
            if (arr[j] > arr[j + 1]) {        // 相邻元素两两对
比
                var temp = arr[j + 1]
                arr[j + 1] = arr[j]
                arr[j] = temp
            }
        }
    }
    return arr
}

// 示例
var arr = [34, 8, 64, 51, 32, 21]
console.log(bubbleSort(arr))
```

在这个代码中，外层循环控制所有需要遍历的次数，内层循环负责具体的比较和交换工作。当某一趟遍历过程中没有进行任何交换，说明序列已经有序，此时可以直接结束算法。但是在这个简单的示例中，我们并没有加入这个判断条件。如果需要优化性能，可以加入这个判断条件。

6.选择排序

选择排序是一种简单直观的排序算法。它的工作原理是首先在未排序序列中找到最小（或最大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（或最大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

以下是使用 JavaScript 实现选择排序的代码：

```
function selectionSort(arr) {
    var len = arr.length
    for (var i = 0
        // 记录当前未排序部分中的最小元素的索引
        var minIndex = i
        for (var j = i + 1
            // 如果找到更小的元素，则更新最小元素的索引
            if (arr[j] < arr[minIndex]) {
                minIndex = j
            }
        }
        // 将找到的最小元素与未排序部分的第一个元素交换位置
        var temp = arr[minIndex]
        arr[minIndex] = arr[i]
        arr[i] = temp
    }
    return arr
}

// 示例
var arr = [64, 25, 12, 22, 11]
console.log(selectionSort(arr))
```

在这个代码中，外层循环用于控制需要排序的轮数，内层循环用于在未排序的元素中找到最小的元素，并将其与未排序部分的第一个元素交换位置。这样，每经过一轮循环，未排序部分的最小元素就会被放到正确的位置上。最终，当所有轮循环结束后，整个数组就排好序了。

7.快速排序

快速排序是一种高效的排序算法，它使用了分治法的策略。在快速排序中，我们选择一个“基准”元素，然后将数组分为两部分：一部分包含所有比基准小的元素，另一部分包含所有比基准大的元素。这个过程递归地应用于子数组，直到整个数组被排序。

以下是用 JavaScript 实现快速排序的代码：

```
function quickSort(arr) {
    if (arr.length < 2) {
        return arr
    }
    var pivotIndex = Math.floor(arr.length / 2)
    var pivot = arr.splice(pivotIndex, 1)[0]
    var left = []
    var right = []

    for (var i = 0
        if (arr[i] < pivot) {
            left.push(arr[i])
        } else {
            right.push(arr[i])
        }
    }

    return quickSort(left).concat([pivot], quickSort(right))
}

// 示例
var arr = [3,6,8,10,1,2,1]
console.log(quickSort(arr))
```

在这个实现中，我们首先检查数组的长度，如果长度小于2，那么数组已经是有序的，直接返回。然后，我们选择数组中间的元素作为基准（pivot），并从数组中移除它。接着，我们遍历剩余的数组元素，将它们分别放入比基准小（left）和比基准大（right）的两个数组中。最后，我们递归地对这两个子数组进行快速排序，并将排序后的子数组与基准合并，得到最终的排序结果。

请注意，这个实现为了简单起见，使用了 `splice` 方法来移除基准元素，这可能会影响原数组。在实际应用中，你可能希望避免修改原数组，或者使用其他方法来选取和移除基准元素。此外，快速排序的性能会受到基准选择的影响，实际应用中可能会采用更复杂的策略来选择基准，如随机选择或三数取中等。

8.插入排序

插入排序是一种简单直观的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。插入排序在实现上，通常采用in-place排序（即只需用到O(1)的额外空间的排序），因而在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

以下是使用 JavaScript 实现插入排序的代码：

```
function insertionSort(arr) {
    var len = arr.length
    var preIndex, current
    for (var i = 1
        preIndex = i - 1
        current = arr[i]
        while (preIndex >= 0 && arr[preIndex] > current) {
            arr[preIndex + 1] = arr[preIndex]
            preIndex--
        }
        arr[preIndex + 1] = current
    }
    return arr
}

// 示例
var arr = [4, 3, 2, 10, 12, 1, 5, 6]
console.log(insertionSort(arr))
```

在这个代码中，我们遍历数组中的每个元素，从第二个元素开始（索引为1）。对于每个元素，我们将其保存到一个临时变量`current`中，然后向前扫描已排序的序列，找到当前元素应该插入的位置。在向前扫描的过程中，我们将比当前元素大的元素向后移动一位。最后，将当前元素插入到正确的位置。这个过程对数组中的每个元素重复，直到整个数组排序完成。

9.列表转成树

在JavaScript中，将列表（数组）转换为树形结构是一个常见的任务。通常，列表中的每个对象都有一个表示其父级对象的字段（例如，`parentId`）。以下是一个简单的示例，展示如何实现这种转换：

```
function listToTree(list, parentId = null) {
    let map = {}
    let node, roots = []
    let i
    for (i = 0
        map[list[i].id] = { ...list[i], children: [] }
    }
    for (i = 0
        node = map[list[i].id]
        if (list[i].parentId === parentId) {
            roots.push(node)
        } else {
            map[list[i].parentId].children.push(node)
        }
    }
    return roots
}

// 示例数据
const list = [
    { id: 1, name: 'Node 1', parentId: null },
    { id: 2, name: 'Node 1.1', parentId: 1 },
    { id: 3, name: 'Node 1.2', parentId: 1 },
    { id: 4, name: 'Node 2', parentId: null },
    { id: 5, name: 'Node 2.1', parentId: 4 },
    { id: 6, name: 'Node 2.2', parentId: 4 },
    { id: 7, name: 'Node 2.1.1', parentId: 5 },
]

// 转换列表为树
const tree = listToTree(list)
```

```
// 打印树形结构
function printTree(tree, level = 0) {
    tree.forEach(node => {
        console.log(`${'--'.repeat(level)} ${node.name}`)
        if (node.children.length > 0) {
            printTree(node.children, level + 1)
        }
    })
}

printTree(tree)
```

在这个示例中，`listToTree` 函数接受一个列表（`list`）和一个可选的父级ID（`parentId`）。它首先创建一个映射（`map`），将每个列表项与其ID关联起来，并初始化一个空的`children`数组。然后，它遍历列表，将每个项添加到其父项的`children`数组中，或者如果它没有父项（即它是根节点），则将其添加到`roots`数组中。最后，它返回根节点的数组。

`printTree` 函数是一个辅助函数，用于以易于阅读的格式打印树形结构。它递归地遍历树，并在每个级别前打印相应数量的破折号（`--`），以表示节点的深度。

10.深度优先遍历-对树进行遍历，从第一个节点开始，遍历其子节点，直到它的所有子节点都被遍历完毕，然后再遍历它的兄弟节点

在JavaScript中，深度优先遍历（Depth-First Search, DFS）是一种用于遍历或搜索树或图的算法。这个算法会尽可能深地搜索树的分支。当节点v的所在边都已被探寻过，搜索将回溯到发现节点v的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。

以下是一个简单的JavaScript函数，它实现了对树的深度优先遍历。这个函数假设树是由节点对象组成的，每个节点对象都有一个`children`属性，该属性是一个包含该节点的所有子节点的数组。

```
function depthFirstSearch(root) {
    if (root === null) {
        return;
```

```
    }

    console.log(root.value);


    for (let i = 0; i < root.children.length; i++) {
        depthFirstSearch(root.children[i]);
    }
}


const tree = {
    value: 'root',
    children: [
        {
            value: 'child1',
            children: [
                { value: 'grandchild1', children: [] },
                { value: 'grandchild2', children: [] }
            ]
        },
        {
            value: 'child2',
            children: [
                { value: 'grandchild3', children: [] },
                { value: 'grandchild4', children: [] }
            ]
        }
    ]
};


depthFirstSearch(tree);
```

在这个例子中，`depthFirstSearch`函数首先检查传入的节点是否为`null`。如果是，函数就返回并不做任何事情。否则，它会打印出当前节点的值，然后遍历并递归调用其所有子节点。这样就保证了每个节点都会被访问，并且是先访问其所有子节点，然后再访问其兄弟节点。

11\. 广度优先遍历-以横向的维度对树进行遍历，从第一个节点开始，依次遍历其所有的兄弟节点，再遍历第一个节点的子节点，一层层向下遍历

在JavaScript中，广度优先遍历（Breadth-First Search, BFS）是一种用于遍历或搜索树或图的算法。这种算法从根节点（或任意节点）开始，访问最靠近根节点的节点。广度优先遍历使用队列数据结构来存储待访问的节点。

以下是一个使用JavaScript实现广度优先遍历的例子。假设我们有一个树形结构，每个节点都有一个`children`数组来存储其子节点。

```
function breadthFirstSearch(root) {
    if (root === null) {
        return;
    }

    const queue = [root];

    while (queue.length > 0) {
        const currentNode = queue.shift();
        console.log(currentNode.value);

        for (const child of currentNode.children) {
            queue.push(child);
        }
    }
}

const tree = {
```

```
    value: 'root',
    children: [
        {
            value: 'child1',
            children: [
                { value: 'grandchild1', children: [] },
                { value: 'grandchild2', children: [] }
            ]
        },
        {
            value: 'child2',
            children: [
                { value: 'grandchild3', children: [] },
                { value: 'grandchild4', children: [] }
            ]
        }
    ]
};


breadthFirstSearch(tree);
```

在这个例子中，`breadthFirstSearch`函数首先检查传入的根节点是否为`null`。如果不是，它将根节点放入队列中。然后，它进入一个循环，只要队列不为空，就持续执行以下操作：

1. 从队列中取出第一个节点（即最早入队的节点）。
2. 访问（在这里是打印）该节点的值。
3. 将该节点的所有子节点加入队列。

这个过程将确保树的遍历按照广度优先的方式进行，即先遍历所有兄弟节点，再遍历子节点，一层层向下遍历。

12\. 查找树形结构中符合要求的节点

在JavaScript中，查找树形结构中符合特定要求的节点通常涉及递归遍历树的所有节点，并检查每个节点的属性是否满足条件。以下是一个示例函数，它接受一个树形结构和一个回调函数作为参数，回调函数定义了节点必须满足的条件。函数会返回所有满足条件的节点数组。

```
function findNodes(tree, predicate) {
    const result = [];

    function traverse(node) {
        if (predicate(node)) {
            result.push(node);
        }
        for (const child of node.children) {
            traverse(child);
        }
    }

    traverse(tree);
    return result;
}


const tree = {
    value: 'root',
    children: [
        {
            value: 'child1',
            children: [
                { value: 'grandchild1', children: [] },
                { value: 'grandchild2', children: [] }
            ]
        },
        {
            value: 'child2',
            children: [
                { value: 'grandchild3', children: [] },
```

```
                { value: 'grandchild4', children: [] }
            ]
        }
    ]
};


const predicate = (node) => node.value.startsWith('grandchil
d');
const matchingNodes = findNodes(tree, predicate);

console.log(matchingNodes);
```

在这个例子中，`findNodes` 函数接受一个树形结构 `tree` 和一个回调函数 `predicate`。`predicate` 函数应该接受一个节点作为参数，并返回一个布尔值，表示该节点是否满足搜索条件。`findNodes` 函数通过递归调用 `traverse` 函数来遍历树的每个节点，并在遇到满足条件的节点时将其添加到结果数组中。最后，函数返回所有满足条件的节点数组。

你可以根据需要调整 `predicate` 函数来定义不同的搜索条件。例如，你可以查找具有特定属性的节点，或者基于节点的值或其他属性进行更复杂的匹配。

13\. 二叉查找树-判断一个数组，是否为某二叉查找树的前序遍历结果，二叉查找树特点是所有的左节点比父节点的值小，所有的右节点比父节点的值大

要判断一个数组是否为二叉查找树的前序遍历结果，我们可以利用二叉查找树的特性：所有左子节点的值都小于父节点，所有右子节点的值都大于父节点。在前序遍历中，遍历的顺序是根节点、左子树、右子树。

下面是一个使用JavaScript实现的函数，该函数可以判断一个数组是否是某个二叉查找树的前序遍历结果：

```
function verifyPreorder(preorder) {
    if (preorder.length === 0) {
        return true
    }
```

```
    const rootVal = preorder[0]
    const leftSubtree = []
    const rightSubtree = []

    // 分离左子树和右子树
    for (let i = 1
        if (preorder[i] < rootVal) {
            leftSubtree.push(preorder[i])
        } else {
            rightSubtree.push(preorder[i])
        }
    }

    // 递归验证左子树和右子树
    return (
        verifyPreorder(leftSubtree) && // 左子树必须是有效的二叉查找树前序遍历
        verifyPreorder(rightSubtree) && // 右子树必须是有效的二叉查找树前序遍历
        // 右子树中的所有元素必须都大于根节点的值
        rightSubtree.every(val => val > rootVal)
    )
}

// 示例
const preorderTraversal = [8, 5, 1, 7, 10, 12]
const notPreorderTraversal = [8, 5, 10, 1, 7, 12]

console.log(verifyPreorder(preorderTraversal))
console.log(verifyPreorder(notPreorderTraversal))
```

这个算法的思路是：

1. 如果数组为空，那么它肯定是一个有效的二叉查找树的前序遍历结果。
2. 否则，我们取数组的第一个元素作为根节点的值。

3. 遍历数组中剩余的元素，将它们分为左子树和右子树两部分。左子树中的元素值都小于根节点，右子树中的元素值都大于或等于根节点。
4. 递归地对左子树和右子树进行验证，确保它们也是有效的二叉查找树的前序遍历结果。
5. 同时，验证右子树中的所有元素是否都大于根节点的值。

请注意，这个算法假设输入的数组不包含重复的元素。如果允许重复元素，那么前序遍历的结果可能无法唯一确定一棵二叉查找树，因此需要根据具体问题的需求进行相应调整。

14.买卖股票问题-给定一个整数数组，其中第 i 个元素代表了第 i天的股票价格；

非负整数 fee 代表了交易股票的手续费用，求返回获得利润的最大值

这个问题可以使用动态规划（Dynamic Programming）来解决。我们可以定义两个变量，分别表示手上持有股票时的最大利润和不持有股票时的最大利润。对于每一天，我们都有两种选择：买入股票或者卖出股票。

下面是使用JavaScript实现的代码：

```
function maxProfit(prices, fee) {
    if (prices.length <= 1) {
        return 0
    }

    let cash = 0
    let hold = -prices[0]

    for (let i = 1
        // 如果今天卖出股票，则利润为之前的现金加上今天的股票价格
        cash = Math.max(cash, hold + prices[i] - fee)
        // 如果今天买入股票，则利润为之前的持有股票利润减去今天的股票价
格
        hold = Math.max(hold, cash - prices[i])
    }

    return cash
}
```

```
// 示例
const prices = [1, 3, 2, 8, 4, 9]
const fee = 2
console.log(maxProfit(prices, fee))
```

在这个例子中，我们在第2天买入股票（价格为3），在第3天卖出（价格为2），需要支付手续费2，第5天买入（价格为4），第6天卖出（价格为9），同样需要支付手续费2，所以总利润为8。

注意，我们每次卖出股票时都要支付手续费，这会影响我们的总利润。这个算法的时间复杂度是O(n)，其中n是数组的长度。

15.斐波那契数列-从第3项开始，当前项等于前两项之和：1 1 2 3 5 8 13 21 .....，计算第n项的值

在JavaScript中，可以使用递归或迭代的方式来实现斐波那契数列。由于递归的效率较低，特别是当n较大时，会导致大量的重复计算，因此在这里我将展示迭代的方式来实现斐波那契数列。

以下是一个使用迭代方式计算斐波那契数列第n项值的JavaScript函数：

```
function fibonacci(n) {
    if (n <= 0) {
        return "输入错误，n必须为正整数"
    } else if (n === 1 || n === 2) {
        return 1
    } else {
        let a = 1, b = 1, temp
        for (let i = 3
            temp = a + b
            a = b
            b = temp
        }
        return b
    }
}
```

```
// 测试函数
console.log(fibonacci(10))
```

这个函数首先检查n是否小于等于0，如果是，则返回错误信息。然后，它检查n是否等于1或2，因为斐波那契数列的前两项都是1。如果n大于2，那么函数就会进入一个循环，从第三项开始计算斐波那契数列的值，直到达到第n项。在循环中，我们使用变量a和b来存储前两项的值，并使用temp来存储当前项的值。在每次循环中，我们都会更新a、b和temp的值，以便计算下一项的值。最后，函数返回第n项的值。

16.滑动窗口最大值-给定一个数组 nums，有一个大小为 k 的滑动窗口，从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口中的k个数字。滑动窗口每次只向右移动一位，求返回滑动窗口最大值

要实现滑动窗口最大值的问题，一个常用的方法是使用双端队列（deque）。双端队列在这里的作用是为了保持队列中的元素是单调递减的，这样队列的头部就始终是窗口中的最大值。

以下是使用JavaScript实现滑动窗口最大值的代码：

```
function maxSlidingWindow(nums, k) {
    const deque = [];
    const result = [];


    function removeOutdated(index) {
        while (deque.length > 0 && deque[0] <= index - k) {
            deque.shift();
        }
    }


    function maintainDecreasing() {
        while (deque.length > 1 && nums[deque[deque.length -
1]] < nums[index]) {
            deque.pop();
```

```
        }
    }

    for (let index = 0; index < nums.length; index++) {

        removeOutdated(index);

        maintainDecreasing();

        deque.push(index);


        if (index >= k - 1) {
            result.push(nums[deque[0]]);
        }
    }

    return result;
}


const nums = [1, 3, -1, -3, 5, 3, 6, 7];
const k = 3;
console.log(maxSlidingWindow(nums, k));
```

这段代码首先定义了一个双端队列`deque`，用于存储当前窗口内元素的索引，以保持队列的单调递减性。然后，我们遍历数组`nums`，对于每个元素执行以下操作：

1. 移除队列中超出窗口范围的元素（即索引小于`index - k`的元素）。
2. 保持队列的单调递减性，即如果队列末尾的元素小于当前元素，就将其从队列中移除，直到队列为空或者末尾元素不小于当前元素。
3. 将当前元素的索引添加到队列的末尾。
4. 如果窗口已经形成（即`index >= k - 1`），则将队列头部的元素对应的值（即当前窗口的最大值）添加到结果数组`result`中。

最后，返回结果数组`result`，其中包含了每个滑动窗口的最大值。

17.最长递增子序列-一个整数数组 nums，找到其中一组最长递增子序列的值

在JavaScript中，最长递增子序列（Longest Increasing Subsequence，简称LIS）问题可以使用动态规划（Dynamic Programming）来解决。下面是一个示例实现：

```
function longestIncreasingSubsequence(nums) {
    if (nums.length === 0) return []

    // 初始化一个数组来存储每个位置的最长递增子序列的长度
    let lengths = new Array(nums.length).fill(1)
    // 初始化一个数组来存储每个位置的最长递增子序列的前一个元素的索引
    let prevIndices = new Array(nums.length).fill(-1)

    let maxLength = 1
    let endIndex = 0

    // 遍历数组，计算每个位置的最长递增子序列长度和前一个元素的索引
    for (let i = 1
        for (let j = 0
            if (nums[i] > nums[j] && lengths[i] < lengths[j]
+ 1) {
                lengths[i] = lengths[j] + 1
                prevIndices[i] = j
            }
        }
        // 更新最长递增子序列的长度和最后一个元素的索引
        if (maxLength < lengths[i]) {
            maxLength = lengths[i]
            endIndex = i
        }
    }

    // 根据最后一个元素的索引和前一个元素的索引数组，构建最长递增子序列
```

```
    let lis = []
    while (endIndex !== -1) {
        lis.unshift(nums[endIndex])
        endIndex = prevIndices[endIndex]
    }

    return lis
}

// 示例
const nums = [10, 9, 2, 5, 3, 7, 101, 18]
const lis = longestIncreasingSubsequence(nums)
console.log(lis)
```

在这个实现中，`lengths` 数组用于存储以每个位置为结尾的最长递增子序列的长度，而 `prevIndices` 数组则用于存储每个位置的最长递增子序列中前一个元素的索引。我们通过两次遍历来构建这两个数组，并在遍历过程中更新最长递增子序列的长度和最后一个元素的索引。

最后，我们根据 `endIndex` 和 `prevIndices` 数组回溯构建出最长递增子序列，并返回结果。

需要注意的是，最长递增子序列可能不唯一，因此上面的代码可能输出不同的递增子序列，但它们的长度都是相同的，即数组 `nums` 的最长递增子序列的长度。