From: https://techinterviewhandbook.org

Array

Note that because both arrays and strings are sequences (a string is an array of characters), most of the techniques here will apply to string problems.

Sliding window

Master the sliding window technique that applies to many subarray/substring problems. In a sliding window, the two pointers usually move in the same direction will never overtake each other. This ensures that each value is only visited at most twice and the time complexity is still O(n). Examples: Longest Substring Without Repeating Characters, Minimum Size Subarray Sum, Minimum Window Substring

Template

```
1. Initialize two pointers, start and end, both at the 0th index.

2. Initialize any needed variables. For example:
   - max_sum for storing the maximum sum of subarray
   - current_sum for storing the sum of the current window
   - any other specific variables you need

3. Iterate over the array/string using the end pointer:

   while end < length_of_array:

       a. Add the current element to current_sum (or perform some operation)

       b. While the current window meets a certain condition (like current_sum exceeds
a value, the window has more than a specific number of characters, etc.):
          i. Possibly update max_sum or any other variables
          ii. Remove the element at start pointer from current_sum (or perform the
opposite operation)
          iii. Increment the start pointer to make the window smaller

       c. Increment the end pointer

4. After the loop, the answer could be in max_sum or any other variables you used.
```

Example

```
// Find the maximum sum subarray of size `k`:
function maxSumSubarray(arr: number[], k: number): number {
    let start = 0;
    let currentSum = 0;
    let maxSum = Number.NEGATIVE_INFINITY;

    for (let end = 0; end < arr.length; end++) {
        currentSum += arr[end];
```

```
        if (end - start + 1 === k) {
            maxSum = Math.max(maxSum, currentSum);
            currentSum -= arr[start];
            start++;
        }
    }

    return maxSum;
}
```

Two pointers

Two pointers is a more general version of sliding window where the pointers can cross each other and can be on different arrays. Examples: Sort Colors, Palindromic Substrings

When you are given two arrays to process, it is common to have one index per array (pointer) to traverse/compare the both of them, incrementing one of the pointers when relevant. For example, we use this approach to merge two sorted arrays. Examples: Merge Sorted Array

Template

```
1. Initialize two pointers. Depending on the problem:
   - They can start at the beginning and end of the array (`start = 0, end = length_of_array - 1`), which is typical for sorted arrays.
   - Or they can start both at the beginning (`left = 0, right = 1`) or any other positions based on the requirement.

2. Use a loop (typically a while loop) to iterate while the pointers meet the criteria for traversal, e.g., `start < end`.

3. Inside the loop:
   a. Check the current elements at the two pointers.
   b. Based on the problem, decide how to adjust the pointers. Common decisions are:
      i. If the current elements meet some condition, process the current elements and then adjust the pointers (either moving `start` forward or `end` backward or both).
      ii. If the current elements don't meet the condition, adjust one of the pointers (or both) without processing the elements.

4. Continue until the loop ends.

5. The solution might be obtained during the loop's iterations, or after the loop based on the processed elements.
```

Example

```
// In a sorted array, find a pair of elements that sum up to a given target:
function twoSumSorted(arr: number[], target: number): [number, number] | null {
    let start = 0;
    let end = arr.length - 1;

    while (start < end) {
```

```
        const currentSum = arr[start] + arr[end];

        if (currentSum === target) {
            return [arr[start], arr[end]];
        } else if (currentSum < target) {
            start++;
        } else {
            end--;
        }
    }

    return null; // No pair found
}
```

Traversing from the right

Sometimes you can traverse the array starting from the right instead of the conventional approach of from the left. Examples: Daily Temperatures, Number of Visible People in a Queue

Sorting the array

Is the array sorted or partially sorted? If it is, some form of binary search should be possible. This also usually means that the interviewer is looking for a solution that is faster than O(n).

Can you sort the array? Sometimes sorting the array first may significantly simplify the problem. Obviously this would not work if the order of array elements need to be preserved. Examples: Merge Intervals, Non-overlapping Intervals

Precomputation

For questions where summation or multiplication of a subarray is involved, pre-computation using hashing or a prefix/suffix sum/product might be useful. Examples: Product of Array Except Self, Minimum Size Subarray Sum, LeetCode questions tagged "prefix-sum"

Index as a hash key

If you are given a sequence and the interviewer asks for O(1) space, it might be possible to use the array itself as a hash table. For example, if the array only has values from 1 to N, where N is the length of the array, negate the value at that index (minus one) to indicate presence of that number. Examples: First Missing Positive, Daily Temperatures

Traversing the array more than once

This might be obvious, but traversing the array twice/thrice (as long as fewer than n times) is still O(n). Sometimes traversing the array more than once can help you solve the problem while keeping the time complexity to O(n).

String

Counting characters

Often you will need to count the frequency of characters in a string. The most common way of doing that is by using a hash table/map in your language of choice. If your language has a built-in Counter class like Python, ask if you can use that instead.

If you need to keep a counter of characters, a common mistake is to say that the space complexity required for the counter is O(n). The space required for a counter of a string of latin characters is O(1) not O(n). This is because the upper bound is the range of characters, which is usually a fixed constant of 26. The input set is just lowercase Latin characters.

Anagram

An anagram is word switch or word play. It is the result of rearranging the letters of a word or phrase to produce a new word or phrase, while using all the original letters only once. In interviews, usually we are only bothered with words without spaces in them.

To determine if two strings are anagrams, there are a few approaches:

- Sorting both strings should produce the same resulting string. This takes O(n.log(n)) time and O(log(n)) space.
- If we map each character to a prime number and we multiply each mapped number together, anagrams should have the same multiple (prime factor decomposition). This takes O(n) time and O(1) space. Examples: Group Anagram
- Frequency counting of characters will help to determine if two strings are anagrams. This also takes O(n) time and O(1) space.

Palindrome

A palindrome is a word, phrase, number, or other sequence of characters which reads the same backward as forward, such as madam or racecar.

Here are ways to determine if a string is a palindrome:

- Reverse the string and it should be equal to itself.
- Have two pointers at the start and end of the string. Move the pointers inward till they meet. At every point in time, the characters at both pointers should match.
- The order of characters within the string matters, so hash tables are usually not helpful.

When a question is about counting the number of palindromes, a common trick is to have two pointers that move outward, away from the middle. Note that palindromes can be even or odd length. For each middle pivot position, you need to check it twice - once that includes the character and once without the character. This technique is used in Longest Palindromic Substring.

- For substrings, you can terminate early once there is no match
- For subsequences, use dynamic programming as there are overlapping subproblems

Hash Table

- Describe an implementation of a least-used cache, and big-O notation of it.

Template

```
Initialize an LRU Cache with a given capacity:

1. Set cache capacity.
2. Create an empty hashmap that will hold key-value pairs (key -> node in doubly-
linked list).
3. Create an empty doubly-linked list (nodes will have key-value pairs).

To GET a value from the cache:

1. If the key is not in the hashmap:
   a. Return "Not Found" or equivalent.
2. If the key is in the hashmap:
   a. Use the hashmap to get the node associated with that key from the doubly-linked
list.
   b. Move the accessed node to the head of the doubly-linked list (indicating recent
use).
   c. Return the value of the node.

To PUT a value in the cache:

1. If the key is already in the hashmap:
   a. Use the hashmap to get the node associated with that key from the doubly-linked
list.
   b. Update the value of the node.
   c. Move the node to the head of the doubly-linked list.
2. If the key is not in the hashmap:
   a. Create a new node with the key-value pair.
   b. Add the node to the head of the doubly-linked list.
   c. Add the key and the node reference to the hashmap.
   d. If the size of the hashmap exceeds the cache capacity:
      i. Remove the node from the tail of the doubly-linked list (least recently used
item).
      ii. Remove the corresponding key from the hashmap.

Helper Method - MoveToHead(node):

1. Remove the node from its current position in the doubly-linked list.
2. Add the node to the head of the doubly-linked list.
```

Example

```
class ListNode {
    key: number;
    value: number;
    prev: ListNode | null = null;
    next: ListNode | null = null;

    constructor(key: number, value: number) {
        this.key = key;
        this.value = value;
    }
}
```

```
class LRUCache {
    private capacity: number;
    private map: Map<number, ListNode>;
    private head: ListNode;  // Most recently used
    private tail: ListNode;  // Least recently used

    constructor(capacity: number) {
        this.capacity = capacity;
        this.map = new Map();
        this.head = new ListNode(0, 0);
        this.tail = new ListNode(0, 0);
        this.head.next = this.tail;
        this.tail.prev = this.head;
    }

    get(key: number): number {
        if (!this.map.has(key)) return -1;

        const node = this.map.get(key)!;
        this.moveToHead(node);
        return node.value;
    }

    put(key: number, value: number): void {
        if (this.map.has(key)) {
            const node = this.map.get(key)!;
            node.value = value;
            this.moveToHead(node);
        } else {
            const newNode = new ListNode(key, value);
            this.map.set(key, newNode);
            this.addToHead(newNode);

            if (this.map.size > this.capacity) {
                const tailNode = this.removeTail();
                this.map.delete(tailNode.key);
            }
        }
    }

    private moveToHead(node: ListNode): void {
        this.removeNode(node);
        this.addToHead(node);
    }

    private addToHead(node: ListNode): void {
        node.prev = this.head;
        node.next = this.head.next;
        this.head.next!.prev = node;
        this.head.next = node;
    }
```

```
    private removeNode(node: ListNode): void {
        node.prev!.next = node.next;
        node.next!.prev = node.prev;
    }

    private removeTail(): ListNode {
        const tailNode = this.tail.prev!;
        this.removeNode(tailNode);
        return tailNode;
    }
}
```

Recursion

- Always remember to always define a base case so that your recursion will end.
- Recursion is useful for permutation, because it generates all combinations and tree-based questions. You should know how to generate all permutations of a sequence as well as how to handle duplicates.
- Recursion implicitly uses a stack. Hence all recursive approaches can be rewritten iteratively using a stack. Beware of cases where the recursion level goes too deep and causes a stack overflow (the default limit in Python is 1000). You may get bonus points for pointing this out to the interviewer. Recursion will never be O(1) space complexity because a stack is involved, unless there is tail-call optimization (TCO). Find out if your chosen language supports TCO.
- Number of base cases - In the fibonacci recursion, note that one of the recursive calls invoke fib(n - 2). This indicates that you should have 2 base cases defined so that your code covers all possible invocations of the function within the input range. If your recursive function only invokes fn(n - 1), then only one base case is needed.

Memoization

In some cases, you may be computing the result for previously computed inputs. Let's look at the Fibonacci example again. fib(5) calls fib(4) and fib(3), and fib(4) calls fib(3) and fib(2). fib(3) is being called twice! If the value for fib(3) is memoized and used again, that greatly improves the efficiency of the algorithm and the time complexity becomes O(n).

Sorting

Binary search

When a given sequence is in a sorted order (be it ascending or descending), using binary search should be one of the first things that come to your mind.

Template

```
function binarySearch(array, target):
    1. Define two pointers: "left" initialized to 0 and "right" initialized to (length of array - 1).

    2. While "left" is less than or equal to "right":
```

```
       a. Calculate the middle index: mid = (left + right) / 2.
       b. If array[mid] equals target:
          i. Return mid.
       c. If array[mid] is less than target:
          i. Set left = mid + 1.
       d. Else:
          i. Set right = mid - 1.

    3. Return "Not Found" or an equivalent indicator (like -1).
```

```
function binarySearch(arr: number[], target: number): number {
    let left = 0;
    let right = arr.length - 1;

    while (left <= right) {
        const mid = Math.floor((left + right) / 2);

        if (arr[mid] === target) {
            return mid;  // Target value found, return its index
        }

        if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1;  // Target value not found in the array
}
```

If you want to find the closest value that's less than the target value in a sorted array, you can modify the binary search algorithm slightly. At the end of the standard binary search loop, the right pointer will indicate the position where the target should be if it were in the array (or before it).

You can use this property to get the closest value less than the target. After the loop ends, the value at `arr[right]` would be the closest value less than the target (if right is within the bounds of the array).

```
// White loop here
while (...)

// Check if 'right' is within bounds
if (right >= 0) {
    return arr[right];
}
```

Sorting an input that has limited range

Counting sort is a non-comparison-based sort you can use on numbers where you know the range of values beforehand.

Template

```
function countingSort(inputArray, maxValue):
    1. Initialize an array "count" of zeros with a size of (maxValue + 1).
    2. For each element "x" in inputArray:
       a. Increment count[x] by 1.

    3. Initialize an output array "sortedArray" of the same size as inputArray.
    4. Initialize a position variable "pos" to 0.
    5. For each index "i" from 0 to maxValue:
       a. While count[i] is greater than 0:
          i. Place the value "i" in sortedArray[pos].
          ii. Increment pos by 1.
          iii. Decrement count[i] by 1.

    6. Return sortedArray.
```

Example

```
function countingSort(arr: number[], maxValue: number): number[] {
    // Step 1: Initialize count array
    const count: number[] = new Array(maxValue + 1).fill(0);

    // Step 2: Populate count array with frequencies
    for (let num of arr) {
        count[num]++;
    }

    // Step 3-5: Reconstruct the sorted array using the count array
    let sortedIndex = 0;
    const sortedArray: number[] = new Array(arr.length);

    for (let i = 0; i < count.length; i++) {
        while (count[i] > 0) {
            sortedArray[sortedIndex] = i;
            sortedIndex++;
            count[i]--;
        }
    }

    // Step 6: Return the sorted array
    return sortedArray;
}
```

Matrix

Create an empty X x M matrix:

```
const matrix = Array(X).fill(undefined).map(() => Array(M).fill(-1)); // Replace -1 for whatever default value you want
```

Transposing a matrix

The transpose of a matrix is found by interchanging its rows into columns or columns into rows.

Many grid-based games can be modeled as a matrix, such as Tic-Tac-Toe, Sudoku, Crossword, Connect 4, Battleship, etc. It is not uncommon to be asked to verify the winning condition of the game. For games like Tic-Tac-Toe, Connect 4 and Crosswords, where verification has to be done vertically and horizontally, one trick is to write code to verify the matrix for the horizontal cells, transpose the matrix, and reuse the logic for horizontal verification to verify originally vertical cells (which are now horizontal).

```
function transpose(matrix: number[][]): number[][] {
    // Create a new 2D array of size columns x rows of the original matrix
    const transposed: number[][] = Array.from({ length: matrix[0].length }, () => []);

    for (let i = 0; i < matrix.length; i++) {
        for (let j = 0; j < matrix[0].length; j++) {
            transposed[j][i] = matrix[i][j];
        }
    }

    return transposed;
}
```

Linked List

Sentinel/dummy nodes

Adding a sentinel/dummy node at the head and/or tail might help to handle many edge cases where operations have to be performed at the head or the tail. The presence of dummy nodes essentially ensures that operations will never be done on the head or the tail, thereby removing a lot of headache in writing conditional checks to dealing with null pointers. Be sure to remember to remove them at the end of the operation.

Example

```
class ListNode {
    val: number;
    next: ListNode | null = null;

    constructor(val?: number, next?: ListNode | null) {
        this.val = (val === undefined ? 0 : val);
        this.next = (next === undefined ? null : next);
    }
}

function mergeTwoSortedLists(l1: ListNode | null, l2: ListNode | null): ListNode |
null {
    const dummy = new ListNode(-1);  // Sentinel/dummy node
    let current = dummy;  // Pointer to build the merged list

    while (l1 !== null && l2 !== null) {
```

```
        if (l1.val < l2.val) {
            current.next = l1;
            l1 = l1.next;
        } else {
            current.next = l2;
            l2 = l2.next;
        }
        current = current.next!;
    }

    // If there are remaining nodes in l1 or l2
    if (l1 !== null) {
        current.next = l1;
    } else {
        current.next = l2;
    }

    return dummy.next;  // Return the next of dummy as the merged list's head
}
```

In the `mergeTwoSortedLists` function, we utilize a dummy node as the head of our merged list. By doing this, we don't have to write special logic to initialize the head of the merged list, because `dummy.next` will naturally point to the start of the merged list at the end of the process. The dummy node serves as a placeholder and helps in simplifying the code.

Two pointers

Two pointer approaches are also common for linked lists. This approach is used for many classic linked list problems.

- Getting the kth from last node - Have two pointers, where one is k nodes ahead of the other. When the node ahead reaches the end, the other node is k nodes behind
- Detecting cycles - Have two pointers, where one pointer increments twice as much as the other, if the two pointers meet, means that there is a cycle
- Getting the middle node - Have two pointers, where one pointer increments twice as much as the other. When the faster node reaches the end of the list, the slower node will be at the middle

Using space

Many linked list problems can be easily solved by creating a new linked list and adding nodes to the new linked list with the final result. However, this takes up extra space and makes the question much less challenging. The interviewer will usually request that you modify the linked list in-place and solve the problem without additional storage. You can borrow ideas from the Reverse a Linked List problem.

Elegant modification operations

As mentioned earlier, a linked list's non-sequential nature of memory allows for efficient modification of its contents. Unlike arrays where you can only modify the value at a position, for linked lists you can also modify the next pointer in addition to the value.

Here are some common operations and how they can be achieved easily:

- Truncate a list - Set the next pointer to null at the last element
- Swapping values of nodes - Just like arrays, just swap the value of the two nodes, there's no need to swap the next pointer
- Combining two lists - attach the head of the second list to the tail of the first list

Queue

Most languages don't have a built-in Queue class which can be used, and candidates often use arrays (JavaScript) or lists (Python) as a queue. However, note that the dequeue operation (assuming the front of the queue is on the left) in such a scenario will be O(n) because it requires shifting of all other elements left by one. In such cases, you can flag this to the interviewer and say that you assume that there's a queue data structure to use which has an efficient dequeue operation.

Interval

Sort the array of intervals by its starting point

A common routine for interval questions is to sort the array of intervals by each interval's starting value. This step is crucial to solving the Merge Intervals question.

Checking if two intervals overlap

Be familiar with writing code to check if two intervals overlap.

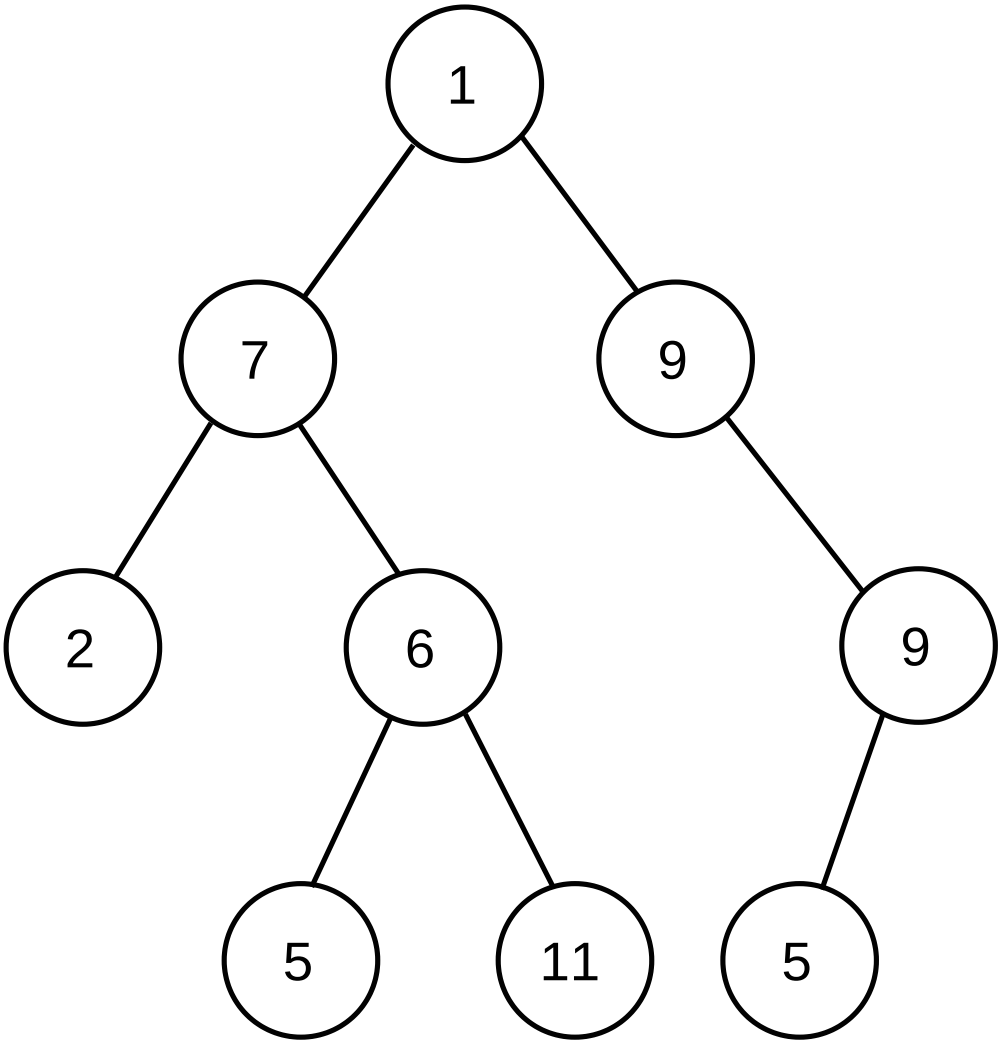
```
def is_overlap(a, b):
  return a[0] < b[1] and b[0] < a[1]
```

Trick to remember: both the higher pos must be greater then both lower pos.

Merging two intervals

```
def merge_overlapping_intervals(a, b):
  return [min(a[0], b[0]), max(a[1], b[1])]
```

Tree



Traversals

- In-order traversal - Left -> Root -> Right.
  Result: 2, 7, 5, 6, 11, 1, 9, 5, 9
- Pre-order traversal - Root -> Left -> Right
  Result: 1, 7, 2, 6, 5, 11, 9, 9, 5
- Post-order traversal - Left -> Right -> Root
  Result: 2, 5, 11, 6, 7, 5, 9, 9, 1

Example (Recursive)

```
class TreeNode {
    val: number;
    left: TreeNode | null = null;
    right: TreeNode | null = null;
```

```
    constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
        this.val = (val === undefined ? 0 : val);
        this.left = (left === undefined ? null : left);
        this.right = (right === undefined ? null : right);
    }
}

// In-Order Traversal (Left, Root, Right)
function inOrderTraversal(root: TreeNode | null): number[] {
    let result: number[] = [];

    function helper(node: TreeNode | null) {
        if (node === null) return;

        helper(node.left);
        result.push(node.val);
        helper(node.right);
    }

    helper(root);
    return result;
}

// Pre-Order Traversal (Root, Left, Right)
function preOrderTraversal(root: TreeNode | null): number[] {
    let result: number[] = [];

    function helper(node: TreeNode | null) {
        if (node === null) return;

        result.push(node.val);
        helper(node.left);
        helper(node.right);
    }

    helper(root);
    return result;
}

// Post-Order Traversal (Left, Right, Root)
function postOrderTraversal(root: TreeNode | null): number[] {
    let result: number[] = [];

    function helper(node: TreeNode | null) {
        if (node === null) return;

        helper(node.left);
        helper(node.right);
        result.push(node.val);
    }
```

```
    helper(root);
    return result;
}
```

Example (Iterative)

```
class TreeNode {
    val: number;
    left: TreeNode | null = null;
    right: TreeNode | null = null;

    constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
        this.val = (val === undefined ? 0 : val);
        this.left = (left === undefined ? null : left);
        this.right = (right === undefined ? null : right);
    }
}

// In-Order Traversal (Left, Root, Right)
function inOrderTraversal(root: TreeNode | null): number[] {
    const result: number[] = [];
    const stack: TreeNode[] = [];
    let current: TreeNode | null = root;

    while (current !== null || stack.length > 0) {
        while (current !== null) {
            stack.push(current);
            current = current.left;
        }

        current = stack.pop()!;
        result.push(current.val);
        current = current.right;
    }

    return result;
}

// Pre-Order Traversal (Root, Left, Right)
function preOrderTraversal(root: TreeNode | null): number[] {
    const result: number[] = [];
    const stack: TreeNode[] = [];
    if (root) stack.push(root);

    while (stack.length > 0) {
        const current = stack.pop()!;
        result.push(current.val);

        if (current.right) stack.push(current.right);
        if (current.left) stack.push(current.left);
    }
```

```
    return result;
}

// Post-Order Traversal (Left, Right, Root)
function postOrderTraversal(root: TreeNode | null): number[] {
    const result: number[] = [];
    const stack: TreeNode[] = [];
    let lastVisited: TreeNode | null = null;

    while (root || stack.length > 0) {
        while (root) {
            stack.push(root);
            root = root.left;
        }

        const peekNode = stack[stack.length - 1];

        if (!peekNode.right || peekNode.right === lastVisited) {
            result.push(peekNode.val);
            lastVisited = stack.pop()!;
        } else {
            root = peekNode.right;
        }
    }

    return result;
}
```

Binary search tree (BST)

In-order traversal of a BST will give you all elements in order.

Be very familiar with the properties of a BST and validating that a binary tree is a BST. This comes up more often than expected.

When a question involves a BST, the interviewer is usually looking for a solution which runs faster than O(n).

Time complexity
Operation Big-O
Access O(log(n))
Search O(log(n))
Insert O(log(n))
Remove O(log(n))

Space complexity of traversing balanced trees is O(h) where h is the height of the tree, while traversing very skewed trees (which is essentially a linked list) will be O(n).

Use recursion

Recursion is the most common approach for traversing trees. When you notice that the subtree problem can be used to solve the entire problem, try using recursion.

When using recursion, always remember to check for the base case, usually where the node is `null`.

Sometimes it is possible that your recursive function needs to return two values.

Traversing by level

When you are asked to traverse a tree by level, use breadth-first search.

Graph

Graph representations

You can be given a list of edges and you have to build your own graph from the edges so that you can perform a traversal on them. The common graph representations are:

- Adjacency matrix
- Adjacency list
- Hash table of hash tables

Using a hash table of hash tables would be the simplest approach during algorithm interviews. It will be rare that you have to use an adjacency matrix or list for graph questions during interviews.

```
// Let's assume you're given a list of edges as input, where each edge is a tuple of
two nodes, e.g., ["A", "B"] means there's an edge between nodes "A" and "B"

type Node = string;
type Graph = Map<Node, Map<Node, boolean>>;

function addEdge(graph: Graph, from: Node, to: Node) {
    if (!graph.has(from)) {
        graph.set(from, new Map<Node, boolean>());
    }
    graph.get(from)!.set(to, true);
}

function buildGraph(edges: [Node, Node][]): Graph {
    const graph = new Map<Node, Map<Node, boolean>>();

    for (const [from, to] of edges) {
        addEdge(graph, from, to);
        addEdge(graph, to, from); // For undirected graph. Remove this for directed
graph.
    }

    return graph;
}

// Test
const edges: [Node, Node][] = [
    ["A", "B"],
    ["A", "C"],
    ["B", "D"],
```

```
    ["C", "D"],
    ["D", "E"]
];

const graph = buildGraph(edges);

console.log(graph);
```

In algorithm interviews, graphs are commonly given in the input as 2D matrices where cells are the nodes and each cell can traverse to its adjacent cells (up/down/left/right). Hence it is important that you be familiar with traversing a 2D matrix. When traversing the matrix, always ensure that your current position is within the boundary of the matrix and has not been visited before.

```
type Matrix = number[][];


// O(rows * cols) -> time and space
function traverseMatrix(matrix: Matrix): void {
    if (!matrix || matrix.length === 0 || matrix[0].length === 0) {
        return;
    }

    const rows = matrix.length;
    const cols = matrix[0].length;
    const visited: boolean[][] = Array.from({ length: rows }, () =>
Array(cols).fill(false));

    function isValid(row: number, col: number): boolean {
        return row >= 0 && row < rows && col >= 0 && col < cols && !visited[row][col];
    }

    function dfs(row: number, col: number): void {
        if (!isValid(row, col)) {
            return;
        }

        console.log(matrix[row][col]); // Process the current cell
        visited[row][col] = true;

        // Traverse up/down/left/right
        dfs(row - 1, col); // Up
        dfs(row + 1, col); // Down
        dfs(row, col - 1); // Left
        dfs(row, col + 1); // Right
    }

    for (let i = 0; i < rows; i++) {
        for (let j = 0; j < cols; j++) {
            if (!visited[i][j]) {
                dfs(i, j);
            }
```

```
        }
    }
}

// Test
const matrix: Matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];

traverseMatrix(matrix);
```

2D Matrix as a graph

Transforming the problem of traversing a 2D matrix into a graph problem involves viewing each cell of the matrix as a node of a graph and the possible movements (e.g., up, down, left, right) from a cell as edges connecting the nodes.

Here's a step-by-step guide on how you can transform the problem:

1. **Nodes Representation**: Each cell in the matrix, identified by its coordinates (i, j), can be treated as a node in the graph.
2. **Edges Representation**: For each cell, consider its adjacent cells. An edge exists between two nodes if you can move from one cell to the adjacent cell. For instance, if movements are restricted to up, down, left, and right, then:
    - The node (i, j) will have an edge to (i-1, j) if (i-1, j) is a valid cell (i.e., moving upwards).
    - The node (i, j) will have an edge to (i+1, j) if (i+1, j) is a valid cell (i.e., moving downwards).
    - The node (i, j) will have an edge to (i, j-1) if (i, j-1) is a valid cell (i.e., moving left).
    - The node (i, j) will have an edge to (i, j+1) if (i, j+1) is a valid cell (i.e., moving right).
3. **Graph Representation**: There are several ways to represent a graph, such as an adjacency list, adjacency matrix, or a hash table of hash tables. In the context of a matrix traversal, an adjacency list is often a good fit. Each node (i.e., matrix cell) would have a list of its adjacent nodes.
4. **Traversal**: With the graph constructed, you can apply standard graph traversal algorithms like DFS or BFS to explore the nodes.
5. **Special Conditions**: If there are certain cells in the matrix that you cannot traverse (like obstacles), you simply skip adding them as valid edges in the graph representation.

This transformation is particularly useful when you have more complex conditions for traversal, or when the problem involves finding paths, connected components, etc. It allows you to leverage a wide range of graph algorithms to solve matrix-based problems.

Example

```
type Node = [number, number];
type Graph = Map<string, Node[]>;

function matrixToGraph(matrix: number[][]): Graph {
    const rows = matrix.length;
    const cols = matrix[0].length;
    const graph: Graph = new Map();

    function getNodeKey(node: Node): string {
        return `${node[0]},${node[1]}`;
    }

    function getAdjacentNodes(row: number, col: number): Node[] {
        const directions: Node[] = [[-1, 0], [1, 0], [0, -1], [0, 1]];  // Up, Down,
Left, Right
        const adjacentNodes: Node[] = [];

        for (const [dx, dy] of directions) {
            const newRow = row + dx;
            const newCol = col + dy;

            if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols) {
                adjacentNodes.push([newRow, newCol]);
            }
        }

        return adjacentNodes;
    }

    for (let i = 0; i < rows; i++) {
        for (let j = 0; j < cols; j++) {
            const node: Node = [i, j];
            const key = getNodeKey(node);
            graph.set(key, getAdjacentNodes(i, j));
        }
    }

    return graph;
}

function traverseGraph(graph: Graph, startNode: Node): void {
    const visited: Set<string> = new Set();

    function dfs(node: Node): void {
        const key = `${node[0]},${node[1]}`;
        if (visited.has(key)) return;

        console.log(node);
        visited.add(key);

        const neighbors = graph.get(key) || [];
        for (const neighbor of neighbors) {
```

```
            dfs(neighbor);
        }
    }

    dfs(startNode);
}

// Test
const matrix: number[][] = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];

const graph = matrixToGraph(matrix);
traverseGraph(graph, [0, 0]);  // Start traversal from top-left corner
```

Time complexity

|V| is the number of vertices while |E| is the number of edges.

Algorithm Big-O
Depth-first search O(|V| + |E|)
Breadth-first search O(|V| + |E|)
Topological sort O(|V| + |E|)

Notes

A tree-like diagram could very well be a graph that allows for cycles and a naive recursive solution would not work. In that case you will have to handle cycles and keep a set of visited nodes when traversing.

Ensure you are correctly keeping track of visited nodes and not visiting each node more than once. Otherwise your code could end up in an infinite loop.

Depth-first search

```python
def dfs(matrix):
  # Check for an empty matrix/graph.
  if not matrix:
    return []

  rows, cols = len(matrix), len(matrix[0])
  visited = set()
  directions = ((0, 1), (0, -1), (1, 0), (-1, 0))

  def traverse(i, j):
    if (i, j) in visited:
      return

    visited.add((i, j))

    # Traverse neighbors.
```

```
    for direction in directions:
      next_i, next_j = i + direction[0], j + direction[1]
      if 0 <= next_i < rows and 0 <= next_j < cols:

        # Add in question-specific checks, where relevant.
        traverse(next_i, next_j)

  for i in range(rows):
    for j in range(cols):
      traverse(i, j)
```

Breadth-first search

```
from collections import deque

def bfs(matrix):
  # Check for an empty matrix/graph.
  if not matrix:
    return []

  rows, cols = len(matrix), len(matrix[0])
  visited = set()
  directions = ((0, 1), (0, -1), (1, 0), (-1, 0))

  def traverse(i, j):
    queue = deque([(i, j)])
    while queue:
      curr_i, curr_j = queue.popleft()

      if (curr_i, curr_j) not in visited:
        visited.add((curr_i, curr_j))

        # Traverse neighbors.
        for direction in directions:
          next_i, next_j = curr_i + direction[0], curr_j + direction[1]
          if 0 <= next_i < rows and 0 <= next_j < cols:

            # Add in question-specific checks, where relevant.
            queue.append((next_i, next_j))

  for i in range(rows):
    for j in range(cols):
      traverse(i, j)
```

Topological sorting

A topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering. Precisely, a topological sort is a graph traversal in which each node v is visited only after all its dependencies are visited.

Topological sorting is most commonly used for scheduling a sequence of jobs or tasks which has dependencies on other jobs/tasks. The jobs are represented by vertices, and there is an edge from x to y if job x must be completed before job y can be started.

Another example is taking courses in university where courses have pre-requisites.

In the context of directed graphs, in-degree of a node refers to the number of incoming edges to that node. In simpler terms, it's a count of how many vertices "point to" or "lead to" the given vertex.

```
type Graph = Map<number, number[]>;

function topologicalSort(graph: Graph): number[] | null {
    const numNodes = graph.size;
    const inDegree: Map<number, number> = new Map();
    const zeroInDegreeQueue: number[] = [];
    const result: number[] = [];

    // Initialize in-degree counts
    for (const [node, neighbors] of graph.entries()) {
        inDegree.set(node, 0);
    }

    for (const [, neighbors] of graph.entries()) {
        for (const neighbor of neighbors) {
            inDegree.set(neighbor, (inDegree.get(neighbor) || 0) + 1);
        }
    }

    // Find all nodes with zero in-degree
    for (const [node, degree] of inDegree.entries()) {
        if (degree === 0) {
            zeroInDegreeQueue.push(node);
        }
    }

    while (zeroInDegreeQueue.length) {
        const current = zeroInDegreeQueue.shift()!;
        result.push(current);

        const neighbors = graph.get(current) || [];
        for (const neighbor of neighbors) {
            inDegree.set(neighbor, inDegree.get(neighbor)! - 1);
            if (inDegree.get(neighbor) === 0) {
                zeroInDegreeQueue.push(neighbor);
            }
        }
    }

    if (result.length !== numNodes) {
        return null;  // Graph has a cycle
    }
```

```
    return result;
}

// Test
const graph: Graph = new Map();
graph.set(5, [2]);
graph.set(4, [0, 2]);
graph.set(2, [3]);
graph.set(3, [1]);
graph.set(1, []);
graph.set(0, []);

const sortedOrder = topologicalSort(graph);
console.log(sortedOrder);  // Possible output: [5, 4, 2, 3, 1, 0]
```

Heap

A heap is a specialized tree-based data structure which is a complete tree that satisfies the heap property.

- Max heap - In a max heap, the value of a node must be greatest among the node values in its entire subtree. The same property must be recursively true for all nodes in the tree.
- Min heap - In a min heap, the value of a node must be smallest among the node values in its entire subtree. The same property must be recursively true for all nodes in the tree.

In the context of algorithm interviews, heaps and priority queues can be treated as the same data structure. A heap is a useful data structure when it is necessary to repeatedly remove the object with the highest (or lowest) priority, or when insertions need to be interspersed with removals of the root node.

Time complexity

Operation Big-O
Find max/min O(1)
Insert O(log(n))
Remove O(log(n))
Heapify (create a heap out of given array of elements) O(n)

Mention of k

If you see a top or lowest k being mentioned in the question, it is usually a signal that a heap can be used to solve the problem, such as in Top K Frequent Elements.

If you require the top k elements use a Min Heap of size k. Iterate through each element, pushing it into the heap (for python heapq, invert the value before pushing to find the max). Whenever the heap size exceeds k, remove the minimum element, that will guarantee that you have the k largest elements.

Trie

Tries are special trees (prefix trees) that make searching and storing strings more efficient. Tries have many practical applications, such as conducting searches and

providing autocomplete. It is helpful to know these common applications so that you can easily identify when a problem can be efficiently solved using a trie.

Be familiar with implementing from scratch, a Trie class and its add, remove and search methods.

Example:

```
class TrieNode {
    children: { [key: string]: TrieNode } = {};
    isEndOfWord: boolean = false;
}

class Trie {
    private root: TrieNode = new TrieNode();

    // Inserts a word into the trie
    insert(word: string): void {
        let currentNode = this.root;
        for (let char of word) {
            if (!currentNode.children[char]) {
                currentNode.children[char] = new TrieNode();
            }
            currentNode = currentNode.children[char];
        }
        currentNode.isEndOfWord = true;
    }

    // Returns if the word is in the trie
    search(word: string): boolean {
        let currentNode = this.root;
        for (let char of word) {
            if (!currentNode.children[char]) {
                return false;
            }
            currentNode = currentNode.children[char];
        }
        return currentNode.isEndOfWord;
    }

    // Returns if there's any word in the trie that starts with the given prefix
    startsWith(prefix: string): boolean {
        let currentNode = this.root;
        for (let char of prefix) {
            if (!currentNode.children[char]) {
                return false;
            }
            currentNode = currentNode.children[char];
        }
        return true;
    }
}
```

```
// Test
const trie = new Trie();
trie.insert("apple");
console.log(trie.search("apple"));    // Expected output: true
console.log(trie.search("app"));      // Expected output: false
console.log(trie.startsWith("app"));  // Expected output: true
trie.insert("app");
console.log(trie.search("app"));      // Expected output: true
```

Time complexity

`m` is the length of the string used in the operation.

Operation Big-O
Search O(m)
Insert O(m)
Remove O(m)

Techniques

Sometimes preprocessing a dictionary of words (given in a list) into a trie, will improve the efficiency of searching for a word of length k, among n words. Searching becomes O(k) instead of O(n).

Geometry

Distance between two points

```
type Point = {
    x: number,
    y: number
};

function distanceBetweenTwoPoints(point1: Point, point2: Point): number {
    return Math.sqrt(Math.pow(point2.x - point1.x, 2) + Math.pow(point2.y - point1.y,
2));
}
```

Overlapping Circles

```
type Circle = {
    center: Point,
    radius: number
};

function areCirclesOverlapping(circle1: Circle, circle2: Circle): boolean {
    const distance = distanceBetweenTwoPoints(circle1.center, circle2.center);
    return distance <= (circle1.radius + circle2.radius);
}
```

Overlapping Rectanges

```
type Rectangle = {
    topLeft: Point,
```

```
    bottomRight: Point
};

function areRectanglesOverlapping(rect1: Rectangle, rect2: Rectangle): boolean {
    // Check if one rectangle is to the left of the other
    if (rect1.topLeft.x > rect2.bottomRight.x || rect2.topLeft.x >
rect1.bottomRight.x) {
        return false;
    }

    // Check if one rectangle is above the other
    if (rect1.topLeft.y < rect2.bottomRight.y || rect2.topLeft.y <
rect1.bottomRight.y) {
        return false;
    }

    return true; // If none of the above cases occurred, rectangles are overlapping
}
```

Notorious Problems

Robot Room Cleaner

Link: https://leetcode.com/problems/robot-room-cleaner/description/

```
/**
 * class Robot {
 *      // Returns true if the cell in front is open and robot moves into the cell.
 *      // Returns false if the cell in front is blocked and robot stays in the
current cell.
 *         move(): boolean {}
 *
 *      // Robot will stay in the same cell after calling turnLeft/turnRight.
 *      // Each turn will be 90 degrees.
 *         turnRight() {}
 *
 *      // Robot will stay in the same cell after calling turnLeft/turnRight.
 *      // Each turn will be 90 degrees.
 *         turnLeft() {}
 *
 *         // Clean the current cell.
 *         clean(): {}
 * }
 */

function cleanRoom(robot: Robot) {
    const dirs = [[-1, 0], [0, 1], [1, 0], [0, -1]];
    const visited = new Set();

    const goBack = () => {
        robot.turnRight();
```

```
        robot.turnRight();
        robot.move();
        robot.turnRight();
        robot.turnRight();
    }

    const backtrack = (row: number, col: number, d: number) => {
        visited.add([row, col].join());
        robot.clean();

        for (let i = 0; i < 4; ++i) {
            const newD = (d + i) % 4;
            const [newRow, newCol] = [row + dirs[newD][0], col + dirs[newD][1]];

            if (!visited.has([newRow, newCol].join()) && robot.move()) {
                backtrack(newRow, newCol, newD);
                goBack();
            }

            robot.turnRight();
        }
    }

    backtrack(0, 0, 0);
};
```

Kadane's Algorithm

Kadane's algorithm is used to find the maximum sum of a contiguous subarray within a one-dimensional array of numbers.

```
function maxSubArraySum(arr: number[]): number {
    if (arr.length === 0) {
        return 0;
    }

    let maxCurrent = arr[0];
    let maxGlobal = arr[0];

    for (let i = 1; i < arr.length; i++) {
        maxCurrent = Math.max(arr[i], maxCurrent + arr[i]);
        maxGlobal = Math.max(maxGlobal, maxCurrent);
    }

    return maxGlobal;
}

// Example usage:
const numbers = [-2, 1, -3, 4, -1, 2, 1, -5, 4];
console.log(maxSubArraySum(numbers));  // Outputs: 6 (because [4, -1, 2, 1] has the largest sum)
```

If you want to track the start and end indices of the subarray with the largest sum, you can use the following code:

```
function maxSubArraySumWithIndices(arr: number[]): { sum: number, start: number, end: number } {
    if (arr.length === 0) {
        return { sum: 0, start: -1, end: -1 };
    }

    let maxCurrent = arr[0];
    let maxGlobal = arr[0];

    let start = 0;
    let end = 0;
    let tempStart = 0;

    for (let i = 1; i < arr.length; i++) {
        if (arr[i] > maxCurrent + arr[i]) {
            maxCurrent = arr[i];
            tempStart = i;
        } else {
            maxCurrent += arr[i];
        }

        if (maxCurrent > maxGlobal) {
            maxGlobal = maxCurrent;
            start = tempStart;
            end = i;
        }
    }

    return { sum: maxGlobal, start: start, end: end };
}

// Example usage:
const numbers = [-2, 1, -3, 4, -1, 2, 1, -5, 4];
const result = maxSubArraySumWithIndices(numbers);
console.log(`Max sum is: ${result.sum} from index ${result.start} to ${result.end}`);
// Outputs: Max sum is: 6 from index 3 to 6 (because [4, -1, 2, 1] has the largest sum)
```

Great Explanation of Dynamic Programming (DP)

- https://leetcode.com/problems/word-break-ii/editorial/

Disjoint Union Set (DSU)

```
/*
Return the lexicographically smallest string that s can be changed to after using the swaps.

Input: s = "dcab", pairs = [[0,3],[1,2]]
```

```
Swap s[0] and s[3], s = "bcad"
Swap s[1] and s[2], s = "bacd"

Input: s = "cba", pairs = [[0,1],[1,2]]
Swap s[0] and s[1], s = "bca"
Swap s[1] and s[2], s = "bac"
Swap s[0] and s[1], s = "abc"

Input: s = "cbfdae", pairs = [[0,1],[3,2],[5,2],[1,4]]
Output: "abdecf"
*/
function smallestStringWithSwaps(s: string, pairs: number[][]): string {
    const parent: number[] = [];
    for (let i = 0; i < s.length; i++) {
        parent[i] = i;
    }

    // Find the parent of a node
    function find(i: number): number {
        if (i !== parent[i]) {
            parent[i] = find(parent[i]);
        }
        return parent[i];
    }

    // Union the two nodes
    function union(i: number, j: number) {
        parent[find(i)] = find(j);
    }

    // Create the disjoint set using the pairs
    for (let [i, j] of pairs) {
        union(i, j);
    }

    // Create mapping of root -> [indices]
    const indexGroups: { [key: number]: number[] } = {};
    for (let i = 0; i < s.length; i++) {
        const root = find(i);
        if (!indexGroups[root]) {
            indexGroups[root] = [];
        }
        indexGroups[root].push(i);
    }

    let chars: string[] = s.split('');
    for (let indices of Object.values(indexGroups)) {
        const sortedChars = indices.map(i => chars[i]).sort();
        for (let i = 0; i < indices.length; i++) {
            chars[indices[i]] = sortedChars[i];
        }
    }
```

```
    return chars.join('');
}

// Test cases
console.log(smallestStringWithSwaps("dcab", [[0, 3], [1, 2]])); // "bacd"
console.log(smallestStringWithSwaps("cba", [[0, 1], [1, 2]]));  // "abc"
console.log(smallestStringWithSwaps("cbfdae", [[0, 1], [3, 2], [5, 2], [1, 4]])); //
"abdecf"
```

Let's dry run the `smallestStringWithSwaps` function with the input string `s = "dcab"` and pairs `[[0,3],[1,2]]`.

Initialization:

```
const parent: number[] = [];
for (let i = 0; i < s.length; i++) {
    parent[i] = i;
}
```

This initializes each character's index as its own parent: `parent = [0, 1, 2, 3]`

Processing the pairs using union:

1. For the pair `[0,3]`:
    - The union function merges the sets containing `0` and `3`.
    - `parent` becomes: `[3, 1, 2, 3]`
2. For the pair `[1,2]`:
    - The union function merges the sets containing `1` and `2`.
    - `parent` becomes: `[3, 2, 2, 3]`

Create mapping of root to its indices: We want to know which indices belong to which set.

```
const indexGroups: { [key: number]: number[] } = {};
for (let i = 0; i < s.length; i++) {
    const root = find(i);
    if (!indexGroups[root]) {
        indexGroups[root] = [];
    }
    indexGroups[root].push(i);
}
```

After this loop, `indexGroups` becomes:

```
{
    3: [0, 3],
    2: [1, 2]
}
```

This means characters at indices `0` and `3` can be swapped with each other, and characters at indices `1` and `2` can be swapped with each other.

Reorder characters within each set: We want the characters in each group to be sorted in ascending order to make the string lexicographically smallest.

```
let chars: string[] = s.split('');
for (let indices of Object.values(indexGroups)) {
    const sortedChars = indices.map(i => chars[i]).sort();
    for (let i = 0; i < indices.length; i++) {
        chars[indices[i]] = sortedChars[i];
    }
}
```

1. For `indices = [0, 3]` (i.e., characters `'d'` and `'b'`):
    - Sort them to get `'b'` and `'d'`.
    - Update the characters in `chars` to reflect the new order: `chars` becomes `['b', 'c', 'a', 'd']`.
2. For `indices = [1, 2]` (i.e., characters `'c'` and `'a'`):
    - Sort them to get `'a'` and `'c'`.
    - Update the characters in `chars` to reflect the new order: `chars` becomes `['b', 'a', 'c', 'd']`.

Return the modified string: The final step is to join the `chars` array:

```
return chars.join('');
```

This returns the string `"bacd"`.

So, for the input `s = "dcab"` and pairs `[[0,3],[1,2]]`, the function `smallestStringWithSwaps` returns the string `"bacd"`.

Monotonic Stack

A monotonic stack is a type of stack that maintains a certain monotonicity property as elements are pushed into it. Depending on the problem at hand, this might be a strictly increasing or strictly decreasing stack. In other words, when you push a new element onto the stack, the stack ensures that its elements are still in either non-decreasing or non-increasing order by potentially popping off the elements that violate the monotonicity.

The primary purpose of a monotonic stack is to efficiently answer questions like:

- For each element in an array, what's the nearest smaller (or larger) element on the left or right of it?
- Finding the maximum rectangle in a histogram.

Let's dive into an example:

Problem Statement: Given an array of numbers, for each element, find the first larger number to its right.

Example: Input: `[4, 3, 2, 5, 1]` Output: `[5, 5, 5, -1, -1]`

Explanation: For `4`, the next larger number is `5`. For `3`, the next larger number is `5`. For `2`, the next larger number is `5`. For `5`, there is no larger number, hence `-1`. For `1`, there is no larger number, hence `-1`.

Typescript Solution using Monotonic Stack:

```
function nextLargerElement(nums: number[]): number[] {
    const result: number[] = Array(nums.length).fill(-1); // initialize result array
with -1s
    const stack: number[] = []; // this will store indices

    for (let i = 0; i < nums.length; i++) {
        while (stack.length && nums[i] > nums[stack[stack.length - 1]]) {
            const idx = stack.pop() as number; // get the last index from the stack
            result[idx] = nums[i]; // we found the next greater element for the
element at index `idx`
        }
        stack.push(i); // push the current index to the stack
    }

    return result;
}

// Testing the function
const input = [4, 3, 2, 5, 1];
console.log(nextLargerElement(input)); // Expected output: [5, 5, 5, -1, -1]
```

In this solution, we're keeping the stack in non-decreasing order. When a larger number is found, we keep popping from the stack until we find a number that's greater than the current one or the stack becomes empty. This helps in finding the next larger number for all the numbers that are smaller than the current number.

Peak Valley (Arrays)

You are given an integer array prices where prices[i] is the price of a given stock on the ith day.

On each day, you may decide to buy and/or sell the stock. You can only hold at most one share of the stock at any time. However, you can buy it then immediately sell it on the same day.

Find and return the maximum profit you can achieve.

```
Input: prices = [7,1,5,3,6,4]
Output: 7
Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4.
Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.
Total profit is 4 + 3 = 7.
```

```
class Solution {
    public int maxProfit(int[] prices) {
        int i = 0;
        int valley = prices[0];
        int peak = prices[0];
        int maxprofit = 0;
        while (i < prices.length - 1) {
            while (i < prices.length - 1 && prices[i] >= prices[i + 1])
```

```
                i++;
            valley = prices[i];
            while (i < prices.length - 1 && prices[i] <= prices[i + 1])
                i++;
            peak = prices[i];
            maxprofit += peak - valley;
        }
        return maxprofit;
    }
}
```

Dijkstra's algorithm

Explanation:

Dijkstra's algorithm is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path costs, producing the shortest path tree. This algorithm is often used in routing and as a subroutine in other graph algorithms.

Here are the basic steps:

1. **Initialization**:
    - Set a tentative distance value for every node: set the initial node's distance to zero and all other nodes' distance to infinity.
    - Set the initial node as the current node.
    - Mark all nodes as unvisited. Create a set of all the unvisited nodes.
2. **Main loop**:
    - For the current node, consider all of its neighbors and calculate their tentative distances through the current node. Compare the newly calculated tentative distance to the current assigned value and update the distance if the new value is smaller.
    - Once you have considered all of the neighbors of the current node, mark the current node as visited. A visited node will not be checked again.
    - Select the unvisited node with the smallest tentative distance, and set it as the new current node. Then go back to the previous step.
3. **Completion**:
    - The algorithm ends when every node has been visited. The algorithm has now constructed the shortest path tree from the source node to all other nodes.

TypeScript Implementation:

```
type Graph = {
    [key: string]: { [key: string]: number };
};

const dijkstra = (graph: Graph, start: string): { distances: { [key: string]: number
}, previous: { [key: string]: string | null } } => {
    let distances: { [key: string]: number } = {};
```

```
    let previous: { [key: string]: string | null } = {};
    let unvisitedNodes = Object.keys(graph);

    for (let node of unvisitedNodes) {
        distances[node] = Infinity;
        previous[node] = null;
    }
    distances[start] = 0;

    while (unvisitedNodes.length !== 0) {
        unvisitedNodes.sort((a, b) => distances[a] - distances[b]);
        let currentNode = unvisitedNodes.shift() as string;

        for (let neighbor in graph[currentNode]) {
            let newDistance = distances[currentNode] + graph[currentNode][neighbor];

            if (newDistance < distances[neighbor]) {
                distances[neighbor] = newDistance;
                previous[neighbor] = currentNode;
            }
        }
    }
    return { distances, previous };
};

// Example Usage:
const exampleGraph: Graph = {
    A: { B: 1, D: 3 },
    B: { A: 1, D: 2, E: 5 },
    D: { A: 3, B: 2, E: 1 },
    E: { B: 5, D: 1 },
};

let result = dijkstra(exampleGraph, "A");
console.log(result.distances);
console.log(result.previous);
```

Dry Run:

Let's dry run the algorithm using the `exampleGraph`:

1. **Initialization**:
    - `distances = { A: 0, B: Infinity, D: Infinity, E: Infinity }`
    - `previous = { A: null, B: null, D: null, E: null }`
    - Current node = `A`
    - Unvisited nodes = `A, B, D, E`
2. **First Iteration**:
    - Current Node: `A`
    - Neighbors: `B` and `D`
        - For `B`: New distance = `0 + 1 = 1`, which is less than `Infinity`
            - `distances[B] = 1`, `previous[B] = A`

        - For `D`: New distance = `0 + 3 = 3`, which is less than `Infinity`
            - `distances[D] = 3`, `previous[D] = A`
    - Mark `A` as visited.
    - New current node = `B` (smallest distance among unvisited nodes)
3. **Second Iteration**:
    - Current Node: `B`
    - Neighbors: `A`, `D`, and `E`
        - For `D`: New distance = `1 + 2 = 3`, which is not less than current `3`
        - For `E`: New distance = `1 + 5 = 6`, which is less than `Infinity`
            - `distances[E] = 6`, `previous[E] = B`
    - Mark `B` as visited.
    - New current node = `D` (smallest distance among unvisited nodes)
4. **Third Iteration**:
    - Current Node: `D`
    - Neighbors: `A`, `B`, and `E`
        - For `E`: New distance = `3 + 1 = 4`, which is less than current `6`
            - `distances[E] = 4`, `previous[E] = D`
    - Mark `D` as visited.
    - New current node = `E` (last unvisited node)
5. **Fourth Iteration**:
    - Current Node: `E`
    - No updates since all neighbors have been visited.
    - Mark `E` as visited.

Completion: `distances` = { A: 0, B: 1, D: 3, E: 4 } `previous` = { A: null, B: 'A', D: 'A', E: 'D' }

The shortest path from A to E is `A -> B -> D -> E` with a distance of `4`.

Time and Space Complexity:

Dijkstra's algorithm's time and space complexity are primarily influenced by the data structures used to implement it. The pseudocode provided earlier uses a basic array for the unvisited set and iterates over it to find the node with the smallest distance, which isn't the most efficient approach. Let's break down the complexities for the provided TypeScript implementation and then discuss how it can be optimized.

For the provided TypeScript implementation:

Time Complexity:

1. Initializing `distances`, `previous`, and `unvisitedNodes` is (O(V)), where (V) is the number of vertices.
2. In the worst-case scenario, the `while` loop runs for every node, i.e., (O(V)).
3. Inside the `while` loop, the `sort` operation is (O(V \log V)).
4. Additionally, inside the `while` loop, we might go through all the edges of a node in the nested `for` loop, i.e., (O(E)) where (E) is the number of edges.

Multiplying these together, the worst-case time complexity is: [O(V \times (V \log V + E))] This can be approximated as (O(V^2 \log V)) in dense graphs where (E \approx V^2) and (O(V^2)) in sparse graphs.

Space Complexity:

1. The `distances` and `previous` maps have space complexity (O(V)).
2. The `unvisitedNodes` array also has space complexity (O(V)).

Summing these up, the overall space complexity is: [O(V)]

Optimized Version using Priority Queue:

You can optimize the time complexity by using a priority queue (or a binary heap) to manage the unvisited nodes. This would allow you to efficiently extract the node with the smallest tentative distance without having to sort the entire set of unvisited nodes each time.

With this optimization:

Time Complexity:

1. Initialization is still (O(V)).
2. The loop runs for every node and edge, i.e., (O(V + E)).
3. Extracting the minimum node from a priority queue is (O(\log V)).

Multiplying these together, the worst-case time complexity becomes: [O((V + E) \log V)]

For a dense graph, this still reduces to (O(V^2 \log V)), but the constant factors are typically much better than the array-based approach. For sparse graphs, this is much faster, at (O(V \log V)).

Space Complexity: The space complexity remains largely unchanged at (O(V)) for the data structures in the algorithm. However, if you include the priority queue's internal structures, it can go up slightly but remains within (O(V)) bounds.

In summary, while the naive version can be quite slow for large graphs, using a priority queue can significantly speed up Dijkstra's algorithm.

Sorting

Quick Sort

Quick Sort is a divide-and-conquer algorithm that works on the principle of choosing a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

1. **Choose a Pivot**: Select an element from the array as the pivot. This can be done in various ways, such as choosing the first element, the last element, the middle element, or even a random element.

2. **Partitioning**: Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it. After this partitioning, the pivot is in its final position.

3. **Recursive Sorting**: Recursively apply the above steps to the sub-array of elements with smaller values and the sub-array of elements with greater values.
4. **Base Case**: The recursion base case is an array with zero or one element, which doesn't need to be sorted.

Now, here's an example of Quick Sort implemented in TypeScript:

```typescript
function quickSort(arr: number[], low: number, high: number): void {
  if (low < high) {
    // Partition the array and get the pivot index
    let pi = partition(arr, low, high);

    // Recursively sort the elements before partition and after partition
    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
  }
}

function partition(arr: number[], low: number, high: number): number {
  // Choose the rightmost element as pivot
  let pivot = arr[high];

  // Pointer for greater element
  let i = low - 1;

  // Traverse through all elements
  // compare each element with pivot
  for (let j = low; j < high; j++) {
    if (arr[j] < pivot) {
      // If element smaller than pivot is found
      // swap it with the greater element pointed by i
      i++;

      // Swapping element at i with element at j
      [arr[i], arr[j]] = [arr[j], arr[i]];
    }
  }

  // Swap the pivot element with the greater element specified by i
  [arr[i + 1], arr[high]] = [arr[high], arr[i + 1]];

  // Return the position from where partition is done
  return i + 1;
}

// Helper function to initiate QuickSort
function quickSortHelper(arr: number[]): number[] {
  quickSort(arr, 0, arr.length - 1);
  return arr;
}

// Example usage:
```

```
const arr = [10, 7, 8, 9, 1, 5];
console.log('Original Array:', arr);
const sortedArray = quickSortHelper(arr);
console.log('Sorted Array:', sortedArray);
```