

Algorithms Interview Questions and Answers

Intern to senior level

Coding interview problems with solutions explained step by step: two sum, palindrome check and flattening nested arrays, with time and space complexity and alternative approaches.

3 questions Read online: <https://www.fredevtemplates.com/interview-questions/algorithms>

1. How do you check if a string is a palindrome in JavaScript?

JavaScript Easy Intern, Junior

Normalise the string (lower case, remove non-alphanumeric characters), then compare characters from both ends moving inward with two pointers. This is $O(n)$ time and $O(1)$ extra space beyond the normalised copy.

A palindrome reads the same forwards and backwards. Interviewers usually mean a "valid palindrome" that ignores case, spaces and punctuation, so ask before coding.

The shortest solution is `s === s.split('').reverse().join('')`, which is fine to mention but builds two extra strings. two-pointer approach shows you understand the problem: compare the first and last characters, move inward, and stop at the first mismatch.

Problem: Given a string `s`, return true if it is a palindrome after converting all letters to lower case and removing all non-alphanumeric characters.

Input: `s = "A man, a plan, a canal: Panama"` -> Output: true

```
function isPalindrome(s) {
  const clean = s.toLowerCase().replace(/[^a-z0-9]/g, '');
  let left = 0;
  let right = clean.length - 1;

  while (left < right) {
    if (clean[left] !== clean[right]) return false;
    left++;
    right--;
  }
  return true;
}

isPalindrome('A man, a plan, a canal: Panama'); // true
isPalindrome('race a car'); // false
```

Time: $O(n)$: every character is processed a constant number of times.

Space: $O(n)$ for the cleaned copy. Skipping invalid characters in place with the pointers makes it $O(1)$.

2. How do you solve the Two Sum problem in JavaScript?

JavaScript Easy Junior, Mid-level

Walk through the array once while storing each number's index in a hash map; for every number, check whether target minus that number is already in the map. This runs in $O(n)$ time and $O(n)$ space.

The brute-force solution checks every pair with two nested loops. It is correct but takes $O(n^2)$ time, which is too slow for large inputs.

The key observation is that for each number `x` you are looking for one specific partner: `target - x`. A Map lets you check whether that partner has already been seen in constant time, so a single pass is enough.

Check for the complement before inserting the current number, otherwise an element could be paired with itself

(for example target 6 and value 3 at one index).

Problem: Given an array of integers `nums` and an integer `target`, return the indices of the two numbers that add up to `target`. Assume exactly one solution exists and you may not use the same element twice.

Input: `nums = [2, 7, 11, 15]`, `target = 9` -> Output: `[0, 1]`

```
function twoSum(nums, target) {
  const seen = new Map(); // value -> index

  for (let i = 0; i < nums.length; i++) {
    const complement = target - nums[i];
    if (seen.has(complement)) {
      return [seen.get(complement), i];
    }
    seen.set(nums[i], i);
  }

  return []; // unreachable when a solution is guaranteed
}

twoSum([2, 7, 11, 15], 9); // [0, 1]
```

Time: $O(n)$: each element is visited once and map operations are $O(1)$ on average.

Space: $O(n)$: in the worst case the map stores every element.

3. How do you flatten a nested array in JavaScript without using `flat()`?

JavaScript Medium Junior, Mid-level

Recursively walk the array: when an element is an array, flatten it and append the result; otherwise append the element. An iterative version uses an explicit stack to avoid recursion limits.

In real code `arr.flat(Infinity)` does this, and you should say so. Interviewers ask you to write it yourself to see recursion, `Array.isArray` and how you reason about depth.

The recursive solution mirrors the structure of the data and is easy to explain. For very deep nesting it can hit the call-stack limit, so mentioning an iterative stack-based version is a good follow-up.

Problem: Write a function that takes an array that may contain nested arrays to any depth and returns a new, single-level array with all the values in their original order.

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

```
function flatten(arr) {
  const result = [];
  for (const item of arr) {
    if (Array.isArray(item)) {
      result.push(...flatten(item));
    } else {
      result.push(item);
    }
  }
  return result;
}

flatten([1, [2, [3, [4]], 5]]); // [1, 2, 3, 4, 5]
```

Time: $O(n)$, where n is the total number of values across all nesting levels.

Space: $O(n)$ for the output plus $O(d)$ call-stack depth, where d is the maximum nesting depth.

These answers are written for real interview conversations. Prefer explaining a trade-off out loud over reciting a definition. Full explanations and related questions: <https://www.fredevtemplates.com/interview-questions/algorithms>