ChatGPT

Deep Dive: Arrays & Hashing Interview Problems

Arrays are one of the most fundamental data structures, and when combined with hashing techniques (using hash sets or hash maps), they unlock powerful capabilities for efficient lookups, duplicate detection, and frequency counting [1]. In coding interviews for top tech companies, **"Arrays & Hashing"** problems are extremely common. Mastering this category will provide a strong foundation for tackling a wide range of algorithmic challenges. In this paper, we will explore nine key problems in this category, breaking down the concepts, solutions, and tips for each. We'll approach each problem as a senior software engineer would – discussing intuitive solutions, optimal approaches, things to memorize, and real-world analogies where appropriate. By the end, you'll understand how to:

- Use sets to quickly detect duplicates and validate constraints [2].
- Leverage sorting and hash maps to solve problems like anagrams efficiently [2].
- Apply hash map lookups for classic problems such as Two Sum [3].
- Use frequency maps and heaps to identify the most common elements [4].
- Design custom encoding schemes for string serialization/deserialization [5].
- Implement prefix/suffix product techniques to avoid division in array products [6].
- Validate complex structures (e.g. Sudoku boards) using hashing to track seen values [7].
- Find sequences (like longest consecutive run) in **O(n)** time using hash sets [8].

Each section below focuses on one problem from the NeetCode "Arrays & Hashing" list, providing an in-depth explanation, solution approach in JavaScript, important observations, and interview tips.

1. Contains Duplicate (LeetCode 217)

Problem: Given an integer array, determine if any value appears at least twice. Return `true` if any element is a duplicate, or `false` if all elements are distinct. This is a straightforward question, but it tests your understanding of time-space trade-offs for searching and storing data.

Discussion: A naive approach would compare every pair of elements (nested loops), which is **$O(n^2)$** time – too slow for large arrays. A more intuitive approach is to sort the array first. After sorting, any duplicates would appear next to each other, so a single linear scan can check adjacent pairs for equality. The sorting method takes **O(n log n)** time and **O(1)** extra space (if sorting in place) [9] [10]. This is acceptable for moderate input sizes and is space-efficient, but not optimal in time for very large inputs.

The optimal solution uses a hash set to track seen numbers as we iterate through the array. For each number, we check if it's already in the set: - If **yes**, we have found a duplicate and can return `true` immediately. - If **no**, add the number to the set and continue.
If we finish the loop without finding any duplicate, return `false`. This approach runs in **O(n)** time on average, since both lookup and insertion in a hash set are amortized constant time [11] [12]. The trade-off is an extra **O(n)** space cost for the set.

Why this works: Hashing provides constant-time membership checks, which is ideal for detecting duplicates quickly. In an interview, you should mention the time-space trade-off: - Using a set: faster (linear time) but uses extra memory. - Sorting first: slower (n log n) but uses little additional memory [13] .

JavaScript Solution (Using a Set):

```
function containsDuplicate(nums) {
  const seen = new Set();
  for (let num of nums) {
    if (seen.has(num)) {
      return true; // Duplicate found
    }
    seen.add(num);
  }
  return false;
}
```

This solution iterates once through the array and uses a `Set` for constant-time lookups. In an interview, you might also mention edge cases: an empty or single-element array (trivially no duplicates), which the code naturally handles by returning false.

Interview Tips: This problem is often one of the first in an interview or on a coding platform because it is simple yet illuminates the candidate's understanding of basic data structures. Make sure to clarify the expected input size (if extremely large, an **O(n log n)** sort might be borderline, so the **O(n)** approach is preferred). Also consider mentioning that in a memory-constrained environment, you *could* sort instead of using a set [14] , showing awareness of trade-offs. However, in most cases, interviewers expect the hash set solution. Recognize related problems: e.g., "Contains Duplicate II/III" involve similar ideas with additional constraints (like checking duplicates within a window). This family of problems reinforces the habit of using hash sets or hash maps for membership tests.

2. Valid Anagram (LeetCode 242)

Problem: Given two strings, determine if one is an anagram of the other. Two strings are anagrams if they contain the same characters in the same frequency, but possibly in a different order. For example, **"listen"** and **"silent"** are anagrams – both consist of the letters **{l, i, s, t, e, n}** just arranged differently [15] .

Discussion: This problem tests if you can effectively count and compare character frequencies. A real-world analogy is checking if two jumbled words contain the same multiset of letters, which has applications in spell-checkers or word games [15] . There are a couple of common approaches: - **Sorting:** Sort both strings alphabetically and compare the results. If the sorted strings are identical, then the originals are anagrams [16] . For example, sorting `"listen"` and `"silent"` each yields `"eilnst"`, which are equal. Sorting takes **O(n log n)** time (where *n* is the length of the string) and uses O(n) space for the sorted copies. - **Hash Map / Frequency Count:** Count the occurrences of each character in the first string (using a hash map or an array of size 26 for English lowercase letters), then decrement the counts using the second string. If all counts end at zero and no mismatches are found, the strings are anagrams. This method runs in **O(n)** time

and uses **O(n)** space (or O(1) space if using a fixed-size 26-element array for letters). It's often the preferred approach in interviews because it's linear and straightforward to implement.

Using the sorting method is perfectly acceptable for an interview, especially if you clarify its complexity. In fact, the sorted solution is succinct and easy to remember:

```
function isAnagram(s, t) {
  if (s.length !== t.length) return false;
  const sortedS = s.split('').sort().join('');
  const sortedT = t.split('').sort().join('');
  return sortedS === sortedT;
}
```

This code sorts the characters of `s` and `t` and then checks for equality. The early length check is a quick win – if the strings differ in length, they can't be anagrams.

For a more optimal linear solution, you can use a character count:

```
function isAnagram(s, t) {
  if (s.length !== t.length) return false;
  const count = {}; // hash map for char counts
  for (let char of s) {
    count[char] = (count[char] || 0) + 1;
  }
  for (let char of t) {
    if (!count[char]) {
      return false; // char in t not in s or count mismatch
    }
    count[char]--;
  }
  // If any count is non-zero, not an anagram
  return Object.values(count).every(val => val === 0);
}
```

This second approach uses a hash map `count` to tally letters in `s` and then ensures `t` has exactly the same tallies. It runs in O(n) time. In interviews, either approach is usually fine – the key is to demonstrate understanding of what an anagram means and how to efficiently compare frequencies.

Interview Tips: Be prepared to discuss why sorting works (anagrams have identical sorted order) and its complexity vs. the counting method. A common mistake is forgetting to check lengths first – mention that upfront. If using a hash map, ensure you handle characters that appear in one string but not the other. Also, consider Unicode or extended character sets if the problem isn't limited to lowercase letters. The Python `Counter` class or JavaScript `Map` could be mentioned as alternatives, but a basic object or array works well. Interviewers may ask about space usage: using an array of size 26 (for English letters) is technically

O(1) additional space, which is very efficient [17]. This problem reinforces the concept of hashing for counting frequencies and comparing distributions of elements.

3. Two Sum (LeetCode 1)

Problem: Given an array of integers and a target value, find indices of two numbers such that they add up to the target. You may assume exactly one solution exists, and you cannot use the same element twice. This is one of the most famous interview problems – in fact, Two Sum is known to be *extremely* common in interviews [18].

Discussion: A brute-force solution is to check every pair of numbers to see if they sum to the target. This requires a double loop (i and j), hence **O(n^2)** time. While easy to write, it's inefficient. We can do better by using a hash map to find complements in one pass.

The **hash map approach** uses a map to store numbers we have seen so far and their indices. As we iterate through the array with index `i` and value `num`: - Compute the **complement** as `target - num`. - Check if this complement is already in the hash map (i.e., we have seen a number that, together with the current `num`, adds up to the target). - If yes, we've found our two numbers – return their indices. - If no, add the current number and its index to the hash map and continue.

This way, we find the solution in one pass (linear time). The hash table lookup for the complement is **O(1)** on average, making the overall complexity **O(n)** [19] [20]. The space complexity is O(n) for the hash map (in the worst case, if no solution is found until the last element).

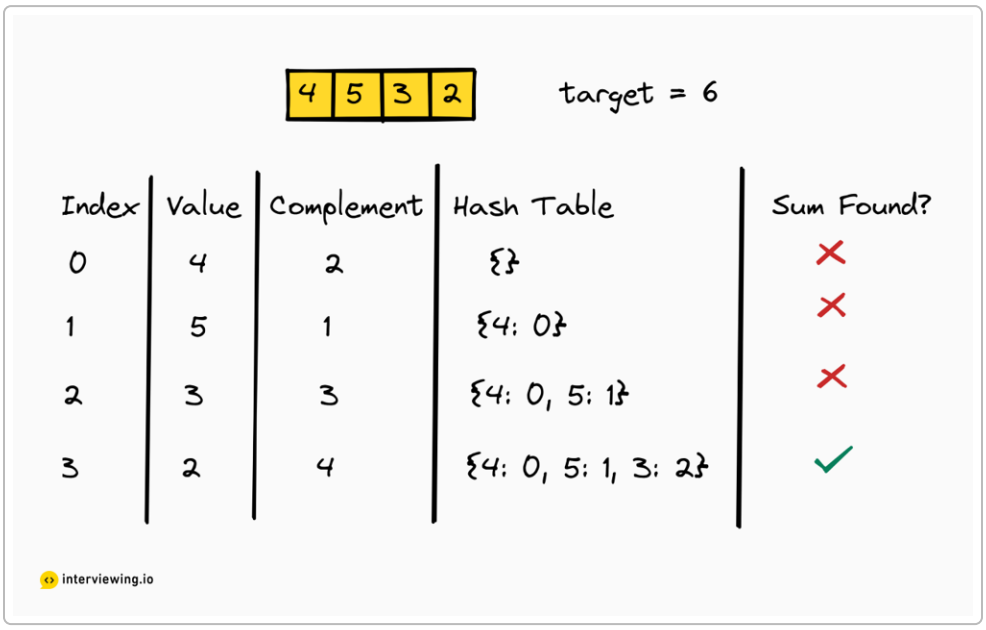

Figure: Using a hash map to solve Two Sum. We iterate through the array, and for each value we check if its complement (target minus the value) is already in the map. The diagram illustrates an example array and target, showing the state of the hash table at each step and when a matching pair is found. Each entry in the map stores a number from the array and its index [21] [22].

In JavaScript, the implementation looks like this:

```
function twoSum(nums, target) {
  const indexMap = new Map(); // number -> index
  for (let i = 0; i < nums.length; i++) {
    const num = nums[i];
    const complement = target - num;
    if (indexMap.has(complement)) {
      return [indexMap.get(complement), i];
    }
    indexMap.set(num, i);
  }
  // Assuming exactly one solution, we won't reach here normally.
}
```

As we loop, we immediately check if the complement of the current number exists from a previous iteration. If so, we've solved it. If not, we record the current number in the map and proceed. This one-pass approach is very efficient.

Why this works: By storing seen numbers in a hash map, we avoid the need for a nested loop. We essentially trade space for time, which is a common theme in using hashing. The map lets us answer "have we seen a number such that number + current = target?" in constant time.

Interview Tips: Two Sum is considered a must-know pattern. Interviewers often use it to test fundamental understanding of hash maps. Be sure to clarify that you understand the difference between returning indices vs. values (the problem usually asks for indices). Also, mention edge cases: if no solution exists (though by problem definition one does), or if the array has negative numbers or zero – the algorithm still works fine. If the input array were sorted, you could also mention the two-pointer approach (which is O(n) without extra space) as an alternative, although for unsorted arrays the hash map is the go-to solution. Recognize that this pattern can extend to variations like 3-sum or 4-sum (which typically require more complex solutions). Two Sum's simplicity is why it's often one of the first problems in interview prep. Showing the ability to derive the O(n) solution and explaining the intuition (find complements using a lookup table) will demonstrate your grasp of array/hash techniques.

4. Group Anagrams (LeetCode 49)

Problem: Given an array of strings, group the anagrams together. An anagram group is a set of words that are all anagrams of each other. For example, given `["eat", "tea", "tan", "ate", "nat", "bat"]`, the output should be groups like `["eat","tea","ate"]`, `["tan","nat"]`, and `["bat"]` (order of groups or order of strings in groups doesn't matter).

Discussion: This problem extends the anagram concept from Problem 2 to multiple strings. It's essentially asking us to categorize strings by their character composition. A real-world scenario for this might be organizing a jumble or scrabble helper, where you group words made of the same letters, or in computational biology grouping DNA sequences with the same nucleotide counts.

The key insight is that **anagrams share a common signature**. We need to choose a representation of that signature and use it to group words: - A simple and popular signature is the **sorted characters**. If you sort the letters of each word alphabetically, all anagrams will result in the same sorted string [23]. For example, `"eat"`, `"tea"`, and `"ate"` all sort to `"aet"`, which can be used as a hash key. - Another signature (that avoids sorting cost) is the **character frequency count**. For each word, count the occurrences of each letter (often 26 letters for simplicity if we assume lowercase a-z). This count can be represented as a tuple or string (like `"a1e1t1"` meaning 1 of a,1 of e,1 of t). All anagrams will have identical frequency representations [17]. This method is O(m) for a word of length m, and forming the key is O(m) instead of O(m log m) for sorting.

The general approach is: 1. Initialize a hash map (dictionary) to collect groups. 2. For each word in the input list: - Compute its **signature** (sorted word or frequency key). - Use the signature as a key in the map, and append this word to the list of words for that signature. 3. After processing all words, the values of the map are our groups of anagrams.

Using sorted strings as keys is straightforward in code:

```
function groupAnagrams(strs) {
  const groups = {};
  for (let word of strs) {
    // Sort the word to get the signature
    const sorted = word.split('').sort().join('');
    if (!groups[sorted]) {
      groups[sorted] = [];
    }
    groups[sorted].push(word);
  }
  // return grouped anagrams as an array of arrays
  return Object.values(groups);
}
```

In this JavaScript solution, we sort each word (which is O(m log m) for word length m) and use it to bucket the words. The overall complexity is O(n * m log m) for n words of average length m. This is efficient for reasonable word sizes. If m is small relative to n (e.g., short strings but many of them), the cost is dominated by n.

If we wanted to optimize further, we could implement the frequency count key. However, that introduces more complexity in code (building a 26-length count array and converting to string for each word). In an interview, it's fine to discuss this optimization verbally: mention that using a frequency tuple key yields O(n * m) time, which for very large n might be slightly better. But often, the sorted approach is acceptable and easier to get correct under pressure.

Example walk-through: Using the input `["eat","tea","tan","ate","nat","bat"]`: - We process `"eat"` -> sorted key `"aet"` -> groups = { "aet": ["eat"] }. - `"tea"` -> sorted key `"aet"` -> groups = { "aet": ["eat","tea"] }. - `"tan"` -> sorted key `"ant"` -> groups adds "ant": { "aet": [...], "ant": ["tan"] }. - `"ate"` -> key `"aet"` -> groups["aet"] = ["eat","tea","ate"]. - `"nat"` -> key `"ant"` -> groups["ant"] =

["tan","nat"]. - `"bat"` -> key `"abt"` -> groups["abt"] = ["bat"].
Final groups (values of the map): `[["eat","tea","ate"], ["tan","nat"], ["bat"]]`, which matches expectation.

Interview Tips: Make sure you clarify what an anagram is, and note that sorting as a solution leverages that definition directly. Also, mention that we ignore things like case or non-letter characters if not specified (usually the problem statement defines the input clearly, e.g., all lowercase). You should mention the complexity: using a hash map data structure leads to an average O(1) insertion for each word's group, so the main cost is computing the key. If the interviewer asks for optimizations, discuss the frequency count approach or even prime number multiplication (an interesting but less common approach where you map each letter to a prime number and use the product as a hash key – but watch out for overflow). Real-world applications include grouping similar items or detecting anagrams quickly in a large dictionary (useful in word puzzles or language processing). Since this problem is a **Medium** difficulty, interviewers expect a solid explanation of the approach. Don't forget to return the groups in whatever format is required (list of lists). Finally, if order doesn't matter in output, don't spend time trying to sort the groups; focus on correctness and efficiency of grouping.

5. Top K Frequent Elements (LeetCode 347)

Problem: Given an array of integers, return the **k** most frequent elements. For example, in `nums = [1,1,1,2,2,3]` with `k = 2`, the answer should be `[1,2]` because 1 appears 3 times and 2 appears 2 times (the two highest frequencies). This problem is essentially asking for a frequency analysis and then extracting the top k categories.

Discussion: This type of problem is very useful in data analysis for identifying trends [24]. Think of real-world scenarios: - Finding the top k most common search queries on a search engine (to see what's trending). - Finding the most frequent log entries or error codes in a server log. - Identifying popular tags or hashtags in a social media dataset. The pattern of finding "top k" frequent items appears in many contexts, so interviewers like to test your ability to use data structures like hash maps, heaps, or sorting to solve it.

A straightforward approach is: 1. Count the frequency of each element using a hash map (or object in JS). This gives us a map of element -> count. 2. We then need to retrieve the k keys with highest counts. Several ways to do this: - **Sorting:** Create an array of the unique elements and sort it by frequency in descending order, then take the first k. Sorting the unique elements array takes O(u log u) where u is the number of unique elements ($u \leq n$). In worst case (all elements unique, u = n), this is O(n log n). For example, after counting frequencies, we might sort the entries by count [25]. - **Heap (priority queue):** Use a min-heap of size k to keep track of the top k frequencies. Iterate through the frequency map, push each entry into the min-heap, and if the heap size exceeds k, pop the smallest frequency. This ensures the heap only stores the k largest frequencies at any time [26] [27]. Building this is O(u log k). This approach is more optimal if n is very large and k is small, because log k can be much smaller than log n. - **Bucket sort:** Since frequency ranges from 1 to n, you can create an array of buckets where index i holds a list of numbers that appear i times. Then iterate from the highest frequency bucket down until you collect k results. This is O(n) time and space. It's a clever approach that avoids explicit sorting and is often used in this problem.

To keep things simple, many candidates (and solutions) use sorting because it's easy to implement and typically fast enough for interview-size inputs. Let's implement using sorting in JavaScript:

```
function topKFrequent(nums, k) {
  const freq = {};
  for (let num of nums) {
    freq[num] = (freq[num] || 0) + 1;
  }
  // Sort unique elements by frequency
  const uniqueElements = Object.keys(freq);
  uniqueElements.sort((a, b) => freq[b] - freq[a]);
  // Get the top k elements, converting keys back to numbers
  return uniqueElements.slice(0, k).map(Number);
}
```

Here we first populate `freq` with counts. Then `Object.keys(freq)` gives us the array of distinct numbers (as strings, hence the `map(Number)` at the end to convert back). We sort those keys by their frequency descending. Finally, take the first k. The complexity is O(n log n) in the worst case (when unique count ~ n).

If we wanted to demonstrate the heap method (to show we can optimize if needed), we would: - Use a min-heap (size k) to store [number, frequency] pairs ordered by frequency. - Iterate over `freq` entries, push each into heap; if heap size > k, remove the smallest freq element. - At the end, extract all elements from heap – those are the top k. (JavaScript doesn't have a built-in heap, but one can be implemented or simulated with an array and sorting operations, or use a library/utility if allowed).

Example: Consider `nums = [1,1,2,2,2,3]`, k = 2. Frequency map: {1:2, 2:3, 3:1}. Sorted unique elements by freq gives [2,1,3]. The top 2 are [2,1]. This matches what we expect (2 appears 3 times, 1 appears 2 times). Order of output doesn't matter unless specified, so [1,2] would also be fine in this case.

Real-world analogy: The problem description can be related to something like, *"If you have a basket of fruits and you want to know which fruits you have the most of, how would you figure it out?"* You'd count each fruit type, then pick the top k counts. In fact, imagine you have a fruit basket with apples, oranges, bananas, etc., and you're asked to only bring the two fruit varieties of which you have the most pieces [28]. You'd tally each type and then pick the largest two tallies. This is exactly what the algorithm does with numbers and frequencies.

Interview Tips: Clarify that you will first count frequencies – this is crucial and straightforward. Then state the approach for getting the top k. If you choose sorting, mention the complexity and that it's often fine given problem constraints. If you mention the heap approach, you might impress by noting its complexity is O(n log k), which for large n and small k is better. However, building a heap in JavaScript is not trivial without writing a lot of code; in an interview you could pseudo-code it or explain conceptually. Always consider k relative to n: - If k is very small compared to n, a heap or partial selection algorithm is beneficial. - If k is near n, sorting is practically as good as anything. It's good to mention that Python, for instance, has a `heapq` or `nlargest` function, and Java has a PriorityQueue, but in JS you'd manage it manually or use sort. If asked for further optimization and if the value range is manageable, talk about bucket sort as a linear time solution. This shows you're aware of multiple techniques. Finally, ensure you return exactly k elements. Off-by-one errors (like taking k+1 elements or something) would be a gotcha to avoid.

6. Encode and Decode Strings (LeetCode 271)

Problem: Design an algorithm to encode a list of strings into a single string, and another algorithm to decode that single string back into the original list of strings. This problem often appears in interviews (especially for companies like Google, Bloomberg, etc.) to test your understanding of string serialization. The twist is that the list of strings may contain any characters, including special characters or even numbers, so you cannot just use a simple delimiter like comma or pipe – those characters might appear in the strings themselves. For example, how do you encode `["hello", "world"]` into a single string? And how do you handle decoding it back?

Discussion: A real-world application of this problem is in designing network protocols or file formats where you need to send arbitrary strings in a single message. It's essentially testing if you know how to implement **serialization and deserialization** of data safely [29]. The key challenge is to ensure the encoding is unambiguous – the decoder should know exactly where one string ends and the next begins, even if the strings contain special characters.

A common and robust solution is **length-prefix encoding** [30] : - For each string, prefix it with its length and a special delimiter. Typically, we use a non-numeric character as delimiter (for example `'#'`) because the length is numeric. - The encoded format for a string `s` becomes `"<length>#<s>"`.
For example, for `["neet","code"]`, we would produce `"4#neet4#code"` – here `"4#neet"` represents the first string (length 4, then the string), and `"4#code"` represents the second [31]. Concatenated together, it's unambiguous because we know to read the length first, then the exact number of characters for the string.

Encoding step: Iterate through the list of strings and build the encoded string:

```
function encode(strs) {
  let encoded = "";
  for (let s of strs) {
    encoded += s.length + "#" + s;
  }
  return encoded;
}
```

If `strs = ["neet","code"]`, this will produce `"4#neet4#code"` as described. If a string is empty, its length is 0, so it would contribute `"0#"` to the encoded form (with no characters after the `#` for that string).

Decoding step: We need to scan the encoded string and extract the original list:

```
function decode(s) {
  const result = [];
  let i = 0;
  while (i < s.length) {
```

```
    // find the position of delimiter '#'
    const j = s.indexOf('#', i);
    const length = parseInt(s.substring(i, j), 10);  // parse the length before
'#'
    const str = s.substring(j + 1, j + 1 + length);  // extract the string of
that length
    result.push(str);
    i = j + 1 + length;  // move to the start of the next encoded string
  }
  return result;
}
```

This decoding logic uses the fact that the length prefix tells us exactly how many characters to take after the `#`. We find the `#`, parse the number before it (this gives the size of the next string), then grab that many characters after `#` as one decoded string. Then we advance our pointer and repeat until we reach the end of the encoded string [32] [33].

Why this works: By including the length, we guard against special characters. Even if a string contains `#` or any other characters, the decoder doesn't get confused because it doesn't rely on finding a terminator in the content; it knows exactly how long the content is. For example, consider encoding `["abc","#def"]`. Using length-prefix, we get `"3#abc4##def"`. The decoder reads length `3` then string `abc`, then reads length `4` then string `#def`. Without length info, an approach like joining by a special sequence could break (e.g., if we just joined with `#` naively, we'd have `"abc##def"`, and it would be unclear if that was `["abc", "", "def"]` or `["abc", "#def"]`).

Interview Tips: This problem is about **designing a protocol**. Make sure to mention that many solutions exist, but the length-prefix approach is a standard one in system design [29]. Another possible solution is to use escaping: e.g., choose a delimiter and then escape that delimiter if it appears in the string. However, that gets tricky to implement correctly. Length-prefix is simpler. Also, consider edge cases: - An empty list of strings should encode to an empty string (depending on design) and decode back to an empty list. - Empty strings within the list: e.g., `["hello", "", "world"]` should be handled. In our scheme, that would encode as `"5#hello0#5#world"` (note the `0#` for the empty string). - Very large strings: our encoding still works as long as we can represent the length as a string of digits. (In interviews, you can assume 32-bit int for length or so). Be ready to walk through a complete encode-decode cycle with an example to show it works. Also, mention that this concept is essentially how certain serialization formats work and it's important in networking (for example, TCP streams often include message lengths for chunking data). The interviewer is looking for your ability to come up with a robust scheme and also implement parsing carefully (off-by-one errors are common when writing the decode). So take care when writing index arithmetic, as we did with `i`, `j`, and calculating the next position.

7. Product of Array Except Self (LeetCode 238)

Problem: Given an array of numbers, produce a new array such that each element at index i of the new array is the product of all the numbers in the original array *except* the one at i. For example, if the input is `[1,2,3,4]`, the output should be `[24,12,8,6]` because: - The product of all except index 0 is 2×3×4 = 24. - Except index 1: 1×3×4 = 12. - Except index 2: 1×2×4 = 8. - Except index 3: 1×2×3 = 6.

This problem specifically disallows using division (otherwise you could simply multiply everything and divide by the element at i, but that fails when zero is present, and also the problem usually forbids it to force a different approach).

Discussion: The challenge is to compute these products efficiently without computing the entire product for each index from scratch (which would be $O(n^2)$). A very elegant solution uses the concept of **prefix and suffix products** [34] [35] : - **Prefix products:** For each index i, imagine the product of all elements to the *left* of i (i.e., before i in the array). We can compute these in one pass from left to right. - **Suffix products:** Similarly, for each index i, consider the product of all elements to the *right* of i (after i in the array). We can compute these in one pass from right to left. - Finally, the product of all numbers except self at i is simply **prefix[i] * suffix[i]** [36] , because prefix[i] is everything before i, and suffix[i] is everything after i.

Concretely: 1. Compute an array `prefix` where `prefix[i]` = product of `nums[0] * nums[1] * ... * nums[i-1]`. By definition, `prefix[0] = 1` (since nothing is to the left of index 0). 2. Compute an array `suffix` where `suffix[i]` = product of `nums[i+1] * nums[i+2] * ... * nums[n-1]`. Define `suffix[n-1] = 1` (nothing to the right of the last element). 3. Now, the answer for index i is `prefix[i] * suffix[i]` [36] .

Example walkthrough for `nums = [1,2,4,6]` : - prefix pass:
- prefix[0] = 1 (by definition, no elements to left)
- prefix[1] = nums[0] = 1
- prefix[2] = nums[0]*nums[1]* = *1*2 = 2
- prefix[3] = nums[0]*nums[1]*nums[2] = 1*2*4 = 8
So `prefix = [1, 1, 2, 8]` which means, for example, before index 3 the product of [1,2,4] is 8 [37] . - suffix pass (from right end):
- suffix[3] = 1 (no elements to right of last index)
- suffix[2] = nums[3] = 6
- suffix[1] = nums[2]*nums[3]* = *4*6 = 24
- suffix[0] = nums[1]*nums[2]*nums[3] = 2*4*6 = 48
So `suffix = [48, 24, 6, 1]` (check: to the right of index 0 are [2,4,6] which multiply to 48, etc.). - result: multiply prefix and suffix element-wise: `[1*48, 1*24, 2*6, 8*1] = [48, 24, 12, 8]`. This matches the expected output [35] .

The above method uses two additional arrays of size n, achieving O(n) time and O(n) space. However, we can optimize space by noticing that we don't actually need to keep all values of prefix and suffix at once – we can calculate the result on the fly. Specifically, we can use the output array itself to store prefix products first, then multiply by suffix products in a second pass. This way, we use only O(1) extra space (aside from the output): - First pass: build `res` such that `res[i] = prefix[i]` (same definition as above). We can do this in one loop without a separate array. - Second pass: traverse from the end, keep a running suffix product, and multiply each `res[i]` by that running suffix. Update the suffix product as we go.

JavaScript optimized solution:

```
function productExceptSelf(nums) {
  const n = nums.length;
  const res = Array(n);
```

```
  // Build prefix products in res
  res[0] = 1;
  for (let i = 1; i < n; i++) {
    res[i] = res[i-1] * nums[i-1];
  }
  // res[i] now holds prefix product up to i-1.
  // Now multiply by suffix products
  let suffixProd = 1;
  for (let i = n - 1; i >= 0; i--) {
    res[i] *= suffixProd;
    suffixProd *= nums[i];
  }
  return res;
}
```

In this code, after the first loop, `res[i]` is the product of all elements to the left of i. The second loop uses `suffixProd` to represent the product of all elements to the right of the current index (initially 1 for the last element). We update each `res[i]` by multiplying with `suffixProd`, then update `suffixProd` by including `nums[i]` (moving leftwards). This yields the correct result without needing separate arrays for prefix and suffix, thus O(1) additional space [38] [39].

Interview Tips: This problem is a favorite for testing knowledge of array manipulation and understanding multiplicative prefix/suffix concepts. Interviewers expect you to come up with the prefix-suffix idea, because the division trick (compute total product and divide by each element) is typically disallowed or leads to corner cases (zeroes). When explaining, clearly state the insight: "the product except self can be broken into product of elements before i and product of elements after i" [36]. Using examples helps. Also, mention how you handle zeros: - If the array has zeros, the logic still works: prefix and suffix products naturally become 0 beyond the zero element. Specifically, if more than one zero is present, all outputs will be 0 (because each position is missing at least one factor). If exactly one zero is present, all outputs except one will be 0, and the position of the zero will be the product of all non-zero elements. Our algorithm handles this inherently. It's good to check a quick example with zeros mentally or in explanation to show robustness. For instance, `[1,2,0,4]` -> output should be `[0*4*?=0, 0*4*?=0, product of [1,2,4] = 8, 0*?*?=0]` -> `[0,0,8,0]`. The prefix-suffix method yields that correctly. Space optimization is a nice touch to mention (interviewers appreciate when you recognize you can reuse the output array to save space). However, clarity is more important, so first ensure the basic two-pass idea is solid. This problem also teaches a pattern that appears in other contexts (like solving certain DP problems or other array manipulation tasks). It's a great example of how thinking in terms of "accumulations from left and right" can lead to a linear solution without brute force.

8. Valid Sudoku (LeetCode 36)

Problem: Determine if a 9x9 Sudoku board is valid. A valid Sudoku board (partially filled) will not have any duplicates in any row, any column, or any of the nine 3x3 sub-grids (boxes). The board may be only partially filled, so empty cells (denoted by `'.'`) are allowed and ignored in the validity check. We just need to ensure no rule is violated by the placed numbers.

Discussion: Sudoku is a familiar puzzle, and this problem is about enforcing its rules. Each number 1-9 must appear at most once per row, column, and 3x3 box [40]. We're essentially asked to check a matrix against these constraints. This is a hashing problem in disguise: we can use sets (or boolean arrays) to track seen numbers for each row, each column, and each subgrid.

A straightforward approach: - Use 3 sets of 9 sets each: - 9 row sets (rowSets[0] through rowSets[8]), - 9 column sets, - 9 box sets. - Iterate through each cell of the board (row `r`, column `c`): - If the cell is `'.'`, skip it (empty). - Otherwise, let the value be `val`. - Check the corresponding row set `rowSets[r]`. If `val` is already in it, the row has a duplicate, invalid. - Check `colSets[c]`. If `val` is in it, column duplicate, invalid. - Compute the box index for (r,c). Each 3x3 box can be indexed from 0 to 8. A common formula is `boxIndex = Math.floor(r/3) * 3 + Math.floor(c/3)` [41]. This maps row and col to the range 0-8 such that: - Box 0: rows 0-2, cols 0-2 - Box 1: rows 0-2, cols 3-5 - Box 2: rows 0-2, cols 6-8 - Box 3: rows 3-5, cols 0-2, etc. - Check the corresponding `boxSets[boxIndex]` for `val` as well. - If `val` is not found in any of the three sets, insert it into each: rowSets[r], colSets[c], boxSets[boxIndex]. - If you finish scanning all cells with no conflict, the board is valid [42] [43].

This approach is **O(81) = O(1)** essentially, since the board is always 9x9 (constant size). Even if you think of it in terms of n for generality, it's $O(n^2)$ for an n×n board, but here n is 9. The space used is also small (a few sets of size at most 9).

Let's illustrate the logic with a small snippet in JavaScript:

```
function isValidSudoku(board) {
  const rows = Array.from({length: 9}, () => new Set());
  const cols = Array.from({length: 9}, () => new Set());
  const boxes = Array.from({length: 9}, () => new Set());

  for (let r = 0; r < 9; r++) {
    for (let c = 0; c < 9; c++) {
      const val = board[r][c];
      if (val === '.') continue;
      if (rows[r].has(val) || cols[c].has(val)) {
        return false; // duplicate in row or col
      }
      const boxIndex = Math.floor(r/3) * 3 + Math.floor(c/3);
      if (boxes[boxIndex].has(val)) {
        return false; // duplicate in 3x3 box
      }
      // mark as seen
      rows[r].add(val);
      cols[c].add(val);
      boxes[boxIndex].add(val);
    }
  }
```

```
    return true;
}
```

Each number is checked in its row, column, and box context. By using sets we ensure that if a number appears a second time in any of those contexts, we catch it and return false immediately.

Things to watch out for: - We ignore `'.'` because empty cells are allowed and not part of validation [44] . - A common mistake is to accidentally allow a number twice in the same row/col/box because of wrong indexing of the box. But the formula above is a reliable way to map (r,c) to a box index 0-8 [41] . - We consider only validity of the current state, not solvability. The problem is not asking us to solve the Sudoku, just to verify that no rule is broken so far. So it's much simpler than a solver.

Interview Tips: Although the board is fixed size, treat the problem systematically, as if it could generalize. It demonstrates your ability to handle 2D arrays and constraints. Explain that using sets is natural to track seen numbers (because checking membership is O(1), and it clearly communicates the intent "have we seen this number before in this context"). Also mention alternatives: you could use boolean arrays of length 9 for each row/col/box to mark digits 1-9 as seen (using index 0-8 to represent number 1-9). That might be slightly more memory efficient in a low-level sense, but in an interview, clarity is more important.

A nice touch: mention that the problem is essentially using hashing to enforce uniqueness constraints. Each row, column, and box is like a separate hash set of seen numbers. This is a common technique for matrix validation problems. Also, consider edge cases: if any cell has an invalid character (not '.' or '1'-'9'), we could decide to return false or ignore, but usually input is guaranteed valid format. If a row or column or box has a duplicate, we stop early – that's fine.

The Sudoku validation doesn't have a direct "real-world application" outside of games, but it's representative of problems where you need to validate a matrix against multiple rules (for example, verifying a Latin square or a solved Sudoku, etc.). The main point is showing you can systematically check all constraints without mixing them up. If asked how you might extend this, you could mention checking a fully filled board for also being a correct Sudoku solution (which would mean also every row/col/box has all digits 1-9, but that's a different condition beyond just validity). In summary, focus on the method of tracking and checking duplicates, which is a pattern useful in many contexts.

9. Longest Consecutive Sequence (LeetCode 128)

Problem: Given an unsorted array of integers, find the length of the longest consecutive elements sequence. The consecutive sequence refers to numerical sequence (not positions in the array). For example, in `[100, 4, 200, 1, 3, 2]`, the longest consecutive sequence is `[1, 2, 3, 4]` which has length 4. We need to return the length (which is 4 in this case).

Discussion: A naive approach would sort the array and then scan for longest run, which would take O(n log n) time due to sorting. However, there's a clever O(n) solution using a hash set to allow O(1) lookups of numbers [45] . The idea is to use the hash set to quickly check the existence of numbers that are next in sequence.

Approach using hashing: 1. Insert all numbers into a hash set (this de-duplicates them as well, which doesn't harm the answer, since duplicates don't extend a consecutive sequence beyond the unique values). 2. Initialize a variable `longestStreak = 0`. 3. Iterate through each number `x` in the set: - If `x-1` is *not* in the set, then `x` is a potential **start** of a sequence [46]. Why? Because if there was a smaller number right before `x`, then `x` would be in the middle of a sequence, and we want to only start counting at the beginning of sequences to avoid recounting. - If `x` is the start of a sequence, then proceed to count how long this consecutive sequence goes: check for `x+1, x+2, x+3, ...` in the set until it breaks. - Compute the length of this sequence and update `longestStreak` if this length is greater. 4. Continue until all numbers are processed. 5. Return `longestStreak`.

Because each number is inserted into the set (O(n)), and in the worst case each number is part of at most one sequence counting process, the overall time is O(n). Each element is essentially visited at most twice – once in the initial insertion, and once in the sequence counting. We never restart counting from a middle element, which prevents superlinear behavior [47].

Example walk-through: For `[100, 4, 200, 1, 3, 2]`: - Put in set: `{100, 4, 200, 1, 3, 2}`. - Iterate: say we pick 100 first. Check 99 (100-1) – not in set, so 100 is start of a sequence. Count forward: 100 in set (start count=1), 101 not in set -> sequence length = 1. longestStreak = 1. - Next, 4. Check 3 (4-1) – 3 *is* in set, so 4 is not a start (it's actually continuation of a sequence that should start at 3 or earlier). So we skip counting from 4. - Next, 200. Check 199 – not in set, so start at 200. 200 (count 1), 201 not in set -> length 1. longestStreak still 1 (ties don't increase). - Next, 1. Check 0 – not in set, start at 1. Now count: 1 (exists), 2 (exists), 3 (exists), 4 (exists), 5 (not in set) -> sequence [1,2,3,4] length = 4. longestStreak becomes 4. - Next, 3. Check 2 – 2 is in set, so 3 isn't a start (we already counted that sequence starting at 1). - Next, 2. Check 1 – 1 is in set, not a start. - Done. longestStreak = 4, which is correct [48] [49].

JavaScript Solution:

```
function longestConsecutive(nums) {
  if (nums.length === 0) return 0;
  const numSet = new Set(nums);
  let longestStreak = 0;

  for (let num of numSet) {
    // only start counting if `num` is start of a sequence
    if (!numSet.has(num - 1)) {
      let currentNum = num;
      let currentStreak = 1;
      // count up the consecutive numbers
      while (numSet.has(currentNum + 1)) {
        currentNum += 1;
        currentStreak += 1;
      }
      longestStreak = Math.max(longestStreak, currentStreak);
    }
  }
```

```
    return longestStreak;
}
```

We check `!numSet.has(num-1)` to ensure we only start when `num` is the smallest in its sequence [46]. The inner `while` then counts how far the sequence goes. This is efficient because each sequence is counted once in total. If a number is not a sequence start, the loop is skipped entirely for that number.

Interview Tips: Emphasize the reasoning for checking `num-1`. This is the crux of avoiding an $O(n^2)$ scenario. Without this check, if you naively for each number tried to scan downwards and upwards, you could repeat work. By ensuring you only start at the beginning of sequences, each number is part of at most one count. Also, it's worth noting the use of a set to get O(1) containment queries, which makes the scanning linear overall [50].

Mention alternative approaches: - Sorting then scanning is simpler to think of, but it's O(n log n). If asked, you can outline that approach (sort, then iterate keeping track of current streak, reset when gap found). - Another approach could use Union-Find (disjoint set union) to union consecutive numbers, but that's overkill and also roughly O(n) amortized; not needed but you could mention it if you want to show knowledge. The hash set approach is the ideal solution for interviews due to its clarity and optimal complexity.

Make sure to consider edge cases: empty array (answer 0), array with one element (answer 1), arrays where all elements are the same (duplicates) – the set will reduce them and answer will be 1. Our code handles these. If duplicates are present, they don't affect the sequence length because a consecutive sequence is defined by distinct numbers; using a set inherently took care of that by ignoring duplicates.

To conclude, highlight that this approach achieves O(n) time on average, which is optimal, and uses O(n) space for the set. It's a beautiful application of hashing to detect sequences. This pattern of looking for sequence starts can be reused in other problems (like longest sequence in graph edges etc., but primarily it's known for this problem). Interviewers often appreciate hearing how you'd ensure no double counting – the `num-1` trick is key to demonstrate.

References: The explanations and approaches above reference various resources and common solutions from the programming community, including insights on using sets for duplicates [51] [12], sorting vs hashing trade-offs [14], using hash maps for Two Sum [19], grouping anagrams by sorted keys [23], heap usage for top-K frequencies [26], length-prefix encoding for strings [32], prefix/suffix product techniques [36], Sudoku validation with sets [42], and hash-set sequence finding [45] [46]. These are well-established strategies in solving algorithm problems efficiently and are crucial knowledge for coding interviews.

---

1 2 3 4 5 6 7 8 16 23 31 34 35 37 41 42 43 45 46 47 50 NeetCode 150 Solutions Explained: Arrays & Hashing | by Akanksha Wagh | Nov, 2025 | Medium
https://akankshawagh.medium.com/neetcode-150-solutions-explained-arrays-hashing-ed0c6184f387

9 10 11 12 13 14 51 How to Solve the “Contains Duplicate” Problem: A Simple Guide | by Mrinmayee Gajanan Rane | Medium
https://medium.com/@mrinmayeerane2810/how-to-solve-the-contains-duplicate-problem-a-simple-guide-e1e1793dab88

15 Group Anagrams Together (With Visualization and Code Examples)
https://www.finalroundai.com/articles/group-anagrams-algorithm

17 deepSWE.io
https://deepswe.io/courses/hashing/group-anagrams

18 Lessons from Leetcode: Two Sum and a brief lesson in HashMaps
https://levelup.gitconnected.com/lessons-from-leetcode-two-sum-and-a-brief-lesson-in-hashmaps-1bbc324181c8

19 20 21 22 Two Sum (Interview Solution)
https://interviewing.io/questions/two-sum

24 25 28 Top K Frequent Elements Guide [Problem + Solution]
https://interviewing.io/questions/top-k-frequent-elements

26 27 347. Top K Frequent Elements - In-Depth Explanation
https://algo.monster/liteproblems/347

29 30 32 33 LeetCode - 271 Encode and Decode Strings
https://www.linkedin.com/pulse/leetcode-271-encode-decode-strings-asbaq-laareb-kac1c

36 38 39 Product of Array Except Self - GeeksforGeeks
https://www.geeksforgeeks.org/dsa/a-product-array-puzzle/

40 44 36. Valid Sudoku - In-Depth Explanation
https://algo.monster/liteproblems/36

48 49 128. Longest Consecutive Sequence - In-Depth Explanation
https://algo.monster/liteproblems/128