PHP Coding Challenges

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

1. Write a PHP function to find the longest palindromic substring in a given string using dynamic programming.

Longest Palindromic Substring

This problem requires identifying the longest contiguous substring that reads the same forwards and backwards. The **dynamic programming approach** uses a 2D table to track palindromes.

```
function longestPalindrome($s) {
    $n = strlen($s);
    if ($n < 2) return $s;
    $dp = array_fill(0, $n, array_fill(0, $n, false));
    $start = 0; $maxLen = 1;
    for ($i = 0; $i < $n; $i++) $dp[$i][$i] = true;
    for ($len = 2; $len <= $n; $len++) {
        for ($i = 0; $i <= $n - $len; $i++) {
            $j = $i + $len - 1;
            if ($s[$i] === $s[$j] && ($len === 2 || $dp[$i+1][$j-1])) {
                $dp[$i][$j] = true;
                $start = $i; $maxLen = $len;
            }
        }
    }
    return substr($s, $start, $maxLen);
}
```

Key points:

- Time complexity: O(n^2)
- Space complexity: O(n^2)
- Handles edge cases like single characters and empty strings
- Can be optimized to O(n) space using expand-around-center approach

2. Implement a PHP class that handles rate limiting using the Token Bucket algorithm with Redis backend.

Token Bucket Rate Limiter

The **Token Bucket algorithm** allows burst traffic while maintaining average rate limits. This implementation uses Redis for distributed rate limiting across multiple servers.

```
class RateLimiter {
    private $redis;
    public function __construct($redis) { $this->redis = $redis; }
    public function allowRequest($key, $capacity, $refillRate) {
        $now = microtime(true);
        $tokenKey = "tokens:$key";
        $timeKey = "time:$key";
        $tokens = $this->redis->get($tokenKey) ?? $capacity;
        $lastTime = $this->redis->get($timeKey) ?? $now;
        $elapsed = $now - $lastTime;
        $tokens = min($capacity, $tokens + $elapsed * $refillRate);
        if ($tokens >= 1) {
            $this->redis->setex($tokenKey, 3600, $tokens - 1);
            $this->redis->setex($timeKey, 3600, $now);
            return true;
        }
        return false;
    }
}
```

Features:

- Distributed rate limiting with Redis
- Configurable capacity and refill rate
- Handles burst traffic gracefully
- Thread-safe with atomic operations

3. Create a PHP function that detects cycles in a directed graph using depth-first search.

Cycle Detection in Directed Graph

Detecting cycles in a **directed graph** requires tracking visited nodes and the current recursion stack. A cycle exists if we encounter a node already in the current path.

```
function hasCycle($graph) {
    $visited = [];
    $recStack = [];
    foreach (array_keys($graph) as $node) {
        if (!isset($visited[$node])) {
            if (dfs($graph, $node, $visited, $recStack)) {
                return true;
            }
        }
    }
    return false;
}
function dfs($graph, $node, &$visited, &$recStack) {
    $visited[$node] = true;
    $recStack[$node] = true;
    foreach ($graph[$node] ?? [] as $neighbor) {
        if (!isset($visited[$neighbor]) && dfs($graph, $neighbor, $visited, $recStack)) return true;
        if (isset($recStack[$neighbor])) return true;
    }
    unset($recStack[$node]);
    return false;
}
```

Algorithm details:

- Uses DFS with recursion stack tracking
- Time complexity: O(V + E)
- Space complexity: O(V)
- Distinguishes between visited and currently processing nodes

4. Write a PHP function to implement the Levenshtein distance algorithm for calculating edit distance between two strings.

Levenshtein Distance Implementation

The **Levenshtein distance** measures the minimum number of single-character edits (insertions, deletions, substitutions) needed to transform one string into another.

```
function levenshteinDistance($str1, $str2) {
    $len1 = strlen($str1);
    $len2 = strlen($str2);
    $dp = array_fill(0, $len1 + 1, array_fill(0, $len2 + 1, 0));
    for ($i = 0; $i <= $len1; $i++) $dp[$i][0] = $i;
    for ($j = 0; $j <= $len2; $j++) $dp[0][$j] = $j;
    for ($i = 1; $i <= $len1; $i++) {
        for ($j = 1; $j <= $len2; $j++) {
            $cost = ($str1[$i-1] === $str2[$j-1]) ? 0 : 1;
            $dp[$i][$j] = min($dp[$i-1][$j] + 1, $dp[$i][$j-1] + 1, $dp[$i-1][$j-1] + $cost);
        }
    }
    return $dp[$len1][$len2];
}
```

Use cases:

- Spell checking and autocorrect
- DNA sequence analysis
- Fuzzy string matching
- Time complexity: O(m × n)

5. Implement a PHP memory-efficient solution to find the kth largest element in an unsorted array using a min-heap.

Kth Largest Element with Min-Heap

Using a **min-heap of size k** is memory-efficient for finding the kth largest element, especially in large datasets or streaming scenarios.

```
function findKthLargest($nums, $k) {
    $heap = new SplMinHeap();
    foreach ($nums as $num) {
        $heap->insert($num);
        if ($heap->count() > $k) {
            $heap->extract();
        }
    }
    return $heap->top();
}
```

Advantages:

- Space complexity: O(k) instead of O(n)
- Time complexity: O(n log k)
- Works with streaming data
- More efficient than full sorting for small k
- PHP's SplMinHeap provides built-in heap operations

This approach maintains only the k largest elements seen so far, making it ideal when k is much smaller than n.

6. Create a PHP function that implements a Trie (prefix tree) with insert, search, and startsWith methods for efficient string operations.

Trie Data Structure Implementation

A **Trie** is a tree-based data structure for efficient string storage and retrieval, commonly used in autocomplete and spell-checking systems.

```
class TrieNode {
    public $children = [];
    public $isEndOfWord = false;
}
class Trie {
    private $root;
    public function __construct() { $this->root = new TrieNode(); }
    public function insert($word) {
        $node = $this->root;
        for ($i = 0; $i < strlen($word); $i++) {
            $char = $word[$i];
            if (!isset($node->children[$char])) $node->children[$char] = new TrieNode();
            $node = $node->children[$char];
        }
        $node->isEndOfWord = true;
    }
    public function search($word) {
        $node = $this->searchPrefix($word);
        return $node !== null && $node->isEndOfWord;
    }
    public function startsWith($prefix) { return $this->searchPrefix($prefix) !== null; }
    private function searchPrefix($prefix) {
        $node = $this->root;
        for ($i = 0; $i < strlen($prefix); $i++) {
            $char = $prefix[$i];
            if (!isset($node->children[$char])) return null;
            $node = $node->children[$char];
        }
        return $node;
    }
}
```

Performance:

- Insert: O(m) where m is word length
- Search: O(m)
- Space: O(ALPHABET_SIZE × N × M)
- Ideal for prefix-based operations

7. Write a PHP function to merge K sorted linked lists efficiently using a priority queue approach.

Merge K Sorted Lists

Merging multiple sorted lists efficiently requires a **min-heap (priority queue)** to always select the smallest element among k candidates.

```
class ListNode {
    public $val;
    public $next;
    public function __construct($val = 0, $next = null) {
        $this->val = $val; $this->next = $next;
    }
}
function mergeKLists($lists) {
    $heap = new SplMinHeap();
    $dummy = new ListNode(0);
    $current = $dummy;
    foreach ($lists as $i => $list) {
        if ($list !== null) $heap->insert([$list->val, $i, $list]);
    }
    while (!$heap->isEmpty()) {
        [$val, $i, $node] = $heap->extract();
        $current->next = $node;
        $current = $current->next;
        if ($node->next !== null) $heap->insert([$node->next->val, $i, $node->next]);
    }
    return $dummy->next;
}
```

Complexity analysis:

- Time: O(N log k) where N is total nodes
- Space: O(k) for the heap
- Better than sequential merging O(kN)

8. Implement a PHP solution for the sliding window maximum problem that finds the maximum value in each window of size k.

Sliding Window Maximum

This problem requires finding the maximum in every contiguous subarray of size k. Using a **deque (double-ended queue)** maintains indices of useful elements in decreasing order.

```
function maxSlidingWindow($nums, $k) {
    $result = [];
    $deque = [];
    for ($i = 0; $i < count($nums); $i++) {
        while (!empty($deque) && $deque[0] < $i - $k + 1) {
            array_shift($deque);
        }
        while (!empty($deque) && $nums[end($deque)] < $nums[$i]) {
            array_pop($deque);
        }
        $deque[] = $i;
        if ($i >= $k - 1) $result[] = $nums[$deque[0]];
    }
    return $result;
}
```

Key insights:

- Time complexity: O(n) - each element processed at most twice
- Space complexity: O(k)
- Deque maintains indices in decreasing order of values
- Front of deque always contains maximum of current window

9. Create a PHP function that solves the N-Queens problem using backtracking and returns all possible solutions.

N-Queens Backtracking Solution

The **N-Queens problem** requires placing N queens on an N×N chessboard so no two queens attack each other. This uses backtracking to explore all valid configurations.

```
function solveNQueens($n) {
    $solutions = [];
    $board = array_fill(0, $n, array_fill(0, $n, '.'));
    backtrack($board, 0, $n, $solutions);
    return $solutions;
}
function backtrack(&$board, $row, $n, &$solutions) {
    if ($row === $n) {
        $solutions[] = array_map(fn($r) => implode('', $r), $board);
        return;
    }
    for ($col = 0; $col < $n; $col++) {
        if (isSafe($board, $row, $col, $n)) {
            $board[$row][$col] = 'Q';
            backtrack($board, $row + 1, $n, $solutions);
            $board[$row][$col] = '.';
        }
    }
}
function isSafe($board, $row, $col, $n) {
    for ($i = 0; $i < $row; $i++) if ($board[$i][$col] === 'Q') return false;
    for ($i = $row - 1, $j = $col - 1; $i >= 0 && $j >= 0; $i--, $j--) if ($board[$i][$j] === 'Q') return false;
    for ($i = $row - 1, $j = $col + 1; $i >= 0 && $j < $n; $i--, $j++) if ($board[$i][$j] === 'Q') return false;
    return true;
```

}

Algorithm features:

- Explores solution space systematically
- Prunes invalid branches early
- Time complexity: O(N!)
- Returns all distinct solutions

10. Write a PHP function to implement the LRU (Least Recently Used) cache with O(1) get and put operations using a hash map and doubly linked list.

LRU Cache Implementation

An **LRU cache** evicts the least recently used item when capacity is reached. Combining a hash map with a doubly linked list achieves O(1) operations.

```
class LRUCache {
    private $capacity, $cache = [], $head, $tail;
    public function __construct($capacity) {
        $this->capacity = $capacity;
        $this->head = new Node(0, 0);
        $this->tail = new Node(0, 0);
        $this->head->next = $this->tail;
        $this->tail->prev = $this->head;
    }
    public function get($key) {
        if (!isset($this->cache[$key])) return -1;
        $node = $this->cache[$key];
        $this->remove($node);
        $this->add($node);
        return $node->value;
    }
    public function put($key, $value) {
        if (isset($this->cache[$key])) $this->remove($this->cache[$key]);
        $node = new Node($key, $value);
        $this->cache[$key] = $node;
        $this->add($node);
        if (count($this->cache) > $this->capacity) {
            $lru = $this->head->next;
            $this->remove($lru);
            unset($this->cache[$lru->key]);
        }
    }
    private function add($node) {
        $prev = $this->tail->prev;
        $prev->next = $node; $node->prev = $prev;
        $node->next = $this->tail; $this->tail->prev = $node;
    }
    private function remove($node) {
        $node->prev->next = $node->next;
        $node->next->prev = $node->prev;
    }
}
class Node { public $key, $value, $prev, $next; public function __construct($k, $v) { $this->key = $k; $this->value = $v; } }
```

Design highlights:

- O(1) get and put operations
- Hash map for fast lookup
- Doubly linked list maintains access order
- Most recently used at tail, least at head

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. Implement a Stack data structure in PHP with push, pop, peek, and isEmpty methods. What is the time complexity of each operation?

Stack Implementation

A stack follows **LIFO (Last In First Out)** principle. Here's a clean implementation using an array:

```
class Stack {
  private $items = [];
  public function push($item) { $this->items[] = $item; }
  public function pop() { return array_pop($this->items); }
  public function peek() { return end($this->items); }
  public function isEmpty() { return empty($this->items); }
  public function size() { return count($this->items); }
}
```

Time Complexity:

- push(): O(1)
- pop(): O(1)
- peek(): O(1)
- isEmpty(): O(1)

2. Write a PHP function to implement an LRU (Least Recently Used) Cache with get and put operations in O(1) time complexity.

LRU Cache Implementation

An **LRU Cache** requires O(1) access and eviction. Use a combination of a doubly linked list and hash map:

```
class LRUCache {
  private $capacity, $cache = [], $order = [];
  public function __construct($cap) { $this->capacity = $cap; }
  public function get($key) {
    if (!isset($this->cache[$key])) return -1;
    $this->updateOrder($key);
    return $this->cache[$key];
  }
  public function put($key, $val) {
    if (count($this->cache) >= $this->capacity && !isset($this->cache[$key])) {
      $lru = array_shift($this->order);
      unset($this->cache[$lru]);
    }
    $this->cache[$key] = $val;
    $this->updateOrder($key);
  }
  private function updateOrder($key) {
    $this->order = array_diff($this->order, [$key]);
    $this->order[] = $key;
  }
}
```

Key Points: Uses array to track access order, removes least recently used item when capacity exceeded.

3. Implement a function to find all pairs in an array that sum to a target value. Optimize for O(n) time complexity.

Two Sum Problem

Use a **hash map** to achieve O(n) time complexity by storing complements:

```
function findPairs($arr, $target) {
  $seen = [];
  $pairs = [];
  foreach ($arr as $num) {
    $complement = $target - $num;
    if (isset($seen[$complement])) {
      $pairs[] = [$complement, $num];
    }
    $seen[$num] = true;
  }
  return $pairs;
}
```

Complexity Analysis:

- Time: O(n) - single pass through array
- Space: O(n) - hash map storage

Example: findPairs([2, 7, 11, 15], 9) returns [[2, 7]]

4. Write a PHP function to implement a Min Heap with insert, extractMin, and getMin operations. What are the time complexities?

Min Heap Implementation

A **Min Heap** maintains the smallest element at the root. PHP's SplMinHeap can be used, or implement manually:

```
class MinHeap extends SplMinHeap {
  public function compare($a, $b) {
    return $b - $a;
  }
}
$heap = new MinHeap();
$heap->insert(5);
$heap->insert(3);
$heap->insert(7);
echo $heap->top(); // 3
echo $heap->extract(); // 3
```

Time Complexity:

- insert(): O(log n) - bubble up
- extractMin(): O(log n) - bubble down
- getMin(): O(1) - peek at root

5. Implement a function to check if a string has balanced parentheses/brackets using a stack. Include support for (), {}, and [].

Balanced Brackets Checker

Use a **stack** to match opening and closing brackets:

```
function isBalanced($str) {
  $stack = [];
  $pairs = ['(' => ')', '{' => '}', '[' => ']'];
  for ($i = 0; $i < strlen($str); $i++) {
    $char = $str[$i];
    if (isset($pairs[$char])) {
      $stack[] = $char;
    } elseif (in_array($char, $pairs)) {
      if (empty($stack) || $pairs[array_pop($stack)] !== $char) return false;
    }
  }
  return empty($stack);
}
```

Logic: Push opening brackets onto stack, pop and match closing brackets. Returns true if stack is empty at end.

6. Write a sliding window algorithm to find the maximum sum of k consecutive elements in an array. Optimize for O(n) time.

Sliding Window Maximum Sum

The **sliding window technique** avoids recalculating the entire sum for each window:

```
function maxSumKConsecutive($arr, $k) {
  $n = count($arr);
  if ($n < $k) return null;
  $windowSum = array_sum(array_slice($arr, 0, $k));
  $maxSum = $windowSum;
  for ($i = $k; $i < $n; $i++) {
    $windowSum = $windowSum - $arr[$i - $k] + $arr[$i];
    $maxSum = max($maxSum, $windowSum);
  }
  return $maxSum;
}
```

Time Complexity: O(n) - single pass. **Space:** O(1). Subtract leftmost, add rightmost element each iteration.

7. Implement a function to detect a cycle in a linked list using Floyd's Cycle Detection Algorithm. What is the time and space complexity?

Cycle Detection in Linked List

Floyd's Algorithm uses two pointers (slow and fast) to detect cycles:

```
class ListNode {
  public $val, $next;
  public function __construct($val) { $this->val = $val; }
}
function hasCycle($head) {
  $slow = $fast = $head;
  while ($fast && $fast->next) {
    $slow = $slow->next;
    $fast = $fast->next->next;
    if ($slow === $fast) return true;
  }
  return false;
}
```

Complexity:

- Time: O(n) - at most 2n iterations
- Space: O(1) - only two pointers

If cycle exists, fast pointer will eventually meet slow pointer.

8. Write a PHP function to implement binary search on a sorted array. Include both iterative and recursive approaches.

Binary Search Implementation

Binary search efficiently finds elements in sorted arrays by repeatedly halving the search space:

```
// Iterative
function binarySearch($arr, $target) {
  $left = 0; $right = count($arr) - 1;
  while ($left <= $right) {
    $mid = floor(($left + $right) / 2);
    if ($arr[$mid] === $target) return $mid;
    if ($arr[$mid] < $target) $left = $mid + 1;
    else $right = $mid - 1;
  }
  return -1;
}
```

Recursive version: Replace while loop with recursive calls adjusting left/right bounds.

Time Complexity: O(log n), **Space:** O(1) iterative, O(log n) recursive (call stack)

9. Implement a Trie (Prefix Tree) data structure with insert, search, and startsWith methods for autocomplete functionality.

Trie Implementation

A **Trie** efficiently stores and searches strings with common prefixes:

```
class TrieNode {
  public $children = [];
  public $isEnd = false;
}
class Trie {
  private $root;
  public function __construct() { $this->root = new TrieNode(); }
  public function insert($word) {
    $node = $this->root;
    foreach (str_split($word) as $char) {
      if (!isset($node->children[$char])) $node->children[$char] = new TrieNode();
      $node = $node->children[$char];
    }
    $node->isEnd = true;
  }
  public function search($word) {
    $node = $this->traverse($word);
    return $node && $node->isEnd;
  }
  public function startsWith($prefix) { return $this->traverse($prefix) !== null; }
  private function traverse($str) {
    $node = $this->root;
    foreach (str_split($str) as $char) {
      if (!isset($node->children[$char])) return null;
      $node = $node->children[$char];
    }
    return $node;
  }
}
```

Time Complexity: O(m) for all operations where m is word length.

10. Write a function to find the kth largest element in an unsorted array without fully sorting it. Optimize using a min heap or quickselect.

Kth Largest Element

Use a **min heap** of size k to maintain k largest elements efficiently:

```
function findKthLargest($arr, $k) {
  $heap = new SplMinHeap();
  foreach ($arr as $num) {
    $heap->insert($num);
    if ($heap->count() > $k) {
      $heap->extract();
    }
  }
  return $heap->top();
}
```

Time Complexity: O(n log k) - better than O(n log n) for full sort.

Alternative: Quickselect algorithm achieves O(n) average time but $O(n^2)$ worst case. Min heap approach is more consistent for interview settings.

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. Write a PHP function to flatten a multi-dimensional array into a single-level array.

Flattening a Multi-Dimensional Array

Here's an efficient recursive solution to flatten nested arrays:

```
function flattenArray($array) {
    $result = [];
    array_walk_recursive($array, function($value) use (&$result) {
        $result[] = $value;
    });
    return $result;
}

// Example usage:
$nested = [1, [2, 3, [4, 5]], 6];
print_r(flattenArray($nested)); // [1, 2, 3, 4, 5, 6]
```

Key Points:

- **array_walk_recursive** traverses all nested elements automatically
- The closure captures **$result** by reference using **use (&$result)**
- Time complexity: O(n) where n is total number of elements
- Alternative: Use iterator_to_array with RecursiveIteratorIterator for large datasets

2. How would you reverse a string in PHP while properly handling multi-byte UTF-8 characters?

Reversing UTF-8 Strings Safely

Using **mb_string** functions ensures proper handling of multi-byte characters:

```
function reverseUtf8String($str) {
    $length = mb_strlen($str, 'UTF-8');
    $reversed = '';
    while ($length-- > 0) {
        $reversed .= mb_substr($str, $length, 1, 'UTF-8');
    }
    return $reversed;
}

// Example: reverseUtf8String('Hello □□') returns '□□ olleH'
```

Important Considerations:

- Never use **strrev()** for UTF-8 strings as it breaks multi-byte characters
- **mb_strlen** and **mb_substr** count characters, not bytes
- Always specify 'UTF-8' encoding explicitly
- For better performance on large strings, use **grapheme_*** functions for grapheme cluster support

3. Write a function to check if a string is a palindrome, ignoring case and non-alphanumeric characters.

Palindrome Checker with Sanitization

This solution normalizes input before comparison:

```
function isPalindrome($str) {
    $cleaned = preg_replace('/[^a-z0-9]/i', '', $str);
    $cleaned = strtolower($cleaned);
    return $cleaned === strrev($cleaned);
}

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

Implementation Details:

- **preg_replace** removes all non-alphanumeric characters using regex
- The 'i' flag makes the pattern case-insensitive

- **strtolower** normalizes case for comparison
- For UTF-8 support, replace strrev with the mb_string approach shown earlier
- Time complexity: O(n), Space complexity: O(n)

4. Explain how to use Xdebug for step debugging and profiling in PHP. What are the key configuration settings?

Xdebug Configuration and Usage

Xdebug is the most powerful debugging tool for PHP, offering step debugging, profiling, and code coverage.

Key php.ini Configuration:

```
zend_extension=xdebug.so
xdebug.mode=debug,profile
xdebug.start_with_request=trigger
xdebug.client_host=127.0.0.1
xdebug.client_port=9003
xdebug.output_dir=/tmp/xdebug
xdebug.profiler_output_name=cachegrind.out.%p
```

Essential Features:

- **Step Debugging:** Set breakpoints in IDEs (PHPStorm, VSCode) and step through code execution
- **Profiling:** Use xdebug.mode=profile to generate cachegrind files, analyze with tools like KCachegrind or Webgrind
- **Stack Traces:** Enhanced error reporting with full variable dumps
- **Remote Debugging:** Debug applications running on remote servers
- Trigger profiling with XDEBUG_PROFILE GET/POST parameter or cookie

5. How do you detect and fix memory leaks in long-running PHP applications? What tools and techniques would you use?

Memory Leak Detection and Prevention

Diagnostic Tools and Techniques:

- **memory_get_usage(true):** Monitor real memory allocation throughout execution
- **memory_get_peak_usage():** Identify maximum memory consumption points
- **Xdebug profiler:** Analyze memory allocation per function call
- **Blackfire.io or Tideways:** Production-grade profiling with memory tracking

Common Memory Leak Causes:

```
// BAD: Circular references
$obj1->ref = $obj2;
$obj2->ref = $obj1;

// FIX: Break references explicitly
unset($obj1->ref, $obj2->ref);
// Or use WeakReference (PHP 7.4+)
$obj1->ref = WeakReference::create($obj2);
```

Prevention Strategies:

- Explicitly unset large variables when done: **unset($largeArray)**
- Use generators for large datasets instead of loading everything into memory
- Avoid static class properties holding references
- Call **gc_collect_cycles()** manually in long-running processes
- Monitor with **gc_mem_caches()** to see internal cache sizes

6. Write a function to find the first non-repeating character in a string with optimal time complexity.

First Non-Repeating Character

Using a hash map approach for O(n) time complexity:

```
function firstNonRepeating($str) {
    $charCount = [];
    for ($i = 0; $i < strlen($str); $i++) {
        $char = $str[$i];
        $charCount[$char] = ($charCount[$char] ?? 0) + 1;
    }
    for ($i = 0; $i < strlen($str); $i++) {
        if ($charCount[$str[$i]] === 1) return $str[$i];
    }
    return null;
}
```

Algorithm Analysis:

- **First pass:** Build frequency map of all characters - O(n)
- **Second pass:** Find first character with count of 1 - O(n)
- **Total complexity:** O(n) time, O(k) space where k is unique characters
- The null coalescing operator **??** provides clean initialization
- For UTF-8 strings, replace strlen with mb_strlen and use mb_substr for character access

7. Explain PHP's exception handling hierarchy and demonstrate custom exception handling with multiple catch blocks and finally.

Advanced Exception Handling

PHP 7+ introduced Throwable as the base interface for all exceptions and errors:

```
class DatabaseException extends Exception {}
class ValidationException extends Exception {}

try {
    if (!$valid) throw new ValidationException('Invalid input');
    if (!$db->connect()) throw new DatabaseException('Connection failed');
} catch (ValidationException $e) {
    log_error('Validation: ' . $e->getMessage());
} catch (DatabaseException $e) {
    log_error('Database: ' . $e->getMessage());
} finally {
    $db->cleanup();
}
```

Exception Hierarchy:

- **Throwable** (interface) - base for all exceptions and errors
- **Exception** - base class for user exceptions
- **Error** - base class for internal PHP errors (TypeError, ParseError, etc.)
- **finally** block always executes, even if exception is thrown or return statement is hit
- Use **catch (Throwable $e)** to catch both exceptions and errors
- PHP 8+ supports **catch (ExceptionA | ExceptionB $e)** for multiple types

8. How would you implement a simple LRU (Least Recently Used) cache in PHP?

LRU Cache Implementation

Using SplDoublyLinkedList for efficient O(1) operations:

```
class LRUCache {
    private $capacity, $cache = [], $list;

    public function __construct($capacity) {
        $this->capacity = $capacity;
        $this->list = new SplDoublyLinkedList();
    }

    public function get($key) {
        if (!isset($this->cache[$key])) return null;
        $this->moveToFront($key);
        return $this->cache[$key]['value'];
    }
```

Key Design Decisions:

- Combine **array** (for O(1) lookup) with **SplDoublyLinkedList** (for O(1) reordering)
- Store both key and value to enable reverse lookup during eviction
- On **get()**: move accessed item to front (most recently used)
- On **put()**: add to front, evict from back if capacity exceeded
- Alternative: Use **array_key_exists** + **unset** + re-add pattern for simpler but less efficient implementation
- For production, consider using Redis or Memcached instead

9. What debugging techniques would you use to troubleshoot a PHP application with intermittent slow response times?

Performance Debugging Strategies

Systematic Troubleshooting Approach:

- **Application Profiling:** Use Blackfire, Tideways, or Xdebug profiler to identify slow functions
- **Query Analysis:** Enable MySQL slow query log, use EXPLAIN on queries
- **APM Tools:** New Relic, Datadog for transaction tracing and bottleneck identification
- **Logging:** Add microtime(true) timestamps at key points to measure execution segments

Code-Level Instrumentation:

```
class PerformanceMonitor {
    private static $timers = [];

    public static function start($label) {
        self::$timers[$label] = microtime(true);
    }

    public static function end($label) {
        $elapsed = microtime(true) - self::$timers[$label];
        error_log("$label: {$elapsed}s");
    }
}
```

Common Culprits:

- N+1 query problems - use eager loading
- Missing database indexes - check query execution plans
- External API calls without timeouts or caching
- Inefficient ORM usage - monitor generated SQL
- Opcode cache issues - verify opcache.enable and opcache.revalidate_freq settings

10. Write a function to implement FizzBuzz with a twist: make it extensible for any number of rules without modifying the core logic.

Extensible FizzBuzz Pattern

Using a rules-based approach with Strategy pattern:

```
class FizzBuzz {
    private $rules = [];

    public function addRule($divisor, $word) {
        $this->rules[] = [$divisor, $word];
        return $this;
    }

    public function generate($n) {
        $result = '';
        foreach ($this->rules as [$divisor, $word]) {
            if ($n % $divisor === 0) $result .= $word;
        }
        return $result ?: (string)$n;
    }
}
```

Usage Example:

```
$fb = new FizzBuzz();
$fb->addRule(3, 'Fizz')->addRule(5, 'Buzz')->addRule(7, 'Bazz');

for ($i = 1; $i <= 15; $i++) {
    echo $fb->generate($i) . PHP_EOL;
}
```

Design Benefits:

- **Open/Closed Principle:** Add new rules without modifying existing code
- **Fluent Interface:** Method chaining with return $this
- Rules can be added dynamically at runtime
- Easy to unit test individual rules
- Could extend to use callable rules for complex conditions