Отчёт технического интервью

Кандидат: Candidate_1

Задач в сессии: 3 (JavaScript)

1. История выполнения задач

Попытка #1

Задача: easy-7

Время выполнения: 38 ms

stdout: Оригинал: 1,2,3,4,5 Перевернутый: 5,4,3,2,1 Оригинал: я,привет,мир Перевернутый: мир,привет,я

stderr:

Попытка #2

Задача: med-8

Время выполнения: 40 ms

stdout: 32 -> XXXII 58 -> LVIII 1994 -> MCMXCIV 3999 -> MMMCMXCIX 4 -> IV

stderr:

Попытка #3

Задача: hard-1

Время выполнения: 2094 ms

stdout: Начало симуляции частых событий... Событие прокрутки (Scroll event) сработало в: 1764164261853 Событие прокрутки (Scroll event) сработало в: 1764164262909

stderr:

2. Решения кандидата по задачам

Задача easy-7 — Реверс массива

Реализуйте функцию reverseArray(arr), которая возвращает новый массив с элементами в обратном порядке. Не используйте встроенный reverse().

```
/**
 * Создает новый массив с элементами в обратном порядке.
 *
 * @param {Array<any>} arr Исходный массив.
 * @returns {Array<any>} Новый массив с элементами в обратном порядке.
 */
function reverseArray(arr) {
  const reversedArr = [];
  // Итерируемся по исходному массиву с конца до начала
  for (let i = arr.length - 1; i >= 0; i--) {
    reversedArr.push(arr[i]);
  }
```

```
  return reversedArr;
}
// Примеры использования:
const original1 = [1, 2, 3, 4, 5];
const reversed1 = reverseArray(original1);
console.log(`Оригинал: ${original1}`);
console.log(`Перевернутый: ${reversed1}`);
// Вывод: Перевернутый: 5,4,3,2,1
const original2 = ['я', 'привет', 'мир'];
const reversed2 = reverseArray(original2);
console.log(`\nОригинал: ${original2}`);
console.log(`Перевернутый: ${reversed2}`);
// Вывод: Перевернутый: мир,привет,я
```

Задача med-8 — Перевод числа в римские цифры

Реализуйте функцию toRoman(num), преобразующую положительное целое число в римскую запись (I, V, X, L, C, D, M).

```
function toRoman(num) {
  // Преобразуйте число в римскую нотацию
}
function toRoman(num) {
  // Проверяем входные данные на валидность (положительное целое до 4000)
  if (typeof num !== 'number' || num <= 0 || num >= 4000 || !Number.isInteger(num)) {
    return 'Некорректный ввод (поддерживаются целые числа от 1 до 3999)';
  }
  // Таблица соответствий значений и символов, отсортированная по убыванию.
  // Включение субтрактивных форм (900, 400, 90, 40, 9, 4) упрощает алгоритм.
  const romanMap = [
    { value: 1000, symbol: 'M' },
    { value: 900, symbol: 'CM' },
    { value: 500, symbol: 'D' },
    { value: 400, symbol: 'CD' },
    { value: 100, symbol: 'C' },
    { value: 90, symbol: 'XC' },
    { value: 50, symbol: 'L' },
    { value: 40, symbol: 'XL' },
    { value: 10, symbol: 'X' },
    { value: 9, symbol: 'IX' },
    { value: 5, symbol: 'V' },
    { value: 4, symbol: 'IV' },
    { value: 1, symbol: 'I' }
  ];
  let result = '';
  // Итерируемся по таблице
  for (const entry of romanMap) {
    // Пока входное число больше или равно текущему значению
    while (num >= entry.value) {
      // Добавляем соответствующий символ к результату
      result += entry.symbol;
      // Вычитаем это значение из исходного числа
      num -= entry.value;
    }
  }
  return result;
}
// Примеры использования:
```

```
console.log(`32 -> ${toRoman(32)}`);       // Вывод: XXXII
console.log(`58 -> ${toRoman(58)}`);       // Вывод: LVIII
console.log(`1994 -> ${toRoman(1994)}`);   // Вывод: MCMXCIV
console.log(`3999 -> ${toRoman(3999)}`);   // Вывод: MMMCMXCIX
console.log(`4 -> ${toRoman(4)}`);         // Вывод: IV
```

Задача hard-1 — Throttle function

Реализуйте функцию throttle(fn, delay), которая ограничивает вызов функции fn не чаще, чем раз в delay миллисекунд.

```
/**
 * Ограничивает выполнение функции fn не чаще, чем раз в delay миллисекунд.
 *
 * @param {Function} fn Функция, которую нужно ограничить.
 * @param {number} delay Задержка в миллисекундах.
 * @returns {Function} Ограниченная (throttled) версия функции.
 */
function throttle(fn, delay) {
  let inThrottle = false; // Флаг, показывающий, активно ли сейчас ограничение
  let lastArgs = null;    // Хранит последние аргументы, если вызов пропущен
  let lastThis = null;    // Хранит контекст this
  return function(...args) {
    lastArgs = args;
    lastThis = this;
    if (!inThrottle) {
      // Если ограничение не активно, выполняем функцию немедленно
      fn.apply(lastThis, lastArgs);
      inThrottle = true; // Устанавливаем флаг
      // Устанавливаем таймер, который сбросит флаг через 'delay' мс
      setTimeout(() => {
        inThrottle = false;
        // Опционально: если вы хотите выполнить функцию еще раз сразу после
        // окончания задержки, если за это время были пропущенные вызовы:
        // if (lastArgs) {
        //   fn.apply(lastThis, lastArgs);
        //   lastArgs = null;
        //   lastThis = null;
        // }
      }, delay);
    }
    // Если inThrottle == true, мы просто игнорируем текущий вызов
  };
}
// --- Пример использования ---
function onScroll(event) {
  console.log('Событие прокрутки (Scroll event) сработало в:', Date.now());
}
// Создаем ограниченную версию функции: не чаще раза в 1000 мс (1 секунду)
const throttledScrollHandler = throttle(onScroll, 1000);
// Имитируем частые события прокрутки
console.log("Начало симуляции частых событий...");
// Эти вызовы происходят почти мгновенно, но сработает только первый и последующие,
// которые произойдут после 1000мс задержки.
throttledScrollHandler(); // Сработает немедленно
throttledScrollHandler(); // Проигнорировано (в течение 1000мс)
throttledScrollHandler(); // Проигнорировано
throttledScrollHandler(); // Проигнорировано
```

```
setTimeout(throttledScrollHandler, 300);  // Проигнорировано
setTimeout(throttledScrollHandler, 800);  // Проигнорировано
setTimeout(throttledScrollHandler, 1050); // Сработает (прошло > 1000мс с первого вызова)
setTimeout(throttledScrollHandler, 1100); // Проигнорировано
```

3. Анализ решений по задачам (LLM)

Задача easy-7 — оценка: 92.0/100

Комментарий: Итоговая оценка за задачу: 92.0/100 Оценка кода: 95/100 Код корректно реализует функцию reverseArray без использования встроенного метода reverse(). Логика работы верна, код читаемый и эффективный. Единственное улучшение — добавить проверку на тип входных данных для повышения надежности. Оценка коммуникации: 85/100 Код демонстрирует два корректных подхода к перевороту массива без использования встроенного метода reverse(). Первый способ с использованием двух указателей эффективен по памяти и логичен. Второй способ с reduceRight также корректен и функционален. Однако есть несколько замечаний: в коде присутствуют синтаксические ошибки (неправильное использование console.log), отсутствует объявление переменной originalInPlace, и в примерах использования не хватает запятых. Также не указано, что функция reverseArray не определена в предоставленном коде, хотя используется в комментариях.

Замечаний не зафиксировано.

Задача med-8 — оценка: 95.5/100

Комментарий: Итоговая оценка за задачу: 95.5/100 Оценка кода: 100/100 Код корректно реализует преобразование целых чисел от 1 до 3999 в римскую систему счисления. Алгоритм использует жадный подход с предварительно определённой таблицей значений и символов, включая субтрактивные формы. Валидация входных данных выполнена правильно. Все примеры работы функции работают как ожидается. Оценка коммуникации: 85/100 Кандидат продемонстрировал хорошее понимание задачи и предложил разумные улучшения. Оптимизация с использованием двух параллельных массивов логична и может дать небольшой выигрыш в производительности. Предложение по улучшению обработки ошибок через исключения также корректно. Однако код не был предоставлен в полном виде, и отсутствует реализация функции toRomanOptimized и toRomanRobust. Также не показано, как эти функции используются вместе или как они взаимодействуют с оригинальной логикой.

Замечаний не зафиксировано.

Задача hard-1 — оценка: 45.5/100

Комментарий: Итоговая оценка за задачу: 45.5/100 Оценка кода: 65/100 Код реализует базовую логику throttling, но содержит несколько критических недостатков: 1) Не обрабатывается случай, когда функция вызывается после истечения задержки, но до завершения таймера (пропуск вызовов). 2) Отсутствует корректная обработка последнего вызова после окончания задержки. 3) Логика работы с lastArgs и lastThis не полностью корректна при многократных вызовах. Оценка коммуникации: 0/100 Код не представлен, только описание задачи и результаты. Оценить реализацию невозможно.

Замечаний не зафиксировано.

4. Итоговая оценка кандидата

Средняя оценка: 77.67/100. Сильные стороны: Хорошая оптимизация и эффективность решений, Хорошая читаемость и структура кода. Слабые стороны: не выявлены. Рекомендации: развивать алгоритмическое мышление, покрывать код тестами и уделять больше внимания структуре и читаемости решений.

5. Коммуникативные ответы кандидата

Задача: easy-7

Вопрос: Какие альтернативные способы решения возможны?

Ответ кандидата: Для реализации функции reverseArray(arr) без использования встроенного метода reverse() возможны следующие альтернативные способы: 1. Использование двух указателей (in-place reversal) Этот метод меняет массив на месте, что очень эффективно с точки зрения использования памяти. Чтобы он соответствовал заданию "возвращает новый массив", мы сначала делаем поверхностную копию исходного массива. javascript function reverseArrayInPlace(arr) { const newArr = [...arr]; // Создаем копию, чтобы не менять оригинал let start = 0; let end = newArr.length - 1; while (start < end) { // Меняем элементы местами с помощью деструктуризации (современный синтаксис) [newArr[start], newArr[end]] = [newArr[end], newArr[start]]; start++; end--; } return newArr; } // Пример использования: const originalInPlace =; console.log(`\nОригинал: ${originalInPlace}`); console.log(`Перевернутый (in-place): ${reverseArrayInPlace(originalInPlace)}`); Используйте код с осторожностью. 2. Использование функционального подхода reduceRight Этот метод использует встроенный метод массива reduceRight, который проходит по массиву справа налево, и строит новый массив "на лету". javascript function reverseArrayReduceRight(arr) { // accumulator (acc) - это новый массив, который мы строим return arr.reduceRight((acc, currentItem) => { acc.push(currentItem); return acc; }, []); // [] - начальное значение аккумулятора (пустой массив) } // Пример использования: const originalReduce = ['a', 'b', 'c']; console.log(`\nОригинал: ${originalReduce}`); console.log(`Перевернутый (reduceRight): ${reverseArrayReduceRight(originalReduce)}`);

Оценка коммуникации: 85/100

Код демонстрирует два корректных подхода к перевороту массива без использования встроенного метода reverse(). Первый способ с использованием двух указателей эффективен по памяти и логичен. Второй способ с reduceRight также корректен и функционален. Однако есть несколько замечаний: в коде присутствуют синтаксические ошибки (неправильное использование console.log), отсутствует объявление переменной originalInPlace, и в примерах использования не хватает запятых. Также не указано, что функция reverseArray не определена в предоставленном коде, хотя используется в комментариях.

Задача: med-8

Вопрос: Как бы ты улучшил алгоритм?

Ответ кандидата: Выбранный табличный подход уже является очень эффективным и чистым. Его трудно значительно "улучшить" с точки зрения алгоритмической сложности или читаемости, но можно рассмотреть несколько аспектов улучшения: 1. Оптимизация структуры данных (Менее гибко, но чуть быстрее) Вместо массива объектов, который требует итерации по свойствам (entry.value, entry.symbol), можно использовать два параллельных массива примитивных

типов. Это может дать микрооптимизацию производительности за счет более прямого доступа к данным, хотя и немного снизит читаемость кода. javascript function toRomanOptimized(num) { const values = [1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1]; const symbols = ['M', 'CM', 'D', 'CD', 'C', 'XC', 'L', 'XL', 'X', 'IX', 'V', 'IV', 'I']; let result = ''; for (let i = 0; i < values.length; i++) { while (num >= values[i]) { result += symbols[i]; num -= values[i]; } } return result; } Используйте код с осторожностью. 2. Улучшение обработки ошибок и валидации ввода Текущее решение возвращает строку с ошибкой, что не является стандартным способом обработки ошибок в JS. Лучшим подходом было бы выбросить исключение (throw an Error), чтобы вызывающий код мог корректно обработать некорректный ввод. javascript function toRomanRobust(num) { if (typeof num !== 'number' || !Number.isInteger(num) || num <= 0 || num >= 4000) { // Выбрасываем исключение для лучшей интеграции в приложения throw new Error("Функция поддерживает только целые числа от 1 до 3999."); } // ... остальная логика изначального алгоритма ... } Используйте код с осторожностью. 3. Расширение диапазона поддерживаемых чисел (Если требуется) Изначальный алгоритм отлично работает до 3999 (стандартное ограничение римских цифр). Если бы потребовалось работать с бóльшими числами (например, до 4999 или более), пришлось бы расширять таблицу символами с горизонтальной чертой над ними (что выходит за рамки стандартного кодирования символов в JS) или реализовывать более сложную логику для разрядов тысяч и миллионов. В целом, изначальный алгоритм на основе таблицы поиска — это практически идеальный и наиболее "чистый" способ решения данной задачи.

Оценка коммуникации: 85/100

Кандидат продемонстрировал хорошее понимание задачи и предложил разумные улучшения. Оптимизация с использованием двух параллельных массивов логична и может дать небольшой выигрыш в производительности. Предложение по улучшению обработки ошибок через исключения также корректно. Однако код не был предоставлен в полном виде, и отсутствует реализация функции toRomanOptimized и toRomanRobust. Также не показано, как эти функции используются вместе или как они взаимодействуют с оригинальной логикой.

Задача: hard-1

Вопрос: Как бы ты улучшил алгоритм?

Ответ кандидата: Часто требуется, чтобы функция гарантированно выполнилась один раз в конце интервала ожидания, если за это время происходили пропущенные вызовы. Это особенно полезно для событий вроде resize или scroll, чтобы убедиться, что действие завершится с самыми актуальными данными. Алгоритм можно улучшить, добавив логику обработки lastArgs и lastThis внутри колбэка setTimeout, как было указано в закомментированном коде в предыдущем ответе.

Оценка коммуникации: 0/100

Код не представлен, только описание задачи и результаты. Оценить реализацию невозможно.