фаза 1 · неделя 2 · день 3

Рекурсия и базовые алгоритмы в JavaScript

ELBRUS/BOOTCAMP

План

1. Введение в рекурсию
2. Базовые алгоритмы с использованием рекурсии
3. Факториал числа
4. Фибоначчи
5. Бинарный поиск
6. Обход дерева
7. Мемоизация

Рекурсия

Концепция, при которой функция вызывает саму себя внутри своего собственного тела. Это позволяет функции решать сложные задачи, разбивая их на более простые подзадачи.

Введение в рекурсию

Рекурсия - это процесс, когда функция вызывает сама себя, напрямую или косвенно, для решения задачи.

- Функция, которая содержит вызов самой себя, является рекурсивной функцией
- Базовый случай — условие, при котором рекурсивный процесс останавливается, и функция возвращает результат без дополнительных вызовов
- Рекурсивный случай — часть функции, в которой вызывается сама функция с "меньшей" версией задачи

Примеры использования рекурсии

```
function factorial(n) {
    // Базовый случай
    if (n === 0 || n === 1) {
        return 1;
    }
    // Рекурсивный случай
    return n * factorial(n - 1);
}
console.log(factorial(5)); // Результат: 120
```

Базовые алгоритмы с рекурсией: плюсы

Преимущества рекурсии:

- Упрощение кода и повышение читаемости: рекурсивные решения могут быть более краткими и интуитивно понятными
- Природа некоторых задач: некоторые задачи, такие как обход деревьев или вычисление чисел Фибоначчи, естественным образом подходят для рекурсивных решений

Базовые алгоритмы с рекурсией: минусы

Недостатки рекурсии:

- Производительность: рекурсия может привести к большому количеству вызовов функций, что может замедлить выполнение кода и увеличить потребление памяти
- Глубина стека вызовов: слишком большая глубина рекурсии может привести к исчерпанию стека вызовов, что вызовет ошибку `"Maximum call stack size exceeded"`

Альтернативы рекурсии

Альтернативы рекурсии:

- Итеративные решения: рекурсивные алгоритмы могут быть заменены итеративными, используя циклы (for, while) и структуры данных, такие как стеки или очереди

Основы работы с рекурсивными алгоритмами

1. Определите базовый случай, который остановит рекурсию
2. Разбейте задачу на более мелкие подзадачи, которые можно решить рекурсивно
3. Проверьте, можно ли оптимизировать рекурсивный алгоритм, используя мемоизацию

Факториал числа

Факториал числа n (обозначается n!) - это произведение всех натуральных чисел от 1 до n включительно. Факториал 0 равен 1.

- Факториал можно вычислить с использованием рекурсивной функции, которая вызывает сама себя с уменьшенным аргументом
- Базовый случай: факториал 0 равен 1
- Рекурсивный случай: факториал n равен произведению n на факториал (n - 1)

Последовательность Фибоначчи

Последовательность Фибоначчи - это числовая последовательность, в которой каждое число равно сумме двух предыдущих чисел. Первые два числа равны 0 и 1.

- Числа Фибоначчи можно вычислить с использованием рекурсивной функции, которая вызывает сама себя с уменьшенными аргументами
- Базовый случай: Фибоначчи(0) равно 0, Фибоначчи(1) равно 1
- Рекурсивный случай: Фибоначчи(n) равно Фибоначчи(n - 1) + Фибоначчи(n - 2)

Бинарный поиск

Бинарный поиск - это алгоритм поиска, который эффективно находит искомый элемент в отсортированном массиве, делая примерно $\log_2(n)$ сравнений, где n - количество элементов в массиве

- Бинарный поиск может быть реализован с помощью рекурсивной функции, которая сокращает массив пополам на каждом шаге
- Базовый случай: искомый элемент найден или массив пуст (элемент не найден)
- Рекурсивный случай: вызов функции с новыми границами массива, исключая половину элементов

Обход дерева

Обход дерева - это алгоритм поиска или обработки элементов в структуре данных "дерево", при котором каждый узел дерева посещается в определенном порядке

- Обход дерева может быть реализован с помощью рекурсивной функции, которая вызывает сама себя для обработки дочерних узлов дерева
- Базовый случай: текущий узел пуст (не существует)
- Рекурсивный случай: вызов функции для обработки дочерних узлов текущего узла

Мемоизация

Мемоизация - это техника оптимизации, при которой результаты вызовов функций с определенными аргументами сохраняются в кэше для последующего использования. Это уменьшает количество повторных вычислений, особенно полезно для рекурсивных функций.

- Мемоизация может существенно улучшить производительность рекурсивных функций, так как она предотвращает повторное вычисление уже полученных результатов

Пример функции мемоизации

```
function memoizedFactorial(n, cache = {}) {
    if (n === 0 || n === 1) return 1;
    if (cache[n]) return cache[n];
    const result = n * memoizedFactorial(n - 1, cache);
    cache[n] = result;
    return result;
}
```