Частное учреждение образования

«Колледж бизнеса и права»

|  |
| --- |
| УТВЕРЖДАЮ  Заведующий  методическим кабинетом  \_\_\_\_\_\_\_\_\_\_Е.В.Паскал  «\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_ |

|  |  |
| --- | --- |
| Специальность:  2-40 01 01 «Программное обеспечение информационных технологий» | Дисциплина: «Скриптовые языки программирования» |

**Лабораторная работа № 30**

**Инструкционно-технологическая карта**

Тема: Рекурсия и стек

Цель: Научиться разрабатывать скрипты с использованием рекурсии и стека.

Время выполнения: 2 часа

# Порядок выполнения работы

1. Изучить теоретические сведения к лабораторной работе.
2. Реализовать алгоритм решения задачи.
3. Отлаженную, работающую программу сдать преподавателю. Работу программы показать с помощью самостоятельно разработанных тестов.
4. Ответить на контрольные вопросы.

# Теоретические сведения

Рекурсия – это приём программирования, полезный в ситуациях, когда задача может быть естественно разделена на несколько аналогичных, но более простых задач. Или, когда задача может быть упрощена до несложных действий плюс простой вариант той же задачи. Или для работы с определёнными структурами данных.

В процессе выполнения задачи в теле функции могут быть вызваны другие функции для выполнения подзадач. Частный случай подвызова – когда функция вызывает сама себя. Это как раз и называется рекурсией.

## Два способа мышления

В качестве первого примера будет приведена функция **pow(x, n)**, которая возводит **x** в натуральную степень **n**. Иначе говоря, умножает **x** на само себя **n** раз.

Приведено два способа её реализации.

Итеративный способ: цикл **for**:

function pow(x, n) {

    let result = 1;

    // умножение result на x n раз в цикле

    for (let i = 0; i < n; i++) {

        result \*= x;

    }

    return result;

}

alert(pow(2, 3)); // 8

Рекурсивный способ: упрощение задачи и вызов функцией самой себя:

function pow(x, n) {

    if (n == 1) {

        return x;

    } else {

        return x \* pow(x, n - 1);

    }

}

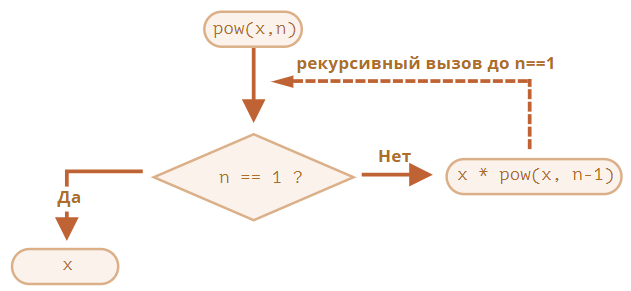
alert(pow(2, 3)); // 8

Когда функция **pow(x, n)** вызывается, исполнение делится на две ветви:

Если **n == 1**, тогда всё просто. Эта ветвь называется базой рекурсии, потому что сразу же приводит к очевидному результату: **pow(x, 1)** равно x.

Можно представить **pow(x, n)** в виде: **x \* pow(x, n - 1)**. Что в математике записывается как: **xn = x \* xn-1**. Эта ветвь – шаг рекурсии: задача сводится к более простому действию (умножение на x) и более простой аналогичной задаче (pow с меньшим n). Последующие шаги упрощают задачу всё больше и больше, пока n не достигает 1.

В таких случаях говорят, что функция **pow** рекурсивно вызывает саму себя до **n == 1**.



Например, рекурсивный вариант вычисления **pow(2, 4)** состоит из шагов:

1. pow(2, 4) = 2 \* pow(2, 3)
2. pow(2, 3) = 2 \* pow(2, 2)
3. pow(2, 2) = 2 \* pow(2, 1)
4. pow(2, 1) = 2

Итак, рекурсию используют, когда вычисление функции можно свести к её более простому вызову, а его – к ещё более простому и так далее, пока значение не станет очевидно.

Общее количество вложенных вызовов (включая первый) называют глубиной рекурсии. В случае из примера она будет равна n.

Максимальная глубина рекурсии ограничена движком JavaScript. Точно можно рассчитывать на 10000 вложенных вызовов, некоторые интерпретаторы допускают и больше, но для большинства из них 100000 вызовов – за пределами возможностей. Существуют автоматические оптимизации, помогающие избежать переполнения стека вызовов («оптимизация хвостовой рекурсии»), но они ещё не поддерживаются везде и работают только для простых случаев.

Это ограничивает применение рекурсии, но она всё равно широко распространена: для решения большого числа задач рекурсивный способ решения даёт более простой код, который легче поддерживать.

## Контекст выполнения, стек

Информация о процессе выполнения запущенной функции хранится в её контексте выполнения (execution context).

Контекст выполнения – специальная внутренняя структура данных, которая содержит информацию о вызове функции. Она включает в себя конкретное место в коде, на котором находится интерпретатор, локальные переменные функции, значение **this** и прочую служебную информацию.

Один вызов функции имеет ровно один контекст выполнения, связанный с ним.

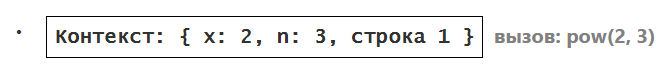
Когда функция производит вложенный вызов, происходит следующее:

1. Выполнение текущей функции приостанавливается.
2. Контекст выполнения, связанный с ней, запоминается в специальной структуре данных – стеке контекстов выполнения.
3. Выполняются вложенные вызовы, для каждого из которых создаётся свой контекст выполнения.
4. После их завершения старый контекст достаётся из стека, и выполнение внешней функции возобновляется с того места, где она была остановлена.

Разберёмся с контекстами более подробно на примере вызова функции **pow(2, 3)**.

В начале вызова **pow(2, 3)** контекст выполнения будет хранить переменные: x = 2, n = 3, выполнение находится на первой строке функции.

Можно схематически изобразить это так:



Это только начало выполнения функции. Условие n == 1 ложно, поэтому выполнение идёт во вторую ветку **if**.

Значения переменных те же самые, но выполнение функции перешло к другой строке, актуальный контекст:



Чтобы вычислить выражение **x \* pow(x, n - 1)**, требуется произвести запуск **pow** с новыми аргументами **pow(2, 2)**.

И так далее.

# Контрольные вопросы

1. Что такое рекурсия?
2. Что такое контекст выполнения?
3. Что хранится в контексте выполнения?
4. Что такое глубина рекурсии?
5. Какая максимальная глубина стека структуры?

# Задания

Реализовать **рекурсивные** функции, решающие поставленную задачу.

|  |  |
| --- | --- |
| Вар | Задание |
|  | Дано число n, десятичная запись которого не содержит нулей. Получить число, записанное теми же цифрами, но в противоположном порядке.  Функция должна возвращать целое число, являющееся результатом работы программы, выводить число по одной цифре нельзя. |
|  | Даны числа a и b. Определить, сколько существует последовательностей из a нулей и b единиц, в которых никакие два нуля не стоят рядом. |
|  | Даны натуральные числа k и s. Определить, сколько существует k-значных натуральных чисел, сумма цифр которых равна d. Запись натурального числа не может начинаться с цифры 0.  В этой задаче можно использовать цикл для перебора всех цифр, стоящих на какой-либо позиции. |
|  | Дана монотонная последовательность, в которой каждое натуральное число k встречается ровно k раз: 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, ...  По данному натуральному n необходимо вывести первые n членов этой последовательности. |
|  | Дана последовательность натуральных чисел, содержащая два числа 0 подряд. Определить, сколько раз в этой последовательности встречается число 1. Числа, идущие после двух нулей, необходимо игнорировать. |
|  | Дана последовательность натуральных чисел, завершающаяся числом 0. Определить, какое количество элементов этой последовательности, равны ее наибольшему элементу. |
|  | Дана последовательность натуральных чисел, завершающаяся числом 0. Определить значение второго по величине элемента в этой последовательности, то есть элемента, который будет наибольшим, если из последовательности удалить наибольший элемент. |
|  | Дана последовательность натуральных чисел, завершающаяся числом 0. Определить среднее значение элементов этой последовательности (без учета последнего нуля). |
|  | Дана последовательность натуральных чисел, завершающаяся числом 0. Определите наибольшее значение числа в этой последовательности. |
|  | Дана последовательность натуральных чисел, завершающаяся числом 0. Вывести первое, третье, пятое и т.д. из введенных чисел. Завершающий ноль выводить не надо. |
|  | Дана последовательность натуральных чисел, завершающаяся числом 0. Вывести все нечетные числа из этой последовательности, сохраняя их порядок. |
|  | Дано слово, состоящее только из строчных латинских букв. Проверить, является ли это слово палиндромом. Вывести YES или NO. |
|  | Дано натуральное число n>1. Вывести все простые делители этого числа в порядке неубывания с учетом кратности. |
|  | Дано натуральное число n>1. Проверить, является ли оно простым. Программа должна вывести слово YES, если число простое и NO, если число составное |
|  | Дано натуральное число N. Вывести слово YES, если число N является точной степенью двойки, или слово NO в противном случае. |

# Литература

1. **Диков, А.В.** Клиентские технологии веб программирования: JavaScript и DOM: учебное пособие / А.В. Диков. – СПб: Лань, 2020 – 124 с.
2. **Читанамбри, Кирупа.** Изучаем React / Кирупа Читанамбри – 2-е изд. – М.: Эксмо, 2019. 368 с.
3. **Хавербеке, Марейн.** Выразительный JavaScript. Современное веб-программирование. / Марейн Хавербеке – 3-е изд. – СПб.: Питер, 2019. – 480 с.:

Преподаватель Рогалевич А.В.

Рассмотрено на заседании цикловой

комиссии программного обеспечения

информационных технологий №10

Протокол № \_\_\_ от \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Председатель ЦК\_\_\_\_\_\_\_\_ В.Ю.Михалевич