

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Два способа мышления

Контекст выполнения, стек

Рекурсивные обходы

Рекурсивные структуры

Итого

Задачи (5)

Комментарии

Поделиться

[Редактировать на GitHub](#)[🏠 → Язык программирования JavaScript](#)
[→ Продвинутая работа с функциями](#) 2-го октября 2020

Рекурсия и стек

Вернёмся к функциям и изучим их более подробно.

Нашей первой темой будет *рекурсия*.

Если вы не новичок в программировании, то, возможно, уже знакомы с рекурсией и можете пропустить эту главу.

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, как мы скоро увидим, 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 напомним функцию `pow(x, n)`, которая возводит `x` в натуральную степень `n`. Иначе говоря, умножает `x` на само себя `n` раз.

```
1 pow(2, 2) = 4
2 pow(2, 3) = 8
3 pow(2, 4) = 16
```



Рассмотрим два способа её реализации.

1. Итеративный способ: цикл `for` :

```
1 function pow(x, n) {
2   let result = 1;
3
4   // умножаем result на x n раз в цикле
5   for (let i = 0; i < n; i++) {
6     result *= x;
7   }
8
9   return result;
10 }
11
12 alert( pow(2, 3) ); // 8
```



2. Рекурсивный способ: упрощение задачи и вызов функцией самой себя:

```
1 function pow(x, n) {
2   if (n == 1) {
3     return x;
4   } else {
5     return x * pow(x, n - 1);
6   }
7 }
8
9 alert( pow(2, 3) ); // 8
```



Обратите внимание, что рекурсивный вариант отличается принципиально.

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

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Два способа мышления

Контекст выполнения, стек

Рекурсивные обходы

Рекурсивные структуры

Итого

Задачи (5)

Комментарии

Поделиться



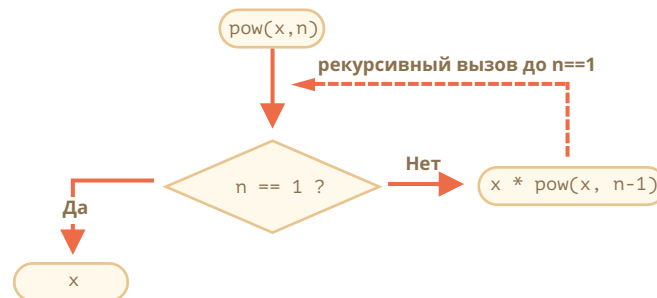
[Редактировать на GitHub](#)

```
1         if n==1 == x
2         /
3 pow(x, n) =
4         \
5         else      = x * pow(x, n - 1)
```



1. Если $n == 1$, тогда всё просто. Эта ветвь называется базой рекурсии, потому что сразу же приводит к очевидному результату: $\text{pow}(x, 1)$ равно x .
2. Мы можем представить $\text{pow}(x, n)$ в виде: $x * \text{pow}(x, n - 1)$. Что в математике записывается как: $x^n = x * x^{n-1}$. Эта ветвь – шаг рекурсии: мы сводим задачу 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.

Говорят, что функция pow рекурсивно вызывает саму себя до $n == 1$.



Например, рекурсивный вариант вычисления $\text{pow}(2, 4)$ состоит из шагов:

1. $\text{pow}(2, 4) = 2 * \text{pow}(2, 3)$
2. $\text{pow}(2, 3) = 2 * \text{pow}(2, 2)$
3. $\text{pow}(2, 2) = 2 * \text{pow}(2, 1)$
4. $\text{pow}(2, 1) = 2$

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.

i Рекурсивное решение обычно короче

Рекурсивное решение задачи обычно короче, чем итеративное.

Используя условный оператор `?` вместо `if`, мы можем переписать $\text{pow}(x, n)$, делая код функции более лаконичным, но всё ещё легко читаемым:

```
1 function pow(x, n) {
2   return (n == 1) ? x : (x * pow(x, n - 1));
3 }
```

Общее количество вложенных вызовов (включая первый) называют глубиной рекурсии. В нашем случае она будет равна ровно 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.

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Два способа мышления

Контекст выполнения, стек

Рекурсивные обходы

Рекурсивные структуры

Итого

Задачи (5)

Комментарии

Поделиться



Редактировать на GitHub



Контекст выполнения, стек

Теперь мы посмотрим, как работают рекурсивные вызовы. Для этого заглянем «под капот» функций.

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` (мы не используем его в данном примере) и прочую служебную информацию.

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

Когда функция производит вложенный вызов, происходит следующее:

- Выполнение текущей функции приостанавливается.
- Контекст выполнения, связанный с ней, запоминается в специальной структуре данных – *стеке контекстов выполнения*.
- Выполняются вложенные вызовы, для каждого из которых создаётся свой контекст выполнения.
- 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)`.

`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:

- **Контекст: { x: 2, n: 3, строка 1 }** вызов: `pow(2, 3)`

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

```
1 function pow(x, n) {
2   if (n == 1) {
3     return x;
4   } else {
5     return x * pow(x, n - 1);
6   }
7 }
8
9 alert( pow(2, 3) );
```

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

- **Контекст: { x: 2, n: 3, строка 5 }** вызов: `pow(2, 3)`

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

`pow(2, 2)`

Для выполнения вложенного вызова JavaScript запоминает текущий контекст выполнения в *стеке контекстов выполнения*.

Здесь мы вызываем ту же функцию `pow`, однако это абсолютно неважно. Для любых функций процесс одинаков:

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Два способа мышления

Контекст выполнения, стек

Рекурсивные обходы

Рекурсивные структуры

Итого

Задачи (5)

Комментарии

Поделиться



[Редактировать на GitHub](#)



1. Текущий контекст «запоминается» на вершине стека.
2. Создаётся новый контекст для вложенного вызова.
3. Когда выполнение вложенного вызова заканчивается – контекст предыдущего вызова восстанавливается, и выполнение соответствующей функции продолжается.

Вид контекста в начале выполнения вложенного вызова `pow(2, 2)`:

- Контекст: { `x: 2`, `n: 2`, строка 1 } вызов: `pow(2, 2)`
- Контекст: { `x: 2`, `n: 3`, строка 5 } вызов: `pow(2, 3)`

Новый контекст выполнения находится на вершине стека (и выделен жирным), а предыдущие запомненные контексты – под ним.

Когда выполнение подвызова закончится, можно будет легко вернуться назад, потому что контекст сохраняет как переменные, так и точное место кода, в котором он остановился. Слово «строка» на рисунках условно, на самом деле запоминается более точное место в цепочке команд.

`pow(2, 1)`

Процесс повторяется: производится новый вызов в строке 5, теперь с аргументами `x=2`, `n=1`.

Создаётся новый контекст выполнения, предыдущий контекст добавляется в стек:

- Контекст: { `x: 2`, `n: 1`, строка 1 } вызов: `pow(2, 1)`
- Контекст: { `x: 2`, `n: 2`, строка 5 } вызов: `pow(2, 2)`
- Контекст: { `x: 2`, `n: 3`, строка 5 } вызов: `pow(2, 3)`

Теперь в стеке два старых контекста и один текущий для `pow(2, 1)`.

Выход

При выполнении `pow(2, 1)`, в отличие от предыдущих запусков, условие `n == 1` истинно, поэтому выполняется первая ветка условия `if`:

```
1 function pow(x, n) {
2   if (n == 1) {
3     return x;
4   } else {
5     return x * pow(x, n - 1);
6   }
7 }
```

Вложенных вызовов больше нет, поэтому функция завершается, возвращая 2.

Когда функция заканчивается, контекст её выполнения больше не нужен, поэтому он удаляется из памяти, а из стека восстанавливается предыдущий:

- Контекст: { `x: 2`, `n: 2`, строка 5 } вызов: `pow(2, 2)`
- Контекст: { `x: 2`, `n: 3`, строка 5 } вызов: `pow(2, 3)`

Возобновляется обработка вызова `pow(2, 2)`. Имея результат `pow(2, 1)`, он может закончить свою работу `x * pow(x, n - 1)`, вернув 4.

Восстанавливается контекст предыдущего вызова:

- Контекст: { `x: 2`, `n: 3`, строка 5 } вызов: `pow(2, 3)`

Самый внешний вызов заканчивает свою работу, его результат: `pow(2, 3) = 8`.

Глубина рекурсии в данном случае составила 3.

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Два способа мышления

Контекст выполнения, стек

Рекурсивные обходы

Рекурсивные структуры

Итого

Задачи (5)

Комментарии

Поделиться



[Редактировать на GitHub](#)



Как видно из иллюстраций выше, глубина рекурсии равна максимальному числу контекстов, одновременно хранимых в стеке.

Обратим внимание на требования к памяти. Рекурсия приводит к хранению всех данных для неоконченных внешних вызовов в стеке, и в данном случае это приводит к тому, что возведение в степень n хранит в памяти n различных контекстов.

Реализация возведения в степень через цикл гораздо более экономна:

```
1 function pow(x, n) {
2   let result = 1;
3
4   for (let i = 0; i < n; i++) {
5     result *= x;
6   }
7
8   return result;
9 }
```

Итеративный вариант функции `pow` использует один контекст, в котором будут последовательно меняться значения `i` и `result`. При этом объём затрачиваемой памяти небольшой, фиксированный и не зависит от `n`.

Любая рекурсия может быть переделана в цикл. Как правило, вариант с циклом будет эффективнее.

Но переделка рекурсии в цикл может быть нетривиальной, особенно когда в функции в зависимости от условий используются различные рекурсивные подвызовы, результаты которых объединяются, или когда ветвление более сложное. Оптимизация может быть ненужной и совершенно нестоящей усилий.

Часто код с использованием рекурсии более короткий, лёгкий для понимания и поддержки. Оптимизация требуется не везде, как правило, нам важен хороший код, поэтому она и используется.

Рекурсивные обходы

Другим отличным применением рекурсии является рекурсивный обход.

Представьте, у нас есть компания. Структура персонала может быть представлена как объект:

```
1 let company = {
2   sales: [{
3     name: 'John',
4     salary: 1000
5   }, {
6     name: 'Alice',
7     salary: 600
8   }],
9
10  development: {
11    sites: [{
12      name: 'Peter',
13      salary: 2000
14    }, {
15      name: 'Alex',
16      salary: 1800
17    }],
18
19    internals: [{
20      name: 'Jack',
21      salary: 1300
22    }]
23  }
24 };
```

Другими словами, в компании есть отделы.

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Два способа мышления

Контекст выполнения, стек

Рекурсивные обходы

Рекурсивные структуры

Итого

Задачи (5)

Комментарии

Поделиться



[Редактировать на GitHub](#)



- Отдел может состоять из массива работников. Например, в отделе sales работают 2 сотрудника: Джон и Алиса.
- Или отдел может быть разделён на подотделы, например, отдел development состоит из подотделов: sites и internals. В каждом подотделе есть свой персонал.
- Также возможно, что при росте подотдела он делится на подразделения (или команды).

Например, подотдел sites в будущем может быть разделён на команды siteA и siteB. И потенциально они могут быть разделены ещё. Этого нет на картинке, просто нужно иметь это в виду.

Теперь, допустим, нам нужна функция для получения суммы всех зарплат. Как мы можем это сделать?

Итеративный подход не прост, потому что структура довольно сложная. Первая идея заключается в том, чтобы сделать цикл for по верху объекта company с вложенным циклом над отделами 1-го уровня вложенности. Но затем нам нужно больше вложенных циклов для итераций над сотрудниками отделов второго уровня, таких как sites ... А затем ещё один цикл по отделам 3-го уровня, которые могут появиться в будущем? Если мы поместим в код 3-4 вложенных цикла для обхода одного объекта, то это будет довольно некрасиво.

Давайте попробуем рекурсию.

Как мы видим, когда наша функция получает отдел для подсчёта суммы зарплат, есть два возможных случая:

1. Либо это «простой» отдел с массивом – тогда мы сможем суммировать зарплаты в простом цикле.
2. Или это объект с N подотделами – тогда мы можем сделать N рекурсивных вызовов, чтобы получить сумму для каждого из подотделов, и объединить результаты.

Случай (1), когда мы получили массив, является базой рекурсии, тривиальным случаем.

Случай (2), при получении объекта, является шагом рекурсии. Сложная задача разделяется на подзадачи для подотделов. Они могут, в свою очередь, снова разделиться на подотделы, но рано или поздно это разделение закончится, и решение сведётся к случаю (1).

Алгоритм даже проще читается в виде кода:

```
1 let company = { // тот же самый объект, сжатый для крат
2   sales: [{name: 'John', salary: 1000}, {name: 'Alice',
3     development: {
4       sites: [{name: 'Peter', salary: 2000}, {name: 'Alex
5         internals: [{name: 'Jack', salary: 1300}]
6       }
7     }];
8
9   // Функция для подсчёта суммы зарплат
10  function sumSalaries(department) {
11    if (Array.isArray(department)) { // случай (1)
12      return department.reduce((prev, current) => prev +
13    } else { // случай (2)
14      let sum = 0;
15      for (let subdep of Object.values(department)) {
16        sum += sumSalaries(subdep); // рекурсивно вызывает
17      }
18      return sum;
19    }
20  }
21
22  alert(sumSalaries(company)); // 6700
```

Код краток и прост для понимания (надеюсь?). В этом сила рекурсии. Она работает на любом уровне вложенности отделов.

Схема вызовов:

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Два способа мышления

Контекст выполнения, стек

Рекурсивные обходы

Рекурсивные структуры

Итого

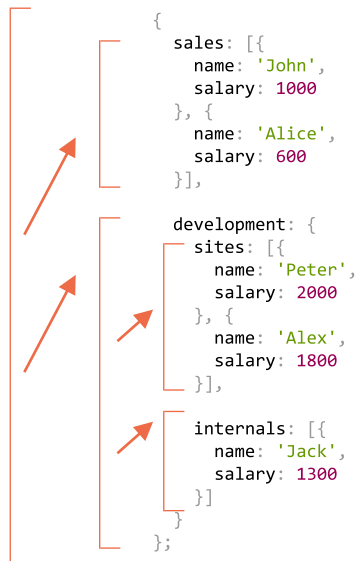
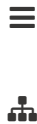
Задачи (5)

Комментарии

Поделиться



[Редактировать на GitHub](#)



Принцип прост: для объекта `{...}` используются рекурсивные вызовы, а массивы `[...]` являются «листьями» дерева рекурсии, они сразу дают результат.

Обратите внимание, что в коде используются возможности, о которых мы говорили ранее:

- Метод `arr.reduce` из главы [Методы массивов](#) для получения суммы элементов массива.
- Цикл `for(val of Object.values(obj))` для итерации по значениям объекта: `Object.values` возвращает массив значений.

Рекурсивные структуры

Рекурсивная (рекурсивно определяемая) структура данных – это структура, которая повторяет саму себя в своих частях.

Мы только что видели это на примере структуры компании выше.

Отдел компании – это:

- Либо массив людей.
- Либо объект с *отделами*.

Для веб-разработчиков существуют гораздо более известные примеры: HTML- и XML-документы.

В HTML-документе *HTML-тег* может содержать:

- Фрагменты текста.
- HTML-комментарии.
- Другие *HTML-теги* (которые, в свою очередь, могут содержать фрагменты текста/комментарии или другие теги и т.д.).

Это снова рекурсивное определение.

Для лучшего понимания мы рассмотрим ещё одну рекурсивную структуру под названием «связанный список», которая в некоторых случаях может использоваться в качестве альтернативы массиву.

Связанный список

Представьте себе, что мы хотим хранить упорядоченный список объектов.

Естественным выбором будет массив:

```
1 let arr = [obj1, obj2, obj3];
```

...Но у массивов есть недостатки. Операции «удалить элемент» и «вставить элемент» являются дорогостоящими. Например, операция `arr.unshift(obj)` должна переиндексировать все элементы, чтобы

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Два способа мышления

Контекст выполнения, стек

Рекурсивные обходы

Рекурсивные структуры

Итого

Задачи (5)

Комментарии

Поделиться



[Редактировать на GitHub](#)



освободить место для нового `obj`, и, если массив большой, на это потребуется время. То же самое с `arr.shift()`.

Единственные структурные изменения, не требующие массовой переиндексации – это изменения, которые выполняются с конца массива: `arr.push/pop`. Таким образом, массив может быть довольно медленным для больших очередей, когда нам приходится работать с его началом.

Или же, если нам действительно нужны быстрые вставка/удаление, мы можем выбрать другую структуру данных, называемую **связанный список**.

Элемент **связанного списка** определяется рекурсивно как объект с:

- `value`,
- `next` – свойство, ссылающееся на следующий элемент **связанного списка** или `null`, если это последний элемент.

Пример:

```
1 let list = {
2   value: 1,
3   next: {
4     value: 2,
5     next: {
6       value: 3,
7       next: {
8         value: 4,
9         next: null
10      }
11    }
12  }
13 };
```

Графическое представление списка:



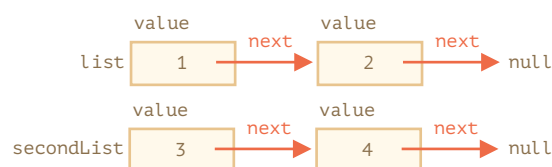
Альтернативный код для создания:

```
1 let list = { value: 1 };
2 list.next = { value: 2 };
3 list.next.next = { value: 3 };
4 list.next.next.next = { value: 4 };
```

Здесь мы можем ещё лучше увидеть, что есть несколько объектов, каждый из которых имеет `value` и `next`, указывающий на соседа. Переменная `list` является первым объектом в цепочке, поэтому, следуя по указателям `next` из неё, мы можем попасть в любой элемент.

Список можно легко разделить на несколько частей и впоследствии объединить обратно:

```
1 let secondList = list.next.next;
2 list.next.next = null;
```



Для объединения:

```
1 list.next.next = secondList;
```


Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Два способа мышления

Контекст выполнения, стек

Рекурсивные обходы

Рекурсивные структуры

Итого

Задачи (5)

Комментарии

Поделиться



Редактировать на GitHub



И, конечно, мы можем вставить или удалить элементы из любого места.

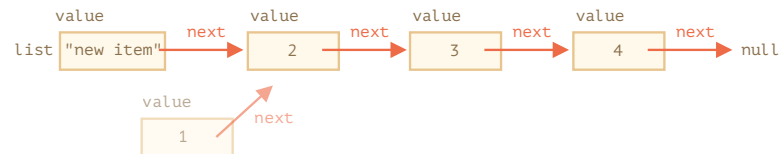
Например, для добавления нового элемента нам нужно обновить первый элемент списка:

```
1 let list = { value: 1 };
2 list.next = { value: 2 };
3 list.next.next = { value: 3 };
4 list.next.next.next = { value: 4 };
5
6 // добавление нового элемента в список
7 list = { value: "new item", next: list };
```



Чтобы удалить элемент из середины списка, нужно изменить значение `next` предыдущего элемента:

```
1 list.next = list.next.next;
```



`list.next` перепрыгнуло с 1 на значение 2. Значение 1 теперь исключено из цепочки. Если оно не хранится где-нибудь ещё, оно будет автоматически удалено из памяти.

В отличие от массивов, нет перенумерации, элементы легко переставляются.

Естественно, списки не всегда лучше массивов. В противном случае все пользовались бы только списками.

Главным недостатком является то, что мы не можем легко получить доступ к элементу по его индексу. В простом массиве: `arr[n]` является прямой ссылкой. Но в списке мы должны начать с первого элемента и перейти в `next` N раз, чтобы получить N-й элемент.

...Но нам не всегда нужны такие операции. Например, нам может быть нужна очередь или даже [двухсторонняя очередь](#) – это упорядоченная структура, которая позволяет очень быстро добавлять/удалять элементы с обоих концов, но там не нужен доступ в середину.

Списки могут быть улучшены:

- Можно добавить свойство `prev` в дополнение к `next` для ссылки на предыдущий элемент, чтобы легко двигаться по списку назад.
- Можно также добавить переменную `tail`, которая будет ссылаться на последний элемент списка (и обновлять её при добавлении/удалении элементов с конца).
- ...Возможны другие изменения: главное, чтобы структура данных соответствовала нашим задачам с точки зрения производительности и удобства.

Итого

Термины:

- *Рекурсия* – это термин в программировании, означающий вызов функцией самой себя. Рекурсивные функции могут быть использованы для элегантного решения определённых задач.

Когда функция вызывает саму себя, это называется *шагом рекурсии*. База рекурсии – это такие аргументы функции, которые делают задачу

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Два способа мышления

Контекст выполнения, стек

Рекурсивные обходы

Рекурсивные структуры

Итого

Задачи (5)

Комментарии

Поделиться



[Редактировать на GitHub](#)



настолько простой, что решение не требует дальнейших вложенных вызовов.

- **Рекурсивно определяемая** структура данных – это структура данных, которая может быть определена с использованием самой себя.

Например, связанный список может быть определён как структура данных, состоящая из объекта, содержащего ссылку на список (или null).

```
1 list = { value, next -> list }
```

Деревья, такие как дерево HTML-элементов или дерево отделов из этой главы, также являются рекурсивными: они разветвляются, и каждая ветвь может содержать другие ветви.

Как мы видели в примере `sumSalary`, рекурсивные функции могут быть использованы для прохода по ним.

Любая рекурсивная функция может быть переписана в итеративную. И это иногда требуется для оптимизации работы. Но для многих задач рекурсивное решение достаточно быстрое и простое в написании и поддержке.

✓ Задачи

Вычислить сумму чисел до данного [↗](#)

важность: 5

Напишите функцию `sumTo(n)`, которая вычисляет сумму чисел $1 + 2 + \dots + n$.

Например:

```
1 sumTo(1) = 1
2 sumTo(2) = 2 + 1 = 3
3 sumTo(3) = 3 + 2 + 1 = 6
4 sumTo(4) = 4 + 3 + 2 + 1 = 10
5 ...
6 sumTo(100) = 100 + 99 + ... + 2 + 1 = 5050
```

Сделайте три варианта решения:

1. С использованием цикла.
2. Через рекурсию, т.к. $\text{sumTo}(n) = n + \text{sumTo}(n-1)$ for $n > 1$.
3. С использованием формулы **арифметической прогрессии**.

Пример работы вашей функции:

```
1 function sumTo(n) { /*... ваш код ... */ }
2
3 alert( sumTo(100) ); // 5050
```

P.S. Какой вариант решения самый быстрый? Самый медленный? Почему?

P.P.S. Можно ли при помощи рекурсии посчитать `sumTo(100000)` ?

решение

Вычислить факториал [↗](#)

важность: 4

Факториал натурального числа – это число, умноженное на "себя минус один", затем на "себя минус два", и так далее до 1. Факториал n обозначается как $n!$

Определение факториала можно записать как:

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Два способа мышления

Контекст выполнения, стек

Рекурсивные обходы

Рекурсивные структуры

Итого

Задачи (5)

Комментарии

Поделиться



[Редактировать на GitHub](#)



```
1 n! = n * (n - 1) * (n - 2) * ...*1
```

Примеры значений для разных n :

```
1 1! = 1
2 2! = 2 * 1 = 2
3 3! = 3 * 2 * 1 = 6
4 4! = 4 * 3 * 2 * 1 = 24
5 5! = 5 * 4 * 3 * 2 * 1 = 120
```

Задача – написать функцию `factorial(n)`, которая возвращает $n!$, используя рекурсию.

```
1 alert( factorial(5) ); // 120
```

P.S. Подсказка: $n!$ можно записать как $n * (n-1)!$ Например: $3! = 3*2! = 3*2*1! = 6$

решение

Числа Фибоначчи

важность: 5

Последовательность [чисел Фибоначчи](#) определяется формулой $F_n = F_{n-1} + F_{n-2}$. То есть, следующее число получается как сумма двух предыдущих.

Первые два числа равны 1, затем $2(1+1)$, затем $3(1+2)$, $5(2+3)$ и так далее: 1, 1, 2, 3, 5, 8, 13, 21....

Числа Фибоначчи тесно связаны с [золотым сечением](#) и множеством природных явлений вокруг нас.

Напишите функцию `fib(n)` которая возвращает n -е число Фибоначчи.

Пример работы:

```
1 function fib(n) { /* ваш код */ }
2
3 alert(fib(3)); // 2
4 alert(fib(7)); // 13
5 alert(fib(77)); // 5527939700884757
```

P.S. Все запуски функций из примера выше должны работать быстро. Вызов `fib(77)` должен занимать не более доли секунды.

решение

Вывод односвязного списка

важность: 5

Допустим, у нас есть односвязный список (как описано в главе [Рекурсия и стек](#)):

```
1 let list = {
2   value: 1,
3   next: {
4     value: 2,
5     next: {
6       value: 3,
7       next: {
8         value: 4,
9         next: null
10      }
11   }
12 }
```

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Два способа мышления

Контекст выполнения, стек

Рекурсивные обходы

Рекурсивные структуры

Итого

Задачи (5)

Комментарии

Поделиться



Редактировать на GitHub

```
11     }  
12   }  
13 };
```



Напишите функцию `printList(list)`, которая выводит элементы списка по одному.



Сделайте два варианта решения: используя цикл и через рекурсию.

Как лучше: с рекурсией или без?

решение

Вывод односвязного списка в обратном порядке

важность: 5

Выведите односвязный список из предыдущего задания [Вывод односвязного списка](#) в обратном порядке.

Сделайте два решения: с использованием цикла и через рекурсию.

решение

Проводим [курсы по JavaScript и фреймворкам](#).



Комментарии

перед тем как писать...

