25 августа 2023 г.

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 напишем функцию `pow(x, n)`, которая возводит `x` в натуральную степень `n`. Иначе говоря, умножает `x` на само себя `n` раз.

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

Рассмотрим два способа её реализации.

1. Итеративный способ: цикл `for`:

```
function pow(x, n) {
  let result = 1;

  // умножаем result на x n раз в цикле
  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}

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

2. 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
```

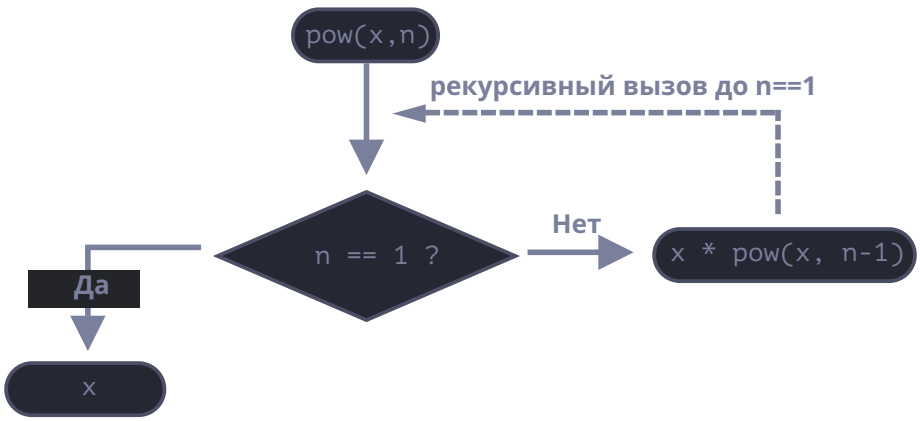
Обратите внимание, что рекурсивный вариант отличается принципиально.

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

```
               if n==1  = x
              /
pow(x, n) =
              \
               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: `pow(x, 1)` равно `x`.
2. Мы можем представить `pow(x, n)` в виде: `x * 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 `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.

Рекурсивное решение обычно короче

Рекурсивное решение задачи обычно короче, чем итеративное.

Используя условный оператор `?` вместо `if`, мы можем переписать `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 `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

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:

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**.

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;
```

```
  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}
```

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:

```
let company = {
  sales: [{
    name: 'John',
    salary: 1000
  }, {
    name: 'Alice',
    salary: 600
  }],

  development: {
    sites: [{
      name: 'Peter',
      salary: 2000
    }, {
      name: 'Alex',
      salary: 1800
    }],

    internals: [{
      name: 'Jack',
      salary: 1300
    }]
  }
};
```

Другими словами, в компании есть отделы.

- 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` поверх 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:

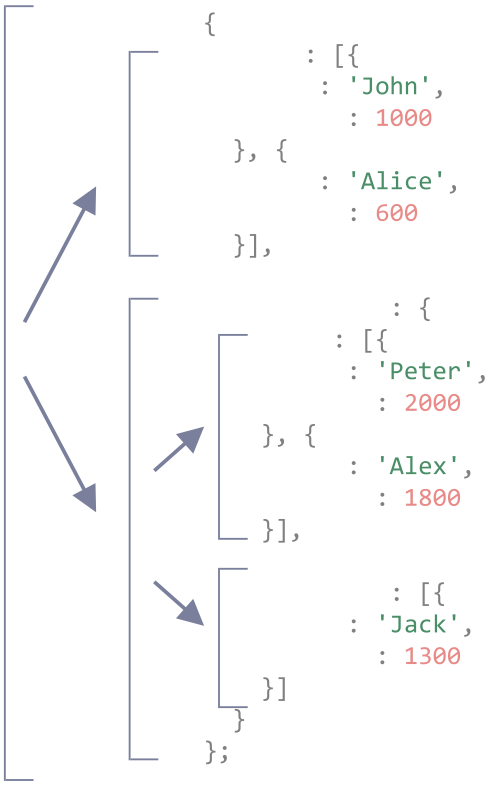
```
let company = { // тот же самый объект, сжатый для краткости
  sales: [{name: 'John', salary: 1000}, {name: 'Alice', salary: 600 }],
  development: {
    sites: [{name: 'Peter', salary: 2000}, {name: 'Alex', salary: 1800 }],
    internals: [{name: 'Jack', salary: 1300}]
  }
};

// Функция для подсчёта суммы зарплат
function sumSalaries(department) {
  if (Array.isArray(department)) { // случай (1)
    return department.reduce((prev, current) => prev + current.salary, 0); //
  } else { // случай (2)
    let sum = 0;
    for (let subdep of Object.values(department)) {
      sum += sumSalaries(subdep); // рекурсивно вызывается для подотделов, сум
    }
    return sum;
  }
}

alert(sumSalaries(company)); // 6700
```

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

Схема вызовов:



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:

```
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 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.

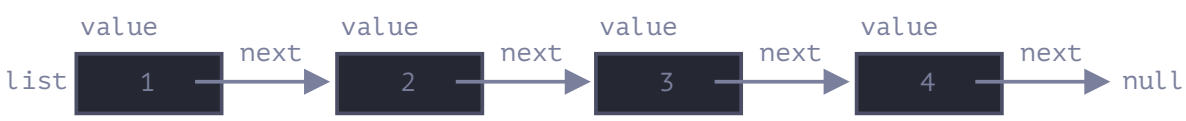
Элемент *связанного списка* определяется рекурсивно как объект с:

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

Пример:

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

Графическое представление списка:



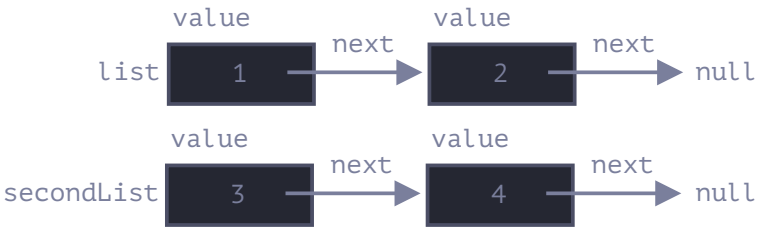
Альтернативный код для создания:

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

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:

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



Для объединения:

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

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

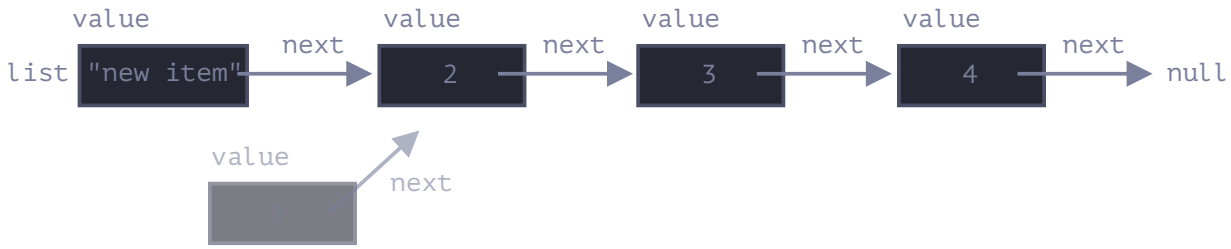
Например, для добавления нового элемента нам нужно обновить первый элемент списка:

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

// добавление нового элемента в список
list = { value: "new item", next: list };
```

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

```
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 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).

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

  Деревья, такие как дерево HTML-элементов или дерево отделов из этой главы, также являются рекурсивными: у них есть ветви, 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 + ... + n`.

Например:

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

Сделайте три варианта решения:

1. С использованием цикла.
2. Через рекурсию, т.к. `sumTo(n) = n + sumTo(n-1)` for `n > 1`.
3. С использованием формулы арифметической прогрессии.

Пример работы вашей функции:

```
function sumTo(n) { /*... ваш код ... */ }

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

Решение с помощью цикла:

```
function sumTo(n) {
  let sum = 0;
  for (let i = 1; i <= n; i++) {
    sum += i;
  }
  return sum;
}

```

```
9
alert( sumTo(100) );
```

Решение через рекурсию:

```
1  function sumTo(n) {
2    if (n == 1) return 1;
3    return n + sumTo(n - 1);
4  }
5
6  alert( sumTo(100) );
```

Решение по формуле: `sumTo(n) = n*(n+1)/2` :

```
1  function sumTo(n) {
2    return n * (n + 1) / 2;
3  }
4
5  alert( sumTo(100) );
```

P.S. Надо ли говорить, что решение по формуле работает быстрее всех? Это очевидно. Оно использует всего три операции для любого n, а цикл и рекурсия требуют как минимум n операций сложения.

Вариант с циклом – второй по скорости. Он быстрее рекурсии, так как операций сложения столько же, но нет дополнительных вычислительных затрат на организацию вложенных вызовов. Поэтому рекурсия в данном случае работает медленнее всех.

P.P.S. Некоторые движки поддерживают оптимизацию «хвостового вызова»: если рекурсивный вызов является самым последним в функции, без каких-либо других вычислений, то внешней функции не нужно будет возобновлять выполнение и не нужно запоминать контекст его выполнения. В итоге требования к памяти снижаются. Но если JavaScript-движок не поддерживает это (большинство не поддерживают), будет ошибка: максимальный размер стека превышен, так как обычно существует ограничение на максимальный размер стека.

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:

```
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

По определению факториал `n!` можно записать как `n * (n-1)!`.

Другими словами, `factorial(n)` можно получить как `n` умноженное на результат `factorial(n-1)`. И результат для `n-1`, в свою очередь, может быть вычислен рекурсивно и так далее до `1`.

```
1  function factorial(n) {
2    return (n != 1) ? n * factorial(n - 1) : 1;
3  }
4
5  alert( factorial(5) ); // 120
```

Базисом рекурсии является значение `1`. А можно было бы сделать базисом и `0`, однако это добавило рекурсии дополнительный шаг:

```
1  function factorial(n) {
2    return n ? n * factorial(n - 1) : 1;
3  }
4
5  alert( factorial(5) ); // 120
```

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-e` число Фибоначчи.

Пример работы:

```
function fib(n) { /* ваш код */ }

alert(fib(3)); // 2
alert(fib(7)); // 13
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

Сначала решим через рекурсию.

Числа Фибоначчи рекурсивны по определению:

```
function fib(n) {
  return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}

alert( fib(3) ); // 2
alert( fib(7) ); // 13
// fib(77); // вычисляется очень долго
```

При больших значениях `n` такое решение будет работать очень долго. Например, `fib(77)` может повесить браузер на некоторое время, съев все ресурсы процессора.

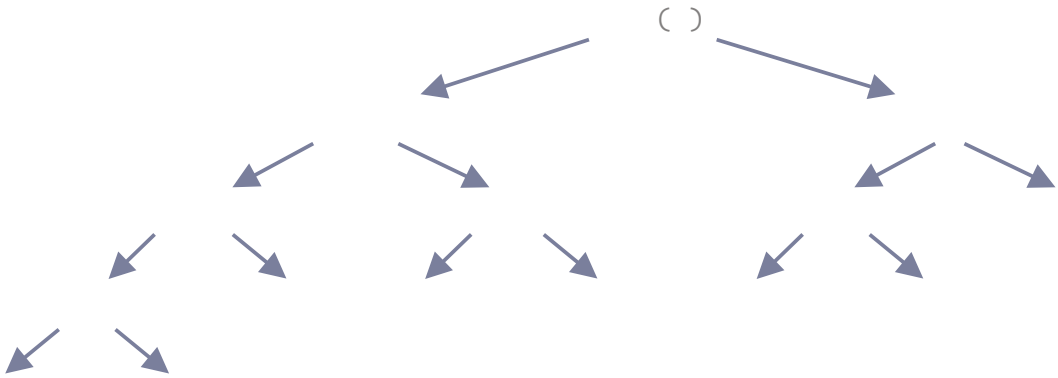
Это потому, что функция порождает обширное дерево вложенных вызовов. При этом ряд значений вычисляется много раз снова и снова.

Например, посмотрим на отрывок вычислений для `fib(5)`:

```
...
fib(5) = fib(4) + fib(3)
fib(4) = fib(3) + fib(2)
...
```

Здесь видно, что значение `fib(3)` нужно одновременно и для `fib(5)` и для `fib(4)`. В коде оно будет вычислено два раза, совершенно независимо.

Полное дерево рекурсии:



Можно заметить, что `fib(3)` вычисляется дважды, а `fib(2)` – трижды. Общее количество вычислений растёт намного быстрее, чем `n`, что делает его огромным даже для `n=77`.

Можно это оптимизировать, запоминая уже вычисленные значения: если значение, скажем, `fib(3)` вычислено однажды, затем мы просто переиспользуем это значение для последующих вычислений.

Другим вариантом было бы отказаться от рекурсии и использовать совершенно другой алгоритм на основе цикла.

Вместо того, чтобы начинать с `n` и вычислять необходимые предыдущие значения, можно написать цикл, который начнёт с `1` и `2`, затем из них получит `fib(3)` как их сумму, затем `fib(4)` как сумму предыдущих значений, затем `fib(5)` и так далее, до финального результата. На каждом шаге нам нужно помнить только значения двух предыдущих чисел последовательности.

Вот детальные шаги нового алгоритма.

Начало:

```
// a = fib(1), b = fib(2), эти значения по определению равны 1
let a = 1, b = 1;

// получим c = fib(3) как их сумму
let c = a + b;

/* теперь у нас есть fib(1), fib(2), fib(3)
a  b  c
1, 1, 2
*/
```

Теперь мы хотим получить `fib(4) = fib(2) + fib(3)`.

Переставим переменные: `a,b`, присвоим значения `fib(2),fib(3)`, тогда `c` можно получить как их сумму:

```
a = b; // теперь a = fib(2)
b = c; // теперь b = fib(3)
c = a + b; // c = fib(4)

/* имеем последовательность:
   a  b  c
1, 1, 2, 3
*/
```

Следующий шаг даёт новое число последовательности:

```
a = b; // now a = fib(3)
b = c; // now b = fib(4)
c = a + b; // c = fib(5)

/* последовательность теперь (на одно число больше):
      a  b  c
1, 1, 2, 3, 5
*/
```

...И так далее, пока не получим искомое значение. Это намного быстрее рекурсии и не требует повторных вычислений.

Полный код:

```
function fib(n) {
  let a = 1;
  let b = 1;
  for (let i = 3; i <= n; i++) {
    let c = a + b;
    a = b;
    b = c;
  }
  return b;
}

alert( fib(3) ); // 2
alert( fib(7) ); // 13
alert( fib(77) ); // 5527939700884757
```

Цикл начинается с `i=3`, потому что первое и второе значения последовательности заданы `a=1`, `b=1`.

Такой способ называется динамическое программирование снизу вверх.

Вывод односвязного списка

важность: 5

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

```
let list = {
  value: 1,
  next: {
    value: 2,
    next: {
      value: 3,
      next: {
        value: 4,
        next: null
```

```
      }
    }
  }
};
```

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

Решение с использованием цикла

Решение с использованием цикла:

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

function printList(list) {
  let tmp = list;

  while (tmp) {
    alert(tmp.value);
    tmp = tmp.next;
  }

}

printList(list);
```

Обратите внимание, что мы используем временную переменную `tmp` для перемещения по списку. Технически, мы могли бы использовать параметр функции `list` вместо неё:

```
function printList(list) {

  while(list) {
    alert(list.value);
    list = list.next;
  }

```

```
8
  }
```

…Но это было бы неблагоразумно. В будущем нам может понадобиться расширить функцию, сделать что-нибудь ещё со списком. Если мы меняем `list`, то теряем такую возможность.

Говоря о хороших именах для переменных, `list` здесь – это сам список, его первый элемент. Так и должно быть, это просто и понятно.

С другой стороны, `tmp` используется исключительно для обхода списка, как `i` в цикле `for`.

Решение через рекурсию

Рекурсивный вариант `printList(list)` следует простой логике: для вывода списка мы должны вывести текущий `list`, затем сделать то же самое для `list.next`:

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

function printList(list) {

  alert(list.value); // выводим текущий элемент

  if (list.next) {
    printList(list.next); // делаем то же самое для остальной части сп
  }

}

printList(list);
```

Какой способ лучше?

Технически, способ с циклом более эффективный. В обеих реализациях делается то же самое, но для цикла не тратятся ресурсы для вложенных вызовов.

С другой стороны, рекурсивный вариант более короткий и, возможно, более простой для понимания.

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

С использованием рекурсии

Рекурсивная логика в этом случае немного сложнее.

Сначала надо вывести оставшуюся часть списка, а *затем* текущий элемент:

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

function printReverseList(list) {

  if (list.next) {
    printReverseList(list.next);
  }

  alert(list.value);
}

printReverseList(list);
```

С использованием цикла

Вариант с использованием цикла сложнее, чем в предыдущей задаче.

Нет способа сразу получить последнее значение в списке `list`. Мы также не можем «вернуться назад», к предыдущему элементу списка.

Поэтому мы можем сначала перебрать элементы в прямом порядке и запомнить их в массиве, а затем вывести то, что мы запомнили, в обратном порядке:

```
let list = {
  value: 1,
  next: {
    value: 2,
    next: {
```

```
      value: 3,
      next: {
        value: 4,
        next: null
      }
    }
  }
};

function printReverseList(list) {
  let arr = [];
  let tmp = list;

  while (tmp) {
    arr.push(tmp.value);
    tmp = tmp.next;
  }

  for (let i = arr.length - 1; i >= 0; i--) {
    alert( arr[i] );
  }
}

printReverseList(list);
```

Обратите внимание, что рекурсивное решение на самом деле делает то же самое: проходит список, запоминает элементы в цепочке вложенных вызовов (в контексте выполнения), а затем выводит их.

Предыдущий урок | Следующий урок

Поделиться

Карта учебника

Комментарии

- Если вам кажется, что в статье что-то не так - вместо комментария напишите на GitHub.
- Для одной строки кода используйте тег `<code>`, для нескольких строк кода — тег `<pre>`, если больше 10 строк — ссылку на песочницу (plnkr, JSBin, codepen...)
- Если что-то непонятно в статье — пишите, что именно и с какого места.

Джун на фронте

месяц назад

Вы уже освоили JavaScript — что же делать?

Building career — следующий шаг. Я рассказываю о своем пути в IT: от zero до получения первого job offer'a, и это уже 700 days откровений и реального experience!

Мне удалось поработать верстальщиком, editor в одном из аналогов Хабра, а сейчас я активно прохожу interviews на позицию frontend-developer'a.

Приглашаю вас в мой **telegram-channel «Джун на front»**, где мы вместе отправим отклики, выполним тестовые и найдем нашу первую работу!

0 1 Ответить • Поделиться ›

frontforme

2 месяца назад

Изучаете теорию и стремитесь стать веб-разработчиком?!

В своем блоге я делюсь ценной информацией о **самом быстром пути к получению должности фронтенд-разработчика** в крупной компании. Давайте знакомится!

👤Кто я?
Я автор этого канала, **Frontend-специалист**, отвечающий за сервис в известной компании.

Айда к нам!
тг: @davay_v_frontend

0 0 Ответить • Поделиться ›

Neysel

4 месяца назад

0 понимания

0 0 Ответить • Поделиться ›

Д

Дмитрий Иванов Neysel

2 месяца назад

Включи отладчик и прочитай про стеки внимательно. Но это не поможет). Рекурсию нужно принимать

как должное, работает и работает. :)

1 0 Ответить • Поделиться ›

Dedal

4 месяца назад

Альтернативное решение задачи "Числа Фибоначчи" с использованием массива и метода at(), для получения 2-х последних значений при вычисления следующего. Работает быстро со значениями из примера.

```
function fib(n) {
  // создаём начальную последовательность
  // чисел Фибоначчи
  let arr = [1, 1, 2];
  // заполняем последовательность
  // до искомого числа
  while (arr.length != n) {
    arr.push(arr.at(-1) + arr.at(-2));
  }
  return arr[n - 1];
}

alert(fib(3)); // 2
alert(fib(7)); // 13
alert(fib(77)); // 5527939700884757
```

1 0 Ответить • Поделиться ›

Climat Channel → Dedal

4 месяца назад

Задумка интересная, плюсую, только завтыкали с условием в цикле, с таким условием может 'пушиться' бесконечный массив. Нужно было просто '< n'.

1 0 Ответить • Поделиться ›

Dedal → Climat Channel

4 месяца назад

Согласен, условие '< n' будет лучше, но не могу представить ситуацию в данной задаче при каком значении n мог бы получится бесконечный массив.

0 0 Ответить • Поделиться ›

Climat Channel → Dedal

4 месяца назад

В вашем примере это будет fib(2), почему, вы наверняка разберетесь :)

2 0 Ответить • Поделиться ›

Dedal → Climat Channel

4 месяца назад

Точно, спасибо за наводку, да при таком раскладе при условии "!= n" цикл будет бесконечен.

0 0 Ответить • Поделиться ›

A **Anatolii Tarasov (Tarasov.Fron**

5 месяцев назад edited

Я так, чисто себя похвалить)) Проход по дереву компании с суммированием значений зарплат

```
const sumSalaries = (depart) => {
  if (Array.isArray(depart)) {
    return depart.reduce((acc, el) => acc += el.salary,0)
  } else {
    return Object.values(depart).reduce((acc, subDep) => acc += sumSalaries(subDep), 0);
  }
}
```

0 0 Ответить • Поделиться ›

C **Саша Анилова**

5 месяцев назад edited

Моё решение Фибоначчи

```
function fib(n){
    let a = 0, b = 1;
        for(let i = 0; i < n; i++){a = (b + (b = a));}
    return a
}

alert(fib(3)); //2
alert(fib(7)); //13
alert(fib(77)); //5527939700884757
```

1 0 Ответить • Поделиться ›

Aristocrat

5 месяцев назад edited

Обратный порядок с использованием рекурсии

```
function a(list){
if(!list.next){
console.log(list.value)
return true;
}else{
if(a(list.next)){
console.log(list.value)
return true;
};
}
}
```

0 0 Ответить • Поделиться ›

Sergey RJS

6 месяцев назад edited

Код, который показывает, сколько ваш браузер позволяет сделать вложенных вызовов. Проверил на Яндекс, Edge и Chrome, значения отличаются всего на пару десятков единиц. Что действительно имеет

значение, так это размер контекста вызова (количество объявленных переменных в функции). Уже при 4 объявленных переменных внутри такой функции на 10000 вызовов можно не рассчитывать.

```
let count = 0;

function counter() {
count++;
counter();
}

try {
counter();
} catch (err) {
console.log('Вызовов сделано:', count); // 13977 в яндекс браузере
}
```

0 0 Ответить • Поделиться ›

exponenta

6 месяцев назад edited

Решение 4-ой задачи

```
function printListRecursion(list) {
if (list.next) {
console.log(list.value);
printListRecursion(list.next);
}
else console.log(list.value);
}

function printListIterator(list) {
let listNext = list;
do {
console.log(listNext.value);
listNext = listNext.next ?? null;
} while (listNext);
}
```

показать больше

0 0 Ответить • Поделиться ›

porcoder

7 месяцев назад

можно чуть проще вывести список в обратном порядке

```
function printReverseList(list) {

        let arr = [];
        let tmp = list;

        while(tmp) {
                arr.push(tmp.value);
```

```
            tmp = tmp.next;
        }

        for (let elem of arr.reverse()) { //<--- вот тут
            console.log(elem);
        }
```

0 0 Ответить • Поделиться ›

Kate One
8 месяцев назад

Решение 4й задачи:
function printList(list) {
for (let i in list) {
(typeof list[i] === 'object') ? printList(list[i]) : console.log(list[i]);
}
}

0 0 Ответить • Поделиться ›

Kei Presley
8 месяцев назад edited

Если кому интересно, то вот код fn fib() который еще более оптимизирован:

```
function fib(n) {
if (n <= 2) {
return 1;
}

let [a, b] = [1, 1];
for (let i = 3; i <= n; i++) {
[a, b] = [b, a + b];
}
return b;
}

console.log(fib(77));
```

2 0 Ответить • Поделиться ›

Джун на фронте
8 месяцев назад

Я хочу рассказать о своем пути в IT. Давным-давно я мечтал стать разработчиком, но не знал, с чего начать. И тогда я обнаружил этот сайт, который помог мне начать мой путь. 🤓

Я был так вдохновлен обучением, что создал **телеграм-канал "Джун на фронте"**, где записываю свою статистику. Если вы тоже мечтаете о карьере в разработке и дизайне, то присоединяйтесь. 🚀

1 4 Ответить • Поделиться ›

`nevermind
9 месяцев назад

```
function sumTo(n) {
return (n == 1) ? 1 : (n + sumTo(n - 1))
```

}

0 0 Ответить • Поделиться ›

Frinnal

9 месяцев назад

Нихуя не понял

6 0 Ответить • Поделиться ›

Загрузить ещё комментарии