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.

Частный случай подвызова – когда функция вызывает сама себя. Это как раз и называется рекурсией.

Например функция, которая возводит x в натуральную степень n. Иначе говоря, умножает x на само себя n раз.

unction 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:

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: xn = x \* xn-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:

function pow(x, n) {

return (n == 1) ? x : (x \* pow(x, n - 1));

}

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

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](https://learn.javascript.ru/recursion" \l "kontekst-vypolneniya-stek)**

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

[Контекст выполнения](https://tc39.github.io/ecma262/#sec-execution-contexts) – 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.

## [Контекст выполнения, стек](https://learn.javascript.ru/recursion#kontekst-vypolneniya-stek)

как работают рекурсивные вызовы «под капотом»

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

[Контекст выполнения](https://tc39.github.io/ecma262/#sec-execution-contexts) – 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 на вершине 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.

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

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

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:

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 структуры](https://learn.javascript.ru/recursion" \l "rekursivnye-struktury)

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

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.

## [Итого](https://learn.javascript.ru/recursion" \l "itogo)

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](https://en.wikipedia.org/wiki/Recursive_data_type) 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.

Можно ли при помощи рекурсии посчитать сумму всех чисел от 1 до 100000?

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.