

PHASE 1

WEEK 1

# DAY 3



# ОТЛАДКА

- Что делаешь?

- Баг чиню

- Что это значит?

- Баг, это ошибка в коде, вот я её и исправляю

- А зачем вы ошибки в коде пишете?

17:11



# Debug

Отладка – это процесс поиска и исправления ошибок в скрипте. Все современные браузеры и большинство других сред разработки поддерживают инструменты для отладки – специальный графический интерфейс, который сильно упрощает отладку. Он также позволяет по шагам отследить, что именно происходит в нашем коде.

The screenshot shows a browser's developer tools interface with the 'Sources' tab active. The code being debugged is as follows:

```
1 function hello(name) { name = "John"
2   let phrase = `Hello, ${name}!`; phrase = "Hello, John!"
3
4   say(phrase);
5 }
6
7 function say(phrase) {
8   alert(`** ${phrase} **`);
9 }
10
```

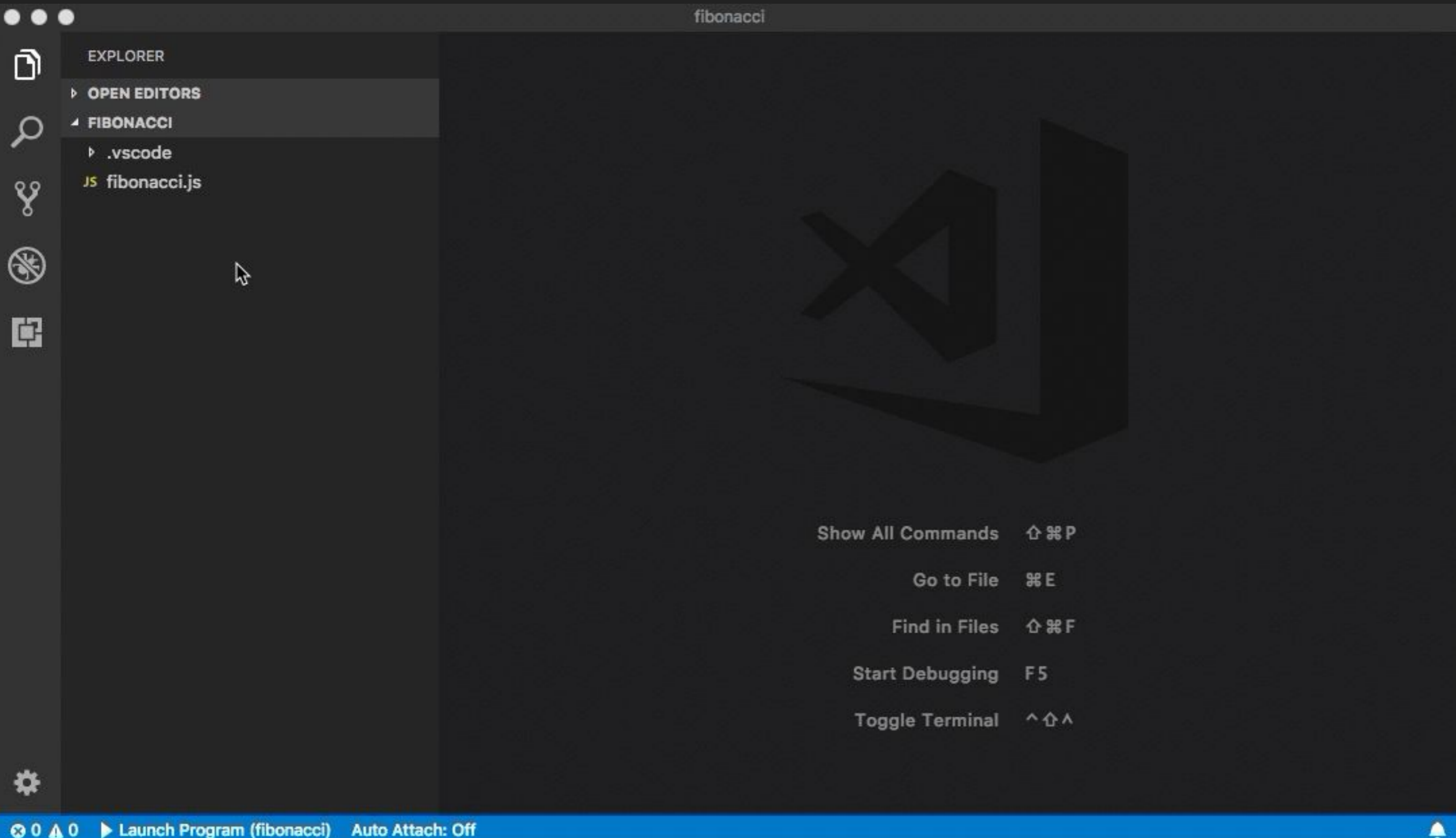
Annotations on the code:

- СМОТРЕТЬ ЗА ВЫРАЖЕНИЯМИ** (Look at expressions) points to the assignment `name = "John"` on line 1.
- ПОСМОТРЕТЬ ДЕТАЛИ ВНЕШНЕГО ВЫЗОВА** (Look at details of external call) points to the function call `say(phrase);` on line 4.
- ТЕКУЩИЕ ПЕРЕМЕННЫЕ** (Current variables) points to the local variable declarations `name` and `phrase` on lines 1 and 2.

The right-hand side of the interface shows the 'Paused on breakpoint' status. The 'Call Stack' shows the current call to `hello` at `hello.js:4` and the call from `(anonymous)` at `index.html:10`. The 'Scope' section shows the following local variables:

- `name`: "John"
- `phrase`: "Hello, John!"
- `this`: Window

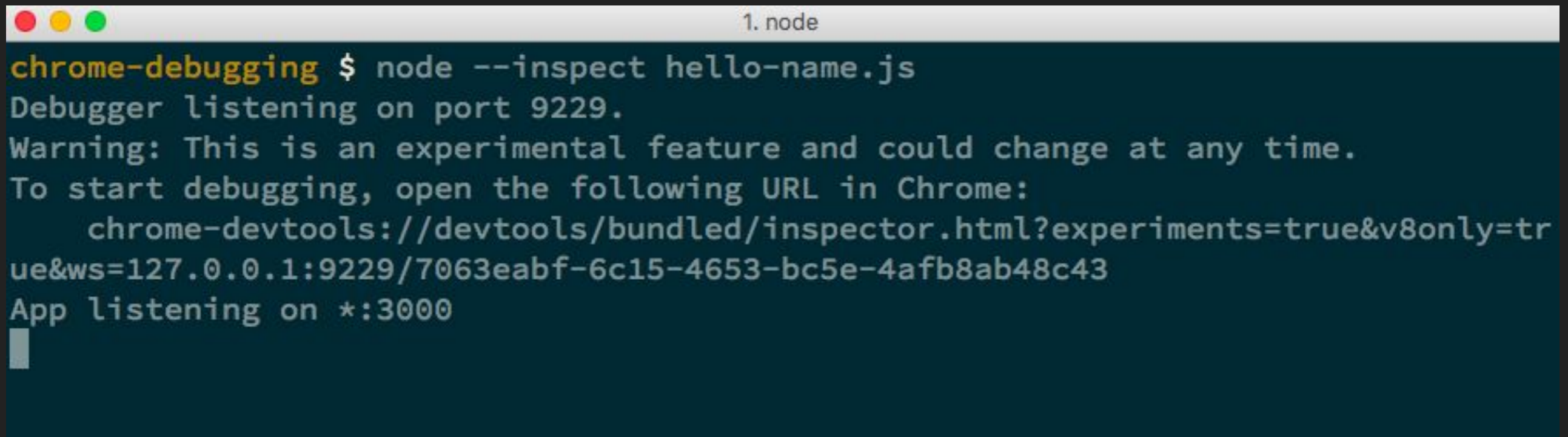
# Debugger VSCode



# Google Chrome для отладки Node.js

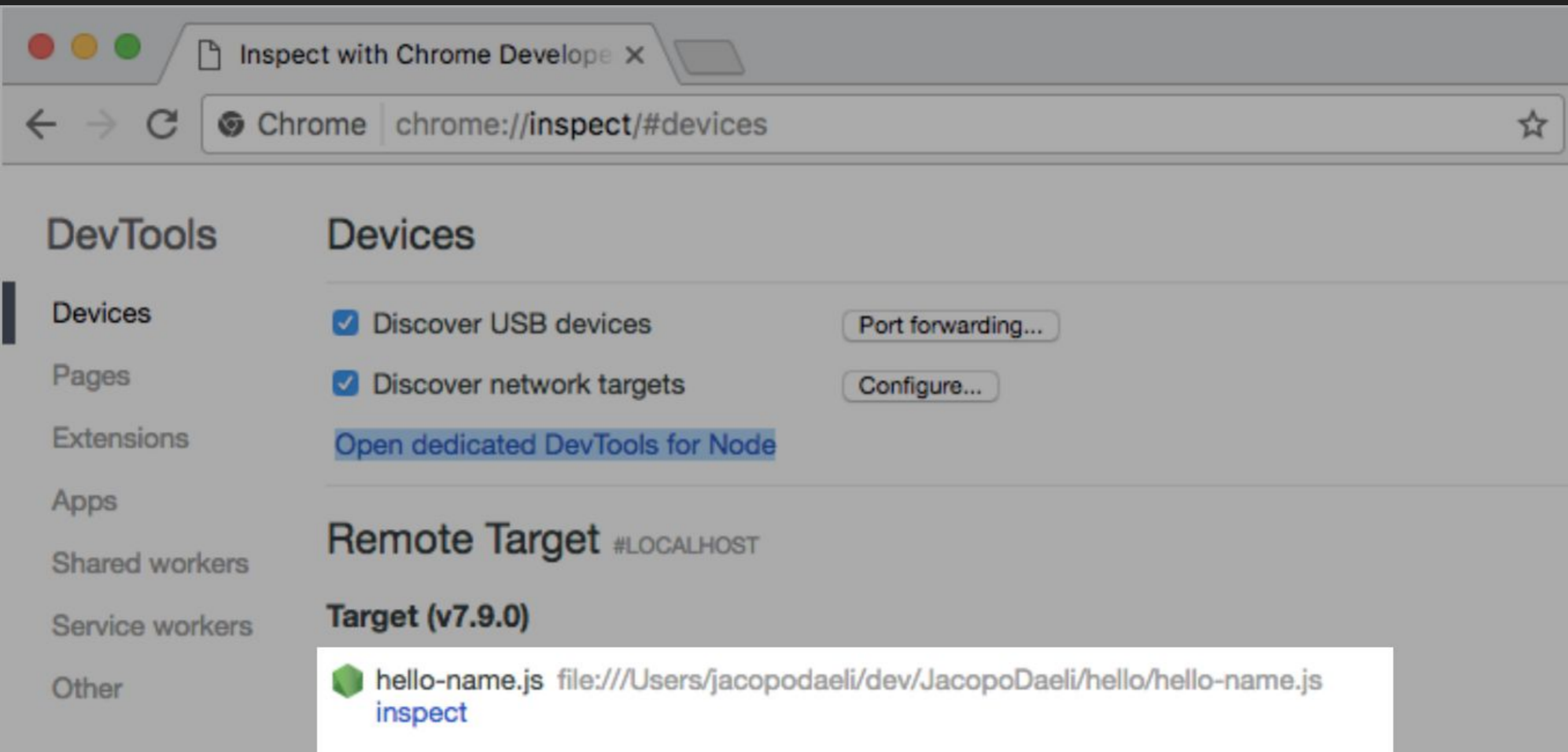
---

```
$ node --inspect <your_file>.js
```

A terminal window titled "1. node" with a dark teal background. The text inside shows the command "chrome-debugging \$ node --inspect hello-name.js" and its output: "Debugger listening on port 9229.", "Warning: This is an experimental feature and could change at any time.", "To start debugging, open the following URL in Chrome:", "chrome-devtools://devtools/bundled/inspector.html?experiments=true&v8only=true&ws=127.0.0.1:9229/7063eabf-6c15-4653-bc5e-4afb8ab48c43", and "App listening on \*:3000".

```
1. node  
chrome-debugging $ node --inspect hello-name.js  
Debugger listening on port 9229.  
Warning: This is an experimental feature and could change at any time.  
To start debugging, open the following URL in Chrome:  
chrome-devtools://devtools/bundled/inspector.html?experiments=true&v8only=true&ws=127.0.0.1:9229/7063eabf-6c15-4653-bc5e-4afb8ab48c43  
App listening on *:3000
```

# Google Chrome для отладки Node.js



# Google Chrome для отладки Node.js

The image shows the Google Chrome Developer Tools interface for debugging a Node.js application. The main pane displays the source code of a file named `hello-name.js`. The code is as follows:

```
1 (function (exports, require, module, __filename, __dirname) { 'use strict'  
2  
3 const express = require('express')  
4 const app = express()  
5  
6 const PORT = process.env.PORT || 3000  
7  
8 function capitalize (str) {  
9   const firstLetter = str.charAt(0) // we can check what's inside here  
10  return `${firstLetter.toUpperCase()}${str.slice(1)}`  
11 }  
12  
13 app.get('/:name?', (req, res) => {  
14   const name = req.params.name ? capitalize(req.params.name) : 'World'  
15   res.send(`Hello ${name}!`)  
16 })  
17  
18 app.listen(PORT, () => console.log(`App listening on *:${PORT}`))  
19  
20 });
```

The code is paused at line 14, column 16. The right-hand pane shows the 'Breakpoints' section with two active breakpoints:

- hello-name.js:9  
const firstLetter = str.charAt(0) // we can ch
- hello-name.js:14  
const name = req.params.name ? capitalize(req.

The status of the breakpoints is 'Not Paused'. The bottom status bar indicates the current position: `{}` Line 14, Column 16.



# РЕКУРСИЯ

# РЕКУРСИЯ

---

Рекурсия - вызов функции из неё самой.



# РЕКУРСИЯ

---

Рекурсия хороша, когда задача предполагает разделение на несколько аналогичных и простых действий.

Два основных условия: базовый случай и шаг (рекурсивный случай)

# РЕКУРСИЯ

```
function countdown(i) {
```

```
  console.log(i)
```

```
  if (i <= 1) {
```

```
    return;
```

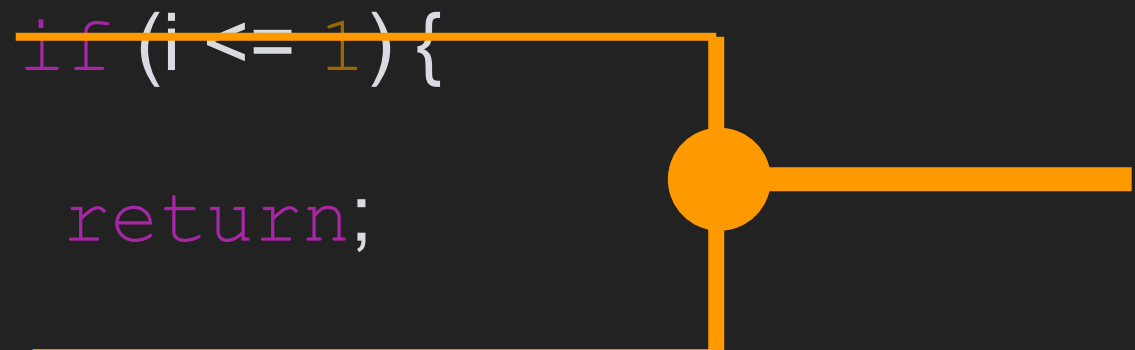
```
  } else {
```

```
    countdown(i - 1)
```

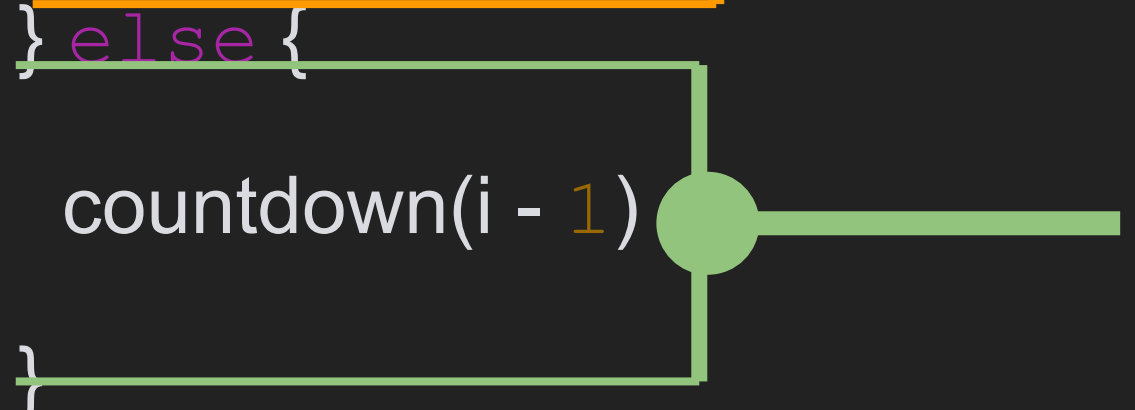
```
  }
```

```
}
```

```
countdown(5);
```



Базовый  
случай



Рекурсивный  
случай

# РЕКУРСИЯ

```
function recurSum(n) {  
  if (n === 1) {  
    return n  
  }  
  return n + recurSum(n - 1)  
}
```

Базовый  
случай

Рекурсивный  
случай

```
const res = recurSum(10); // 55
```

$\text{recurSum}(10) = 10 + \text{recurSum}(10-1)$

$9 + \text{recurSum}(9-1)$

$8 + \text{recurSum}(8-1)$

$7 + \text{recurSum}(7-1) \dots$

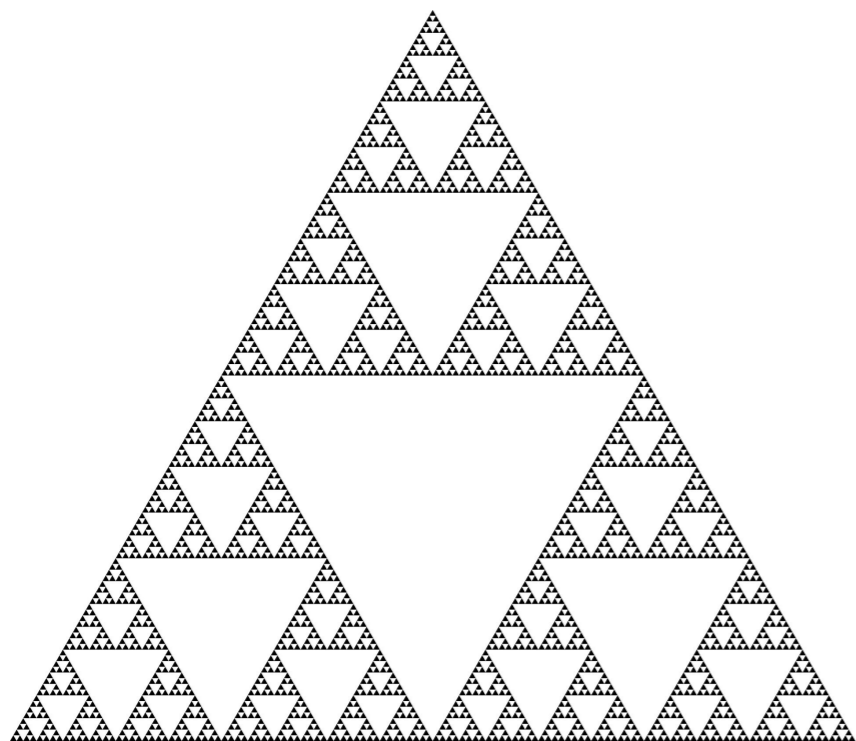
# ИТЕРАЦИЯ

---

- Повторение, но не вызов самого себя
- Например, цикл `for`

# ИТЕРАЦИЯ VS РЕКУРСИЯ

---



# АЛГОРИТМЫ



# АЛГОРИТМ ЭТО...

---

«Конечная совокупность точно заданных правил решения произвольного класса задач или набор инструкций, описывающих порядок действий исполнителя для решения некоторой задачи.»

(бесполезное определение из Википедии)

# БИНАРНЫЙ ПОИСК

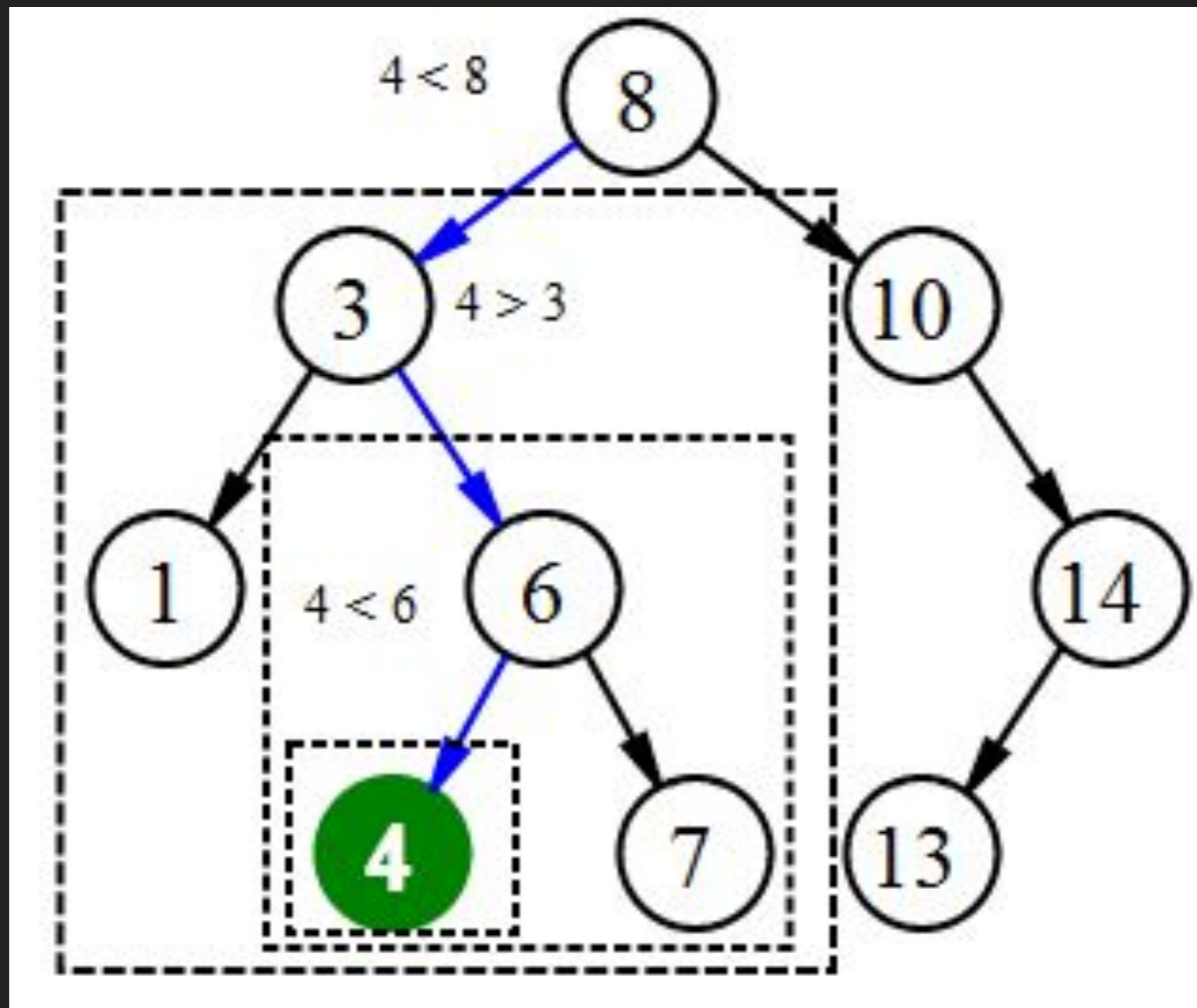
---

Выполняется по отсортированному массиву.

Бинарный поиск выполняется путем проверки того, является ли искомое значение больше, меньше или равно среднему значению в нашем массиве:

- Если оно меньше, мы можем удалить правую половину массива.
- Если оно больше, мы можем удалить левую половину массива.
- Если оно равно, мы возвращаем значение

# БИНАРНЫЙ ПОИСК



# ЛИНЕЙНЫЙ ПОИСК

---

Алгоритм линейного поиска (linear search) просто по очереди сравнивает элементы заданного списка с ключом поиска до тех пор, пока не будет найден элемент с указанным значением ключа (успешный поиск) или весь список будет проверен, но требуемый элемент не найден (неудачный поиск).

# СОРТИРОВКА ПУЗЫРЬКОМ

---

Алгоритм состоит из повторяющихся проходов по сортируемому массиву.

За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов.

Проходы по массиву повторяются  $N-1$  раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован.

При каждом проходе алгоритма по внутреннему циклу, очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива («всплывает» до нужной позиции, как пузырёк в воде — отсюда и название алгоритма).

# ВИЗУАЛИЗАЦИЯ

---

В виде танцев:

<https://www.youtube.com/watch?v=lyZQPjUT5B4>

# СОРТИРОВКА ПУЗЫРЬКОМ

---

5	2	1	3	9	0	4	6	8	7
---	---	---	---	---	---	---	---	---	---

```
const arr = [5, 2, 1, 3, 9, 0, 4, 6, 8, 7];
```

```
for (let i = 0; i < arr.length; i += 1) {  
  for (let j = 0; j < arr.length - i; j += 1) {  
    if (arr[j] > arr[j + 1]) {  
      const temp = arr[j];  
      arr[j] = arr[j + 1];  
      arr[j + 1] = temp;  
    }  
  }  
}
```

# QUICKSORT

---

В начале выбирается “опорный” элемент массива. Это может быть любое число, но от выбора этого элемента сильно зависит эффективность алгоритма. Если нам известна медиана, то лучше выбирать элемент, который как можно ближе к медиане. В нашей реализации алгоритма, мы будем брать самый левый элемент, который в результате займет свое место.

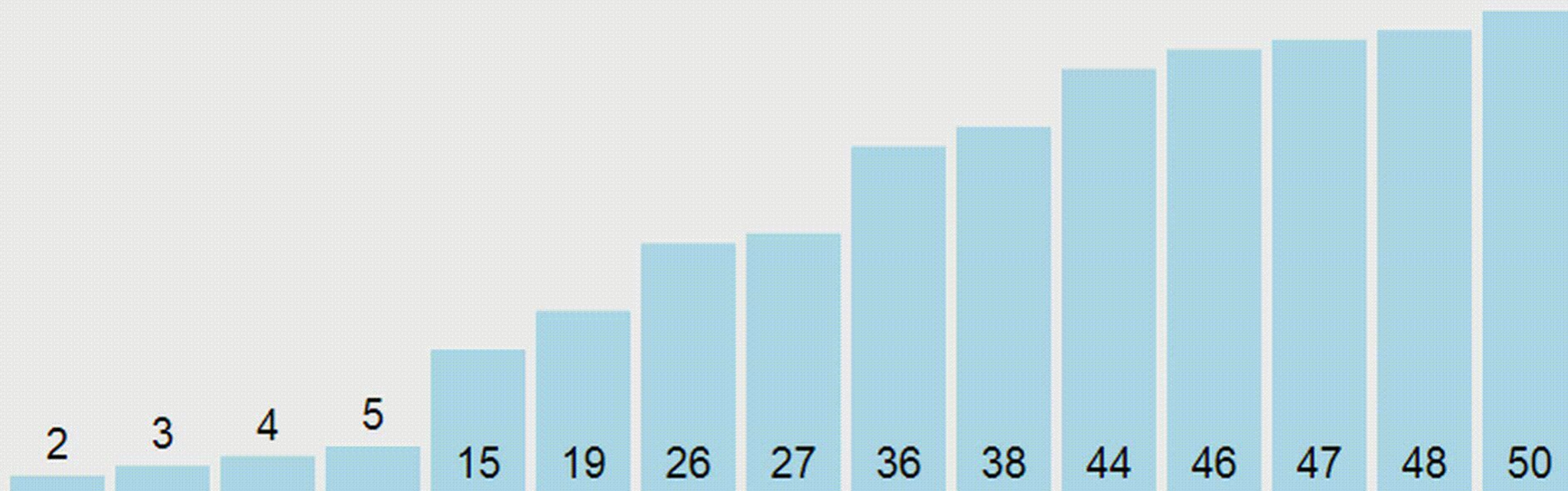
Элементы в массиве делятся на две части: слева те кто меньше опорного элемента, справа те кто больше. Таким образом опорный элемент занимает свое место и больше никуда не двигается.

Для левого и правого массива действия повторяются рекурсивно.



# ВИЗУАЛИЗАЦИЯ

---



В виде танцев:

<https://www.youtube.com/watch?v=ywWBy6J5qz8>

# QUICKSORT

---

```
const arr = [15, 4, 10, 100, 2, 34, 6, 8];
function quickSort(items, left, right) {
  let index;
  if (items.length > 1) {
    index = partition(items, left, right);
    if (left < index - 1) {
      quickSort(items, left, index - 1);
    }
    if (index < right) {
      quickSort(items, index, right);
    }
  }
  return items;
}
quickSort(arr, 0, arr.length - 1);
```

# РАЗБИЕНИЕ МАССИВА НА 2 ЧАСТИ

---

```
function partition(items, left, right) {
  let pivot = items[Math.floor((right + left) / 2)],
      i = left,
      j = right;
  while (i <= j) {
    while (items[i] < pivot) {
      i++;
    }
    while (items[j] > pivot) {
      j--;
    }
    if (i <= j) {
      const temp = items[i];
      items[i] = items[j];
      items[j] = temp;
      i++;
      j--;
    }
  }
  return i;
}
```