

Лекция 18

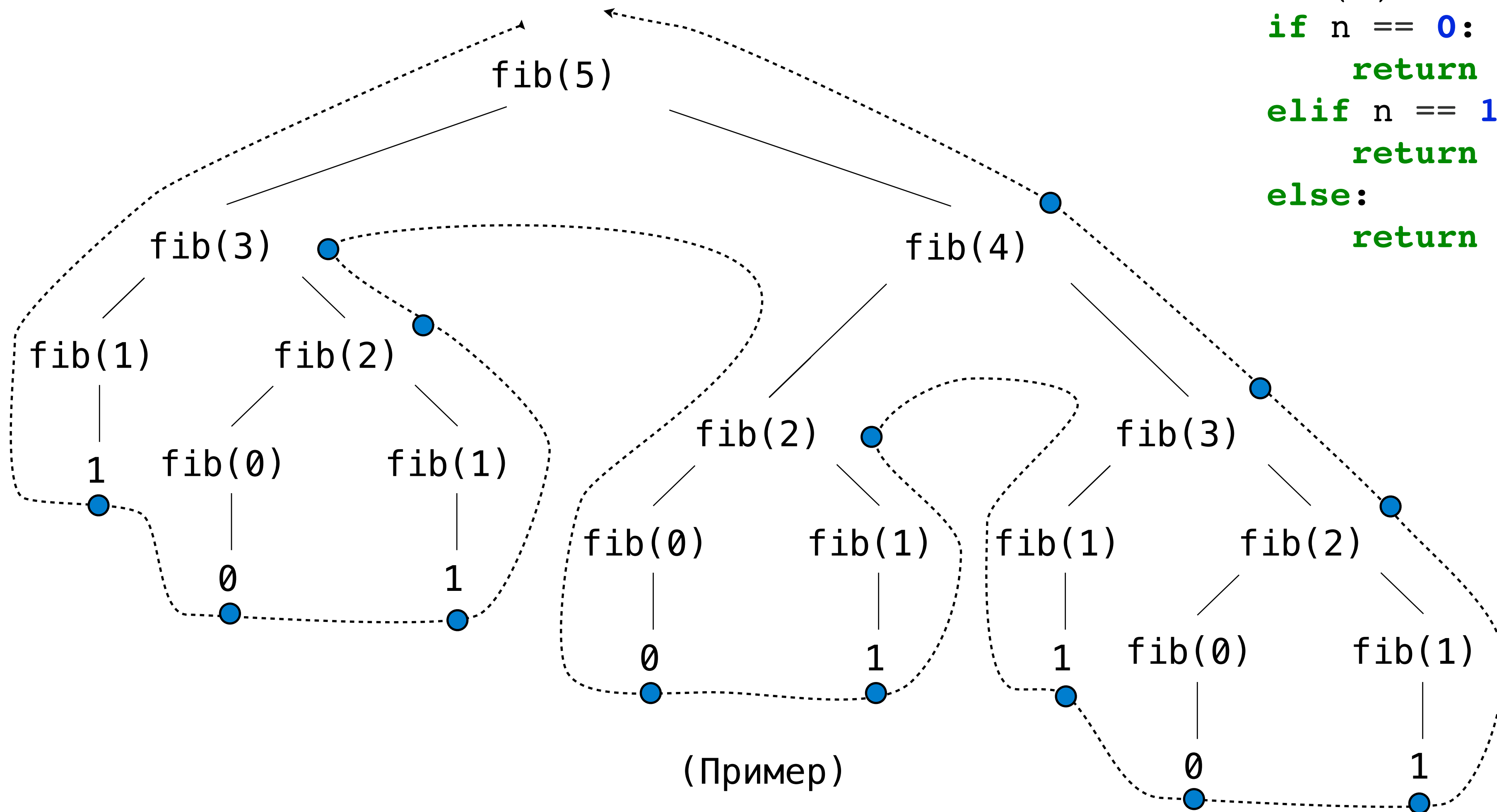
Перевод и адаптация материалов Джона ДеНиро (John DeNero). Используется с разрешения автора.

Измерение эффективности

Рекурсивное вычисление последовательности Фибоначчи

Первый пример о древовидной рекурсии:

```
def fib(n):  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return fib(n-2) + fib(n-1)
```



Запоминание

Запоминание

Идея: запоминать результаты, полученные ранее

```
def memo(f):
```

```
    cache = {}
```

```
    def memoized(n):
```

```
        if n not in cache:
```

```
            cache[n] = f(n)
```

```
        return cache[n]
```

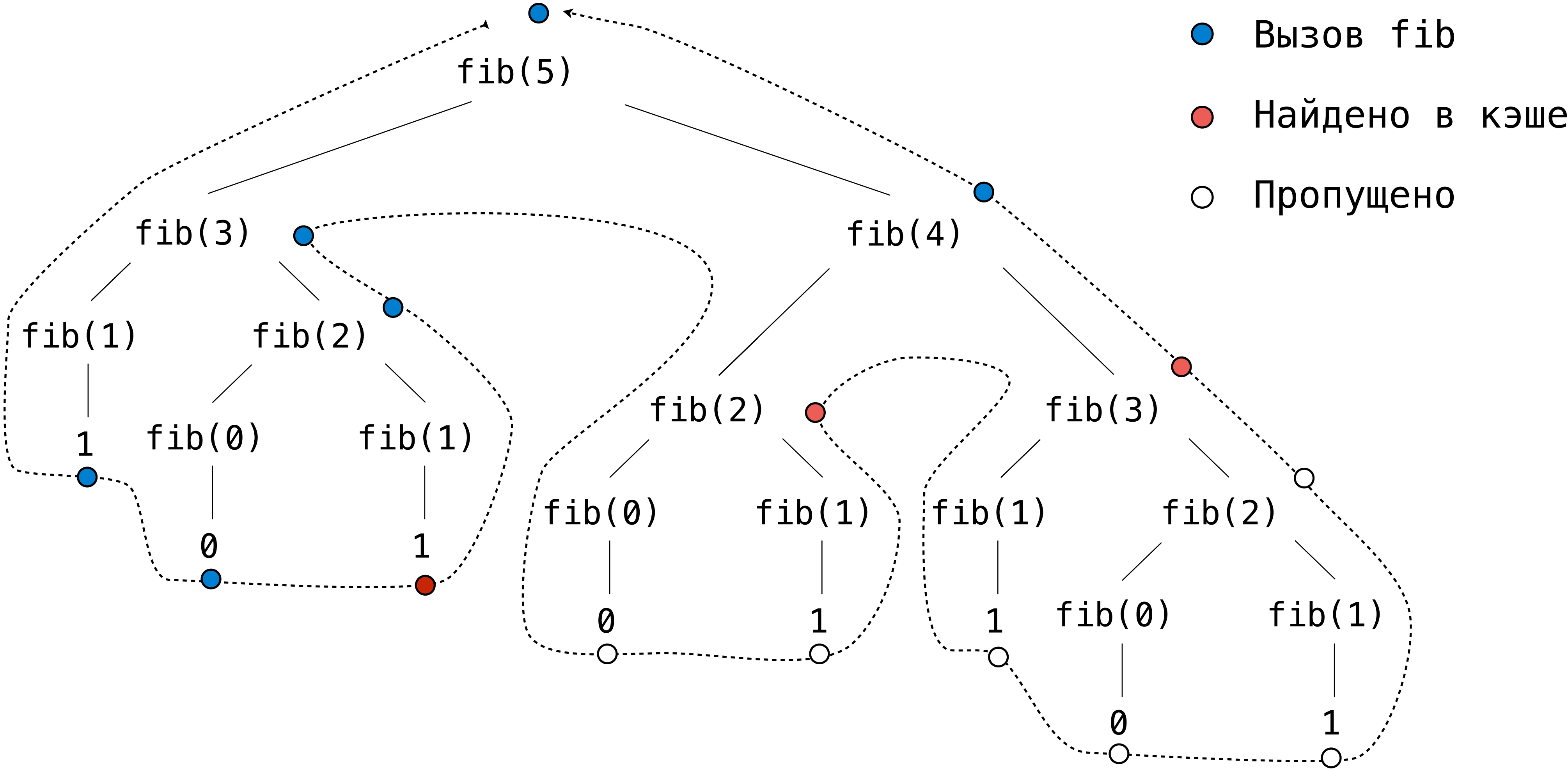
```
    return memoized
```

Ключи – аргументы,
значения – возвращаемые
значения

То же поведение, что и у f,
если f – чистая функция

(Пример)

Запоминание в древовидной рекурсии



Пространство

Затраты пространства

Фреймы каких окружений нужно помнить во время вычисления?

В любой момент времени существует набор активных окружений

Значения и фреймы в активных окружениях потребляют память (т.е. место, т.е. пространство)

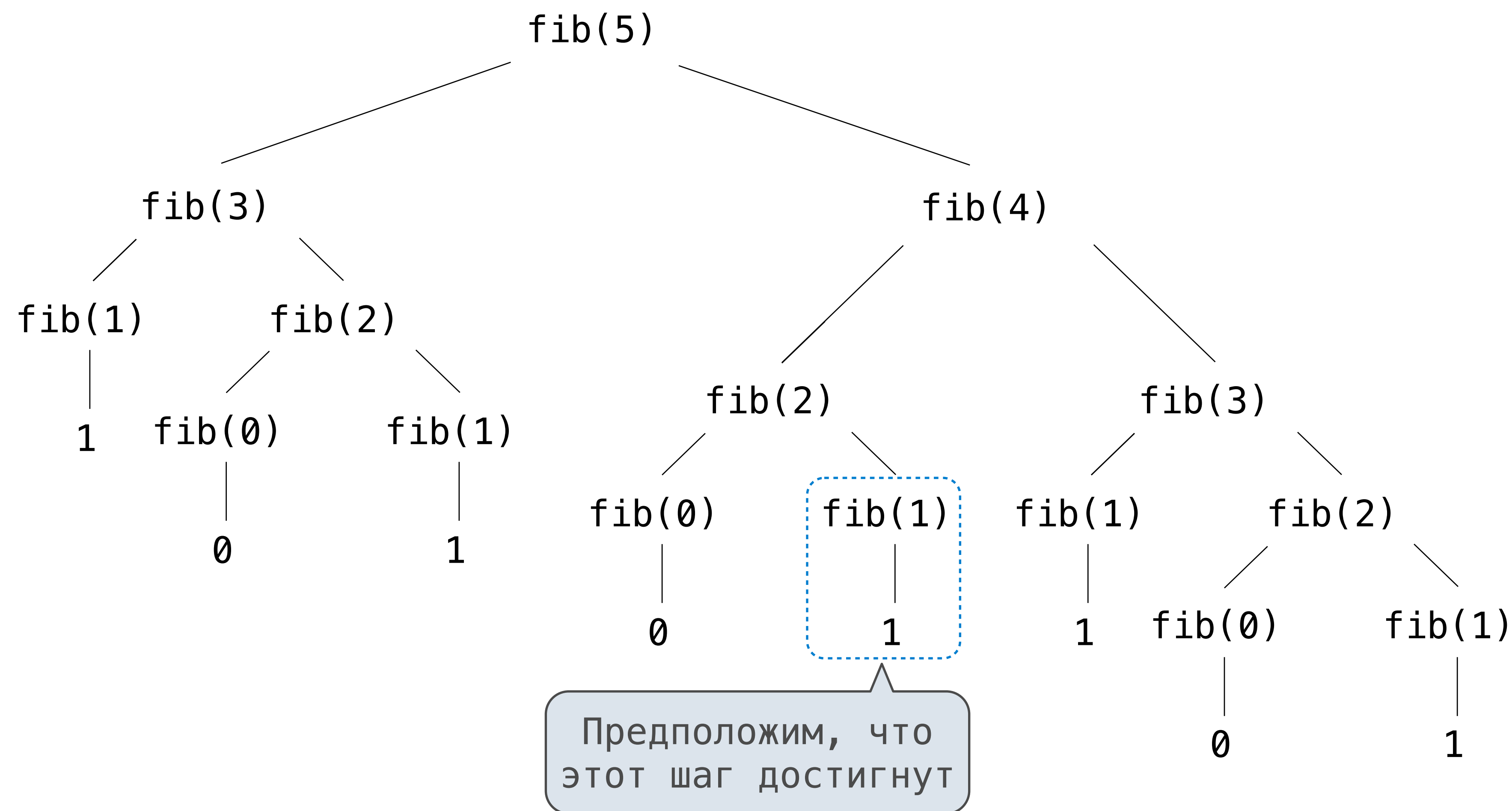
Память для других значений и фреймов может быть освобождена

Активные окружения:

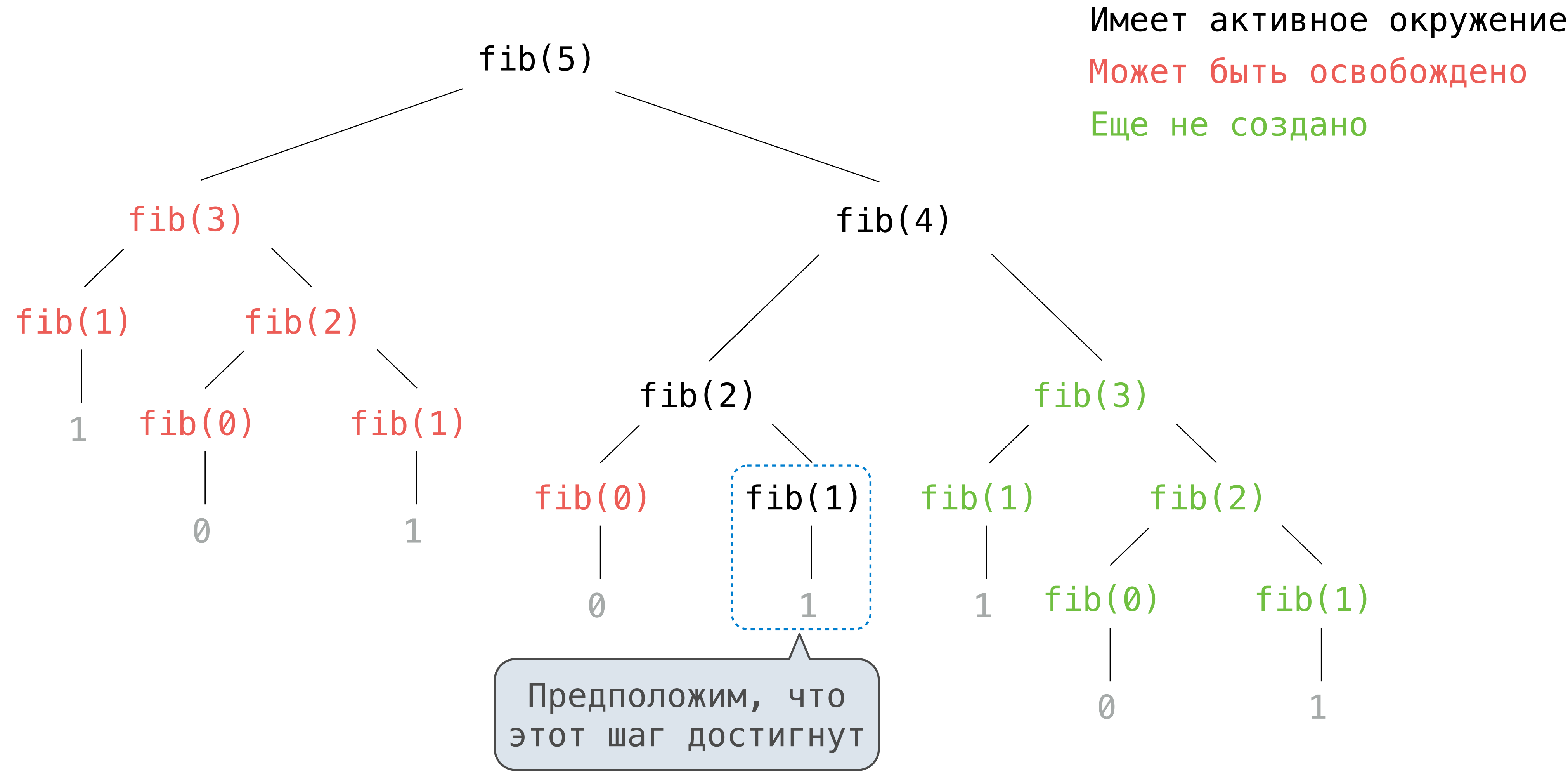
- Окружения для любых вызовов функции в процессе текущего вычисления
- Родительские окружения для функций указанных в активных окружениях

(Пример)

Пространственные затраты для последовательности Фибоначчи



Пространственные затраты для последовательности Фибоначчи



Время

Сравнение реализаций

Реализация одной и той же функциональной абстракции может требовать разного количества времени.

Задача: Сколько множителей у положительного n ?

Множитель k числа n — это положительное целое, которое делит n нацело.

`def factors(n):`

Время (количество делений)

Медленно: Проверяем каждый k от 1 до n

n

Быстро: Проверяем каждый k от 1 до корня из n
Для любого k , n/k также множитель!

Наибольшее целое, меньшее $\sqrt{n}$

(Пример)

Порядок роста

Порядки роста

Метод ограничения ресурсов, используемых функцией в терминах «размера» задачи

n: размер задачи

R(n): измерение некоторого использованного ресурса (время или пространство)

$$R(n) = \Theta(f(n))$$

означает, что существуют положительные константы **k₁** и **k₂** такие, что

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$

для всех **n** больших некоторого минимума **m**

Подсчет множителей

Количество действий, необходимых для подсчета множителей n , используя `factors_fast`: $\Theta(\sqrt{n})$

Для проверки *нижней границы*, выбираем $k_1 = 1$:

- Инструкции за пределами `while`: 4 или 5
- Инструкции внутри `while` (включая заголовок): 3 или 4
- Количество итераций `while`: между $\sqrt{n} - 1$ и $\sqrt{n}$
- Общее количество инструкций: по меньшей мере $4 + 3(\sqrt{n} - 1)$

Для проверки *верхней границы*:

- Максимальное количество инструкций: $5 + 4\sqrt{n}$
- Максимальное количество действий на инструкцию: некоторое p
- Выбираем $k_2 = 5p$ и $m = 25$

```
def factors_fast(n):  
    sqrt_n = sqrt(n)  
    k, total = 1, 0  
    while k < sqrt_n:  
        if divides(k, n):  
            total += 2  
        k += 1  
    if k * k == n:  
        total += 1  
    return total
```

Предположение: каждая инструкция (например инкремент, используя оператор `+=`), требует выполнения некоторого количества действий

Порядок роста для подсчета множителей

Реализация одной и той же функциональной абстракции может требовать разного количества времени.

Задача: Сколько множителей у положительного n ?

Множитель k числа n — это положительное целое, которое делит n нацело.

`def factors(n):`

Медленно: Проверяем каждый k от 1 до n

Время	Место
-------	-------

$\Theta(n)$	$\Theta(1)$
-------------	-------------

Быстро: Проверяем каждый k от 1 до корня из n
Для любого k , n/k также множитель!

$\Theta(\sqrt{n})$	$\Theta(1)$
--------------------	-------------

Предположение:
целое занимает
фиксированный
размер

(Пример)

Возведение в степень

Возведение в степень

Цель: возвести b в степень n

```
def exp(b, n):  
    if n == 0:  
        return 1  
    else:  
        return b * exp(b, n-1)
```

```
def square(x):  
    return x*x
```

```
def exp_fast(b, n):  
    if n == 0:  
        return 1  
    elif n % 2 == 0:  
        return square(exp_fast(b, n//2))  
    else:  
        return b * exp_fast(b, n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ (b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\ b \cdot b^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

(Пример)

Возведение в степень

Цель: возвести b в степень n

```
def exp(b, n):  
    if n == 0:  
        return 1  
    else:  
        return b * exp(b, n-1)
```

```
def square(x):  
    return x*x
```

```
def exp_fast(b, n):  
    if n == 0:  
        return 1  
    elif n % 2 == 0:  
        return square(exp_fast(b, n//2))  
    else:  
        return b * exp_fast(b, n-1)
```

Время

Место

$\Theta(n)$

$\Theta(n)$

$\Theta(\log n)$

$\Theta(\log n)$

Сравнение порядков роста

Свойства порядков роста

Константы: множители не влияют на порядок роста процесса

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta\left(\frac{1}{500} \cdot n\right)$$

Логарифмы: основания логарифма не влияют на порядок роста процесса

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Вложения: когда внутренний процесс повторяется на каждом шаге внешнего процесса, общее число шагов равно произведению количеств шагов внешнего и внутреннего процессов

```
def overlap(a, b):  
    count = 0  
    for item in a:  
        if item in b:  
            count += 1  
    return count
```

Внешний: длина a

Внутренний: длина b

Если a и b оба имеют длину **n**,
то overlap потребует $\Theta(n^2)$ шагов

Члены низкого порядка: Быстрорастущая часть вычисления в итоге доминирует

$$\Theta(n^2) \qquad \Theta(n^2 + n) \qquad \Theta(n^2 + 500 \cdot n + \log_2 n + 1000)$$

Сравнение порядков роста (n — размер задачи)

