Лекция 3
Рекурсия

(c) Д.Н. Лавров
2017

Рекурсия

- ***Рекурсией*** называется приём программирования, когда подпрограмма вызывает сама себя либо непосредственно, либо через другие подпрограммы.
- Другими словами, рекурсия – вызов функции (процедуры) из неё же самой, непосредственно (простая рекурсия) или через другие функции (сложная или косвенная рекурсия).
- Функция называется ***рекурсивной***, если во время её обработки возникает её повторный вызов, либо непосредственно, либо косвенно, путём цепочки вызовов других функций.

Примеры

```
# Бесконечная рекурсия
def ShortStory():
    print("У попа была собака, он ее любил.")
    print("Она съела кусок мяса, он ее убил,")
    print("В землю закопал и надпись написал:")
    ShortStory()
```

Примеры

```
# Факториал

def factorial (n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Примеры

```
# Вычисление биномиального коэффициента.
# Используется свойство треугольника Паскаля.

def c(k, n):
    if (n == k):
        return 1;
    if (k == 0):
        return 1;
    return c(k - 1, n - 1) + c(k, n - 1);
```

Как реализована рекурсия

- В точке вызова в стек помещаются параметры, передаваемые функции, и адрес возврата, а также состояние процессора (если это assembler).
- Вызываемая функция в ходе работы размещает в стеке собственные локальные переменные.
- По завершении вычислений функция очищает стек от своих локальных переменных, записывает результат (обычно – в один из регистров процессора).
- Команда возврата из функции считывает из стека адрес возврата и выполняет переход по этому адресу, восстанавливается состояние процессора. Одновременно стек очищается от параметров.

Достоинства и недостатки

- При отсутствии условия остановки стек, который в большинстве языков программирования ограничен, переполняется и программа прекращает работу с соответствующей ошибкой. Если использовать стек неограниченного размера, то ситуация сильно не улучшится, так как он всё равно ограничен объёмом доступной оперативной памяти.
- Основным ***достоинством*** рекурсивных функций является то, что с их помощью упрощается реализация некоторых алгоритмов и самиалгоритмы становится понятнее и яснее.

Хвостовая концевая рекурсия

- ***Хвостовая рекурсия*** – частный случай рекурсии, при котором любой рекурсивный вызов является последней операцией перед возвратом из функции.
- Подобный вид рекурсии может быть легко заменён на итерацию путём формальной и гарантированно корректной перестройки кода функции.
- Оптимизация хвостовой рекурсии путём преобразования её в плоскую итерацию реализована во многих оптимизирующих компиляторах.
- В некоторых функциональных языках программирования спецификация гарантирует обязательную оптимизацию хвостовой рекурсии.

Хвостовая концевая рекурсия

- В Python не реализован и не планируется алгоритм преобразования хвостовой рекурсии в итерацию.
- Причины объясняет автор Python https://habrahabr.ru/post/111768/
- Но сделать это не сложно и самостоятельно и не только для хвостовой рекурсии. Достаточно завести свой стек и организовать свою реализацию вызова подпрограмм.

Итерация vs Рекурсии

- Итерация не работает с системным стеком, не вызовет ошибку его переполнения.
- С помощью рекурсии проще реализовывать алгоритмы.
- Итерация, как правило, работает быстрее, так как сохраняет контекст самостоятельно и можно контролировать, что сохранять, а что нет.
- Меньше времени тратит на переключение контекста, так как нет вложенного вызова функций.
- Рекурсия – отличный приём программирования, но для повышения эффективности, часто приходиться её исключать заменяя на итерацию.

Эквивалентность

- В теоретическом программировании существует теорема, которая утверждает, что рекурсия и итерация ***эквивалентны***.
- Это значит, что любой алгоритм, который можно реализовать рекурсивно, может быть реализован и итеративно, и наоборот.
- Тем, кто знаком с теорией формальных языков, теорией конечных автоматов, теорией алгоритмов и рекурсивных функций, – она хорошо известна.
- А пока примите это на веру!!! Аминь!

Принцип исключения концевой рекурсии

```
def f(x):
    …операторы...
    return f(y)


def f(x):
    while True:
        …операторы…
        x=y
```

Полное удаление рекурсии

- Предыдущий подход может исключать, только концевые рекурсии, когда рекурсивный вызов находится в конце кода функции
- Общий подход, позволяющий преобразовать любую рекурсивную функцию в нерекурсивную, вводит определяемый пользователем стек. В общем случае этот стек хранит следующее.

1. Текущие значения параметров функции.

2. Текущие значения всех локальных переменных процедуры.

3. Адрес возврата, т.е. адрес места, куда должно перейти управление после завершения вызванной функции.

Перед рекурсивным вызовом мы сохраняем все эти параметры в стэк и передаем управление на начало программы. А после того как рекурсивный вызов отработал, мы восстанавливаем все эти параметры, извлекая их из стека, а управление передаём по адресу возврата, так же извлечённого из стека.

Пример с концевой рекурсией: Точная степень двойки

- Дано натуральное число N. Написать функцию, возвращающую *True*, если число N является точной степенью двойки, или слово *False* в противном случае.
- Операцией возведения в степень пользоваться нельзя!

Точная степень двойки

- Шаг 1 (Рекурсия)

```
def f(n):
    if n==1:
        return True
    elif n%2==1:
        return False
    else: return f(n//2)
```

- Шаг 2 (Исключение концевой рекурсии)

```
def f(n):
    while True:
        if n==1:   return True
        elif n%2: return False
        else:
            n=n//2
```

Точная степень двойки

- Шаг 2 (Исключение концевой рекурсии)

```
def f(n):
    while True:
        if n==1:    return True
        elif n%2: return False
        else:
            n=n//2
```

- Шаг 3 (Упрощение)

```
def f(n):
    while n != 1:
        if n % 2: return False
        n //= 2
    return True
```

Упрощение:

1. Условие окончания рекурсии преобразуем в условие выполнения цикла
2. Уменьшение числа в 2 раза

не выполнится, если отработает хотья бы один return. Можно убрать else

Все шаги, для сравнения

- Шаг 1 (Рекурсия)

```
def f(n):
    if n==1:
        return True
    elif n%2==1:
        return False
    else: return f(n//2)
```

- Шаг 2 (Исключение концевой рекурсии)

```
def f(n):
    while True:
        if n==1:   return True
        elif n%2: return False
        else:
            n=n//2
```

- Шаг 3 (Упрощение)

```
def f(n):
    while n != 1:
        if n % 2: return False
        n //= 2
    return True
```

Принцип декомпозиции

- Принцип декомпозиции
  или «Разделяй и властвуй»
- Метод предполагает такую декомпозицию (разбиение) задачи размера *n* на более мелкие задачи (меньшей чем *n* размерности), что на основе решений этих более мелких задач можно легко получить решение исходной задачи.
- Далее можно пользуясь принципом исключения рекурсии перейти к итерации

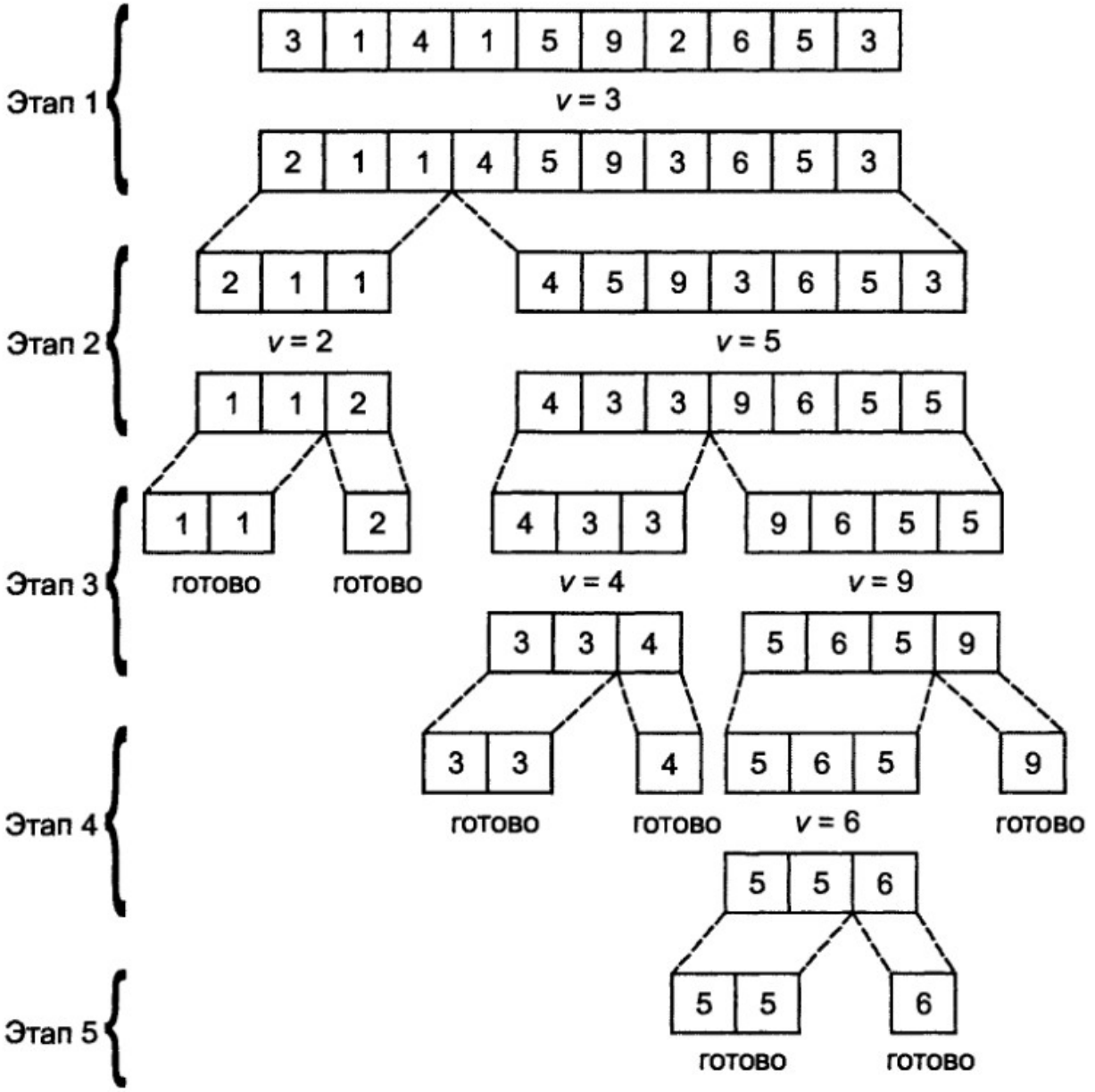
Пример

- Построение алгоритма «быстрой» сортировки.
- Быстрая сортировка (англ. quicksort), часто называемая qsort по имени реализации в стандартной библиотеке языка Си – широко известный алгоритм сортировки, разработанный английским информатиком Чарльзом Хоаром в 1960 году. Один из быстрых известных универсальных алгоритмов сортировки массивов (в среднем O(n log n) обменов при упорядочении n элементов)

Идея алгоритма

- ШАГ 1: Выбор «опорного элемента»
- ШАГ 2: «Разделение» и рекурсивный вызов

Как это работает



Возможная реализация

```
def qsort(lst):
    if lst:
        head=lst[0] # Выбор опорного -- первый из списка (не лучший выбор).
        tail =lst[1:]
        leftPart=[]
        rightPart=[]
        for x in tail:
            if x <= head: leftPart.append(x)
            else:           rightPart.append(x)
        return qsort(leftPart) + [head] + qsort(rightPart)
    return []
```

Математический вариант

```
def qsort(L):
    if L:
        return \
            qsort(filter(lambda x: x < L[0], L[1:])) + \
            L[0:1] + \
            qsort(filter(lambda x: x >= L[0], L[1:]))
    return []
```

Без дополнительной памяти

```
def QuickSort(A, l, r):
    if l >= r:  return
    else:
        v = random.choice(A[l : r+1]) # случайный опорный элемент
        i, j = l, r
        while i <= j:
            while A[i] < v:
                i += 1
            while A[j] > v:
                j -= 1
            if i <= j:
                A[i], A[j] = A[j], A[i]
                i += 1
                j -= 1
                QuickSort(A, l, j)
                QuickSort(A, i, r)
```

Трудоемкость

- В среднем $O(n \log(n))$
- В худшем $O(n^2)$

Вопросы