Практика 7. Порядки роста и связные списки

Порядки роста

Когда говорят об эффективности функции, зачастую интересуются следующим: как увеличение количества входных данных скажется на времени работы функции? И что такое «время работы»?

Вызов `square(1)` требует выполнения одной простой операции — умножения. Вызов `square(100)` также требует всего одно умножение. Не важно, какое число будет передано в качестве аргумента, для выполнения `square` всегда потребуется одна операция.

Вход	Вызов функции	Возвращаемое значение	Количество операций
1	`square(1)`	1 □ 1	1
2	`square(2)`	2 □ 2	1
□	□	□	□
100	`square(100)`	100 □ 100	1
□	□	□	□
n	`square(n)`	n □ n	1

Вызов `factorial(1)` требует одного перемножения, однако `factorial(100)` требует уже 100 перемножений. По мере увеличения входного аргумента требуемое количество простых операций растёт линейно.

Вход	Вызов функции	Возвращаемое значение	Количество операций
1	`factorial(1)`	1 □ 1	1
2	`factorial(2)`	2 □ 1 □ 1	2
□	□	□	□
100	`factorial(100)`	100 □ 99 □□□ 1 □ 1	100
□	□	□	□
n	`factorial(n)`	n □ (n-1) □□□ 1 □ 1	n

Для оценки сложности функции здесь используется Θ (тета) нотация. Например, если говорят, что сложность функции $\Theta(n^2)$, то это означает, что количество простых действий при увеличении n будет увеличиваться пропорционально n^2.

Отбрасывание младших порядков

Если функция требует $n^3 + 3n^2 + 5n + 10$ операций при заданном размере входа n, тогда сложность этой функции $\Theta(n^3)$. По мере увеличения n, слагаемые младших порядков станут незначительными по сравнению с n^3.

Отбрасывание констант

Если функция требует $5n$ операций при заданном размере входа n, то сложность этой функции будет $\Theta(n)$. Константа не учитывается, поскольку значение имеет только то, как асимптотически увеличивается сложность. Поскольку $5n$ остаётся асимптотически

линейной, то константа не влияет на порядок роста.

Виды роста

Вот наиболее часто встречающиеся порядки роста. Они расставлены от отсутствия роста до самого быстрого роста:

Θ(1)

Постоянное время — требуется одно и то же количество действий независимо от размера входа.

Θ(log n)

Логарифмический рост.

Θ(n)

Линейный рост.

Θ(n log n)

Линарифмический (?), квазилинейный (?) рост. Эн-лог-эн. Linearithmic!

***Θ(n^2), Θ(n^3)* и так далее**

Полиномиальный рост.

***Θ(2^n), Θ(3^n)* и так далее**

Экспоненциальное время (это не просто ужас, это ужас-ужас).

Кроме того, некоторые программы требуют бесконечного времени, например, если автор по невнимательности напрограммировал бесконечный цикл.

Вопрос 1

Каков порядок роста для следующих функций:

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

Ответ

```
def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n=n/2
    return total
```

Ответ

```

```

```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

Ответ

```

```

Связные списки

Для представления абстракции последовательности в Python существует множество возможностей. Ниже представлена реализация связного списка.

Связный список может быть пустым, либо быть экземпляром класса `Link`, содержащим значение `first` и связный список `rest`.

Для проверки связного списка на пустоту, его можно сравнивать с атрибутом класса `Link.empty`:

```
if link is Link.empty:
    print('Этот связный список пуст!')
else:
    print('Этот связный список вовсе не пуст!')
```

Реализация

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest:
            rest_str = ', ' + repr(self.rest)
        else:
            rest_str = ''
        return 'Link({0}{1})'.format(repr(self.first), rest_str)

@property
def second(self):
    return self.rest.first

@second.setter
def second(self, value):
        self.rest.first = value

def __str__(self):
    string = '<'
    while self.rest is not Link.empty:
        string += str(self.first) + ' '
        self = self.rest
    return string + str(self.first) + '>'
```

Вопрос 2

Напиши функцию, принимающую Python-список связных списков и перемножающую их поэлементно. Функция должна возвращать новый связный список.

Если длины связных списков не одинаковы, то результирующих связный список должен быть минимально возможной длины. Считай, что во входных связных списках нет вложенности и в списке связных списков `lst_of_lnks` есть хотя бы один элемент.

```
def multiply_lnks(lst_of_lnks):
    """
    >>> a = Link(2, Link(3, Link(5)))
    >>> b = Link(6, Link(4, Link(2)))
    >>> c = Link(4, Link(1, Link(0, Link(2))))
    >>> p = multiply_lnks([a, b, c])
    >>> p.first
    48
    >>> p.rest.first
    12
    >>> p.rest.rest.rest is Link.empty
    True
    """
```

Ответ

Вопрос 3

Напиши функцию, которая принимает отсортированный связный список целых чисел и изменяет его так, чтобы убрать из него все повторяющиеся элементы.

```
def remove_duplicates(lnk):
    """
    >>> lnk = Link(1, Link(1, Link(1, Link(1, Link(5)))))
    >>> remove_duplicates(lnk)
    >>> lnk
    Link(1, Link(5))
    """
```

Ответ

Подготовка к контрольной

Вопрос 4

Напиши функцию, которая принимает список и возвращает список, содержащий только четные элементы исходного списка, умноженные на их исходный индекс.

```
def even_weighted(lst):
    """
    >>> x = [1, 2, 3, 4, 5, 6]
    >>> even_weighted(x)
    [0, 6, 20]
    """
```

Ответ

```
    return [_________________________________________________]
```

Вопрос 5

Алгоритм сортировки `quicksort` — эффективный и часто используемый алгоритм упорядочивания элементов списка:

- Выбрать из списка элемент `pivot`, называемый опорным. Это может быть любой из элементов списка.
- Сравнить все остальные элементы с опорным и переставить их в списке так, чтобы разбить его на три непрерывных отрезка, следующих друг за другом: «элементы меньшие опорного», «равные» и «большие».
- Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

Реализуй функцию `quicksort_list`. В качестве опорного выбирай первый элемент списка. Можешь считать, что все элементы списка различаются.

```
def quicksort_list(lst):
    """
    >>> quicksort_list([3, 1, 4])
    [1, 3, 4]
    """

    if ____________________________________________________:

        ____________________________________________________

    pivot = lst[0]

    less = _____________________________________________

    greater = _____________________________________________

    return _____________________________________________
```

Вопрос 6

Напиши функцию, которая принимает список и возвращает наибольшее произведение, которое может быть получено из непоследовательных элементов. Входной список может содержать только натуральные числа.

```
def max_product(lst):
    """
    >>> max_product([10,3,1,9,2]) # 10 * 9
    90
    >>> max_product([5,10,5,10,5]) # 5 * 5 * 5
    125
    >>> max_product([])
    1
    """
```

Ответ

Вопрос 7

Дополни функцию `redundant map`, которая принимает дерево `t`, функцию `f` и применяет `f` к каждому узлу 2^d раз, где `d` — глубина узла. Корень имеет глубину `0`. Функция должна изменить существующее дерево, а не создавать новое.

```
def redundant_map(t, f):
    """
    >>> double = lambda x: x*2
    >>> tree = Tree(1, [Tree(1), Tree(2, [Tree(1, [Tree(1)])])])
    >>> redundant_map(tree, double)
    >>> print_levels(tree)
    [2]    # 1 * 2 □ (1) ; Применяет double один раз
    [4, 8] # 1 * 2 □ (2), 2 * 2 □ (2) ; Применяет double два раза
    [16]   # 1 * 2 □ (2 □ 2) ; Применяет double четыре раза
    [256]  # 1 * 2 □ (2 □ 3) ; Применяет double восемь раз
    """

    t.label = ____________________________________________

    new_f = ______________________________________________

    ______________________________________________________

        ______________________________________________________
```

Ответ