Практика 7. Порядки роста и связные списки

материалы преподавателя

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

```
Порядок роста —  θ(n^2)
```

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

Ответ

```
Порядок роста —  θ(log(n))
```

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

Ответ

```
Порядок роста — θ(1)
```

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

```
    product = 1
    for lnk in lst_of_lnks:
        if lnk is Link.empty:
            return Link.empty
        product *= lnk.first
    lst_of_lnks_rests = [lnk.rest for lnk in lst_of_lnks]
    return Link(product, multiply_lnks(lst_of_lnks_rests))
```

For our base case, if we detect that any of the lists in the list of Links is empty, we can return the empty linked list as we're not going to multiply anything. Otherwise, we compute the product of all the firsts in our list of Links. Then, the subproblem we use here is the rest of all the linked lists in our list of Links. Remember that the result of calling multiply_lnks will be a linked list! We'll use the product we've built so far as the first item in the returned Link, and then the result of the recursive call as the rest of that Link.

Итеративное решение:

```
import operator
from functools import reduce
def prod(factors):
    return reduce(operator.mul, factors, 1)
head = Link.empty
tail = head
while Link.empty not in lst_of_lnks:
    all_prod = prod([l.first for l in lst_of_lnks])
    if head is Link.empty:
        head = Link(all_prod)
        tail = head
    else:
        tail.rest = Link(all_prod)
        tail = tail.rest
    lst_of_lnks = [l.rest for l in lst_of_lnks]
return head
```

The iterative solution is a bit more involved than the recursive solution. Instead of building the list "backwards" as in the recursive solution (because of the order that the recursive calls result in, the last item in our list will be finished first), we'll build the resulting linked list as we go along.

We use head and tail to track the front and end of the new linked list we're creating. Our stopping condition for the loop is if any of the Links in our list of Links runs out of items.

Finally, there's some special handling for the first item. We need to update both head and tail in that case. Otherwise, we just append to the end of our list using tail, and update tail.

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

```
if lnk is Link.empty or lnk.rest is Link.empty:
    return
if lnk.first == lnk.rest.first:
    lnk.rest = lnk.rest.rest
    remove_duplicates(lnk)
else:
    remove_duplicates(lnk.rest)
```

Now consider two possible cases: — If there is a duplicate of the first item, we will find that the first and second items in the list will have the same values (that is, lnk.first == lnk.rest.first). We can confidently state this because we were told that the input linked list is in sorted order, so duplicates are adjacent to each other. We'll remove the second item from the list. Finally, it's tempting to recurse on the remainder of the list (lnk.rest), but remember that there could still be more duplicates of the first item in the rest of the list! So we have to recurse on lnk instead. Remember that we have removed an item from the list, so the list is one element smaller than before. Normally, recursing on the same list wouldn't be a valid subproblem. — Otherwise, there is no duplicate of the first item. We can safely recurse on the remainder of the list.

Итеративное решение:

```
while lnk is not Link.empty and lnk.rest is not Link.empty:
    if lnk.first == lnk.rest.first:
        lnk.rest = lnk.rest.rest
    else:
        lnk = lnk.rest
```

The loop condition guarantees that we have at least one item left to consider with lnk. For each item in the linked list, we pause and remove all adjacent items that have the same value. Once we see that lnk.first != lnk.rest.first, we can safely advance to the next item. Once again, this takes advantage of the property that our input linked list is sorted.

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 [i * lst[i] for i in range(len(lst)) if i % 2 == 0]
    # return [i * lst[i] for i in range(0, len(lst), 2)]
```

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 _____________________________________________________
```

Ответ

```
def quicksort_list(lst):
    """
    >>> quicksort_list([3, 1, 4])
    [1, 3, 4]
    """
    if not lst or len(lst) == 1:
        return lst
    pivot = lst[0]
    less = [i for i in lst[1:] if i < pivot]
    greater = [i for i in lst[1:] if i > pivot]
    return quicksort_list(less) + [pivot] + quicksort_list(greater)
```

Решение для списка, в котором элементы могут повторяться:

```
def quicksort_list(lst):
    """
    >>> quicksort_list([3, 1, 3, 7])
    [1, 3, 3, 7]
    """
    if not lst or len(lst) == 1:
        return lst
    pivot = [i for i in lst if i == lst[0]]
    less = [i for i in lst[1:] if i < pivot[0]]
    greater = [i for i in lst[1:] if i > pivot[0]]
    return quicksort_list(less) + pivot + quicksort_list(greater)
```

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

```
def max_product(lst):
    if not lst:
        return 1
    elif len(lst) == 1:
        return lst[0]
    else:
        max(max_product(lst[1:]), lst[0] * max_product(lst[2: ]))
```

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

```
    t.label =f(t.label)
    new_f = lambda x: f(f(x))
    for b in t.branches:
        redundant_map(b, new_f)
```