Залікова робота

Залікова робота з курсу “Алгоритми та структури данних”

студента 2 курсу групи МП-21

Захарова Д.О.

Завдання 3, 16

Завдання 3.

> **Умова.** Рекурсивні алгоритми, їх коректність. Стек рекурсивних викликів. Задача про ханойські вежі.

Відповідь. *Рекурсивним алгоритмом*, якщо говорити нестрого, називають алгоритм, який у процесі виконання викликає сам себе.

Зазвичай у більшості мов програмування використання рекурсивних функцій виглядає приблизно так (не варто сприймати як строге означення, наведено чисто для прикладу):

```
function recursive_function(args):
    if some_condition_with_args(args):
        return result
    // Some processing of args
    return some_function_from_args(args)*recursive_function(new_args)
```

На початку ставимо умову при якій рекурсивна функція припиняє своє виконання — зазвичай це якісь доволі тривіальні базові кейси. Далі ми робимо обробку аргументів і викликаємо функцію знову, але з модифікованими аргументами.

Як приклад розглянемо таку задачу:

> **Задача.** Реалізувати функцію `pow(a,n)` яка повертає $a^n, a \in \mathbb{R}, n \in \mathbb{N}$.

Звичайно можна реалізувати задачу ітеративно (реалізація на мові *Python*):

```
def pow(a,n):
  result = 1
  for _ in range(n):
    result = result * a
  return result
```

Проте спробуємо реалізувати цю задачу рекурсивним алгоритмом. Помітимо, що справедлива наступна формула:

$$a^n = a \cdot a^{n-1}$$

Тому на кожному кроці ми можемо повертати `a*pow(a,n-1)`. Проте якщо просто повертати таким чином значення без додаткової перевірки на початку, то просто отримаємо нескінченну рекурсію. Тому нам потрібно врахувати, що якщо на якомусь етапі $n = 1$ (а такий момент точно настане, бо $n \in \mathbb{N}$), то ми просто повертаємо a. Інакше кажучі,

$$a^n = \begin{cases} a, \ n = 1 \\ a \cdot a^{n-1}, \ n > 1 \end{cases}$$

Тому можемо реалізувати алгоритм наступним чином:

```
def pow(a,n):
    return a*pow(a,n-1) if n > 1 else a
```

Умова $n = 1 \implies a^n = a$ називається *базою рекурсії*. В якомусь сенсі це нагадує доведення за індукцією: маємо базу індукції — перевірка перших елементарних випадків, а далі перехід — використовуючи вже відомий факт(и), переходити на наступний (звісно, якщо описувати дуже спрощено). В цьому випадку навпаки — ми йдемо з кінця до початку — бази рекурсії.

Поговоримо про *стек викликів*. Розглянемо на прикладі нашої функції `pow(a,n)` зверху. Отже, нехай ми запустили `pow(2.0,5)`. Оскільки $5 > 1$, то ми викликаємо функцію `2.0*pow(2.0,4)`. При цьому в стек ми зберігаємо (дуже грубо кажучі, бо це дуже залежить від конкретної мови):

function	a	n
return a*pow(a,n-1)	2.0	5

Важливо, що при цьому ми поки нічого не повертаємо, а зберігаємо цей “стан” функції. Далі ми запустимо `2.0*pow(2.0,3)` і до стеку ми додаємо (верхньою строкою я позначаю вершину стеку):

function	a	n
return a*pow(a,n-1)	2.0	4
return a*pow(a,n-1)	2.0	5

Так ми продовжимо поки не викличемо `2.0*pow(2.0,1)`. В такому разі наша функція поверне `return 2.0` і тому загалом наш стек буде виглядати як:

function	a	n
return a	2.0	1
return a*pow(a,n-1)	2.0	2
return a*pow(a,n-1)	2.0	3
return a*pow(a,n-1)	2.0	4
return a*pow(a,n-1)	2.0	5

І вже тільки після цього компілятор пройдеться по всьому стеку і викликає усі функції підряд зверху-вниз, повертаючи $2.0^5 = 32.0$. У деяких мов програмування є оптимізація рекурсії і тому не використовують стек викликів.

Як бачимо, у рекурсивних алгоритмів є суттєвий недолік — для того, щоб вони працювали, потрібно в аргументах функції вказувати усі параметри, що описують “стан” функції і тому вони можуть займати багато пам'яті.

Перейдемо до ханойської вежі. Спочатку сформулюємо її.

> **Ханойські вежі.** Маємо 3 палиці (A, B, C) та n дисків, що поставлені на першу палицю і розташовані по спаданню діаметрів (якщо дивитися знизу палиці). Задача перемістити усю цю "піраміду" з палиці A на палицю C, переміщаючи диски по одному, причому можна брати лише диски зверху палиці та не можна ставити диск більшого діаметру на диск меншого діаметру.

Алгоритм, який ми можемо використати, наступний:

- Беремо $n-1$ диск і якимось чином (нам поки не важливо як саме) переставляємо їх з палиці A на палицю B, використовуючі палицю C як допоміжну.
- Беремо диск, що залишився, і кидаємо його на палицю C.
- Беремо усю піраміду, що залишилась, і переставляємо на палицю C (знову ж таки, поки не важливо як саме), використовуючі палицю A як допоміжну.

Тому можемо запропонувати таку реалізацію:

```
def HanoiTower(n, From='A',To='C',Helper='B'):
    if n == 0:
        return
    HanoiTower(n-1, From=From,To=Helper,Helper=To)
    print('Moving disk {} from {} to {}'.format(n, From, To))
    HanoiTower(n-1, From=Helper,To=To,Helper=From)
```

І далі можемо запустити:

```
HanoiTower(3)
```

Отримаємо результат:

```
Moving disk 1 from A to C
Moving disk 2 from A to B
Moving disk 1 from C to B
Moving disk 3 from A to C
Moving disk 1 from B to A
Moving disk 2 from B to C
Moving disk 1 from A to C
```

Проаналізуємо кількість операцій. Нехай кількість операцій, що необхідна для того, щоб перемістити n дисків з A до C дорівнює H_n. В такому разі рекурсивна залежність:

$$H_n = 2H_{n-1} + 1$$

З початковою умовою $H_1 = 1$. Доволі нескладно бачити, що

$$H_n = 2H_{n-1} + 1 = 2(2H_{n-2} + 1) + 1 = 2^2 H_{n-2} + 2 + 1 =$$
$$2^2(2H_{n-3} + 1) + 2 + 1 = 2^3 H_{n-3} + 2^2 + 2^1 + 2^0 = \dots$$

Отже бачимо, що

$$H_n = 2^{n-1}H_1 + \sum_{k=0}^{n-2} 2^k = 2^{n-1}H_1 + 2^{n-1} - 1 = 2^n - 1$$

Отже складність алгоритму $\mathcal{O}(2^n)$. *Space complexity* в цьому випадку $\mathcal{O}(n)$, оскільки на кожному кроці рекурсії ми понижуємо n на 1 і таким чином глибина рекурсії n.

Завдання 16.

> **Умова.** Побудова зворотного польського запису, алгоритм "сортувальної станції".

Відповідь. Зворотній польский запис — це допоміжна структура запису математичних виразів для їх обрахування. Наприклад, розглянемо наступний математичний вираз:

$$(2 + 3) \times 4 + 7 \times (6 - 5)$$

Зворотній польский запис буде виглядати наступним чином:

$$[2, 3, +, 4, \times, 7, 6, 5, -, \times, +]$$

Щоб із польского запису отримати значення, потрібно спочатку створити стек для чисел, а потім зліва направо почати зчитувати символи. На кожному кроці робимо наступні дії:

- Якщо перед нами число, додаємо його в стек.
- Якщо перед нами операція, то беремо 2 останніх числа в стеку, застосовуємо до них цю операцію і далі додаємо результат до стеку.
- Якщо ми закінчити зчитування, то просто повертаємо значення зі стеку. Якщо значення залишилось більше за 1, то запис було сформовано неправильно (або алгоритм реалізовано неправильно 🙂).

Наприклад, для нашого випадку результат алгоритму буде виглядати наступним чином:

Input	Stack	Operation
2	$[2]$	$-$
3	$[2,3]$	$-$
$+$	$[5]$	$2+3=5$
4	$[5,4]$	$-$
$\times$	$[20]$	$4\times 5=20$
7	$[20,7]$	$-$
6	$[20,7,6]$	$-$
5	$[20,7,6,5]$	$-$
$-$	$[20,7,1]$	$6-5=1$
$\times$	$[20,7]$	$1\times 7=7$
$+$	$[27]$	$20+7=27$

Імплементація може виглядати, наприклад, наступним чином:

```
supported_signs = {
    '+': lambda x, y : y + x,
    '-': lambda x, y : y - x,
    '*': lambda x, y : y * x,
    '/': lambda x, y : y / x,
    '^': lambda x, y : y ** x
}

def array_as_string(array):
    result = ''
    for element in array[:(-1)]:
        result += (str(element) + ',')
    return result + str(array[-1])

def calculate_reverse_polish_notation(converted_expression):
    stack = []
    for character in converted_expression:
        if character.isdigit():
            stack.append(float(character))
            print('Current stack: ' + array_as_string(stack))
        elif character in supported_signs:
            fn = supported_signs[character]
            stack.append(float(fn(float(stack.pop()), float(stack.pop()))))
            print('Current stack: ' + array_as_string(stack))
    return stack[0]
```

Тоді якщо запустити:

```
print(calculate_reverse_polish_notation(['2','3','+','4','*','7','6','5','-','*','+']))
```

То отримаємо:

```
Current stack: 2.0
Current stack: 2.0,3.0
Current stack: 5.0
Current stack: 5.0,4.0
Current stack: 20.0
Current stack: 20.0,7.0
Current stack: 20.0,7.0,6.0
Current stack: 20.0,7.0,6.0,5.0
Current stack: 20.0,7.0,1.0
Current stack: 20.0,7.0
```

```
Current stack: 27.0
27.0
```

Тепер питання: як, маючи строку з формулою, отримати зворотній польский запис? Для цього є *алгоритм Дейкстри* (або *алгоритм сортувальної станції*). Виглядає він наступним чином: нехай ми прочитали елемент строки (число/операція/дужка), тоді:

- Якщо ми прочитали число, то додаємо його у стек `output`.
- Це операція — поки на вершині стеку `operations` ми зустрічаємо операції, що такого самого або більшого пріоритету, перекладаємо їх у `output`. Як тільки ми зустріли операцію меншого пріоритету або стек виявився пустим, додаємо цю операцію до стеку `operations`.
- Якщо нам зустрілась `(`, то додаємо її в `operations`.
- Якщо нам зустрілась `)`, то допоки на вершині `operations` ми не зустрінемо `(`, перекладаємо операції у стек `output`. Саму дужку `(` просто видаляємо.

Коли ми закінчили все зчитувати, перекладаємо увесь стек `operations` у `output`.

Як приклад розглянемо $2 \times (3+4) + 5 \times 6$. Отже:

Input	Output	Operations
2	$[2]$	$[]$
$\times$	$[2]$	$[\times]$
$($	$[2]$	$[\times, (]$
3	$[2, 3]$	$[\times, (]$
$+$	$[2, 3]$	$[\times, (, +]$
4	$[2, 3, 4]$	$[\times, (, +]$
$)$	$[2, 3, 4, +]$	$[\times]$
$+$	$[2, 3, 4, +, \times]$	$[+]$
5	$[2, 3, 4, +, \times, 5]$	$[+]$
$\times$	$[2, 3, 4, +, \times, 5]$	$[+, \times]$
6	$[2, 3, 4, +, \times, 5, 6]$	$[+, \times]$

Отже наш результат: $[2, 3, 4, +, \times, 5, 6, \times, +]$. Приклад реалізації можна побачити нижче:

```
# Just a helper function to print current step
def print_step(input_symbol, output, stack):
    print('Input: {}'.format(input_symbol))
    print('Output: ' + ','.join(output))
    print('Stack: ' + ','.join(stack))
    print('-'*32)

# Dictionary that stores the importance of each sign (also it stores all supported signs)
signs_importance = {
    '+': 1,
    '-': 1,
    '*': 2,
    '/': 2,
    '^': 2,
    '(': 0,
}

# Function for converting expression into the reverse polish notation
def dijkstra_conversion(expression: str) -> str:
    output, stack = [], []
    k = 0
```

```
    while k < len(expression):
        # If the expression is a number, iterating while we encounter a number
        # to handle numbers with number of digits greater than 1
        if expression[k].isdigit():
            number = ''
            while expression[k].isdigit():
                number += expression[k]
                k = k + 1

            # Appending a character
            output.append(number)
            print_step(number,output,stack)
            continue
        elif expression[k] == '(':
            # Just appending the left bracket
            stack.append(expression[k])
            print_step(expression[k],output,stack)
        elif expression[k] in signs_importance:
            # Only if the stack contains at least something, we start iterating through it
            if len(stack) > 0:
                # While importance of the top element in stack is greater than the importance
                # of a current character, add top element in stack to the output
                while signs_importance[stack[-1]] >= signs_importance[expression[k]]:
                    output.append(stack.pop())
                    # If length of a stack is 0, break out of the loop
                    if len(stack) == 0:
                        break
            # Append character in stack
            stack.append(expression[k])
            print_step(expression[k],output,stack)
        elif expression[k] == ')':
            # Stack should not be empty since it must contain at least '('
            if len(stack) == 0:
                print('ERROR: Stack should not be empty. Check whether you have enterred brackets properly')
                break
            # While top element in stack is not '(', moving this element to the output
            while stack[-1] != '(':
                output.append(stack.pop())
                # If length of a stack is 0, break out of the loop
                if len(stack) == 0:
                    break
            # Removing last element in stack since it is '('
            stack.pop()
            print_step(expression[k],output,stack)
        else:
            # In case none of the conditions were satisfied, return an error
            print('ERROR: Unsupported character')
            break
        # Do not forget to add 1 to k
        k = k + 1

    while len(stack) > 0:
        output.append(stack.pop())
    return output
```

Запустимо нашу програму на прикладі $(12 \cdot 4)^{2 \cdot (3+5)/4}$:

```
# Defining our expression
expression = '(12*4)^((2*(3+5))/4)'
# Removing spaces
expression = expression.replace(' ', '')

# Calling our function
converted_expression = dijkstra_conversion(expression)
print('Result is {}'.format(converted_expression))
```

Отримаємо наступний результат:

```
Input: (
Output:
Stack: (
--------------------------------
Input: 12
Output: 12
Stack: (
--------------------------------
Input: *
Output: 12
Stack: (,*
--------------------------------
Input: 4
Output: 12,4
Stack: (,*
--------------------------------
Input: )
Output: 12,4,*
Stack:
--------------------------------
Input: ^
Output: 12,4,*
Stack: ^
--------------------------------
Input: (
Output: 12,4,*
Stack: ^,(
--------------------------------
Input: (
Output: 12,4,*
Stack: ^,(,(
--------------------------------
Input: 2
Output: 12,4,*,2
Stack: ^,(,(
--------------------------------
Input: *
Output: 12,4,*,2
Stack: ^,(,(,*
--------------------------------
Input: (
Output: 12,4,*,2
Stack: ^,(,(,*,(
--------------------------------
Input: 3
Output: 12,4,*,2,3
Stack: ^,(,(,*,(
--------------------------------
Input: +
Output: 12,4,*,2,3
Stack: ^,(,(,*,(,+
--------------------------------
Input: 5
Output: 12,4,*,2,3,5
Stack: ^,(,(,*,(,+
--------------------------------
Input: )
Output: 12,4,*,2,3,5,+
Stack: ^,(,(,*
--------------------------------
Input: )
Output: 12,4,*,2,3,5,+,*
Stack: ^,(
--------------------------------
Input: /
Output: 12,4,*,2,3,5,+,*
```

```
Stack: ^,(,/
--------------------------------
Input: 4
Output: 12,4,*,2,3,5,+,*,4
Stack: ^,(,/
--------------------------------
Input: )
Output: 12,4,*,2,3,5,+,*,4,/
Stack: ^
--------------------------------
Result is ['12', '4', '*', '2', '3', '5', '+', '*', '4', '/', '^']
```

Можна також зазначити, що складність обох алгоритмів $\mathcal{O}(n)$, де n — довжина строки для алгоритма Дейкстри та довжина масиву для переводу з зворотнього запису до результату.