THE TLV PARSER PROBLEM

Datecs Ltd.
Geri Ilieva

2017 | spring

УВОД

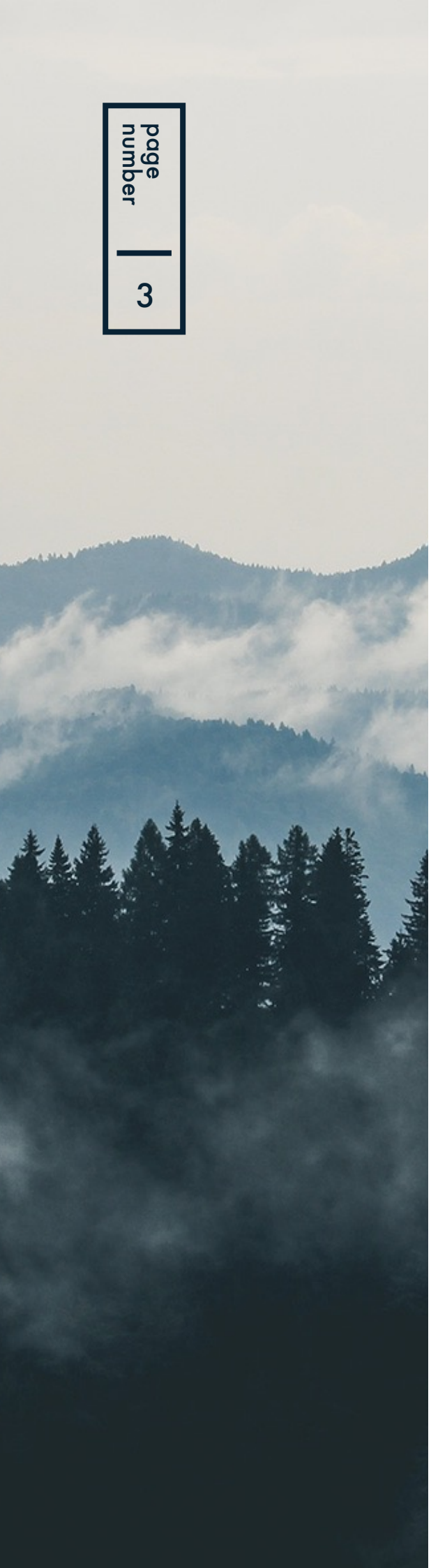
Време е за една задача, която в бъдеще ще ползвате в реален проект, а не поредната досадна и измислена задача от интернет пространството. Но, за да я направим добре, трябва да ползваме мразената от много програмисти рекурсия.

Какво е рекурсия?

Това е похват в програмирането, в който, решението на един проблем се базира на решението на по-малка част от същия проблем. В общи линии рекурсия казваме, когато една функция вика себе си. Както се шегуват: за да разбереш рекурсия, трябва да разбереш рекурсия.

Защо хората избягват рекурсия?

По-опитните програмисти смятат, че рекурсията е едно от нещата, което отличава добрите програмисти от слабите. Истината е, че много неща могат да се оплетат, ако кодът не се напише правилно → например, ако не се дефинира добре базовият случай, кодът ви няма да може да излеле, ще се завърти в един безкраен цикъл от влизане във функцията, докато не изяде напълно стака (всеки път, когато се викне функция, на частта от паметта позната като стак, се копира контекста на функцията – какви са аргументите и къде да се върне, когато излезе – но стака е ограничен като памет и в един момент ще почнете да пишете върху heap-а (където ви е динамичната памет) и надолу към статичната памет, и така докато не крашне цялостно програмата поради някаква причина свързана с омазаната памет). Не е никак красиво! Но когато е добре написана, рекурсията може да ни помогне да решим заплетен проблем просто и елегантно, както ще видим в днешната задача. И все пак, дори добрите програмисти ще ползват решение с цикли, ако е възможно (а повечето рекурсии могат да се заменят с цикъл) – основно, заради описаният по-горе overhead при викане на функция, който за у-во с малка памет може да направи големи бели.



РЕКУРСИЯ

Как да пишем правилно рекурсивни функции?

За да работи рекурсията трябват две неща:
1. Базов случай – това е нещо като крайна цел, или нещото, което ще спре рекурсията
2. Процес, в който задачата е намалена, така че да стигне до крайната цел

Пример 1: Можем да ползваме рекурсия за смятането на факториел.
Нека да помислим върху това как може да разгледаме тази задача като един куп по-малко задачи.

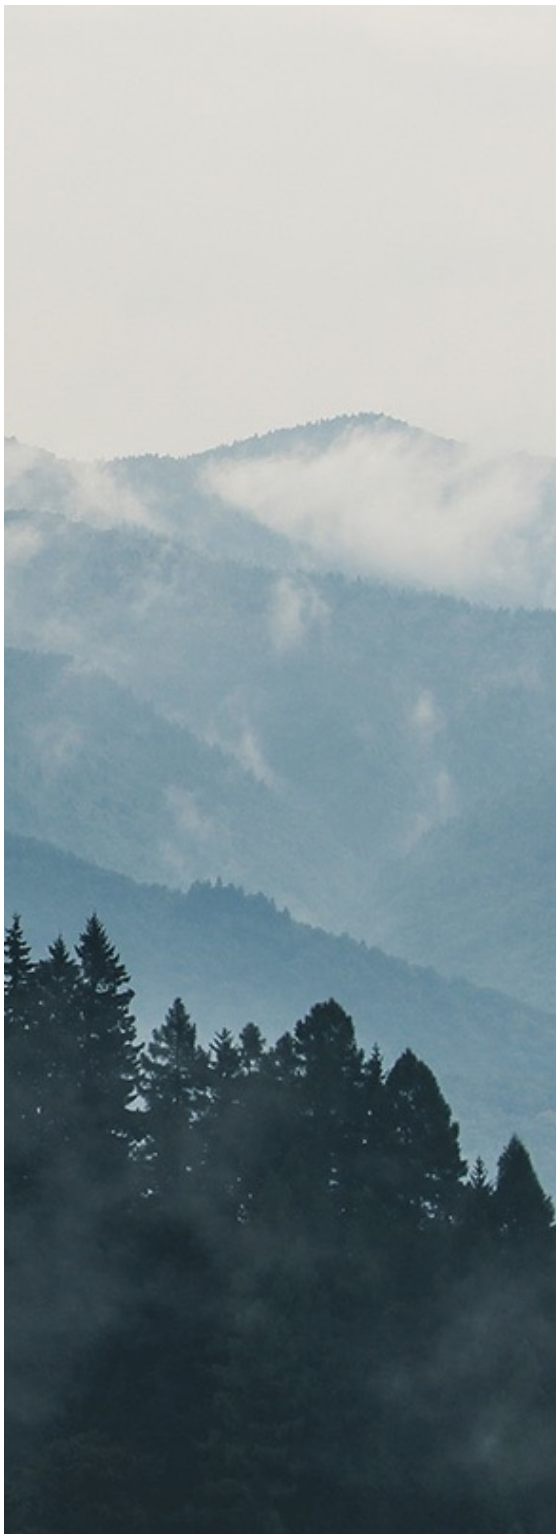
Например 5! е 1x2x3x4x5, но това може да го напишем и като 5x4!, 4! е и 4x3!, 3! = 3x2!, 2! = 2x1! и 1! = 1

Какво виждаме по-горе? Че наистина можем да разделим голямата на по-малки – като умножим сегашното число по факториел от предишното. Тук идва и базовия случай – кога ще спрем рекурсията? Ами когато стигнем до най-простия факториел – 1! = 1
Добре е да добавим и 0 като базов случай, защото user-ите са винаги бабаити и ще пробват най-невероятните неща :P

И така функцията ни в псевдо-код ще стане:

```
factorial(n){

     //base case
     if(n==0 || n==1)
          return 1;

     return n * factorial(n-1);
}
```

Пример 2: Искаме да намерим най-голям общ делител на две числа, да ги наречем A и B, и ще приемем, че A е винаги по-голямо или равно на B. За да намерим НОД може да ползваме алгоритъма на Евклид.



(Нека A = 84, B = 18)

1. A / B = 84/18 = 4 + остатък 12 (R1)
2. B/ R1 = 18/12 = 1 + остатък 6 (R2)
3. R1/R2 = 12/6 = 2 + остатък 0

=> НОД(84, 18) = R2 = 6

Какво виждаме от алгоритъма? Първо, ясно е, че базовият случай е когато остатъкът е 0 и трябва да върнем числото, на което сме делили (или тъй като не ни интересува самото делене → модуло операцията). Ето как ще изглежда като псевдо-код:

```
int GCD(A, B){
    remainder = A%B;

    if( remainder == 0)
        return B;

    return GCD(B, remainder)
}
```

Забележете, че горните две рекурсии всъщност са различни видове. Първата е augmenting recursion, защото има допълнителна операция (умножение), която трябва да се извърши след връщане от рекурсивната функция. Втората е tail recursion, защото няма допълнителни операции след връщане от рекурсивната функция.

ЗАДАЧА 0

Да видим дали разбрахте

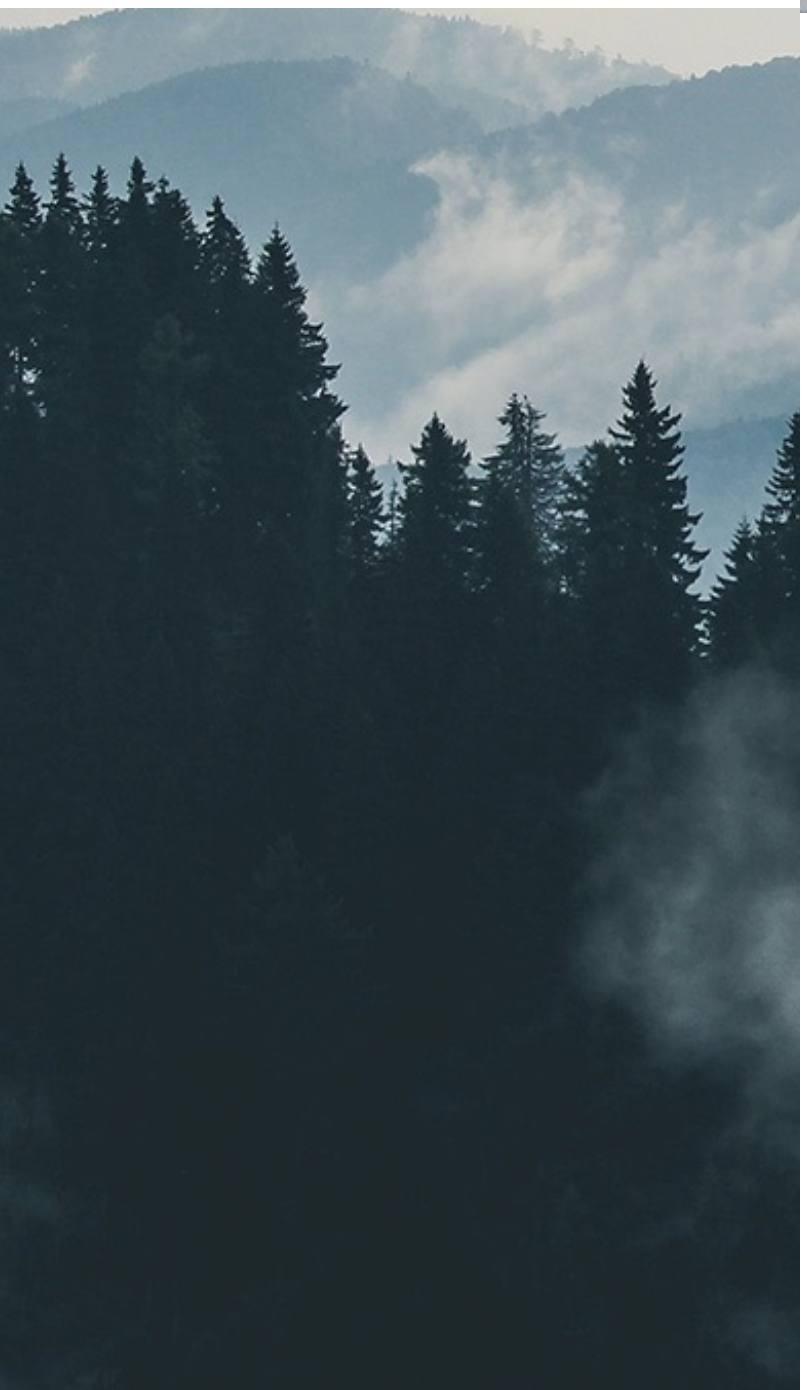
Ок, опитайте се на ръка да решите какво ще върне долната функция за дадените инпути и какво всъщност прави функцията. Знам, че е лесно да я препишете и да я пуснете, но идеята е да се научите. Няколкото минути, които ще изгубите, ще са ви от полза :)

```
mystery(2, 25) = _________________

mystery(3, 35) = _________________

mystery(a, b):
```

```
public int mystery(int a, int b){
 if (b == 0)
  return 0;
 else if (b % 2 == 0)
  return mystery(a + a, b / 2);
 else
  return mystery(a + a, b / 2) + a;
}
```

ЗАДАЧА 1

Най-после задачата!!!!



Хари Бовик е доста нервен програмист. Той иска възможно най-бързо да му направите програмка, която да парсва BER-TLV пакети, върнати като данни от разплащателни карти, за да може той да си върши по-лесно работата. Има следните изисквания към самата програмка:

i. При стартиране да може си въведе пакета от данни
ii. програмата да може да се справи с пакети, в които всички байтове са залепени един за друг (“9F02060000000001000”) и такива, в които има празно място между байтовете (“9F 02 06 00 00 00 00 10 00”).
iii. Програмката да му принтира всички тагове в подадения стринг заедно с дължината и стойността им. За бонус точки – подредени йерархично, едно под друго.

BER-TLV 1

BER-TLV е стандартизиран формат за данни по ISO/IEC 8825. Форматът съдържа в себе си три елемента и се ползва най-вече в банковите карти. Трите елемента в BER-TLV дата са:

1. Таг (T) - състои се от един или повече поредни байта. Определя класа, вида и номера на тага.
2. Дължина (L - Length) - състои се от един или повече поредни байта. Показва дължината на данни за определения таг.
3. Стойност (V - Value) - данните, които се отнасят за този таг.

Ето един пример за TLV дата обект: 9F02060000000001000

Според правила, които ще опиша по-надолу, този обект всъщност е

Таг: 9F02 с дължина 6 байта, като тези 6 байта са “000000001000”

Има две категории ТЛВ обекти:

1. Примитивни като 9F02, които съдържат в себе си само стойността на тага

2. Конструирани (constructed), където стойността съдържа един или повече примитивни или конструирани тагове. Стойността на такъв таг се нарича template. (може би се досещате вече къде ще ни е нужа рекурсията).

По-долу са описани правилата за как да определите кои байтове са тага и дължината:

BER-TLV 2

B1 Coding of the Tag Field of BER-TLV Data Objects

Table 35 describes the first byte of the tag field of a BER-TLV data object:

b8	b7	b6	b5	b4	b3	b2	b1	Meaning
0	0							Universal class
0	1							Application class
1	0							Context-specific class
1	1							Private class
		0						Primitive data object
		1						Constructed data object
			1	1	1	1	1	See subsequent bytes
			Any other value <31					Tag number

Table 35: Tag Field Structure (First Byte) BER-TLV

According to ISO/IEC 8825, Table 36 defines the coding rules of the subsequent bytes of a BER-TLV tag when tag numbers ≥ 31 are used (that is, bits b5 - b1 of the first byte equal '11111').

b8	b7	b6	b5	b4	b3	b2	b1	Meaning
1								Another byte follows
0								Last tag byte
	Any value > 0							(Part of) tag number

Table 36: Tag Field Structure (Subsequent Bytes) BER-TLV

Before, between, or after TLV-coded data objects, '00' bytes without any meanin may occur (for example, due to erased or modified TLV-coded data objects).

BER-TLV 3

B2 Coding of the Length Field of BER-TLV Data Objects

When bit b8 of the most significant byte of the length field is set to 0, the length field consists of only one byte. Bits b7 to b1 code the number of bytes of the value field. The length field is within the range 1 to 127.

When bit b8 of the most significant byte of the length field is set to 1, the subsequent bits b7 to b1 of the most significant byte code the number of subsequent bytes in the length field. The subsequent bytes code an integer representing the number of bytes in the value field. Two bytes are necessary to express up to 255 bytes in the value field.

В общи линии задачата ви е използвайки информацията по-горе да напишете код, използващ рекурсия, който от масив с данни в TLV формат да върне таговете и данните, които са в този масив. Ако искате да видите примерно решение в действие:
https://www.emvlab.org/tlvutils/
http://extranet.cryptomathic.com/tlvutils/index

Примерен инпут:
61 1F 4F 08 A0 00 00 00 25 01 05 01 50 10 50 65 72 73 6F 6E 61 6C 20 41 63 63 6F 75 6E 74 87 01 01

61 1E 4F 07 A0 00 00 00 29 10 10 50 10 50 65 72 73 6F 6E 61 6C 20 41 63 63 6F 75 6E 74 87 01 02

77 22 82 02 78 00 94 1C 10 01 04 00 10 05 05 00 08 06 06 01 08 07 07 01 08 08 09 01 08 0A 0A 00 08 01 04 00

77 1E 9F 27 01 80 9F 36 02 02 13 9F 26 08 2D F3 83 3C 61 85 5B EA 9F 10 07 06 84 23 00 31 02 08

Още полезна информация:
https://www.openscdp.org/scripts/tutorial/emv/tlv.html