Observações sobre Recursão primitiva

Recursão primitiva funciona sobre naturais

In [2]:

```
def parNat(n):
# pré-condição: n >= 0
    if n == 0:
        return True
    else:
        return not parNat(n-1)

parNat(-2)
```

Out[2]:

```
True
```

parNat(-2) → not parNat(-3) → not not parNat(-4) → ...

Recursão infinita é uma *dor de cabeça* comum em iniciantes

Podemos estender a definição para todos os inteiros assim:

In [3]:

```
def par(n):
    if n >= 0:
        return parNat(n)
    else:
        return parNat(-n)

par(-2)
```

Out[3]:

```
True
```

Recursão Geral

Na recursão primitiva sobre um natural `n` a chamada recursiva é com `n-1` (ou um elemento a menos em strings ou listas).

No entanto, nem tudo é possível com recursão primitiva assim como também existem soluções recursivas que não são primitivas.

Por exemplo, outra solução para verificar se um número é par é

In [6]:

```
def parNat(n):
    if n == 0:
        return True
    elif n == 1:
        return False
    else:
        return parNat(n-2)

par(9)
```

Out[6]:

```
False
```

Como definir com segurança funções recursivas

A metáfora do amigo ainda funciona, mas precisa dos seguintes cuidados:

1. A chamada recursiva é feita com um **tamanho de instância do problema** menor (cada chamada se aproxima dos casos base)
2. Garantir que sempre é alcançado algum caso base
3. Verificar se os casos base e a parte recursiva são consistentes com as respectivas condições

Exemplo: Dados dois números m e n, queremos calcular o produto de todos os números que estão entre m e n, inclusive

$$m \times (m+1) \times (m+2) \ldots \times (n-1) \times n$$

Tentemos recursão em n

In [9]:

```
def produtoDeAte(m, n):
    if m > n:
        return 1
    else:
        return produtoDeAte(m, n-1) * n

produtoDeAte(5, 5)
```

Out[9]:

```
5
```

Alternativamente temos esta outra solução

In [10]:

```
def produtoDeAte(m, n):
    if m > n :
        return 1
    else:
        return m * produtoDeAte(m+1, n)

produtoDeAte(3, 5)
```

Out[10]:

```
60
```

A chamada é feita com um número maior? Neste caso a recursão não é em `m`, mas sim no tamanho do intervalo `[m,n]`

Em outras palavras, o tamanho da instância é o tamanho do intervalo.

Exemplo: Divisão inteira

Definir uma função que calcule a divisão inteira de dois números sem usar `//`

Idéia: subtrações sucessivas. A resposta é o número de vezes que conseguimos substrair o divisor do dividendo

Exemplo, queremos dividir 16 fatias de bolo para 5 pessoas

$$16 \rightarrow 11 \rightarrow 6 \rightarrow 1$$

3 com resto 1

In [ ]:

```
def div(p, q):
# pré-condição: q != 0
    if p < q :
        return 0
    else:
         1 + div(p-q, q)
```

Podemos também usar recursão geral em listas ou strings tendo os mesmos cuidados citados acima.

Exemplo: Verificar se um texto é palíndromo

Roma me tem amor

In [9]:

```
def frasePalindroma(txt):
    txt = txt.replace(" ", "")   # txt sem espaços
    txt = txt.upper()            # txt sem espaços e em maiúsculas
    return palavraPalindroma(txt)

frasePalindroma("Roma me tem amor")
```

Out[9]:

```
True
```

palavraPalindroma:

$$"ROMAMETEMAMOR"$$
$$\uparrow \qquad\qquad\qquad\qquad \uparrow$$

In [6]:

```
def palavraPalindroma(txt):
    if len(txt) <=1 :
        return True
    else:
        return txt[0] == txt[len(txt)-1] and palavraPalindroma(txt[1 : len(txt)-1])
```