Instituto Superior Técnico

Lic. em Matemática Aplicada e Computação
Mestrado Integrado em Eng. Biomédica

Elementos de Programação

2 de Fevereiro de 2018 Exame 2 Duração: 2h30

Número: _____ **Nome:** _____

Grupo I (5 valores)

Pelo *teorema fundamental da aritmética*, todo o número inteiro positivo tem uma factorização única como produto de números primos.

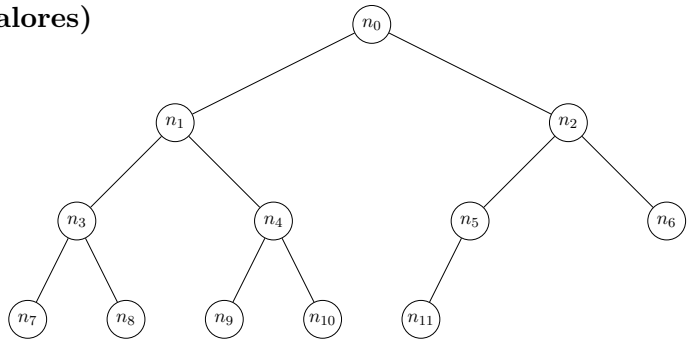
Defina imperativamente em *Python* uma função `factor` que dado um inteiro positivo `n` devolva a factorização de `n` na forma de uma lista de pares $[(p_1, e_1), \ldots, (p_k, e_k)]$ onde cada p_i é um número primo e cada e_i é um inteiro positivo tais que $n = {p_1}^{e_1} \times \cdots \times {p_k}^{e_k}$.

Por exemplo, `factor(200)` deverá ser $[(2,3),(5,2)]$.

Resolução:

```
def factor(n):
    res=[]
    p=2
    while n!=1:
        e=0
        while n%p==0:
            n=n//p
            e=e+1
        if e!=0:
            res=res+[(p,e)]
        p=p+1
    return res
```

Grupo II (4+4 valores)



A árvore binária desenhada acima tem os nós numerados sequencialmente, de cima para baixo e da esquerda para a direita (cada n_i é o valor contido no nó i). Como em qualquer árvore binária, cada nó pai (num certo nível da árvore) tem no máximo dois nós filhos (no nível imediatamente abaixo). Por exemplo, o nó 1 tem como filhos os nós 3 e 4. O nó 5 tem apenas um filho, o nó 11. O nó 11, por sua vez, não tem filhos. A árvore acima diz-se *completa* pois todos os seus níveis estão completamente preenchidos, possivelmente à excepção do último, e nesse último nível os nós ocupados estão encostados à esquerda. O próximo nó disponível na árvore corresponderia ao nó 12, o filho mais à direita do nó 5; depois os dois nós filhos do nó 6; e a seguir os filhos do nó 7, etc.

Considere o tipo de dados *árvore binária completa de inteiros*, em que os nós contêm números inteiros, com as seguintes operações:

- `void()`: árvore vazia;
- `add(t,n)`: árvore que resulta de adicionar o inteiro `n` no próximo nó disponível da árvore `t`;
- `val(t,i)`: valor contido no nó `i` da árvore `t`;
- `swap(t,i)`: árvore que resulta da árvore `t` trocando o valor contido no nó `i` com o valor do seu pai;
- `size(t)`: número de nós da árvore `t`;
- `heapQ(t)`: `True` se e só se a árvore binária completa de inteiros `t` é uma *heap*, isto é, se cada nó contém um valor maior ou igual que os valores contidos nos seus filhos.

a) Desenvolva em *Python* uma implementação eficiente deste tipo de dados, de modo a que cada árvore binária completa de inteiros seja representada simplesmente por uma lista da forma $[\mathtt{n_0}, \mathtt{n_1}, \mathtt{n_2}, \ldots, \mathtt{n_{k-1}}]$ onde `k` é o número de nós da árvore, e cada $\mathtt{n_i}$ é o valor contido no nó i.

Resolução:

```
def void():
    return []

def add(t,n):
    return t+[n]

def val(t,i):
    assert i<len(t)
    return t[i]

def pai(i):
    assert i!=0
    return (i-1)//2

def swap(t,i):
    assert i!=0
    j=pai(i)
    t[i],t[j]=t[j],t[i]
    return t

def size(t):
    return len(t)

def heapQ(t):
    return all(t[i]<=t[pai(i)] for i in range(1,len(t)))
```

b) Desenvolva em *Python*, sobre a camada de abstracção acima desenvolvida e assegurando a independência da implementação, uma função `siftUP` que, recebendo uma *heap* `t` e um inteiro `n` devolve uma *heap* com todos os valores dos nós de `t` e um nó adicional com o valor `n`. Sugestão: adicione `n` no fundo de `t` e use a operação `swap` para manipular a árvore obtida de forma a garantir a propriedade de *heap*.

Resolução:

```
def pai(i):
    assert i!=0
    return (i-1)//2

def siftUP(t,n):
    i=size(t)
    t=add(t,n)
    while i!=0 and val(t,i)>val(t,pai(i)):
        t=swap(t,i)
        i=pai(i)
    return t
```

Grupo III (4 valores)

Neste exercício não pode usar recursão, ciclos ou atribuições. Pode usar, sem necessitar de os definir, os combinadores `map`, `reduce`, `any`, `all`, `filter`, `nest`, `fixedpoint`, bem como definições `lambda` e por compreensão.

Atente nas seguintes definições:

- inteiros positivos $n_1, \ldots, n_k$ dizem-se *coprimos entre si* se para qualquer número primo p, se p divide n_i então p não divide n_j para nenhum $j \neq i$;
- se a factorização em primos de um inteiro positivo é $n = p_1^{e_1} \times \cdots \times p_k^{e_k}$ então define-se o seu *radical* $\mathrm{rad}(n) = p_1 \times \cdots \times p_k$, i.e., $\mathrm{rad}(n)$ é o produto dos primos que dividem n;
- a *qualidade* $q(a, b, c)$ de um triplo a, b, c de inteiros positivos coprimos entre si é definida por

$$q(a, b, c) = \frac{\log(c)}{\log(\mathrm{rad}(a).\mathrm{rad}(b).\mathrm{rad}(c))}.$$

A *conjectura abc*, um famoso e importante resultado em aberto em teoria de números, afirma que para qualquer $\epsilon > 0$ existe apenas uma quantidade finita de triplos a, b, c inteiros positivos e coprimos entre si, com $a + b = c$, tais que $q(a, b, c) > 1 + \epsilon$.

Implemente funcionalmente em *Python* a função `best_abc` que dado um inteiro positivo `c` devolve `(a,b,q)`, onde `a,b,c` é o triplo com o maior valor de `q(a,b,c)` de entre os triplos possíveis, fixado `c`, e `q` é essa mesma qualidade. Pode (e deve) usar a função `factor` do **Grupo I**, mesmo se não a definiu.

Resolução:

```
def coprimesQ(a,b,c):
    return all([p[0]!=q[0] for p in factor(a) for q in factor(b)]) and all(
        [p[0]!=q[0] for p in factor(a) for q in factor(c)]) and all(
        [p[0]!=q[0] for p in factor(b) for q in factor(c)])

def rad(n):
    return reduce(lambda x,y:x*y,[p[0] for p in factor(n)],1)

def q(a,b,c):
    return log(c)/log(rad(a)*rad(b)*rad(c))

def best_abc(c):
    return max([(q(a,c-a,c),a,c-a) for a in range(1,1+(c-1)//2) if coprimesQ(a,c-a,c)])
```

A resolução anterior é bastante ineficiente, pois a factorização é uma operação pesada. Apresenta-se abaixo uma solução mais eficiente, que evita calcular (ou recalcular) factorizações. Em particular, redefine-se a coprimalidade usando o *máximo divisor comum* via algoritmo de Euclides. Também se evita recalcular a factorização de `c`, armazenando o valor do seu radical.

```
def gcd(a,b):
    def euclid(pair):
        if pair[0]==0:
            return pair
        else:
            return (pair[1]%pair[0],pair[0])

    return fixedpoint(euclid,(a,b))[1]

def coprimesQ(a,b,c):
    return gcd(a,b)==gcd(a,c)==gcd(b,c)==1

def rad(n):
    return reduce(lambda x,y:x*y,[p[0] for p in factor(n)],1)

def best_abc(c):
    def best_ab(rc):
        def q(a,b):
            return log(c)/log(rad(a)*rad(b)*rc)

        return max([(q(a,c-a),a,c-a) for a in range(1,1+(c-1)//2) if coprimesQ(a,c-a,c)])

    return best_ab(rad(c))
```

Grupo IV (3 valores)

Considere o seguinte programa imperativo PROG.

```
i=len(w)
r=0
while i!=0:
    i=i-1
    r=w[i]+x*r
```

Demonstre que é válida a asserção

$$\{\texttt{x} == 1\}\ \texttt{PROG}\ \{\texttt{r} == \sum_{0 \leq \texttt{j} < \texttt{len(w)}} \texttt{w[j]}\}.$$

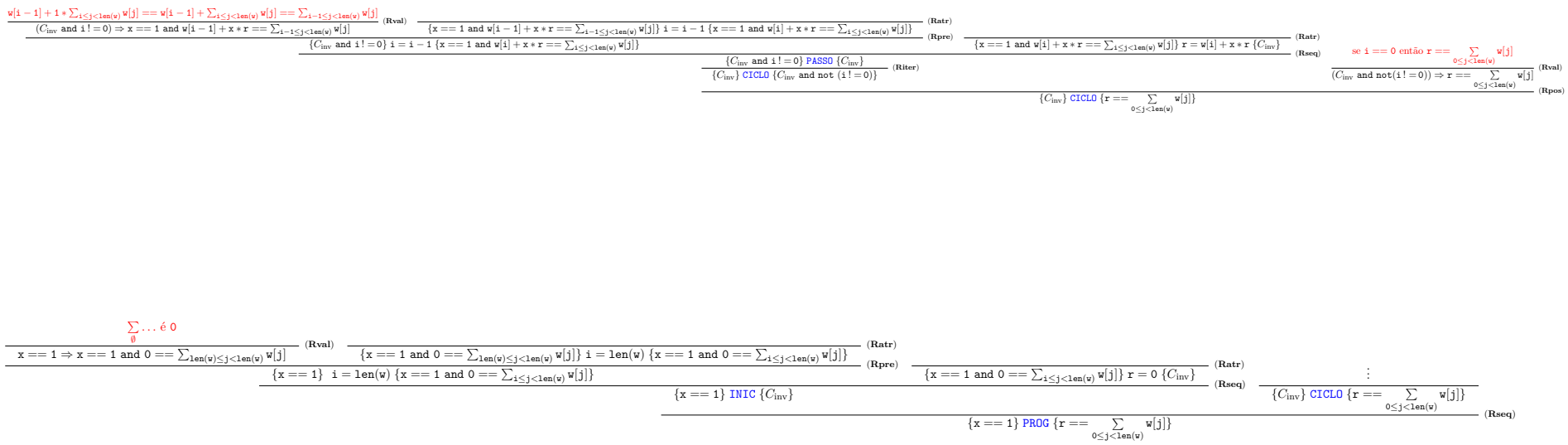
Resolução: Considera-se a estrutura de ciclo inicializado

$$\texttt{PROG} \left\{ \begin{array}{l} \left.\begin{array}{l} \texttt{i} = \texttt{len(w)} \\ \texttt{r} = 0 \end{array}\right\} \texttt{INIC} \\ \left.\begin{array}{l} \texttt{while i} \,!= 0: \\ \quad \left.\begin{array}{l} \texttt{i} = \texttt{i} - 1 \\ \texttt{r} = \texttt{w[i]} + \texttt{x} * \texttt{r} \end{array}\right\} \texttt{PASSO} \end{array}\right\} \texttt{CICLO} \end{array} \right.$$

e a condição invariante do ciclo

$$C_{\text{inv}} \equiv (\texttt{x} == 1 \texttt{ and } \texttt{r} == \sum_{\texttt{i} \leq \texttt{j} < \texttt{len(w)}} \texttt{w[j]}).$$

A demonstração da asserção segue abaixo, onde surgem a vermelho justificações para a validade das condições Booleanas.


w[i − 1] + 1 ∗ ∑ i≤j<len(w) w[j] == w[i − 1] + ∑ i≤j<len(w) w[j] == ∑ i−1≤j<len(w) w[j]
(C inv and i != 0) ⇒ x == 1 and w[i − 1] + x ∗ r == ∑ i−1≤j<len(w) w[j]
(Rval)
{x == 1 and w[i − 1] + x ∗ r == ∑ i−1≤j<len(w) w[j]} i = i − 1 {x == 1 and w[i] + x ∗ r == ∑ i≤j<len(w) w[j]}
(Ratr)
{C inv and i != 0} i = i − 1 {x == 1 and w[i] + x ∗ r == ∑ i≤j<len(w) w[j]}
(Rpre)
{x == 1 and w[i] + x ∗ r == ∑ i≤j<len(w) w[j]} r = w[i] + x ∗ r {C inv}
(Ratr)
{C inv and i != 0} PASSO {C inv}
{C inv} CICLO {C inv and not (i != 0)}
(Riter)
(Rseq)
se i == 0 então r == ∑ 0≤j<len(w) w[j]
(C inv and not(i != 0)) ⇒ r == ∑ 0≤j<len(w) w[j]
(Rval)
{C inv} CICLO {r == ∑ 0≤j<len(w) w[j]}
(Rpos)
∑ ∅ ... é 0
x == 1 ⇒ x == 1 and 0 == ∑ len(w)≤j<len(w) w[j]
(Rval)
{x == 1 and 0 == ∑ len(w)≤j<len(w) w[j]} i = len(w) {x == 1 and 0 == ∑ i≤j<len(w) w[j]}
(Ratr)
{x == 1} i = len(w) {x == 1 and 0 == ∑ i≤j<len(w) w[j]}
(Rpre)
{x == 1 and 0 == ∑ i≤j<len(w) w[j]} r = 0 {C inv}
(Ratr)
{x == 1} INIC {C inv}
(Rseq)
⋮
{C inv} CICLO {r == ∑ 0≤j<len(w) w[j]}
(Rseq)
{x == 1} PROG {r == ∑ 0≤j<len(w) w[j]}