Instituto Superior Técnico

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

Elementos de Programação

18 de Janeiro de 2019 Exame 1 Duração: 2h30

Grupo I (5 valores)

Neste exercício não pode usar definições por compreensão nem métodos. As únicas operações sobre listas permitidas são: lista vazia ([]), acesso aos elementos da lista por posição (lista[posição]), seccionamento da lista (lista[posição:posição]), comparação com a lista vazia (==[]), cálculo do comprimento (`len`) e concatenação (+).

Suponha que a cada lista de números `w` da forma $[\mathtt{a}_0, \mathtt{a}_1, ..., \mathtt{a}_n]$ se associa o polinómio $p_{\mathtt{w}}(x) = \mathtt{a}_0 + \mathtt{a}_1 x + \mathtt{a}_2 x^2 + ... + \mathtt{a}_n x^n$.

Defina imperativamente em *Python* uma função `prod` que dadas duas listas de números `u` e `v` devolva a lista `r` tal que $p_{\mathtt{r}}(x) = p_{\mathtt{u}}(x) \times p_{\mathtt{v}}(x)$. Por exemplo, `prod`$([3,1,1],[1,2])$ deverá resultar na lista $[3,7,3,2]$, já que se tem $(3 + x + x^2) \times (1 + 2x) = 3 + 7x + 3x^2 + 2x^3$.

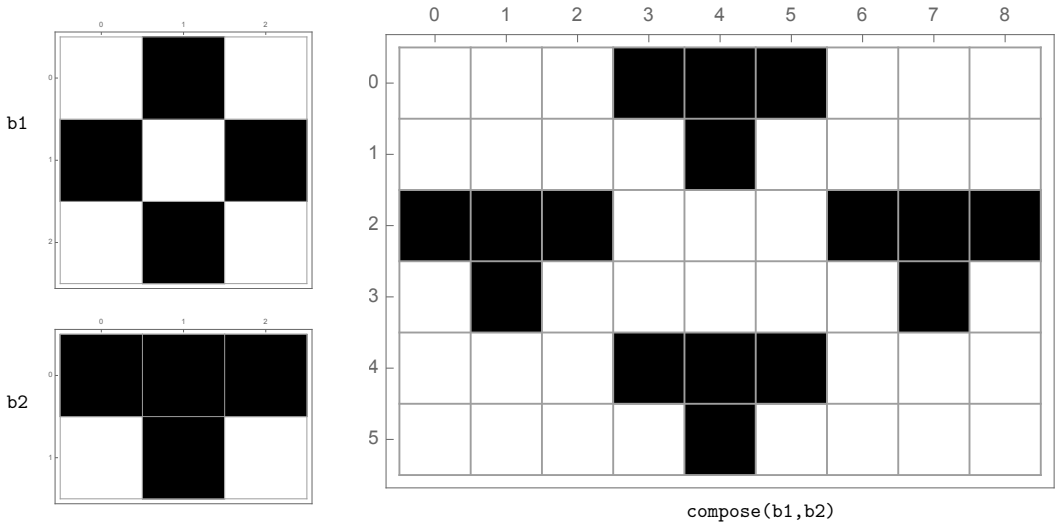
Resolução:

```
def prod(u,v):
    n=len(u)-1
    m=len(v)-1
    r=[]
    k=0
    while k<=n+m:
        c=0
        i=0
        while i<=k and i<=n:
            j=k-i
            if j<=m:
                c=c+u[i]*v[j]
            i=i+1
        r=r+[c]
        k=k+1
    return r
```

Grupo II (4+4 valores) (pesa a nota do projecto)

Considere *bitmaps* (figuras rectangulares, com *pixels* a preto-e-branco) e *filmes* (sequências de bitmaps com as mesmas dimensões) com as operações:

- `white(n,m)`: bitmap com n linhas, m colunas e todos os pixels em branco;
- `pixflip(b,w)`: bitmap que resulta de trocar a cor dos pixels do bitmap `b` nas posições correspondentes à linha `i` coluna `j` para cada par `(i,j)` que ocorre na lista `w`;
- `isblack(b,i,j)`: `True` se é preto o pixel correspondente à linha `i` coluna `j` do bitmap `b`, e `False` caso contrário;
- `compose(b1,b2)`: bitmap em que cada pixel do bitmap `b1` passa a ter as dimensões do bitmap `b2`, ficando em branco se o pixel de `b1` está em branco, ou contendo cópia de `b2` se o pixel de `b1` está a preto (ver exemplo na figura);
- `start(b)`: filme cujo primeiro e único bitmap é `b`;
- `append(f,b)`: filme que resulta de juntar o bitmap `b` no final do filme `f`;
- `size(f)`: número de bitmaps do filme `f`;
- `last(f)`: último bitmap do filme `f`;
- `endat(f,k)`: filme com apenas os primeiros `k` bitmaps do filme `f`.



Em *Python*, pretende-se representar um bitmap como uma matriz de entradas 0,1 (0 para pixels a branco, 1 para pixels a preto), e representar um filme como um par `(b,[d1,...,dk])` onde `b` é o primeiro bitmap do filme, `d1` é a lista das posições que é necessário trocar no primeiro bitmap do filme para obter o segundo, `d2` a lista das posições que é necessário trocar no segundo bitmap do filme para obter o terceiro, e assim por diante.

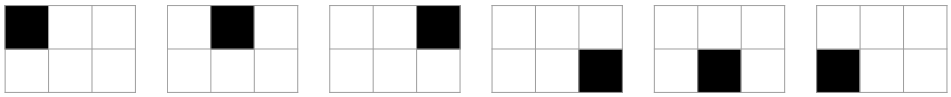
a) Apresente implementações eficientes para as operações identificadas.

Resolução:

```
def white(n,m):
    return [[0 for j in range(m)] for i in range(n)]

def pixflip(bmp,w):
    res=[u[:] for u in bmp]
    for (i,j) in w:
        res[i][j]=1-res[i][j]
    return res

def isblack(bmp,i,j):
    return bmp[i][j]==1

def compose(base,patt):
    res=white(len(base)*len(patt),len(base[0])*len(patt[0]))
    for i in range(len(base)):
        for j in range(len(base[0])):
            if base[i][j]==1:
                for a in range(len(patt)):
                    for b in range(len(patt[0])):
                        if patt[a][b]==1:
                            res[a+i*len(patt)][b+j*len(patt[0])]=1
    return res

def start(bmp):
    return (bmp,[])

def append(mov,bmp):
    (fst,w)=mov
    lst=last(mov)
    return (fst,w+[[(i,j) for i in range(len(bmp)) for j in range(len(bmp[0])) if bmp[i][j]!=lst[i][j]]])

def size(mov):
    (fst,w)=mov
    return 1+len(w)

def last(mov):
    (fst,w)=mov
    return reduce(pixflip,w,fst)

def endat(mov,k):
    (fst,w)=mov
    return (fst,w[:k-1])
```

b) Desenvolva, sobre a camada de abstracção obtida acima e assegurando a independência da implementação, as seguintes funções:

b1) `lap`, que recebendo inteiros positivos `n,m` devolve o filme que resulta de, começando no canto superior esquerdo e sem repetir bitmaps, fazer circular um pixel preto na moldura de um bitmap branco com `n` linhas e `m` colunas no sentido dos ponteiros do relógio;



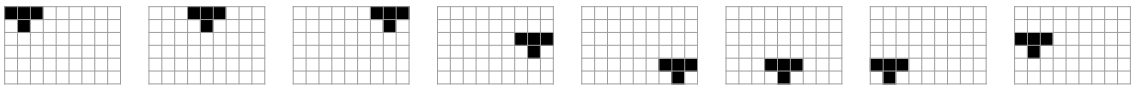
`lap(2,3)`

Resolução:

```
def lap(n,m):
    movie=start(pixflip(white(n,m),[(0,0)]))
    (i,j)=(0,0)
    k=2*(n-1)+2*(m-1)-1
    while k>0:
        if i==0 and j<m-1:
            j=j+1
        elif j==m-1 and i<n-1:
            i=i+1
        elif i==n-1 and j>0:
            j=j-1
        else:
            i=i-1
        movie=append(movie,pixflip(white(n,m),[(i,j)]))
        k=k-1
    return movie
```

b2) `makemovie`, que recebendo um natural `t` devolve o filme correspondente à sequência de bitmaps na figura abaixo (com 1 volta), mas completando `t` voltas.

Sugestão: use a função `lap` como auxiliar, mesmo se não a definiu.



`makemovie(1)`

Resolução:

```
def makemovie(t):
    patt=pixflip(white(2,3),[(0,0),(0,1),(0,2),(1,1)])
    cycle=lap(3,3)
    s=size(cycle)
    for k in range(t*s):
        if k==0:
            movie=start(compose(last(endat(cycle,1)),patt))
        else:
            movie=append(movie,compose(last(endat(cycle,(k%s)+1)),patt))
    return movie
```

Grupo III (4 valores)

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

Um *problema de satisfatibilidade* é representado por uma matriz `m` com entradas `0`, `1` ou `-1`. Dado um tal problema, uma solução de `m` consiste de uma lista `s` com entradas `1` ou `-1`, de comprimento igual ao número de colunas de `m`, que satisfaça a seguinte condição:

- para cada linha `i` de `m` existe pelo menos uma coluna `j` tal que `m[i][j]` é igual a `s[j]`.

Por exemplo, `[1, -1, 1]` é uma solução do problema `[[1,1,1],[-1,-1,0]]`.

Implemente funcionalmente em *Python* uma função `sols` que dado um problema de satisfatibilidade devolve a lista de todas as suas soluções.

Resolução:

```
def sat(prob,sol):
    return all(map(lambda c:any([c[i]==sol[i] for i in range(len(sol))]),prob))

def tolist(n,k):
    def more(t):
        if t[2]==0:
            return t
        else:
            return ([-1 if t[1]%2==0 else 1]+t[0],t[1]//2,t[2]-1)
    return fixedpoint(more,([],n,k))[0]

def sols(prob):
    def evalandnext(p):
        if p[1]==2**len(prob[1]):
            return p
        elif sat(prob,tolist(p[1],len(prob[1]))):
            return (p[0]+[p[1]],p[1]+1)
        else:
            return (p[0],p[1]+1)

    return list(map(lambda n:tolist(n,len(prob[1])),fixedpoint(evalandnext,([],0))[0]))
```

Grupo IV (3 valores)

Considere o seguinte programa imperativo PROG.

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

Demonstre que é válida a asserção

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

Resolução: Considera-se a estrutura de ciclo inicializado

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

e a condição invariante do ciclo

$$C_{\text{inv}} \equiv (\texttt{r} == \sum_{0 \leq \texttt{j} < \texttt{i}} (-1)^{\texttt{j}} * \texttt{w[j]})\ \texttt{and}\ (\texttt{s} == (-1)^{\texttt{i}}).$$

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


se r == $\sum_{0\le j<i}(-1)^j * w[j]$ e a == $(-1)^i$ então
r + a * w[i] é $\sum_{0\le j<i+1}(-1)^j * w[j]$ e
−a é $(-1)^{i+1}$
(Cinv and i ! = len(w)) ⇒ (r + a * w[i] == $\sum_{0\le j<i+1}(-1)^j * w[j]$) and (−a == $(-1)^{i+1}$)
(Rval)
{Cinv and i ! = len(w)} r = r + a * w[i] {(r == $\sum_{0\le j<i+1}(-1)^j * w[j]$) and (−a == $(-1)^{i+1}$)}
(Ratr)
{Cinv and i ! = len(w)} PASSO {Cinv}
{Cinv} CICLO {Cinv and not(i ! = len(w))}
(Riter)
{Cinv} CICLO {r == $\sum_{0\le j<len(a)}(-1)^j * w[j]$}
(Rpre)
{(r == $\sum_{0\le j<i+1}(-1)^j * w[j]$) and (−a == $(-1)^{i+1}$)} i = i + 1 {(r == $\sum_{0\le j<i}(-1)^j * w[j]$) and (−a == $(-1)^i$)}
(Ratr)
{(r == $\sum_{0\le j<i}(-1)^j * w[j]$) and (−a == $(-1)^i$)} a = −a {Cinv}
(Ratr)
(Rseq)
se r == $\sum_{0\le j<i}(-1)^j * w[j]$ e i == len(w) então
r == $\sum_{0\le j<len(a)}(-1)^j * w[j]$
(Cinv and not(i ! = len(w))) ⇒ r == $\sum_{0\le j<len(a)}(-1)^j * w[j]$
(Rval)
$\sum_{\emptyset}$ ... é 0 e $(-1)^0$ é 1
True ⇒ (0 == $\sum_{0\le j<0}(-1)^j * w[j]$) and (1 == $(-1)^0$)
(Rval)
{(0 == $\sum_{0\le j<0}(-1)^j * w[j]$) and (1 == $(-1)^0$)} r = 0 {(r == $\sum_{0\le j<0}(-1)^j * w[j]$) and (1 == $(-1)^0$)}
(Ratr)
{True} r = 0 {(r == $\sum_{0\le j<0}(-1)^j * w[j]$) and (1 == $(-1)^0$)}
(Rpre)
{(r == $\sum_{0\le j<0}(-1)^j * w[j]$) and (1 == $(-1)^0$)} i = 0 {(r == $\sum_{0\le j<i}(-1)^j * w[j]$) and (1 == $(-1)^i$)}
(Ratr)
{(r == $\sum_{0\le j<i}(-1)^j * w[j]$) and (1 == $(-1)^i$)} a = 1 {Cinv}
(Ratr)
(Rseq)
⋮
{Cinv} CICLO {r == $\sum_{0\le j<len(w)}(-1)^j * w[j]$}
(Rseq)
{True} INIC {Cinv}
{True} PROG {r == $\sum_{0\le j<len(w)}(-1)^j * w[j]$}