## 21.1 Aula 21: 24/OUT/2019

## 21.2 Aula passada

Consumo de tempo de operações em listas (`list`) e dicionários (`dict`).

## 21.3 Hoje

Mais programação dinâmica: *Longest common substring*

## 21.4 Arquivos

Há alguns arquivos para fazermos testes.

O arquivo `sub_seq.py` lê duas strings da linha de comando e verifica se a 1a é subsequência da segunda.

O programa `lcsR.py` lê duas strings da linha de comando

```
% python lcsR.py
Uso: python lcsR.py s t [-s]
    s = string
    t = string
    [-s] = mostra chamadas recursivas
```

Já o programa `lcsPD.py` lê o nome de dois arquivos com as strings

```
% python lcsPD.py
Uso: python lcsPD.py s_arq t_arq [-a]
    s_arq = arquivo com string
    t_arq = arquivo com string
```

Os arquivos para brincarmos são `abra.txt` e `yabba.txt`. Há ainda arquivos com strings sobre o alfabeto `"ACGT"`. Um é genoma de um virus `genomeVirus.txt` copiado da COS126 de Princeton. Outros foram geradas aleatoriamente: `dna*.txt`

## 21.5 Subsequência

Uma **subsequência** ou **substring** é uma sequência que aparece na mesma ordem relativa, mas não é necessariamente contígua.

Exemplos:

- `"abc"` é subsequência de `"abcdefg"`.
- `"abg"` é subsequência de `"abcdefg"`.
- `"bdf"` *é subsequência de `"abcdefg"`.
- `"aeg"` é subsequência de `"abcdefg"`.
- `"acefg"` é subsequência de `"abcdefg"`.
- `"gef"` NÃO é subsequência de `"abcdefg"`.

### Problema

Dadas duas strings `z` e `x`, verificar se `z` é uma subsequência de `x`.

```
% sub_seq.py

def sub_seq (z, x):
    ''' (str, str) -> bool
    Recebe duas strings e verifica se z é uma subsequência de x.
    '''
    i = len(z)-1
    j = len(x)-1
    while i >= 0 and j >= 0:
        if z[i] == x[j]:
            i -= 1
        j -= 1
    return i < 0

python sub_seq.py abadaba abracadabra -s
> a
  r
> b
> a
> d
> a
  c
  a
  r
> b
> a
Resposta:  True
```

## 21.6 Longest commom subsequence

*Longest common subsequence* (LCS) = subsequência comum máxima.

### Problema

Dadas duas strings `s` e `t`, encontrar uma subsequência comum máxima entre `s` e `t`.

Uma **subsequência** é uma sequência que aparece na mesma ordem relativa, mas não é necessariamente contígua. Por exemplo, `"abc"`, `"abg"`, `"bdf"`, `"aeg"`, `"acefg"`, etc são subsequências de `"abcdefg"`.

Exemplos:

- a LCS para as sequências `"ABCDGH"` e `"AEDFHR"` é `"ADH"`, com comprimento 3.
- a LCS para as sequências `"AGGTAB"` e `"GXTXAYB"` é `"GTAB"`, com comprimento 4.

### Força bruta

Gerar todas as subsequências de cada sequência, comparar todas e encontrar a subsequência mais longa.

Um string de comprimento `n` tem `2^n` subsequências diferentes possíveis.

### Ideia

Suponha que `r` é `lcs` de `s` e `t`. Suponha ainda que `r`, `s` e `t` têm comprimentos `k`, `m` e `n`, respectivamente.

- se `s[m-1] == t[n-1]`, então `r[0:k-1]` é `lcs` de `s[0:m-1]` e `t[0:n-1]`.
- se `s[m-1] != t[n-1]` e `r[k-1] != s[m-1]`, então `r` é `lcs` de `s[0:m-1]` e `t`.
- se `s[m-1] != t[n-1]` e `r[k-1] != t[n-1]`, então `r` é `lcs` de `s` e `t[0:n-1]`.

### Solução recursiva

A solução se apoia na ideia anterior.

Resolver os subproblemas resultantes da ideia acima.

```
% python lcsR.py
Uso: python lcsR.py s t [-s]
    s = string
    t = string
    [-s] = mostra chamadas recursivas

% python lcsR.py abracadabra yabbadabbadoo
abadaba

% python lcsR.py aba bab -s
lcs_rec(s[0:3],t[0:3])
  lcs_rec(s[0:2],t[0:3])
    lcs_rec(s[0:1],t[0:2])
      lcs_rec(s[0:0],t[0:1])
```

```
      return ''
    return 'a'
  return 'ab'
  lcs_rec(s[0:3],t[0:2])
    lcs_rec(s[0:2],t[0:1])
      lcs_rec(s[0:1],t[0:0])
      return ''
    return 'b'
  return 'ba'
return 'ba'
ba
```

```
#-----------------------------------------
def lcs_rec(s, m, t, n):
    '''(str, str) -> str

    Recebe uma string s de comprimento m e uma string t de
    comprimento n e retorna uma longest common substring de
    de s e t.
    '''
    if m == 0 or n == 0: return ""

    if s[m-1] == t[n-1]:
        return lcs_rec(s, m-1, t, n-1) + s[m-1]

    lcs_1 = lcs_rec(s, m-1, t,   n)
    lcs_2 = lcs_rec(s,   m, t, n-1)
    if len(lcs_1) > len(lcs_2): return lcs_1
    return lcs_2
```

O consumo de tempo é **exponencial**. Recalcula subproblemas várias vezes.

## 21.7 Programação dinâmica

(Dynamic Programming) Método para resolução de problemas complexos que transforma o problema original em uma coleção de subproblemas mais simples, resolvendo cada subproblema uma única vez e armazenando os seus resultados. Da próxima vez que o mesmo subproblema é encontrado, a solução pré computada é utilizada, economizando tempo mas com um gasto (esperamos que modesto) de memória.

A subsequência comum máxima pode ser encontrada construindo-se uma tabela onde os valores das maiores subsequências até um determinado tamanho são armazenadas. A resolução dos subproblemas segue uma política *bottom-up*.

```
   opt[i][j] = len(lcs(s[0:i],t[0:j]))

   opt[i][j] = 0,                              se i == 0 ou j == 0
   opt[i][j] = opt[i-1][j-1] + 1,              se s[i-1] == t[j-1]
   opt[i][j] = max(opt[i-1][j], opt[i][j-1]), em caso contrário
```

```
# Implementação de programação dinâmica para o problema LCS
# Computa o comprimento do LCS para todo subproblema
```

```
def lcs(s, t):
    '''(str, str) -> array

    Recebe uma string s de comprimento m e uma string t de
    comprimento n e retorna a matriz opt com o comprimento dos lcs
    para todo subproblema.
    '''
    # encontre os comprimentos dos strings
    m = len(s)
    n = len(t)

    # declaring the array for storing the dp values
    # opt = [[0]*(n+1) for i in range(m+1)] # crie_matriz(m+1, n+1, 0)
    opt = np.zeros((m+1,n+1))

    # compute as entradas da matriz opt[][] de uma maneira 'bottom up'
    # opt[i][j] contém o comprimento de uma LCS de s[0:i] e t[0:j]
    for i in range(1,m+1):
        for j in range(1,n+1):
            if s[i-1] == t[j-1]:
                opt[i,j] = opt[i-1,j-1] + 1
            else:
                opt[i,j] = max(opt[i-1,j] , opt[i,j-1])

    # opt[m][n] contém o comprimento da LCS de s[0:n] e t[0:m]
    return opt

#-------------------------------------------------------------
def get_lcs(s, t, opt):
    '''(str, str, array) -> str

    RECEBE uma string s e uma string t e a array opt com o
    comprimento dos lcs para todo subproblema e RETORNA uma
    lcs de s e t.
    '''
    lcs = ''
    m, n = len(s), len(t)
    i, j  = m, n
    while i > 0 and j > 0:
        if s[i-1] == t[j-1]:
            lcs = s[i-1] + lcs
            i -= 1
            j -= 1
        elif opt[i-1,j] >= opt[i,j-1]:
            i -= 1
        else:
            j -= 1
    return lcs
```

## 21.8 `difflib`

O algoritmo para computar a máxima subsequência comum é a base do comando `diff` (um programa para comparação de arquivos que mostra as suas diferenças). O Python possui o módulo `difflib`.