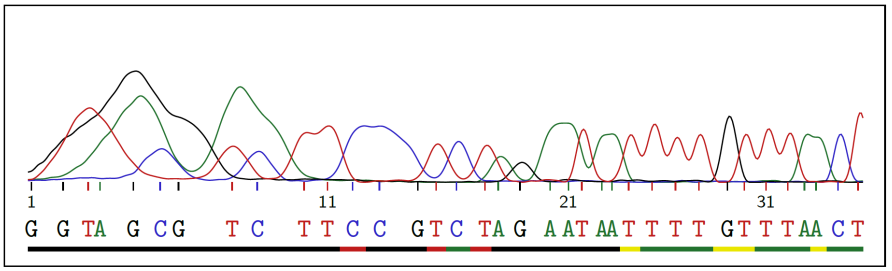
PMR3201 - Computação para Mecatrônica

Prof. Thiago de Castro Martins
Prof. Newton Maruyama
Prof. Marcos de S.G. Tsuzuki
Profa. Larissa Driemeier
Monitor: Eduardo Milanez

Exercício Programa 1 - V2023

Algoritmo de busca da maior subsequência de caracteres em comum: uma aplicação em bioinformática



- **DATA FINAL DE ENTREGA: Terça-Feira 09/05/2023 23h59m**
- **O exercício deve ser feito individualmente.**
- **Submeta através do sistema MOODLE:**
  - **Código fonte formato *.ipynb**

1 Introdução

Existem dois algoritmos de manipulação de caracteres que são bastante utilizados e que eventualmente são confundidos:

1. Algoritmo de busca da maior *substring* em comum (longest commom substring): consiste em comparar duas *strings* e determinar qual a maior *substring* em comum. Nesse caso, *substring* se refere a uma cadeia de caracteres necessariamente contíguos.

2. Algoritmos de busca da maior subsequência em comum (longest commom subsequence): consiste em comparar duas sequências de caracteres e determinar qual a maior subsequência em comum. Nesse caso, subsequência se refere a uma cadeia de caracteres não necessariamente contíguos.

É importante notar que no primeiro caso utiliza-se a terminologia *substring* e no segundo utiliza-se a terminologia subsequência.

Nesse Exercício Programa serão desenvolvidos algoritmos de busca da maior subsequência em comum.

Algoritmos de manipulação de caracteres são utilizados em diversas áreas como: editores de texto, processamento de linguagem natural, máquinas de busca para *web*, filtros de *spam*, bioinformática (busca de padrões em sequências de RNA/DNA ou de proteínas), etc.

Na área de bioinformática uma das principais tarefas é denominada alinhamento de sequências onde duas sequências de DNA são comparadas globalmente e localmente como ilustrado na Figura 1.

Figura 1: Alinhamento global e local.

Durante o processo de sequenciamento são obtidos trechos de DNA incompletos e eventualmente com erros de detecção o que torna a tarefa de alinhamento bastante complexa.

O alinhamento permite tentar descobrir o grau de semelhança entre duas sequências permitindo vários tipos de inferências:

- permite analisar se sequências, que correspondem a proteínas específicas, estão presentes no DNA de alguma espécie ou indivíduo;
- permite analisar se diferenças entre caracteres na sequência de DNA de indivíduos da mesma espécie correspondem a mutações genéticas;
- permite analisar a sequência de DNA de um indivíduo humano em relação a uma base de dados de populações para determinação dos grupos étnicos ancestrais.
- etc.

Como exemplos de *softwares* de bioinformática podemos citar:

- BLAST desenvolvido pelo NCBI National Center for Biotechnology Information [1].
- Biopython desenvolvido na linguagem Python pela comunidade de biologia molecular [2].

Os *softwares* de alinhamento de sequências utilizam algoritmos bastante complexos mas tem em sua base o algoritmo de busca de subsequências que será explorado nesse Exercício Programa.

2 Maior subsequência de caracteres

Uma solução para se obter o comprimento l da maior subsequência de caracteres em comum entre duas cadeias de caracteres X e Y, respectivamente de tamanho m e n, pode ser representada pelo algoritmo recursivo, usualmente denominado *The Naive Solution* que é detalhado a seguir:

O príncipio da recursão consiste em decompor o problema em problemas menores, i.e., comparando cadeias de caracteres com comprimento menor até atingir uma cadeia de caracteres de comprimento nulo quando então começa o retorno para instâncias superiores. Se os caracteres correspondentes à última posição forem coincidentes, i.e., $X[m-1] = Y[n-1]$ então pode-se escrever uma solução

[1] https://blast.ncbi.nlm.nih.gov/Blast.cgi
[2] http://www.biopython.org

Algorithm 1 Maior subsequencia comum: The naive solution

```
1: function LCS(X, Y, m, n)                          ▷ longest common subsequence
2:     if m = 0 or n = 0 then
3:         return(0);
4:     else if X[m − 1] = Y[n − 1] then
5:         return(1 + LCS(X, Y, m − 1, n − 1));
6:     else
7:         return(max(LCS(X, Y, m, n − 1), LCS(X, Y, m − 1, n));
8:     end if
9: end function
```

através da seguinte forma recursiva $\mathrm{LCS}(X, Y, m, n) = 1 + \mathrm{LCS}(X, Y, m-1, n-1)$ caso contrário a solução realiza duas recursões uma retirando o último caracter da cadeia Y e outra retirando o último caracter da cadeia X, $\mathrm{LCS}(X, Y, m, n) = \max(\mathrm{LCS}(X, Y, m, n-1), \mathrm{LCS}(X, Y, m-1, n))$. A solução é dada pelo máximo entre os dois comprimentos.

Note que o algoritmo apresentado devolve apenas o comprimento da maior subsequência comum, se for desejado determinar quais caracteres pertencem a essa subsequência basta armazenar os caracteres ao longo do processo. Pode não existir nenhuma solução ($l = 0$) ou pode existir mais de uma solução.

Por exemplo, para $X =$`"AGGTAB"` ($m = 6$) e $Y =$`"GXTXAYB"` ($n = 7$) a função $\mathrm{LCS}(X, Y, m, n)$ retorna o valor $l = 4$ com a maior subsequência dada por `"GGTAB"`

O algoritmo recursivo tem complexidade exponencial pois testa todas as combinações possíveis, incluindo sequências de X e Y que eventualmente já foram testadas anteriormente.

2.0.1 Programação dinâmica

Para diminuir a complexidade do algoritmo é possível utilizar o conceito de programação dinâmica em que todos os resultados intermediários são armazenados numa tabela como ilustrado no Algoritmo 2.

Algorithm 2 Maior subsequencia comum: utilização de programação dinâmica

```
1: function LCS(X,Y,m,n)
2:     for i ← 0 to m do
3:         for j ← 0 to n do
4:             if i = 0 or j = 0 then
5:                 L[i][j] ← 0
6:             else if X[i − 1] = Y[j − 1] then
7:                 L[i][j] ← L[i − 1][j − 1] + 1
8:             else
9:                 L[i][j] = max(L[i − 1][j], L[i][j − 1])
10:            end if
11:        end for
12:    end for
13:    return(L[m][n])
14: end function
```

Utiliza-se como tabela um *array* L de dimensões $(m+1, n+1)$ para armazenar as soluções intermediárias.

		0	1	2	3	4	5	6	7
		Ø	M	Z	J	A	W	X	U
0	Ø	0	0	0	0	0	0	0	0
1	X	0	0	0	0	0	0	1	1
2	M	0	1	1	1	1	1	1	1
3	J	0	1	1	2	2	2	2	2
4	Y	0	1	1	2	2	2	2	2
5	A	0	1	1	2	3	3	3	3
6	U	0	1	1	2	3	3	3	4
7	Z	0	1	2	2	3	3	3	4

Tabela 1: Tabela $L[][]$ após a execução do algoritmo que utiliza programação dinâmica.

Por exemplo, para as *strings* X =`"XMJYAUZ"` ($m = 7$) e Y =`"MZJAWXU"` ($n = 7$) a função $\text{LCS}(X, Y, m, n)$ retorna o valor $l = 4$ com a maior subsequência dada por `"MJAU"`

O estado final do *array* $L[][]$ está ilustrado na Tabela 1. A célula do *array* $L[m][n]$ contêm o comprimento $l = 4$ da maior subsequência de caracteres em comum.

A complexidade desse algoritmo é quadrática $\mathcal{O}(m \times n)$, como visto em aula

O algoritmo apresentado calcula somente o comprimento l, no entanto, a subsequência de caracteres associada pode ser gerada concomitantemente ao cálculo do comprimento armazenando os caracteres para cada caso positivo de coincidência de caracter. Deve ser lembrado que durante o preenchimento da tabela L, subsequências parciais podem ser de mesmo comprimento.

- Por exemplo, quando da execução da instrução: $\max(L[i-1][j], L[i][j-1])$,
- Se $L[i-1][j] = L[i][j-1]$ então existem duas subsequências que devem ser propagadas.
- Dessa forma, concluímos que eventualmente pode haver mais de uma subsequência de mesmo comprimento l.

Uma forma alternativa para gerar a maior subsequência de caracteres pode ser utilizada percorrendo o *array* $L[][]$ a partir da última célula $L[m][n]$ com índices i e j variando de maneira decrescente.

A Tabela 1 ilustra, na cor amarelo, o caminho a ser percorrido para a obtenção da maior subsequência de caracteres em comum.

Entretanto, esse método apresenta aumento de complexidade quando existe mais de uma solução. Nesse caso é necessário utilizar a técnica de backtracking que vai buscando todas as possíveis soluções de maneira recursiva o que aumenta a complexidade do algoritmo. Uma maneira mais eficiente seria incorporar informações sobre os caminhos percorridos como observado na literatura[3].

No exemplo apresentado no Wikipedia (Veja Tabela 2) $X = GAC$ e $Y = AGCAT$. As maiores subsequências em comum são: (AC), (GC) e (GA).

Note que as flechas na tabela indicam o fluxo do algoritmo e permitem facilmente realizar uma navegação reversa.

Observe que as células que tem número na cor laranja são as que tem caracteres coincidentes. Essas células possuem uma flecha diagonal $\nwarrow$ indicando que o conteúdo dessa célula $L[i][j]$ é resultante do processamento do código:

[3]https://en.wikipedia.org/wiki/Longest_common_subsequence

	ε	A	G	C	A	T
ε	0	**0**	0	0	0	0
G	0	↑←0	↖1	←1	←1	←1
A	0	↖1	↑←1	↑←1	↖2	←2
C	0	↑1	↑←1	↖2	↑←2	↑←2

Tabela 2: Tabela $L[][]$ contendo setas indicativas do fluxo do algoritmo. Fonte: `https://en.wikipedia.org/wiki/Longest_common_subsequence`

- $L[i][j] \leftarrow L[i-1][j-1] + 1$

Por exemplo, o conteúdo da célula $L[1][2] = 1$ é obtida da seguinte forma:

- $L[1][2] \leftarrow L[0][1] + 1$

Existem células com uma flecha horizontal $\leftarrow$ ou vertical $\uparrow$ ou ainda com flechas duplas $\leftarrow\uparrow$. Por exemplo, o conteúdo da célula $L[1][3] = 1$ ($\leftarrow$) é obtida da seguinte forma:

- $L[1][3] = \max(L[1][2], L[0][3])$

A flecha $\leftarrow$ indica que $L[1][2] > L[0][3]$.

Para uma célula com flechas duplas $\leftarrow\uparrow$, por exemplo $L[2][2]$, o conteúdo dessa célula é obtido da seguinte forma:

- $L[2][2] = \max(L[1][2], L[2][1])$

como $L[1][2] = L[2][1] = 1$) significa que são duas subsequências a serem consideradas.

3 Especificações

1. Projetar um código na linguagem Python para realização de buscas da maior subsequência em comum entre duas cadeias de caracteres X e Y. Utilizar o algoritmo baseado em programação dinâmica. O algoritmo deve calcular comprimento l e também gerar uma subsequência de comprimeno l. **Não é necessário gerar todas as subsequências quando houver mais de uma solução, basta gerar apenas uma solução**.

2. Projetar o código do programa `main()` que organiza a realização dos diversos testes propostos.

3. Cada teste envolve duas cadeias de caracteres que representam duas sequências de DNA distintas. Cada cadeia de caracteres está armazenada em um arquivo texto.

4. Os seguintes testes devem ser realizados:

   (a) *strings* simples X ="AGGTAB", Y ="GXTXAYB"

(b) Trecho de DNA humano X ="ATGGGTGATGTTGAGAAAGGCAAGAAGA TTTTTATTATGAAGTGTTCCCAGTGCCACACC",
trecho de DNA Chimpanzé Y ="ATGGGTGATGTTGAGAAAGGCAAGAAGA TTTTTATTATGAAGTGTTCCCAGTGCCATACC"

(c) DNA vírus da Dengue tipo 2 Jakarta
X =`DengueVirus2StrainBA05i_Jakarta.txt` (nome do arquivo aonde se encontra a *string*) e
DNA vírus da Dengue tipo 3 Kuala Lumpur
Y =`DengueVirus3StrainTB55i_KualaLumpur.txt`.

(d) DNA Influenza tipo A H1N1 California X =`InfluenzaTypeA_H1N1_California.txt` e DNA Influenza tipo A H3N2 New York Y =`InfluenzaTypeA_H3N2_NewYork.txt`

(e) RNA SARS Cov2 Spike Toronto2 X =`SARS-COV2-Spike_Toronto2.txt` e RNA SARS Cov2 Spike Wuhan1 Y =`SARS-COV2-Spike_Wuhan1.txt`

5. Devem ser observados os seguintes requisitos

(a) Os resultados devem ser impressos na tela e também escritos em um arquivo texto `"nomedoarquivo.` onde `"nomedoarquivo"` é uma *string* que pode ser selecionada pelo usuário.

(b) Para cada teste deve ser impresso as cadeias de caracteres que estão sendo comparadas (para os dois primeiros testes) ou o nome dos arquivos.

(c) A cadeia de caracteres correspondente à maior subsequência de caracteres em comum e o comprimento correspondente.

4 Leitura de arquivos

O formato dos arquivos contendo sequências de DNA é mostrado a seguir:

```
1 agttgttagt ctacgtggac cgacaaagac agattctttg aggaagctaa gcttaacgta
61 gttctaacag ttttttaatt agagagcaga tctctgatga ataaccaacg gaaaaaggcg
121 agaaatacgc ctttcaatat gctgaaacgc gagagaaacc gcgtgtcaac tgtgcagcag
181 ctgacaaaga gattctcact tggaatgcta cagggacgag gaccattgaa actgttcatg
241 gccctggtgg cattccttcg tttcctaaca atcccgccaa cagcagggat attaaaaaga
301 tggggaacaa tcaaaaaatc aaaggctatc aatgtcttga gagggttcag gaaagagatt
361 ggaaggatgc tgaacatctt gaacaggaga cgcagaacag caggtataat tattatgatg
...
```

Uma possível subrotina que faz a leitura do arquivo e armazena a sequência de DNA numa *string* é apresentado em seguida.

Listing 1: Subrotina para leitura dos arquivos.

```
def LeArquivoDNA(filename):
files=open(filename, 'r')
lists=files.readlines() #this is a matrix of nlines
nlines=len(lists)     # number of lines

a = lists[0].rstrip('\n').split(' ') # separa a linhas em colunas
# observa o espaco ' ' como
# caracter de separacao
# descarta \n
```

```
cadeiacompleta = a[1] + a[2] + a[3] + a[4] + a[5] + a[6]
# concatena as colunas 1..6 ignora a[0]
# agora que cadeiacompleta nao e'vazio faca ate o final
for i in range(1,nlines):
a=lists[i].rstrip('\n').split(' ')
cadeiacompleta = cadeiacompleta + a[1]+a[2]+a[3]+a[4]+a[5]+a[6]
#retorna a string completa
return(cadeiacompleta)
```

5 Referências

1. Algorithms, Robert Sedgewick and Kevin Wayne, Addison-Wesley Professional, 4th Edition, 2011.

2. https://en.wikipedia.org/wiki/Longest_common_subsequence