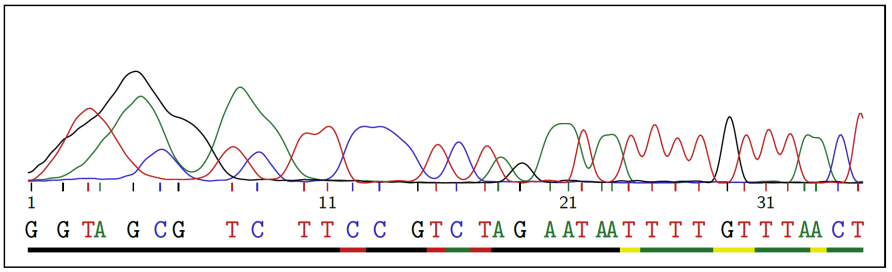
PMR3201 - Computação para Mecatrônica

Prof. Thiago de Castro Martins
Prof. Newton Maruyama
Prof. Marcos de S.G. Tsuzuki
Prof. Rafael Traldi Moura
Monitor:

Exercício Programa 1 - 2019

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



- **DATA FINAL DE ENTREGA: Terça-Feira 09/04/2019**
- **O exercício deve ser feito individualmente.**
- **Submeta através do sistema MOODLE:**
  - **Código fonte.**

1 Introdução

Nesse Exercício Programa será desenvolvido algoritmos de busca da maior subsequência de caracteres contíguos e não contíguos em comum entre duas *strings*. Trata-se de um algoritmo clássico de manipulação de *strings*.

Algoritmos de manipulaçao de *strings* 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 DNA ou de proteínas), detecção de *features* em imagens, 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.

```
--T--CC-C-AGT--TATGT-CAGGGGACACG--A-GCATGCAGA-GAC
  |  || |  ||  | | | |||    || |  | |  | ||||   |
AATTGCCGCC-GTCGT-T-TTCAG----CA-GTTATG--T-CAGAT--C

                    tccCAGTTATGTCAGgggacacgagcatgcagagac
                       ||||||||||||
aattgccgccgtcgttttcagCAGTTATGTCAGatc
```

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 uma 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 os algoritmos de busca de subsequências de caracteres contíguos e não contíguos que serão explorados nesse Exercício Programa.

2 Maior subsequência de caracteres

Uma solução para se obter o comprimento l da maior subsequencia de caracteres em comum entre duas *strings* 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:

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
```

O príncipio da recursão consiste em decompor o problema em problemas menores, i.e., comparando *strings* com comprimento menor até atingir uma *string* de comprimento nulo quando então começa o retorno para as 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 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 *string* Y e outra retirando o último caracter da *string* 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.

Por exemplo, para as *strings* $X =$`"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 *strings* X e Y que eventualmente já foram testadas anteriormente.

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.

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

Por exemplo, para as *strings* $X =$`"XMJYAUZ"` $(m = 7)$ e $Y =$`"MZJAWXU"` $(n = 7)$ a função $\mathrm{LCS}(X, Y, m, n)$ 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 subsequencia 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 obviamente gerada de forma trivial. Pode-se introduzir uma modificação no algoritmo principal para armazenar o caracter quando há uma coincidência. Alternativamente é possível gerar a maior subsequência de caracteres percorrendo o *array* $L[][]$ com índices i e j decrescentes.

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

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
```

		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.

3 Maior subsequência de caracteres contíguos

A maior subsequência de caracteres contíguos também pode ser derivada a partir do Algoritmo 2 apresentado acima. Pode-se modificar o código inserindo partes para a detecção dos trechos contíguos e os comprimentos correspondentes.

Alternativamente é possível utilizar a própria tabela $L[][]$ gerada pela função $\text{LCS}(X, Y, m, n)$ já que esta tabela resume as comparações realizadas.

4 Especificações

1. Projetar um código na linguagem Python para realização de buscas da maior subsequência de caracteres contíguos e não contíguos em duas *strings* X e Y. OBS: Pode ser realizado através de uma única função ou mais funções.

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

3. Cada teste envolve duas *strings* que representam duas sequências de DNA distintas. Cada *string* 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`

5. Devem ser observados os seguintes requisitos

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

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

   (c) A *string* correspondente à maior subsequência de caracteres não contíguos e o comprimento correspondente.

   (d) A *string* correspondente à maior subsequência de caracteres contíguos e o comprimento correspondente.

5 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)
```

6 Referências

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