

# **Transformações de Burrows-Wheeler**

# Compressão de sequências

Um dos problemas no processamento de sequências de DNA (e outras) passa pelo espaço requerido em memória

Mesmo usando árvores de sufixos compactas, o genoma humano ocupa ainda algo como 60 GB

Uma abordagem passa pelo uso de métodos de **compressão de sequências** para reduzir este valor

Compressão usada terá que ser **invertível** de forma a podermos sempre **recuperar** a sequência original

# Compressão de sequências

Considere a sequência: AAATTTTGGCCCC; esta poderia ser representada de forma compacta como 3A4T2G5C

Infelizmente, os genomas não têm muitas sequências de caracteres iguais pelo que esta estratégia não será muito eficiente

Por outro lado, não podemos alterar a ordem dos caracteres pois tornaríamos a sequência original irrecuperável (se pudéssemos poderíamos representar um genoma com 4 n<sup>º</sup>s – frequências de cada tipo de base)

No entanto, os genomas contêm bastantes **repetições** de padrões de diversos tamanhos: será que esta informação não pode ser útil para compressão ?

# Transformação de Burrows-Wheeler

Em 1994, Burrows e Wheeler propuseram um método cujo objetivo é “converter” repetições (padrões de vários tamanhos) em sequências de símbolos repetidos

Método baseia-se em **rotações cíclicas** da sequência original, que são **ordenadas lexicograficamente**

Vamos ver primeiro como se constrói a chamada **matriz de Burrows-Wheeler** (***M***) de uma sequência

# Matriz de Burrows-Wheeler

Exemplo para a sequência:  $s = \text{TAGACAGAGA}\$$

TAGACAGAGA\$
\$TAGACAGAGA
A\$TAGACAGAG
GA\$TAGACAGA
AGA\$TAGACAG
GAGA\$TAGACA
AGAGA\$TAGAC
CAGAGA\$TAGA
ACAGAGA\$TAG
GACAGAGA\$TA
AGACAGAGA\$T

Ordenação  
Lexicográfica

\$	T	A	G	A	C	A	G	A	G	A
A	\$	T	A	G	A	C	A	G	A	G
A	C	A	G	A	G	A	\$	T	A	G
A	G	A	\$	T	A	G	A	C	A	G
A	G	A	C	A	G	A	G	A	\$	T
A	G	A	G	A	\$	T	A	G	A	C
C	A	G	A	G	A	\$	T	A	G	A
G	A	\$	T	A	G	A	C	A	G	A
G	A	C	A	G	A	G	A	\$	T	A
G	A	G	A	\$	T	A	G	A	C	A
T	A	G	A	C	A	G	A	G	A	\$

Rotações cíclicas da  
sequência  $s$

Matriz  $M(s)$   
Matriz de BW

**BWT:** AGGGTCAAAA\$

# Matriz de Burrows-Wheeler

Na prática, não se guarda toda a matriz mas apenas a 1ª e última colunas

Na matriz, a 1ª coluna é a ordenação lexicográfica dos caracteres do alfabeto (note que pode ser muito facilmente compactada pelas frequências de cada símbolo)

A última coluna é também uma re-ordenação dos caracteres de  $s$ , sendo chamada de **Transformada de Burrows-Wheeler (BWT)** -  $BWT(s) = \text{"AGGGTCAAAA\$"}$

Note-se a repetição dos símbolos "A" e "G" na BWT – este é o padrão típico provocado pela ocorrência de repetições (e.g. padrões "AG" e "GA") que permitem a compactação da BWT


# Construção da BWT

Na prática, existem algoritmos eficientes que não necessitam de gerar a matriz  $M$  para a construção da BWT, mas demasiado complexos para os estudarmos aqui ...

No código que vamos construir vamos adoptar uma implementação menos eficiente que faz a construção de toda a matriz  $M$

# Implementação da construção da BWT

```
class BWT:  
  
    def __init__(self, seq):  
        self.bwt = self.buildbwt(seq)  
  
    def buildbwt(self, text):  
        ...
```



Método que retorna a última coluna da matriz, i.e. a BWT

```
def test():  
    seq = "TAGACAGAGA$"  
    bw = BWT(seq)  
    print (bw.bwt)  
  
test()
```



# Implementação da construção da BWT

```
class BWT:

    def __init__(self, seq):
        self.bwt = self.buildbwt(seq)

    def buildbwt(self, text):
        ls = [ ]
        for i in range(len(text)):
            ls.append(text[i:]+text[:i])
        ls.sort()
        res = " "
        for i in range(len(text)):
            res += ls[i][len(text)-1]
        return res
```

```
def test():
    seq = "TAGACAGAGA$"
    bw = BWT(seq)
    print (bw.bwt)

test()
```

# Recuperação da sequência original da BWT

Um bom método de compressão do genoma só será utilizável se permitir recuperar a sequência original

Por exemplo, suponha que:

$BWT(s) = "ACG\$GTAAAC";$

Será que conseguimos saber qual a sequência original (e a matriz M)?



## Recuperação da sequência original da BWT

Note-se que o 1º símbolo da sequência deve estar a seguir ao \$ em qualquer rotação cíclica. Assim, ele deve ser o símbolo da 1ª coluna na mesma linha onde \$ é o último.

[illegible]

## Recuperação da sequência original da BWT

[illegible][illegible]

# Recuperação da sequência original da BWT

Seguindo o mesmo raciocínio, o próximo símbolo deve ser o que está na 1ª coluna na linha onde “A” está na última. O problema é que existem várias possibilidades ...

\$	A	C	?	?	?	?	?	?	?	A
A	?	?	?	?	?	?	?	?	?	C
A	?	?	?	?	?	?	?	?	?	G
A	?	?	?	?	?	?	?	?	?	\$
A	?	?	?	?	?	?	?	?	?	G
A	?	?	?	?	?	?	?	?	?	T
C	?	?	?	?	?	?	?	?	?	A
C	?	?	?	?	?	?	?	?	?	A
G	?	?	?	?	?	?	?	?	?	A
G	?	?	?	?	?	?	?	?	?	A
T	?	?	?	?	?	?	?	?	?	C

OU

\$	A	G	?	?	?	?	?	?	?	A
A	?	?	?	?	?	?	?	?	?	C
A	?	?	?	?	?	?	?	?	?	G
A	?	?	?	?	?	?	?	?	?	\$
A	?	?	?	?	?	?	?	?	?	G
A	?	?	?	?	?	?	?	?	?	T
C	?	?	?	?	?	?	?	?	?	A
C	?	?	?	?	?	?	?	?	?	A
G	?	?	?	?	?	?	?	?	?	A
G	?	?	?	?	?	?	?	?	?	A
T	?	?	?	?	?	?	?	?	?	C

?

# Recuperação da sequência original da BWT

Para ultrapassar este problema, temos que numerar as ocorrências de cada símbolo ...

$\$_0$	?	?	?	?	?	?	?	?	?	$A_0$
$A_0$	?	?	?	?	?	?	?	?	?	$C_0$
$A_1$	?	?	?	?	?	?	?	?	?	$G_0$
$A_2$	?	?	?	?	?	?	?	?	?	$\$_0$
$A_3$	?	?	?	?	?	?	?	?	?	$G_1$
$A_4$	?	?	?	?	?	?	?	?	?	$T_0$
$C_0$	?	?	?	?	?	?	?	?	?	$A_1$
$C_1$	?	?	?	?	?	?	?	?	?	$A_2$
$G_0$	?	?	?	?	?	?	?	?	?	$A_3$
$G_1$	?	?	?	?	?	?	?	?	?	$A_4$
$T_0$	?	?	?	?	?	?	?	?	?	$C_1$

Pode provar-se que a ordem de ocorrência de cada símbolo na 1ª coluna de M é igual à ordem de ocorrência na última !

# Recuperação da sequência original da BWT

Assim, podemos avançar com o nosso método ...

$\$0$	?	?	?	?	?	?	?	?	?	$\mathbf{A}_0$
$\mathbf{A}_0$	?	?	?	?	?	?	?	?	?	$\mathbf{C}_0$
$\mathbf{A}_1$	?	?	?	?	?	?	?	?	?	$\mathbf{G}_0$
$\mathbf{A}_2$	?	?	?	?	?	?	?	?	?	$\$0$
$\mathbf{A}_3$	?	?	?	?	?	?	?	?	?	$\mathbf{G}_1$
$\mathbf{A}_4$	?	?	?	?	?	?	?	?	?	$\mathbf{T}_0$
$\mathbf{C}_0$	?	?	?	?	?	?	?	?	?	$\mathbf{A}_1$
$\mathbf{C}_1$	?	?	?	?	?	?	?	?	?	$\mathbf{A}_2$
$\mathbf{G}_0$	?	?	?	?	?	?	?	?	?	$\mathbf{A}_3$
$\mathbf{G}_1$	?	?	?	?	?	?	?	?	?	$\mathbf{A}_4$
$\mathbf{T}_0$	?	?	?	?	?	?	?	?	?	$\mathbf{C}_1$

[illegible]



## Recuperação da sequência original da BWT

... sucessivamente

[illegible][illegible]

# Recuperação da sequência original da BWT

... até determinar a sequência !



(...)



\$ <sub>0</sub>	A	C	T	A	G	A	G	A	C	A <sub>0</sub>
A <sub>0</sub>	?	?	?	?	?	?	?	?	?	C <sub>0</sub>
A <sub>1</sub>	?	?	?	?	?	?	?	?	?	G <sub>0</sub>
A <sub>2</sub>	?	?	?	?	?	?	?	?	?	\$ <sub>0</sub>
A <sub>3</sub>	?	?	?	?	?	?	?	?	?	G <sub>1</sub>
A <sub>4</sub>	?	?	?	?	?	?	?	?	?	T <sub>0</sub>
C <sub>0</sub>	?	?	?	?	?	?	?	?	?	A <sub>1</sub>
C <sub>1</sub>	?	?	?	?	?	?	?	?	?	A <sub>2</sub>
G <sub>0</sub>	?	?	?	?	?	?	?	?	?	A <sub>3</sub>
G <sub>1</sub>	?	?	?	?	?	?	?	?	?	A <sub>4</sub>
T <sub>0</sub>	?	?	?	?	?	?	?	?	?	C <sub>1</sub>

Sequência: ACTAGAGACA\$

Teste com a BWT (s2) = acm\$intobriifoa – Qual a sequência s2 ?

# Implementação da recuperação da sequência da BWT

Métodos/funções auxiliares:

Recuperar a primeira coluna

```
def get_first_col (self):  
    ...
```

Descobrir posição da i-ésima  
ocorrência de um símbolo numa lista  
(retorna -1 de não ocorre)

```
def find_ith_occ(l, elem, index):  
    ...
```

# Implementação da recuperação da sequência da BWT

Métodos/funções auxiliares:

```
def get_first_col (self):  
    firstcol = []  
    for c in self.bwt:  
        firstcol.append(c)  
    firstcol.sort()  
    return firstcol
```

```
def find_ith_occ(l, elem, index):  
    j,k = 0,0  
    while k < index and j < len(l):  
        if l[j] == elem:  
            k = k + 1  
            if k == index: return j  
        j += 1  
    return -1
```

```
def inverse_bwt(self):  
    firstcol = self.get_first_col()  
    res = ""  
    c = "$"  
    occ = 1  
    for i in range(len(self.bwt)):  
        ...  
    return res
```

## Implementação da recuperação da sequência da BWT

```
def test2():  
    bw = BWT("")  
    bw.setBWT ("ACG$GTAAAAC")  
    print (bw.inverse_bwt())  
test2()
```

```
def inverse_bwt(self):
    firstcol = self.get_first_col()
    res = ""
    c = "$"
    occ = 1
    for i in range(len(self.bwt)):
        pos = find_ith_occ(self.bwt, c, occ)
        c = firstcol[pos]
        occ = 1
        k = pos-1
        while firstcol[k] == c and k >= 0:
            occ += 1
            k -= 1
        res += c
    return res
```

## Implementação da recuperação da sequência da BWT

```
def test2():
    bw = BWT("")
    bw.setBWT("ACG$GTAAAAC")
    print (bw.inverse_bwt())
test2()
```

# Procura de padrões com a BWT

A BWT permite a procura eficiente de padrões existentes na sequência original

A primeira ideia a reter é que cada linha da matriz M começa por um sufixo da sequência

Como estes estão ordenados todos os matches de qualquer padrão aparecem em linhas sucessivas

\$	T	A	G	A	C	A	G	A	G	A
A	\$	T	A	G	A	C	A	G	A	G
A	C	A	G	A	G	A	\$	T	A	G
A	G	A	\$	T	A	G	A	C	A	G
A	G	A	C	A	G	A	G	A	\$	T
A	G	A	G	A	\$	T	A	G	A	C
C	A	G	A	G	A	\$	T	A	G	A
G	A	\$	T	A	G	A	C	A	G	A
G	A	C	A	G	A	G	A	\$	T	A
G	A	G	A	\$	T	A	G	A	C	A
T	A	G	A	C	A	G	A	G	A	\$

# Procura de padrões com a BWT

No entanto, não queremos guardar nem construir a matriz M completa ...

Assim, temos que pensar num algoritmo que só use a 1ª e a última colunas

A estratégia passa por identificar o padrão pela ordem inversa dos seus símbolos movendo-nos nas linhas da 1ª e última colunas (tal como no caso da reconstrução)



# Procura de padrões com a BWT

O 1º passo passa por identificar o último símbolo do padrão e ver onde ele faz match na última coluna

Padrão = “AG**A**”

Procurar linhas com “A” na última coluna

\$ <sub>0</sub>	...	A <sub>0</sub>
A <sub>0</sub>	...	G <sub>0</sub>
A <sub>1</sub>	...	G <sub>1</sub>
A <sub>2</sub>	...	G <sub>2</sub>
A <sub>3</sub>	...	T <sub>0</sub>
A <sub>4</sub>	...	C <sub>0</sub>
C <sub>0</sub>	...	A <sub>1</sub>
G <sub>0</sub>	...	A <sub>2</sub>
G <sub>1</sub>	...	A <sub>3</sub>
G <sub>2</sub>	...	A <sub>4</sub>
T <sub>0</sub>	...	\$ <sub>0</sub>

# Procura de padrões com a BWT

Próximo passo: identificar posições desses símbolos na primeira linha, atualizando os índices T (top) e B (bottom)

Padrão = “AG**A**”

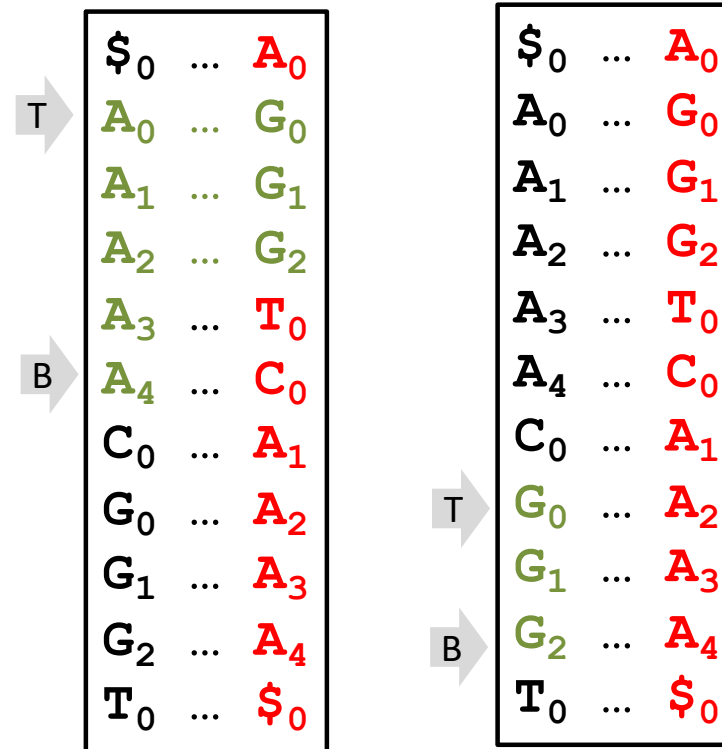
	F		L
T	\$ <sub>0</sub> ...		<b>A</b> <sub>0</sub>
	<b>A</b> <sub>0</sub> ...		<b>G</b> <sub>0</sub>
	<b>A</b> <sub>1</sub> ...		<b>G</b> <sub>1</sub>
	<b>A</b> <sub>2</sub> ...		<b>G</b> <sub>2</sub>
	<b>A</b> <sub>3</sub> ...		<b>T</b> <sub>0</sub>
	<b>A</b> <sub>4</sub> ...		<b>C</b> <sub>0</sub>
	<b>C</b> <sub>0</sub> ...		<b>A</b> <sub>1</sub>
	<b>G</b> <sub>0</sub> ...		<b>A</b> <sub>2</sub>
	<b>G</b> <sub>1</sub> ...		<b>A</b> <sub>3</sub>
	<b>G</b> <sub>2</sub> ...		<b>A</b> <sub>4</sub>
B	<b>T</b> <sub>0</sub> ...		<b>\$</b> <sub>0</sub>

	F		L
T	<b>\$</b> <sub>0</sub> ...		<b>A</b> <sub>0</sub>
	<b>A</b> <sub>0</sub> ...		<b>G</b> <sub>0</sub>
	<b>A</b> <sub>1</sub> ...		<b>G</b> <sub>1</sub>
	<b>A</b> <sub>2</sub> ...		<b>G</b> <sub>2</sub>
	<b>A</b> <sub>3</sub> ...		<b>T</b> <sub>0</sub>
B	<b>A</b> <sub>4</sub> ...		<b>C</b> <sub>0</sub>
	<b>C</b> <sub>0</sub> ...		<b>A</b> <sub>1</sub>
	<b>G</b> <sub>0</sub> ...		<b>A</b> <sub>2</sub>
	<b>G</b> <sub>1</sub> ...		<b>A</b> <sub>3</sub>
	<b>G</b> <sub>2</sub> ...		<b>A</b> <sub>4</sub>
	<b>T</b> <sub>0</sub> ...		<b>\$</b> <sub>0</sub>

# Procura de padrões com a BWT

Próximo passo: procurar nas linhas selecionadas (entre T e B) as ocorrências do segundo símbolo do padrão na última coluna, identificá-los na 1ª coluna e atualizar T e B

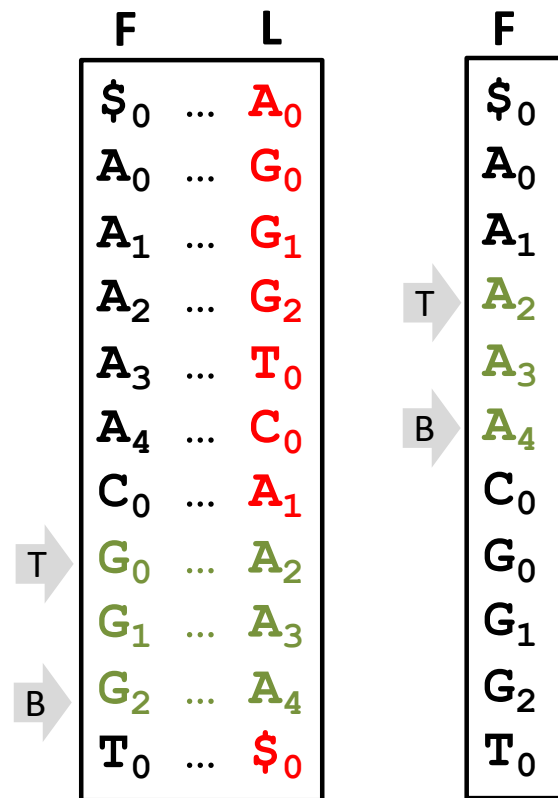
Padrão = "AGA"



# Procura de padrões com a BWT

- Passo final: identificar o primeiro símbolo do padrão

Padrão = “AGA”



Padrão descoberto:  
Linhas 3, 4, 5

# Procura de padrões com a BWT

Para realizar o processo anterior eficientemente deve construir-se uma tabela que indica para cada índice de um símbolo na última coluna, qual a sua posição na 1ª

<i>i</i>	<i>First</i>	<i>Last</i>	<i>LastToFirst</i>
0	\$ <sub>0</sub>	A <sub>0</sub>	1
1	A <sub>0</sub>	G <sub>0</sub>	7
2	A <sub>1</sub>	G <sub>1</sub>	8
3	A <sub>2</sub>	G <sub>2</sub>	9
4	A <sub>3</sub>	T <sub>0</sub>	10
5	A <sub>4</sub>	C <sub>0</sub>	6
6	C <sub>0</sub>	A <sub>1</sub>	2
7	G <sub>0</sub>	A <sub>2</sub>	3
8	G <sub>1</sub>	A <sub>3</sub>	4
9	G <sub>2</sub>	A <sub>4</sub>	5
10	T <sub>0</sub>	\$ <sub>0</sub>	0

# Implementação da procura de padrões a partir da BWT

```
def last_to_first(self):  
    res = []  
    ...  
    return res
```

```
def test():  
    seq = "TAGACAGAGA$"  
    bw = BWT(seq)  
    print (bw.bwt)  
    print (bw.last_to_first())  
  
test()
```

Método que cria a tabela anterior, i.e cria a conversão da última coluna para a primeira

# Implementação da procura de padrões a partir da BWT

```
def last_to_first(self):  
    res = []  
    firstcol = self.get_first_col()  
    for i in range(len(firstcol)):  
        c = self.bwt[i]  
        ocs = self.bwt[:i].count(c) + 1  
        res.append(find_ith_occ(firstcol, c, ocs))  
    return res
```

```
def test():  
    seq = "TAGACAGAGA$"  
    bw = BWT(seq)  
    print (bw.bwt)  
    print (bw.last_to_first())
```

```
test()
```

Método que cria a tabela anterior, i.e cria a conversão da última coluna para a primeira

```

def bw_matching(self, patt):
    lf = self.last_to_first()
    res = []
    top = 0
    bottom = len(self.bwt)-1
    flag = True
    while flag and top <= bottom:
        if patt != "":
            symbol = patt[-1]
            patt = patt[:-1]
            lmat = self.bwt[top:(bottom+1)]
            if symbol in lmat:
                topIndex = lmat.index(symbol) + top
                bottomIndex = bottom - lmat[::-1].index(symbol)
                top = lf[topIndex]
                bottom = lf[bottomIndex]
            else: flag = False
        else:
            for i in range(top, bottom+1): res.append(i)
            flag = False
    return res

```

## Implementação da procura de padrões a partir da BWT

```

def test():
    seq = "TAGACAGAGA$"
    bw = BWT(seq)
    print (bw.bwt)
    print (bw.bw_matching("AGA"))

test()

```



# Procura de padrões com a BWT

Algoritmo / função anterior não permite facilmente identificar as posições iniciais do padrão (apenas o nº de ocorrências)

Posição inicial de cada padrão pode ser recuperada seguindo o processo 1ª coluna – última coluna até ao símbolo inicial (\$), mas este algoritmo é demasiado lento

Alternativa passa pela utilização de estruturas de dados eficientes para guardar informação sobre os sufixos associados a cada posição da BWT (arrays de sufixos parciais)

# Arrays de sufixos

Alternativa eficiente às árvores de sufixos

Representam lista com as posições iniciais de cada sufixo ordenados lexicograficamente (i.e. posição inicial de cada linha da matriz M)

Array de sufixos permitem procura da posição de matches com BWT

Array de sufixos parcial (por exemplo apenas 1/K das posições iniciais) pode ser suficiente para acelerar processo de procura sem utilizar demasiada memória

Posição inicial	Sufixo ordenado
10	\$
9	A\$
3	ACAGAGA\$
7	AGA\$
1	AGACAGAGA\$
5	AGAGA\$
4	CAGAGA\$
8	GA\$
2	GACAGAGA\$
6	GAGA\$
0	TAGACAGAGA\$

*SuffixArray("TAGACAGAGA\$") = (10, 9, 3, 7, 1, 5, 4, 8, 2, 6, 0)*

# Implementação da procura de padrões a partir da BWT

```
class BWT:
    def __init__(self, seq = "", buildsufarray = False):
        self.bwt = self.buildbwt(seq, buildsufarray)

    def buildbwt(self, text, buildsufarray = False):
        ls = []
        for i in range(len(text)):
            ls.append(text[i:] + text[:i])
        ls.sort()
        res = ""
        for i in xrange(len(text)):
            res += ls[i][len(text)-1]
        if buildsufarray:
            self.sa = []
            for i in range(len(ls)):
                stpos = ls[i].index("$")
                self.sa.append(len(text) - stpos - 1)
        return res
```

```
def test3():
    seq = "TAGACAGAGA$"
    bw = BWT(seq, True)
    print("Suffix array:", bw.sa)

test3()
```

# Implementação da procura de padrões a partir da BWT

```
def bw_matching_pos(self, patt):  
    res = []  
    ...  
    return res
```

```
def test3():  
    seq = "TAGACAGAGA$"  
    bw = BWT(seq, True)  
    print("Suffix array:", bw.sa)  
    print (bw.bw_matching_pos("AGA"))  
  
test3()
```

Método que procura os matches de um padrão (assume que foi criado SA)

# Implementação da procura de padrões a partir da BWT

```
def bw_matching_pos(self, patt):  
    res = []  
    matches = self.bw_matching(patt)  
    for m in matches:  
        res.append(self.sa[m])  
    res.sort()  
    return res
```

```
def test3():  
    seq = "TAGACAGAGA$"  
    bw = BWT(seq, True)  
    print("Suffix array:", bw.sa)  
    print (bw.bw_matching_pos("AGA"))  
  
test3()
```

Método que procura os matches de um padrão (assume que foi criado SA)

# BWTs na prática

As BWTs são usadas no alinhamento de leituras de sequenciação (NGS) contra (genomas de) referência

Com esta técnica pode criar-se uma representação do genoma humano com menos de 3 GB

Exemplos são os softwares *Bowtie*, *BWA* e *SOAP2*, um dos mais usados no alinhamento de dados de DNAseq/ RNAseq contra referências:

<http://bowtie-bio.sourceforge.net/index.shtml>

<http://bio-bwa.sourceforge.net/>

<http://soap.genomics.org.cn/soapaligner.html>