Lista 6 - Francisco Braz
1-)
a-)

```
função achaPadrao( T, P, n, m, k): vazio
início
    i := 1; j := 1; index := 0
    m = k
    enquanto index = 0 e i <= n faça
    início
        se P[j] = T[i] então
            j := j+1
            i := i+1
        senão
            i := i-j+2
            j := 1
        se j = m+1 então
            index := i-m
            imprima index
        se index := 0 então imprima -1
    fim
fim
```

b-) Como as alterações realizadas no algoritmo foram apenas alterações de atribuição e verificação, nossa complexidade será a mesma do algoritmo 'achaPadrão' visto na aula 14, já que no pior caso para cada posição do nosso vetor T (onde queremos procurar o padrão), são realizadas m comparações com todos os caracteres disponíveis no padrão (P), até que chegamos no final do padrão e concluímos que ele não existe em T. Portanto, vamos ter:

T(n,m) = $\sum_{i=1}^{n} c * m$

= $\sum_{i=1}^{n} c * m * (n - 1 + 1)$

= $c * m * n$

Ignorando a constante, temos que nossa complexidade é O(m*n).
c-) Implementação no arquivo achaPadroes.c

2-)
a-)

```
função achaPadrao(T, P, n, m): vazio
início
     i := 1; j := 1; index := 0
     enquanto index = 0 e i <= n faça
     início
          se P[j] = T[i] então
               j := j+1
               i := i+1
           senão
               se P[j] = ‘#’ então j = j + 1
               senão se P[j-1] = ‘#’ então i = i + 1
               senão
                    i := i + 1
                    j := 1
           se j = m+1 então
                imprima “Encontrado”
      fim
fim
```

b-) Assim como na primeira questão e no exemplo da aula 14, o pior caso da nossa função ‘achaPadrão’ seria quando o padrão (P) não existisse na nossa expressão T e consequentemente para cada posição do nosso vetor T (onde queremos procurar o padrão), são realizadas m comparações com todos os caracteres disponíveis no padrão (P), até que chegamos no final do padrão e concluímos que ele não existe em T. Portanto, vamos ter:

T(n,m) = $\sum_{i=1}^{n} c \ * \ m$

$$= \sum_{i=1}^{n} c * m * (n - 1 + 1)$$

$$= c * m * n$$

Ignorando a constante, temos que nossa complexidade é O(m*n).

c-) Implementação no arquivo acharPadraoEspecial.c

3-)

a-)

```
função ComputaNext(P, m, next)
início
    next[1] := -1; next[2] := 0
    para i = 3 até m faça
    início
       j := next[i-1] + 1
       enquanto P[i-1] <> P[j] e j > 0 faça j := next[j] + 1
       next[i] := j
    fim
fim

função kmp(T,n,P,m, next)
início
   ComputaNext(P,m,next)
   i := 1; j := 1; index :=0
   enquanto index = 0 e i <= n faça
   início
      se P[j] = T[i] então
           j := j+1
           i := i+1
      senão
          j := next[j]+1
          se j = 0 então {j := 1; i := i+1}
      se j = m+1 então index := i-m
   fim
fim
```

```
função inicaKMP(Matriz, n, P, m)
início
    ComputaNext(P,m,next)
    encontrouLinha := 0
    encontrouColuna := 0
    para i = 1 até n faça
    início
        padraoHorizontal:= kmp(Matriz[i], n, P, m, next)
        se padraoHorizontal<> -1 então
        início
            encontrouLinha := 1
            posPrimeira[1] := i
            posPrimeira[2] = padraoHorizontal
            posUltima[1] = i
            posUltima[2] = padraoHorizontal+ (m-1)
        fim

        para j = 1 até n faça
        início
            arrayColuna := Matriz[j][i]
        fim

        padraoVertical = kmp(arrayColuna, n, P, m, next)
        se padraoVertical <> -1 então
        início
            encontrouLinha := 1
            posPrimeira[1] := i
            posPrimeira[2] = padraoVertical
            posUltima[1] = i
            posUltima[2] = padraoVertical + (m-1)
        fim

        se encontrouColuna <> 0 ou encontrouLInha <> 0 então
           imprima P
            para i = 1 até 2 faça
```

```
        imprima posPrimeira[i]

    para i = 1 até 2 faça
        imprima posUltima[i]
```

b-) Nossa complexidade seria:
Nossa função computaNext seria O(m), pois mesmo que ocorra diferença entre os caracteres(atuais) que estão sendo comparados, não retornaremos nossa comparação do início, pois já encontramos casamentos entre elementos anteriores, ou seja, não iremos retroceder para o início.
função kmp: O(n)
função iniciaKMP: $O(n^2)$, pois temos um loop “para” que irá percorrer toda nossa coluna atual dentro de um outro loop “para” que percorreria as linhas de nossa matriz.

Logo temos: $m + n + n^2$, considerando somente o maior termo temos: $O(n^2)$

c-) Implementação no arquivo Main.java

4-)
a-)

```
função Horspool(T, n, P, m): inteiro
início
   ComputaDeslocamento(P, m, D);
   i := m-1; index := -1
   enquanto i <= n-1 e index = -1 faça
        k := 0
        enquanto k <= m-1 e P[m-1-k] = T[i-k] faça k := k+1
        se k = m então index := i-m+1
        senão i := i + D[T[i]]
fim

função buscarSufixo(T, n, P, m): vazio
início
```

```
    index := -1
    i := m
    enquanto i > 0 e index = -1
    início
        index = Horspool(T, n, P, i)
    fim

    se index <> -1
         imprima index, index+1
    senão
         imprima -1
fim
```

b-) Nosso pior caso irá ocorrer quando as comparações dão erro sempre no primeiro elemento do padrão (ou último, considerando a ordem da direita para esquerda) e ao mesmo tempo o deslocamento desse caractere for 1. Pois desse modo sempre iremos percorrer o padrão (P) por completo e avançaremos somente uma posição no deslocamento.

c-) Implementação no arquivo algoritmoBMH.c