Micael Andrade Dos Santos

Questão 1

a. `Explicação`

> 💡 Precisamos varrer toda a string para determinar a última ocorrência de P em T. Para isso, podemos definir uma variável `ultimaOcorr` que irá armazenar a última ocorrência naquele momento em que o texto está sendo processado. Essa variável começa com `-1`, e irá sendo atualizada com as ocorrências encontradas. Quando finalizar a varredura do texto, retornamos `últimaOcorr`.

b. `pseudo-linguagem`

```
algoritmo KMP_algoritmo(T, N, P, M)
{-Entrada: Texto T de tamanho N e um padrão P de tamanho M.
 -SAÍDA: Ultima ocorrência de P em T caso exista, -1 caso contrário.
.}
inicio
    i := 1
    j := 1
    ultimaOcorr := -1
    ARR_PRE := ComputaNext(P,M) -- pré-processa next(não precisa alterar)
    enquanto i <= N: --Até o tamanho do texto.
        se(T[i] = P[j]) então
            i := i + 1
            j := j + 1
        senão:
            se(j <> 0) então
                j := ARR_PRE[j]
            senão:
                i := i + 1
        se(j = self.M) então -- Deu matched
            ultimaOcorr = i-j
            j = self.ARR_PRE[j]

    retorne ultimaOcorr
fim
```

```
algoritmo ComputaDeslocamento(P, M)
{-
Entrada: Padrão P de tamanho M.
Saída: Tabela T indexada pelos caracteres do alfabeto e
       preenchido com tamanhos de deslocamento calculados.
-}
inicio
    tabela = [256]  -- Tabela com 256 posições (ASCII)
    para i = 0 ate M faça tabela[i] := M -- Iniciliza posições com M

    para i=1 ate M-1 faça
        tabela[padrao[i]] = M - 1 - i

    retorne tabela
fim
```

```
algoritmo horspoolMatching(T, N, P, M)
{-Entrada: Texto T de tamanho N e um padrão P de tamanho M.
 -SAÍDA: Ultima ocorrência de P em T caso exista, -1 caso contrário.
.}
inicio
    tabelaPre := ComputaDeslocamento(P, M) --Gerando a tabela de deslocamentos.
```

```
    i := M - 1 --Posição da extremidade direita do padrão.
    ultimaOcorr = -1
    enquanto(i <= N-1):
        k := 0
        enquanto(k <= M-1 e padrao[M-1-k] = T[i-k]) faça
            k := k + 1
        se(k = M) então
            ultimaOcorr = i-M+1
        i := i + tabelaPre[T[i]]
    retorne ultimaOcorr
fim
```

> Aqui a única diferença do slide(15) foi o fato de não parar após encontrar a primeira ocorrência. Ou seja, precisamos verificar todo o texto. Além disso, `ComputaDeslocamento` não precisa ser alterado.

c. `Discussão da complexidade da solução`

🧮 Utilizando a abordagem KMP. Temos que a complexidade seria `T(ComputaNext)` + `T(enquanto)`. Sabemos que `T(ComputaNext) = Θ(M)` , ou seja, proporcional ao tamanho do padrão. A complexidade do `enquanto` é proporcional ao tamanho do texto (pois o texto não retrocede), e como queremos a última ocorrência, precisamos verificar todo o texto `T(enquanto) = Θ(n)`. Assim, `T(KMP_algoritmo)` = `Θ(n)` + `Θ(m)` = `Θ(n)`.

🧮 Utilizando o algoritmo de `Horspool` : `T(horspoolMatching)` = `T(ComputaDeslocamento)` + `T(enquanto1)`
Temos que o `enquanto1` será executado `N-1` o que seria linear, ou seja, `Θ(n)` o `enquanto` interno será executado apenas quando ocorrer de fato um casamento, o qual será executado `M-1` vezes começando do lado direito do padrão realizando `C` operações de atribuições. Logo, no pior caso temos `O(nm)`, porém, esse algoritmo é `Θ(n)` para textos aleatórios segundo *(Levetin Pg 262).*

Questão 2

a. `explicação`

💡 Podemos estruturar essa solução em duas partes. Primeiro, precisamos de uma função que calcula todos os prefixos de um determinado padrão e retorna uma lista com todos eles.
Depois, podemos adaptar o algoritmo do `Karp` para cada prefixo encontrado anteriormente verificá-lo se há ocorrências no texto.

b. `pseudo-linguagem`

```
algoritmo findPrefix(P,M,prefixs)
{-Entrada: Uma String P de tamanho M, e um vetor prefixos inicialmente vazio.
  Saída: Array com todos o possíveis prefixos de P
  Ex: 'casa' -> ['c','ca','cas','casa']
.}
inicio
    se(M = 0) entao
        retorne prefixs -- CASO BASE
    senão
        sufix.insiraNaCabeca(pat) -- Inicialmente Insere 'casa'
        pat.removeCalda() -- Inicialmente 'casa' vira 'cas'
        retorne findPrefix(pat, M-1, prefixs) --Chamada recursiva
fim
```

> Bem, podemos considerar que as listas encadeadas `sufixos, pat` contém um ponteiro para cabeça e para a calda. Logo isso seria, `O(1)`, pois bastam manipulações simples nesses ponteiros.
> Temos que T(findPrefix)= `T(0) = C1` e `T(M) = T(M-1) + c2` utilizando o método de substituição temos:
> T(M) = T(M-1) + C2
> = T(M-2) + C2 + C2
> = T(M-2) + 2C2
> = T(M-3) + 2C2 + C2
> = T(M-3) + 3C2
> ....
> = T(M-K) + KC2
> Tomando K = M, temos T(M) = T(M-M) + MC2 =
> = T(0) + MC2
> = C1 + MC2
>
> $T(findPrefix) = \Theta(M)$

```
algoritmo rabin_karp_MATCHER(T, N, ArrPrefix, K, D, Q)
{-Entrada: String T de tamanho N, vetor ArrPrefix de Strings de Tamanho K
           D total digitos do alfabeto, numero primo Q.
  Saída: Total de ocorrências de todos os prefixos de ArrPrefix em T.
.}
inicio
  totalOcorr := 0
    para i=1 até K faça
        tamanhoPrefix := ArrPrefix[i].tamanho
        h = D^(tamanhoPrefix-1) mod Q
        p = 0  -- Valor hash para o padrão
        t = 0  -- Valor hash para o texto

        para j=1 ate tamanhoPrefix faça  -- Pré-Processamento do prefixo
            p = (D*p + ArrPrefix[i][j]) mod Q
            t = (D*t + texto[i]) mod Q

        para s=0 ate (N-tamanhoPrefix):
            se(p = t) então
                se(ArrPrefix[i] == T[s + 1...s + tamanhoPrefix]
                    totalOcorr := totalOcorr + 1
            se(s < N-tamanhoPrefix):
                t = (D*(t-T[s+1]*h) + T[s + tamanhoPrefix + 1]*h mod Q
    retorne totalOcorr
fim
```

> A adaptação só muda a parte que precisamos agora de um `para` externo para verificar cada prefixo. Ou seja, agora não temos mais um padrão, e sim Ms. Note que para cada prefixo nós precisamos executar o algoritmo original de `Karp` logo temos:
> `Σ(i=1)(K) * T(algoritmoOriginalKarpCasoMedio)` =

c. `Discussão da complexidade da solução`

Logo temos,

$O(M) + O(K * (tamanhoPrefix + N))$ =
$O(K * (tamanhoPrefix + N))$

Questão 3

a. `explicação`

> 💡 Podemos inverter a string *"parcialmente"* para capturar o sentido inverso da string circular. Por exemplo, considere a palavra `CASA`, invertida ficaria `ASAC`, colocando a primeira letra após a inversão `(A)` no final ficaríamos com: `SACA`. Agora, tendo os dois sentidos: `CASA` e `SACA`, basta aplicar o KMP na primeira e verificar se existe uma substring, caso não exista, precisamos verificar se ela existe em `SACA`. Se não existir em nenhuma basta retorna `Falso`.

b. `pseudo-linguagem`

```
algoritmo invertString(S, M)
{-Entrada: String S de tamanho M.
  Saída: String W de tamanho M representando a inversão de S, porém com
  o primeiro caracter no final.
    Ex: (original) CASA
        (invertida) ASAC
        (desloca o primeiro caracter para o final) SACA
-}
inicio
  W := ''
  ultimoChar := ''
  para i=M descrescendo ate 1 faça
      se(i != M) então
          W := W + S[i]
      senão:
          ultimoChar = S[i]

  retorne W + ultimoChar
fim
```

> `T(invertString) = Σ(i=1)(M) = M-1+1 = O(M)`

```
algoritmo isCircular(T, N, S, M)
  ENTRADA: Padrao T de tamanho N e um string circular S de tamanho M
  SAÍDA: Verdadeiro se T for substring de S, Falso caso contrário.
inicio
  primeiroTeste := KMP(S, M, T, N) --Verificamos se T é substring de S.
  se (primeiroTeste = 0) --Se não for, vamos verificar se é substring do texto invertido.
    stringInvertida := invertString(S, M)
    primeiroTeste = KMP(stringInvertida, M, T, N)
  retorne primeiroTeste
fim
```

> Ao iniciar `isCircular` executamos uma primeira tentativa para verificar se T é substring de S. No pior caso, a primeira tentativa pode falhar, assim entramos no `se` o qual irá inverter a string e realizar uma nova tentativa com o KMP. Logo, temos $T(KMP) + T(invertString) + T(KMP) = O(M)$

c. `Discussão da complexidade da solução`

Dessa forma, nossa complexidade é linear, mesmo chamando o KMP duas vezes com apenas uma chamada da função de inverter uma string.

Questão 4

a. `explicação`

> 💡 Parar encontrar uma palavra na matriz precisamos checar se essa palavra está na `vertical de cima para baixo` , `vertical de baixo para cima`, `horizontal da esquerda para direita` e `horizontal da direita para esquerda`. Precisamos de duas funções importantes. `invertString` *(inverte uma string)* e `getVerticalWord` *(forma palavras na vertical).*
> Quando estivermos na primeira linha da Matriz precisamos verificar não apenas no sentido `horizontal` mas também no vertical *(de forma inversa também)*. Quando estivermos em qualquer outra linha basta verificar no sentido `horizontal`.
>
> A função `findWord` será responsável por imprimir todas as palavras encontradas e suas respectivas posições. Podemos usar o algoritmo de `Karp` para checar as ocorrências.

b. `pseudo-linguagem`

```
algoritmo getVerticalWord(MATRI, N, C)
{-
  ENTRADA:  Uma matriz quadarada de tamanho N.
  um inteiro C representando a coluna.
  SAÍDA: Uma string S no sentindo vertical da matriz da coluna C.
-}
inicio
  palavra = ''
  para i=1 ate N: -- Salta para próxima linha da matriz mantendo a coluna.
      palavra := palavra + MATRI[i][C]
  retorne palavra
fim
```

> Exemplo: Para MATRI =
> L O L O
> O L B A
> B A U T
> O K P J
> `getVerticalWord(MATRI, 4, 1)` → `"LOBO"`

```
algoritmo findWord(words, M, matriz, N)
{-
  ENTRADA:  Array de palavras de tamanho M. Ex: ['CASA', 'BOLO' ...]
  Matriz N*N representando um caça palavras.
  SAÍDA: Palavras encontradas no formato (PLAVRA, (LIN, COL), (LIN, COL))
-}
inicio
  para j=1 ate M faça -- Percorrendo cada palavra do array.
    para i=1 ate N faça -- Percorrendo as linhas da matriz.
      se (i = 1) então -- Precisamos 'caçar' na vertical.
          para c=1 ate N faça -- Percorrendo as colunas.
              inicio
                word_vertical = getVerticalWord(matriz, N, c) -- Palavra formada na vertical de cima para baixo.
                word_vertical_rever = invertString(word_vertical) -- Palavra formada na vertical de baixo para cima.
```

```
                res_vertical = karp(word_vertical, words[j]) -- Formato(texto, padrão)
                res_vertical_rever = karp(word_vertical_rever, words[j]) -- Formato(texto, padrão)

                se (res_vertical <> 0) então -- Verifica se foi encontrado algo na vertical de cima para baixo
                    imprima (words[j], (res_vertical, c), (res_vertical + words[j].tamanho, c))

                se (res_vertical_rever <> 0) então -- Verifica se foi encontrado algo na vertical de baixo para cima
                    imprima (words[j], (word_vertical_rever.tamanho - res_vertical_rever, c),
                        (abs((res_vertical_rever + words[j].tamanho) - word_vertical_rever.tamanho), c))
              fim

      res = karp(matriz[i], words[j]) -- Horizontal direita esquerda para direita.
      res_revers_horizon = karp(matriz[i], invertString(word[j])) -- Horizontal direita para esquerda.

      se (res <> 0) então
        imprima (word[j], (i, res), (i, res + words[j].tamanho))

      se (res_revers_horizon <> 0) então
          imprima (words[j], (i, res_revers_horizon + words[j]), (i, res_revers_horizon))


fim
```

> Quando estivermos na primeira linha da matriz precisamos percorrer todas suas colunas para formar as palavras verticalmente. É justamente o que o primeiro `se` verifica.

c. `discussão da complexidade da solução`

$\Sigma(j=1)(M) * [\Sigma(i=1)(N) * \Sigma(C=1)(N)]$ Resolvendo o último `para` (que percorrer as colunas temos que para cada coluna precisamos invocar a função de formar palavras e a função de inverter e por fim o algoritmo de Karp duas vezes.) Ficamos:
$\Sigma(j=1)(M) * [\Sigma(i=1)(N) * N(T(getVerticalWord) + T(invertString) + T(Karp))]$ =
$\Sigma(j=1)(M) * [N * N(T(getVerticalWord) + T(invertString) + T(Karp))]$ =
$\Sigma(j=1)(M) * [N^2(T(getVerticalWord) + T(invertString) + T(Karp))]$ =
$O(M * N^2(T(getVerticalWord) + T(invertString) + T(Karp))$ =
O(M * N^2 (O(N) + O(N) + O(N+M) =
$T(findWord) = O(M * N^2(N + M))$

Questão 5

a. `explicação`

> Precisamos das seguintes funções:
> `countFreq` → *r*ecebe uma string e contabiliza as frequências. *(será usada para construir a trie)*
> `buildTrie` → recebe as frequências e constrói nossa Trie.
> `buildCodeTable` → Recebe uma árvore arvore Huffman e retorna algo parecido como *{ A:'01', B:'0' .... }*
> `isLeaf` → Verifica se um dado nó na trie é folha.
> `dataCompress` → Recebe a tabela de código criada por `buildCode`, e a string a ser comprimida.
> `expand` → Recebe nossa `Trie`, `NoAtual` *(inicialmente é a Trie, ou seja a raiz)* e os dados codificados. Retornando assim a mensagem recomposta.

> Além disso, precisamos de uma estrutura(classe / struct) para representar `nós` em nossa árvore e de um Min-Heap que irá auxiliar na criação de nossa `Trie`.

b. `pseudo-linguagem`

```
algoritmo NoHof(ch, freq, Esq, Dir)
{-
  ENTRADA:  Um caarctere da tabelas ASCII, sua frequencia no texto,
  filhoEsquerdo e um filhoDir(ponteiros para mesma estrutura).
-}
inicio
    simbolo:= ch
    frequencia := freq
    filhoEsq := Esq
    filhoDir := Dir
fim
```

```
algoritmo NoHof(No)
{-
  ENTRADA:  Um NoHof.
  SAÍDA: Verdadeiro se o nó for uma folha, Falso caso contrário.
-}
inicio
   retorne No.filhoEsq = Null e No.filhoDir = Null
fim
```

```
algoritmo countFreq(T, M)
{-
  ENTRADA:  String T de tamanho M
  SAÍDA: Vetor de Frequencias para cada letra na string.
-}
inicio
    frequencias := [256] --Tabela ASCII
    para i=1 até 256 faça frequencia[i] = 0 -- inicializando as frequencias.
    para i=1 até M faça:
       frequencia[T[i]] := frequencia[T[i]] + 1
    retorne frequencias
fim
```

```
algoritmo buildTrie(F)
{-
  ENTRADA:  Recebe vetor F representando as frequencias.
  SAÍDA: Um ponteiro que aponta para raiz de nossa Trie.
-}
inicio
  minPq := CriarNossoHeapMin()
  para i=1 ate 256 faça -- Inserindo nossos caracteres no heap. (criando uma floresta)
      se (F[i] <> 0) então
          minPq.insert(i, F[i]) -- Estamos usando representação ASCII ex: i=65 = 'A'

  enquanto (minPq.tamanho > 1) faça
          NodeEsq := minPq.heap_extract_min()
          NodeDir := minPq.heap_extract_min()
          NodeParent := NoHof(NodeEsq.frequencia + NodeDir.frequencia, NodeEsq, NodeDir)
          minPq.insert(NodeParent)
  retorne minPq.heap_extract_min -- O último NoHof que restou é a nossa Trie.
fim
```

> Precisamos fazer uma pequena mudança no nosso Heap, pois nosso critério de comparação é um inteiro que representa a frequência. Isso pode ser feito com uma simples função O(1) que dado um `NoHof` ela retorna a frequência. Logo, para cada no `NoHof` basta aplicar essa função.

```
algoritmo buildCodeTable(ArvHof)
{-
  ENTRADA:  Recebe uma Trie e cria nossa tabela de codificação.
  SAÍDA: Um dicinário/TabelaHash com a tabela de codificação do nosso texto. Ex
{ o:'00', a: '010', c:'0110', u:'01110' ...}
-}
inicio
    tableCode := CriarDicionarioVazio() --
    buildCodeseRec(tableCode, ArvHof, "")
    return st
fim
----------------------------------------------------------------------------------
procedimento buildCodeseRec(tabela, Nodex, s)
{-
ENTRADA: Tabela de código vazia, Nó raiz da Trie e uma string vazia.
-}
inicio
    se Nodex.isLeaf() então -- É uma folha? Então Vamos adicionar o ch na tabela, com todo o percurso feito até aquela folha.
        tabela[Nodex.simbolo] = s -- Insere como chave o caractere, e o percurso até encontrá-lo
        retorne
    senão
        buildCodeseRec(tabela, Nodex.filhoEsq, s + '0')
        buildCodeseRec(tabela, Nodex.filhoDir, s + '1')
fim
```

```
algoritmo dataCompress(tabelaCodificacao, string, M)
{-
  ENTRADA: Tabela de codificação de nosso texto e o texto em estado original de tamanho M.
  SAÍDA: Dados comprimidos. Ex: '01010101010...'
-}
inicio
    dados_codificados := ''
    para i=1 ate M faça
        dados_codificados := dados_codificados + tabelaCodificacao[string[i]]
    retorne dados_codificados
fim
```

```
algoritmo expand(arv, noAtual, dadosCompc, string)
{-
  ENTRADA: Árvore de Huffman, um determinado nó (inicialmente será a raiz) os dados compactados no formato '10101010...'
  e uma string inicilmente vazia, que irá sendo incrementanda a cada chamada de decodifica.
  SAÍDA: Um texto representando os dados originais.
-}
inicio
    se (noAtual.isLeaf()) então -- É uma folha ? Então vamos pegar seu conteúdo e inserir na nossa string. E voltar para Raiz.
        retorne expand(Trie, Trie, dadosCompc, dadosOri + noAtual.simbol)

    se (dadosCompc = Null) então -- CASO BASE. Acabou os dados compactados ? Então retorne a string.
        return string

    cabeca := dadosCompc.removeCabeca() -- Captura a cabeça da string de dados compactados e remove-a.
    se (cabeca = '0'): -- Se for um '0' vamos descer em nossa árvore pelo lado esquerdo.
        retorne expand(arv, noAtual.filhoEsq, dadosCompc, string)
    senão:
        return expand(arv, noAtual.filhoDir, dadosCompc, string) -- Caso contrário, vamos pelo lado direito.
fim
```