Micael Andrade Dos Santos

1. Questão

   a. Pseudo-linguagem

```
--Inspirado no Levitin (Cap.8 pg.287)
algoritmo minCut(P, n, m)
{-
Aplicando programação dinâmica para encontrar cortes que maximizam
os lucros.
ENTRADA: Um array P de preços de tamanho n, onde cada índice do array é seu respectivo tamanho.
         E um inteiro m representando o tamanho do fio original.
SAÍDA: Um inteiro que representa a quantidade de cortes para maximizar o lucro.
-}
Inicio
  F := [m]
  para i = 1 até m faça: --Inicializa a tabela
    F[i] := 0

  para i = 1 até n faça:
    temp := INFINITO
    j = 1
    enquanto (j <= m && i>= P[j-1]):
      temp := min(F[i-P[j-1]], temp)
      j += 1
    F[i] := temp + 1
  retorne F[n]

fim
```

   b. complexidade

Temos que a complexidade de espaço do algoritmo é `O(m)` pois precisamos criar um vetor do tamanho do fio de entrada.
A complexidade de Tempo é `T(minCut) = Σ(i=1)(m)C4 + Σ(i=1)(n)C2 * Σ(j=1)(m)C3` =
T(minCut) = (m-1+1) + Σ(i=1)(n)[m-1+1] =
T(minCut) = O(m) + Σ(i=1)(n)[m] =
T(minCut) = O(m) + O(n*m) =

$$T(minCut) = O(n * m)$$

---

2. Questão

   a. Pseudo-linguagem

```
--Inspirado no Levitin (Cap.8)
algoritmo mochilaSemLimites(W, P, n, V)
{-
Resolvendo o problema da mochila com itens ilimitados à disposição.
ENTRADA: Um inteiro W representando a capacidade da mochila, Vetor de Precos de tamanho n
vetor V de valores com tamanho n.
SAÍDA: Valor máximo da mochila podendo conter itens repetidos.
-}
Inicio
  maxTab =: [W] -- tabela de diferentes tamanhos de mochila
  para i = 1 até W faça: --Inicializando nossa tabela
    maxTab[i] := 0
```

```
  para i = 1 até W
    para j=1 até n
      se(P[j] <= i) então --Se cabe na capacidade da mochila atual
        maxTab[i] := max(maxTab[i], maxTab[i-P[j]] + V[j]) --Escolhe o maior entre a solução atual e a anterior

  retorne maxTab[W]
fim
```

b. Complexidade

Temos que a complexidade de espaço do algoritmo é `O(W)` onde W é a capacidade máxima da mochila, ou seja, precisamos criar uma tabela com [1, 2, 3... W] tamanhos de mochilas diferentes. Além disso o tempo para inicializar essa tabela é trivial, ou seja, `O(W)`.

A complexidade de Tempo para encontrar o valor máximo é `T(mochilaSemLimites) = C1Σ(i=1)(W)Σ(i=1)(n)` =
T(mochilaSemLimites) = C1Σ(i=1)(W)[n-1+1] =
T(mochilaSemLimites) = (W-1+1)(n-1+1) =
T(mochilaSemLimites) = O(W*n) =
$T(mochilaSemLimites) = O(W * n)$

---

3. Questão

a. Pseudo-linguagem

```
algoritmo ocorrSubSeq(X,m,Y,n, TotSubSeq)
{- A ideia aqui é parecida com slide 19(Mais Longa Subsequência Comum)
  Foi feita apenas alguns ajustes.
entrada: sequências X e Y de tamanhos m e n;
saída: Total de vezes que Y aparece como subsequencia de X -}

início

  -- CASO BASE 1, a primeira estring seja vazia
  para i = 0 até n faça  TotSubSeq[0,i].tam := 0

  -- CASO BASE 2, caso a segunda string seja vazia, temos que todo conjunto vazio
  -- é subconjunto de qualquer conjunto por isso inicializamos com 1.
  para j = 0 até m faça TotSubSeq[j,0].tam := 1

  -- caso geral
  para i = 1 até m
    para j = 1 até n
      se X[i] = Y[j] -- Se os último caracteres forem iguais
        então TotSubSeq[i,j].tam := TotSubSeq[i-1,j-1].tam + TotSubSeq[i-1, j].tam
      senão -- Caso contrário ignoramos o caracter da primeira string.
        TotSubSeq[i,j].tam =: TotSubSeq[i-1,j].tam
  retorne TotSubSeq[m][n].tam
fim
```

b. Complexidade

Temos que a complexidade de **espaço** é dada pela construção da Matriz `TotSubSeq` a qual é parecida com a `LCS` do slide 19 a única diferença é que não precisamos dos atributo `dir`. $T(espaço) = O(n * m)$

A complexidade de tempo é dominada pelas operações dentro do `for j` aninhado. Temos `CΣ(i=1)(m)Σ(j=1)(n)` = `Cm*n` = $O(m * n)$

---

4. Questão

    a. Pseudo-linguagem

```
algoritmo superMinSubSeq(stringX,n,stringY,m)
{-
  Entra: Duas string e seus respectivos tamanhos.
  SAÍDA: Tamanho da supersequência mais curta das duas strings.
  A estratégia aqui é utilizar o algoritmo da mais longa subsequencia.Visto no Slids
-}

início
  lcs := longaSubSeqComum(stringx,n, stringy,m, LCS) -- Retorna um inteiro com o tamanho da Subsequência
  retorne (n + m) - lcs
fim
```

    b. Complexidade

    `T(superMinSubSeq)` é dominada totalmente pela função `longaSubSeqComum`. Como vimos nos slides `T(longaSubSeqComum)` = Σ(i=1)(m)Σ(j=1)(n)C = C*n*m = O(n*m).

---

5. Assim como o problema discutido no slide 19 *(mais longo sub-sequencia)* o problema do número mínimo de edição de sequência quando pensando em uma abordagem recursiva chegamos a uma solução exponencial, pois vamos **utilizar a indução muitas** vezes para reduzir em soluções não muito menores do que o problema original. Logo, podemos usufruir da programação dinâmica para armazenar as soluções menores em uma tabela.
   Com esta última abordagem reduzimos o tempo de execução que era exponencial para $O(mn)$ o que é uma GRANDE vantagem, porém precisamos de espaço adicional também $O(mn)$ assim como o problema dos slides.