Lista 6

Victor Sena Molero - 8941317

April 15, 2016

Ex 8. Descreva um algoritmo que, dados n inteiros no intervalo de 1 a k, preprocesse sua entrada e então responda em $O(1)$ qualquer consulta sobre quantos dos n inteiros dados caem em um intervalo $[a..b]$. O preprocessammento efetuado pelo seu algoritmo deve consumir tempo $O(n+k)$.

Resposta. O algoritmo para encontrar o tamannho das subsequências máximas foi explicado em aula. Segue uma implementação que considera as strings dadas nos vetores X e Y de tamanhos m e n respectivamente. Além disso, a função assume a existência de uma matriz global dp que vai guardar a resposta dos estados.

```
1: function LCS
2:     i ← 1
3:     while i ≤ m do
4:         j ← 1
5:         while j ≤ n do
6:             if X[i] = Y[j] then
7:                 dp[i][j] ← dp[i − 1][j − 1] + 1
8:             else
9:                 dp[i][j] ← max(dp[i − 1][j], dp[i][j − 1])
10:            end if
11:            j ← j + 1
12:        end while
13:        i ← i + 1
14:    end while
15: end function
```

Agora, precisamos conseguir imprimir todas as subsequências possíveis. Para isso, precisamos explorar todos os caminhos possíveis de subsequencias. A forma mais fácil de fazer isso é com um algoritmo recursivo, vou chamar de Imprime-Sequencias. Esta função recebe como parâmetro a posição a ser explorada na matriz calculada na programação dinâmica.

Nós vamos montar as sequências uma a uma, de trás para frente, ou seja, da ultima letra para a primeira. Toda vez que estivermos numa posição i, j com $X[i] = Y[j]$, sabemos que todas as strings que contém o final já montado contém, também, a letra da posição $X[i]$ (e $Y[j]$), logo, ela deve fazer parte do sufixo atual. Além disso, toda vez que estivermos numa posição i, j onde $X[i] \neq Y[j]$ e $dp[i][j] > dp[i][j-1]$, sabemos que a posição $i, j-1$ não faz

parte de nenhuma LCS que contenha aquele final, pois a maior subsequência daquele estado é menor do que a do estado atual e nenhuma letra foi ganha para transitar de um para o outro. O O pensamento análogo se aplica quando $dp[i][j] > dp[i-1][j]$.
Podemos, então, escrever nossa função. Assumindo uma string S vazia já alocada globalmente.

```
1: function Imprime-Sequencias(i, j)
2:     if dp[i][j] = 0 then
3:         Imprime(S)
4:     else
5:         if X[i] = Y[j] then
6:             S[dp[i][j]] ← X[i]
7:             Imprime-Sequencias(i − 1, j − 1)
8:         else
9:             if dp[i − 1][j] = d[i][j] then
10:                Imprime-Sequencias(i − 1, j)
11:            end if
12:            if dp[i][j − 1] = d[i][j] then
13:                Imprime-Sequencias(i, j − 1)
14:            end if
15:        end if
16:    end if
17: end function
```

Basta chamar esta função com os argumentos m, n e as sequências desejadas serão impressas. □

Ex 21. Escreva um programa que, dado o número n de faixas a serem colocadas na vitrine, calcule o número de maneiras de satisfazer as condições do proprietário.

Resposta. Basta observar que a primeira faixa a ser colocada nunca será uma azul, pois toda azul está entre uma branca e uma vermelha. Depois de colocarmos uma bandeira branca ou vermelha, temos duas opções, colocar uma faixa de cor oposta ou colocar uma faixa azul. Depois de colocar uma faixa azul não temos nenhuma opção, sempre colocaremos uma faixa da cor oposta à que veio antes da azul.
Assim, podemos criar uma pd que assume que sempre acabamos de colocar uma faixa vermelha ou branca e calcule quantas formas de colocar faixas nos restam. Não precisamos saber se a ultima faixa foi vermelha ou branca pois a resposta é igual nos dois casos e não precisamos separar o caso onde acabamos de colocar uma azul pois a resposta será igual à do outro caso só que com uma bandeira a menos precisando ser colocada (a que vem depois da azul, que deve sempre ser diferente da que vem antes da azul).
A resposta será, então, o dobro deste calculo feito para $n - 1$ (basta definir se a primeira faixa é vermelha ou branca.

```
1: function BANDEIRAS(n)
2:     res[0] ← 1
3:     res[1] ← 1
4:     i ← 2
5:     while i < n do
6:         res[i] ← res[i − 1] + res[i − 2]
7:     end while
8:     return 2 ∗ res[n − 1]
9: end function
```

Isso é exatamente o dobro do fibonacci de $n - 1$. E, por isso, esse problema pode ser resolvido em $O(\lg(n))$

□