Trabalho de Análise e Complexidade de Algoritmos

Rubens de Araújo Rodrigues Mendes

Novembro 2020

1 INTRODUÇÃO

Por meio deste documento serão relatados os processos para resolução dos exercícios propostos no trabalho. Cada problema será dividido em processos, sendo eles:

- Fazer uma definição formal do problema a ser resolvido.
- Dividir o problema em subproblemas menores, afim de encontrar uma solução por divisão e conquista.
- Implementar um algoritmo recursivo que resolva o problema por meio da divisão e conquista.
- Buscar uma subestrutura ótima, para a solução por programação dinâmica.
- Implementar um algoritmo utilizando a subestrutura ótima para resolver o problema usando memoização(*top-down*) ou tabulação(*bottom-up*).
- Fazer a análise assintótica dos algoritmos resolvidos por meio de divisão e conquista e programação dinâmica.

2 PROBLEMAS RESOLVIDOS

Aqui serão abordados os problemas escolhidos para a análise e resolução.

2.1 Strings binárias sem 1's consecutivos

O intuito desse exercício é retornar o número total de strings sem 1's consecutivos possíveis, para um dado tamanho de string **n**. Por exemplo, para strings de tamanho **3**, existem **5** combinações possíveis: **000, 001, 010, 100 e 101**, portanto o programa deverá retornar o valor **5** como saída final, caso o tamanho seja **4**, existem **8** combinações: **0000, 0010, 0100, 1000, 1010, 0101, 1001 e 0001**, tendo como retorno o valor **8**.

2.1.1 Solução por divisão e conquista

A abordagem para a solução por divisão em conquista se baseia na ideia de um vetor de caracteres, no caso uma *string* vazia onde os caracteres binários vão sendo inseridos a cada iteração, afim de verificar dígito à dígito quantas possibilidades existem para cada inserção. Por exemplo, caso o último dígito inserido na *string* seja **0**, abrem duas possibilidades para o próximo dígito à ser inserido, pois como não há restrições para **0's** consecutivos, o próximo número pode ser tanto **0** quanto **1**, por outro lado quando o último dígito é **1**, temos apenas 1 possibilidade, que é inserir mais um número **0** na string. A seguir podemos ver a árvore gerada pela lógica aplicada.

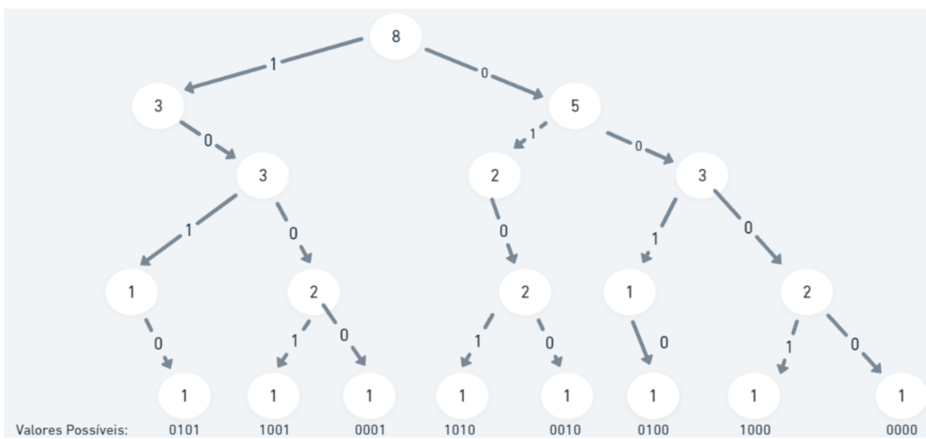

Figura 1: Árvore gerada pela solução por divisão e conquista para n = 4

Podemos notar três coisas nessa árvore, primeiro de tudo, que a o elemento **n** é dependente da soma dos elementos filhos **n-1**, sendo que, caso o último dígito inserido seja **0**, o nó gera 2 filhos, caso seja **1**, gera apenas um filho à sua direita, sendo que cada filho é uma nova possibilidade para a continuação da string. Segundamente, vemos que o lado direito da árvore pertence aos números possíveis que terminem em **0**, enquanto do lado esquerdo temos os valores possíveis que terminam em **1**, por último podemos notar subproblemas sobrepostos na árvore. A primeira observação será mais útil para a implementação recursiva, enquanto as duas últimas serão mais importantes em para chegar a subestrutura ótima, e por consequência, a solução por programação dinâmica.

2.1.2 Pseudocódigo e Análise da Implementação Recursiva

O algoritmo recebe como entrada o tamanho desejado da string e a raiz da árvore que deve ser inicializado como **0**. O pseudocódigo é o seguinte:

BS-COUNTER-REC(n, lastN)

1. **if** n == 0
2.   **return** 0
3. **if** n == 1
4.   **return** lastN == 1 ? 1 : 2
5. **if** lastN == 0
6.   BS-COUNTER-REC(n-1, 1) + BS-COUNTER-REC(n-1, 0)
7. **else if** lastN == 1
8.   BS-COUNTER-REC(n-1, 0)

A equação de tempo dessa recursão seria representado da seguinte maneira:

$$Str0(N) = Str0(N-1) + Str1(N-1) \quad se \quad lastN = 0$$

$$Str1(N) = Str0(N-1) \quad se \quad lastN = 1$$

Porém, nota-se que há uma substituição possível para unir as equações em apenas uma, que é obtida levando em consideração que $Str1(N-1) = str0(N-2)$ portanto a nova equação ficaria da seguinte forma:

$$Str0(N) = Str0(N-1) + Str0(N-2)$$

Fazendo a análise assintótica da melhor solução para o pior caso, verifica-se que o algoritmo possui tempo $\mathbf{O(2^n)}$, sendo portanto, exponencial, o que torna o algoritmo inviável para utilização. Todavia, na próxima seção apontaremos a subestrutura ótima que será utilizada para tornar o tempo para o pior caso, linear, $\mathbf{O(N)}$.

2.1.3 Subestrutura ótima

Para a solução por programação dinâmica, foi adotada a estratégia de adicionar tabulação(*bottom-up*), para lidar com subproblemas sobrepostos, na árvore da figura 1. Para gerar a subestrutura ótima, foi mantida a lógica de adicionar **0's** ou **1's** em uma determinada string. como caso base nós teremos **Str0**, que conterá todas as strings com final **0** e sem **1's** consecutivos e **Str1** que conterá todas as strings com final 1, **1's** consecutivos, ambas iguais a **1**, pois inicialmente teremos como soluções válidas apenas as *strings* "0"e "1"de tamanho **1**. A partir dos casos base, **Str0(n)** pegará as strings de **Str0(n-1)** e **Str0(n-1)** e adicionará um número **0** à elas, enquanto **Str1(n)** adicionará às strings de **Str0(n-1)** um número **1**, sendo assim, para n = 2, teremos para **Str0(n)** as strings "00"e "10", enquanto **Str1(n)** ficará com a string "01", totalizando 3 strings, portanto, a soma da quantidade de strings contidas em Str0(n) e Str1(n) é a solução ótima para o problema.

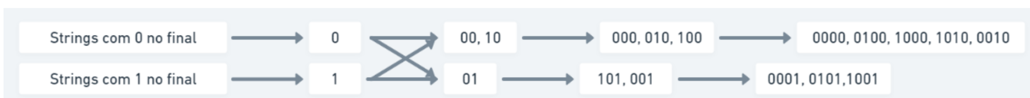

Figura 2: Solução ótima para n = 4, totalizando 8 strings no final.

2.1.4 Pseudocódigo e Análise da Implementação por Programação Dinâmica

O algoritmo recebe como entrada o tamanho desejado da string. O pseudocódigo é o seguinte:

BS-COUNTER-DIN(n)

1. **if** n == 0
2.   **return** 0
3. Str0 = Str1 = 1
4. **for** i = 1 **to** n
5.   tempStr0 = Str0
6.   Str0 = Str0 + Str1
7.   Str1 = tempStr0
8. **return** Str0 + Str1

O algoritmo possui um loop que vai de **1** até **n** e a complexidade de espaço é **O(1)** portanto a análise assintótica da solução para o pior caso é **O(N)**, o que o torna um algoritmo muito mais eficiente que o implementado utilizando recursão, que tinha complexidade $\mathbf{O}(2^n)$.

2.2 Maior subsequência para um palíndromo

Esse problema lida com palíndromos, que são basicamente frases ou palavras que possuem a mesma estrutura independentemente de serem lidas da esquerda para a direita ou da direita para a esquerda, como por exemplo a frase "**Anotaram a data da maratona**", se não for levada em consideração a quantidade de espaços, diferenciação de letras minúsculas ou maiúsculas ou acentuação, pode-se notar que a ordem das letras se mantém a mesma. Outros exemplos para palíndromos podem ser as palavras: **aba, arara, rever e salas**. O algorítmo proposto receberá uma *string* como entrada e deverá retornar o maior tamanho de um palíndromo utilizando as letras da *string* recebida. Por exemplo, para a palavra **balaio**, o retorno deverá ser **3**, podendo haver vários palíndromos de tamanho **3** como **ala, aia, aoa e aba**. A figura a seguir mostra uma intuição inicial para a solução do exercício.

	b	a	l	a	i	o
b	0	0	0	0	0	0
a	0	0	0	1	0	0
l	0	0	0	0	0	0
a	0	1	0	0	0	0
i	0	0	0	0	0	0
o	0	0	0	0	0	0

Figura 3: Matriz montada como intuição inicial do problema.

Inicialmente nota-se que procurar letras iguais na palavra nos dá uma boa base para o algoritmo que vem a seguir, cada letra igual na palavra, em posições diferentes aumenta em **2** o tamanho do palíndromo, na tabela da figura 3, ao encontrar uma letra igual, é adicionado 1 à posição ij correspondente, totalizando na soma, **2**, porém em muito exemplos, pode-se adicionar uma outra letra da palavra no meio do palíndromo para aumentar o seu tamanho em **1**.

2.2.1 Solução por divisão e conquista

O processo utilizado para se chegar a tabela da figura 3, foi basicamente utilizar strings **x** e **y**, para encontrar letras repetidas, deve-se levar em consideração que ambas são idênticas a string de entrada. Primeiramente, deve-se zerar a coluna diagonal, pois são as posições que correspondem a mesma letra, na mesma posição da palavra, em seguida é feita a comparação de cada uma das letras da palavra com as outras. Para a solução, começaremos a comparação da primeira letra, com a última, dependendo da igualdade, certas operações recursivas serão implementadas. levando em consideração **i** como o início da string, **f** como o fim, e **s** sendo um vetor de caracteres contendo a palavra de entrada, caso $s[i] = s[f]$ acrescentamos **2** ao tamanho final do palíndromo, visto que ambas as letras nas posições $s[i]$ e $s[f]$ farão parte do palíndromo final, ao somar, também será feita a chamada recursiva passando $s[i+1, f-1]$, já que encontramos uma igualdade nas letras. Caso ambas não sejam iguais, não se adiciona nada ao tamanho final do palíndromo e deverão ser feitas 2 chamadas recursivas passando $s[i+1, f]$ e $s[i, f-1]$ para testar não deixar que escape uma verificação na posição **i** ou **f**, afim de pegar o valor máximo de retorno das recursões, ambas serão envoltas de uma função que retorne o maior valor de retorno, portando a recursão seria a seguinte: $max(s[i+1, f], s[i, f-1])$. Como casos base, também haverá a verificação de igualdade, sendo o primeiro caso base quando, $s[i]$ e $s[f]$ são iguais, retornando **1** e quando são iguais, e um caso base para caso tenham apenas 2 letras, e elas sejam iguais. Essa última mais opcional, porém importante para a integridade do algoritmo. Formalmente temos:

$$T(i, f) = T(i+1, f-1) \;\; se \;\; s[i] = s[f]$$

$$T(i, f) = max(T(i+1, f), T(i, f-1)) \;\; se \;\; s[i] \neq s[f]$$

2.2.2 Pseudocódigo e Análise da Implementação Recursiva

P-COUNTER-REC(s, i, f)

1. **if** i == f
2.   **return** 0
3. **if** s[i] == s[f] and i+1 = f
4.   **return** 2
5. **if** s[i] == s[f]
6.   **return** P-COUNTER-REC(s, i+1, f-1)
7. **else if** s[i] != s[f]
8.   **return max**(P-COUNTER-REC(s, i+1, f),P-COUNTER-REC(s, i, f-1)

Para fazer a análise do pior caso, devemos observar que, caso uma letra seja diferente da outra, 2 recursões são chamadas, enquanto caso sejam iguais, apenas 1 será chamada, portanto, o pior caso é quando não há letras iguais na palavra, gerando a cada iteração, 2 novas recursões. Como podemos ver na figura 4, no primeiro nível da árvore temos 2^0 nós, enquanto nos níveis 2,3,4 e 5 temos respectivamente 2^1, 2^2, 2^3 e 2^4, notamos um comportamento exponencial no aumento do número de nós da árvore, sendo para cada nível i, temos 2^{i-1} nós. Portando possui limite assintótico superior $\mathbf{O(}2^n\mathbf{)}$.

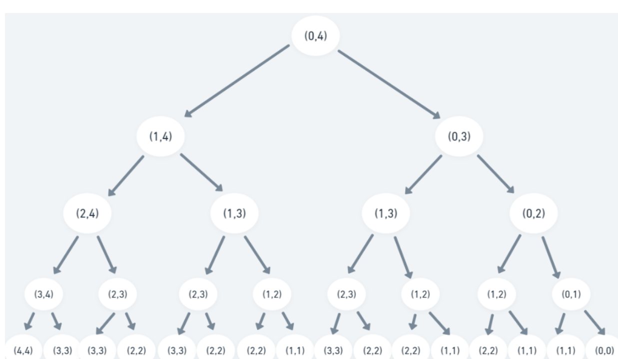

Figura 4: Árvore gerada pela solução recursiva no pior caso, quando não há letras repetidas na palavra.

2.2.3 Subestrutura Ótima

A solução recursiva gerou uma árvore semelhante à esta:

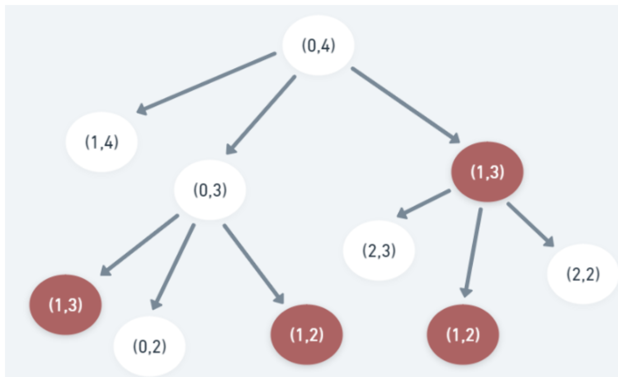

Figura 5: Árvore gerada pela solução recursiva, com problemas sobrepostos em vermelho.

Visto que encontramos problemas sobrepostos, podemos usar a estratégia de programação dinâmica, utilizaremos uma estratégia *bottom-up*,que para mim, lembra uma LCS na maneira de resolver. Basicamente, a lógica de verificar a primeira e última letra se mantém, no entanto as iterações não irão se basear nas letras, a subestrutura se encontrará em uma matriz igual à figura 3. Por exemplo, o maior palíndromo para sequência de tamanho **1** é **1**, sendo assim, a coluna diagonal principal seria preenchida com **1's**, ao fim da primeira iteração o tamanho da sequência sobe em 1 e caso essa sequência tenha 2 letras iguais, a posição $M[i, j]$ da matriz é igual à $M[i+1, j-1]+2$, caso não sejam iguais, a posição $M[i, j]$ pega o valor máximo entre $M[i+1, j]$ e $M[i, j-1]$ de maneira semelhante à uma LCS, na figura abaixo podemos ver um exemplo com a palavra **balaio**.
Sendo M a matriz da subestrutura ótima, s a string de entrada, i o início da sequência, f o fim da sequência e j o tamanho da sequência, formalizando, temos:

$$M[i][f] = m[i+1][f-1] + 2 \;\; se \;\; s[i] = s[f]$$

$$M[i][f] = 2 \;\; se \;\; s[i] = s[f] \;\; e \;\; j = 2$$

$$M[i][f] = max(m[i][f-1], m[i+1][f]) \;\; se \;\; s[i] \neq s[f]$$

O resultado ficará contido na posição $M[0, f-1]$ da matriz. Acredito que

	b	a	l	a	i	o
b	1	1	1	3	3	3
a		1	1	3	3	3
l			1	1	1	1
a				1	1	1
i					1	1
o						1

Figura 6: Exemplo da estratégia *bottom up* para a palavra balaio.

tenha sido uma boa maneira de se manter utilizando as recursões obtidas por divisão e conquista e aplicar um pouco da lógica utilizada na aula para solução em tabela da LCS.

2.2.4 Pseudocódigo e Análise das Implementações Dinâmicas

Para solução por tabulação, recebemos como entrada uma string **S**, o mesmo foi implementado em Python, mais observações estarão contidas nos comentários do código que será enviado, assim como a solução memoizada que também foi implementada em Python.

P-COUNTER-DIN-TAB(S)

```
for i = 0 to S.length
    m[i][i] = 1
for j = 2 to S.length + 1
    for i = 0 to S.length - j + 1
        if s[i] = s[f]
            if j = 2
                m[i][f] = 2
            else
                m[i][f] = m[i+1][f-1] + 2
        else if s[i] ≠ s[f]
            m[i][f] = max(m[i][f-1], m[i+1][f])
```

A solução memoizada, será aqui descrita apenas como uma função auxiliar que recebe como parâmetro a função recursiva já descrita nas seções anteriores. O intuito da mesma é guardar os subproblemas que já foram recebidos em cache, para que caso eles precisem ser resolvidos de novo, o tempo levado por cada problema seja apenas O(1), otimizando o tempo da solução recursiva.

MEMOIZE-PALINDROME(func)

1. palindromeCache = [ ]
2.   **function access**(s,i,f)
3.   **if**(s,i,f) $\notin$ palindromeCache
4.   palindromeCache[s,i,f] = func(s,i,f)
5.   **return** palindromeCache[s,i,f]
6. **return** access

Para o código por tabulação, fazendo a análise pelos loops, levando em consideração que **n** é o tamanho da string, e j o tamanho da subsequência analisada temos um loop que vai de **2** até **n**, portanto **O(n)** e outro loop que vai de **0** até **n-j**, também **O(n)**, temos que a solução por tabulação leva tempo $\mathbf{O}(n^2)$, assim como a solução por memoização que também leva tempo $\mathbf{O}(n^2)$.

2.3 Maior sub-matriz quadrada de 1's em uma matriz binária

Esse exercício, recebe como entrada uma matriz composta de 0's e 1's, o objetivo aqui é retornar o tamanho da maior sub-matriz quadrada composta de **1's** na matriz de entrada. Por exemplo: para a matriz:

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

A maior matriz quadrada possível é **3x3**, que vai de A[3][2] até A[5][4], ou então A[3][3] até A[5][5]. Portanto o resultado de saída do algorítmo será **3**.

2.3.1 Solução por divisão e conquista

Para este problema, inicialmente acabei encontrando uma solução por programação dinâmica antes de uma recursiva, porém a divisão e conquista se mantém para ambas as aplicações. Basicamente a solução passa pela análise dos vizinhos de um determinado nó de uma matriz **M**. Aqui a análise, será utilizando os vizinhos de cima, do lado, e da diagonal superior esquerda. An-

1	1	1	1	1
1	2	2	2	2
1	2	3	3	3
1	2	3	4	4
1	2	3	4	5

Figura 7: Matriz montada como intuição inicial do problema.

tes de chegar a qualquer conclusão eu já tinha alguma ideia de como poderia fazer esse exercício, foi utilizado o excel para ver se algum cálculo poderia fazer sentido para retornar o resultado 5, já que me acostumei a resolver por tabulação e o problema já está nesse formato. A figura 7 mostra o resultado da análise. Basicamente, se na posição **M[row][col]** o número for

1, significa que essa posição pode fazer parte de uma sub-matriz válida, o contador que será chamado de **tempCount**, recebe o mínimo dos vizinhos analisados + 1, ou seja: **tempCount** $= min(M[row-1][col-1], M[row-1][col], M[row][col-1]) + 1$. E caso o número for 0, executamos a mesma operação com **min**, porém sem somar 1, já que **0's** não fazem parte da sub-matriz. Ao fim, pegamos o maior valor de contagem, no caso da figura 7 seria o número **5** a ser retornado. Formalizando temos:

$$T(row, col) = min(T(row-1, col-1), T(row-1, col), T(row, col-1)) + 1 \;\; se \;\; M[row][col] = 1$$

$$T(row, col) = min(T(row-1, col-1), T(row-1, col), T(row, col-1)) \;\; se \;\; M[row][col] \neq 1$$

2.3.2 Pseudocódigo e Análise da Implementação Recursiva

MAT-QUAD-REC(M, row, col, count)

1. **tempCount** = 0
2. **if**(row < 0 or col < 0)
3.   **return** 0
4. **if**(M[row][col] = 1)
5.   tempCount = min(MAT-QUAD-REC(M, row, col-1, count),MAT-QUAD-REC(M, row-, col, count),MAT-QUAD-REC(M, row-, col-1, count)) + 1
6. **else**
7.   tempCount = min(MAT-QUAD-REC(M, row, col-1, count),MAT-QUAD-REC(M, row-, col, count),MAT-QUAD-REC(M, row-, col-1, count))
8. count = max(count,tempCount)
9. **return** count

Como cada caso, abre sempre 3 recursões, é fácil de se analisar que para cada nível **i**, teremos 3^i nós, portanto crescendo até 3^{n-1} então, o limite assintótico superior para a solução recursiva é $\mathbf{O(3^n)}$.

2.3.3 Subestrutura Ótima

A recursão gerou a seguinte árvore, nela podemos ver os subproblemas sobrepostos pintados de vermelho: Para lidar com isso uma proposta por tabulação

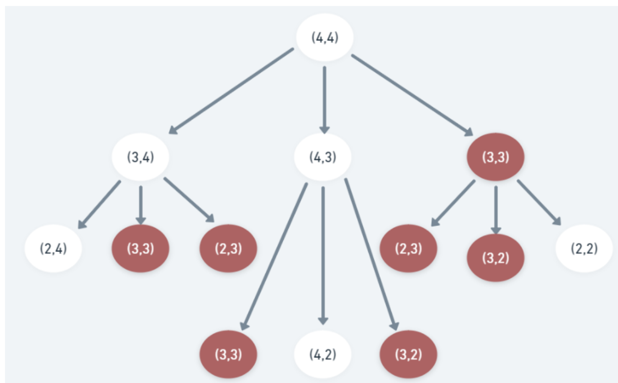

Figura 8: Árvore gerada pela solução recursiva, com subproblemas sobrepostos pintados de vermelho.

é trazida, a mesma lógica dos vizinhos será aplicada aqui. Porém dessa vez utilizaremos uma matriz que será nossa subestrutura ótima, semelhante à figura 7 no tópico de solução por divisão e conquista. Tendo **N** como a matriz ótima e **M** como a matriz de entrada temos que:

$$N[row][col] = min(N[row-1][col-1], N[row-1][col], N[row][col-1])+1 \ \ se \ \ M[row][col] = 1$$

$$N[row][col] = 0 \ \ se \ \ M[row][col] \neq 1$$

Após a matriz ótima ser computada, basta pegarmos o valor máximo dentro da mesma, que essa será a saída do algoritmo.

2.3.4 Pseudocódigo e Análise da Implementação Dinâmica

O algoritmo recebe como entrada apenas uma matriz a ser analisada.
MAT-QUAD-DIN(M)

1. rows = M.length
2. cols = M[0].length
3. N = [rows][cols]
4. **for** row = 0 **to** rows
5.   **for** col = 0 **to** cols
6.   **if** M[row][col] = 1
7.   N[row][col]=min(N[row-1][col-1],N[row-1][col],N[row][col-1]) + 1
8.   **else**
9.   N[row][col] = 0
10. arrM = [ ]
11. **for** row = 0 **to** rows
12.   **for** col = 0 **to** cols
13.   arrM.add(N[row][col])
14. **return** max(arrM)

Levando em consideração que **m** é o número de linhas e n o número de colunas visto que o primeiro loop percorre de 0 até m e o segundo de 0 até n, podemos afirmar que o algoritmo possui limite assintótico superior **O(n*m).** Que é muito mais eficiente em relação à solução recursiva. Obs: no código em python também há uma função para vetorizar a matriz e recolher o valor máximo, como ela também é **O(n*m).** pode-se afirmar que a função como um todo é **O(n*m).**

3 Considerações Finais

Acredito que o exercício do palíndromo foi mais difícil de resolver, pois a parte recursiva foi até rápida de chegar, porém a parte da subestrutura ótima e programação dinâmica foram um pouco complicadas de chegar, enquanto no exercício da matriz eu cheguei de maneira rápida à solução por programação dinâmica mas demorei para achar a solução recursiva, ainda que fosse um pouco fácil, a solução que imaginei tinha que usar um contador passado por referência, e usar o c++ foi a melhor opção para mim naquela hora. Tenho certeza que é um trabalho que agregou bastante no conhecimento dos alunos apesar de ser trabalhosa a documentação e demorada a análise para chegar na lógica dos problemas. O exercício 1 foi feito em C++, o 6 todo em python, onde fiz até algumas análises por tempo no código,e o 2 teve a recursão feita em C++ e a programação dinâmica feita em python.