Pedro Henrique Andrade Trindade

# 1 Prova Final Programação inteira e Combinacional

## 1.1 Questão 1: (Arranha-céus!)

**(a) Quadrados latinos.** Proponha um modelo de otimização linear inteira para resolver o jogo quadrados latinos n × n. Suponha, para o seu modelo, que algumas posições do quadrado estão preenchidas inicialmente. Indique claramente as variáveis de decisão adotadas e justifique as restrições utilizadas.

**Resposta:**

Para modelar o problema, as variáveis de decisão representarão "qual número será colocado na posição $(i, j)$".

Como o problema utiliza variáveis discretas, utilizei variáveis de auxílio binárias $z_{ijk} \in \mathbb{B}$ que definem qual valor entre os $n$ possíveis será utilizado naquela posição, qual valor foi utilizado é representado pelo índice $k$. Assim também foi necessário incluir uma restrição que garanta que apenas um dos números será utilizado por posição, isto é:

$$\sum_{k=1}^{n} z_{ijk} = 1, \ \forall i, j \quad \text{(somente um valor por célula)}$$

Mais explicitamente as variáveis de decisão foram:

$$z_{ijk} \in \{0, 1\} \text{ para } i, j \in \{1, \dots, n\}, k \in \{1, \dots, n\}$$

E para encontrarmos qual valor de fato $x_{ij} \in \{1, 2, \dots, n\}$ se encontrará na posição $(i, j)$ é só uma questão de fazer uma soma ponderada das variáveis de decisão, isto é

$$x_{ij} = \sum_{k=1}^{n} k.z_{ijk}$$

que podemos imaginar como variáveis de decisão a parte e colocar na formulação (como eu fiz) ou pensar na formulação só em função de $z_ijk$ e calcular $x_{ij}$ por fora.

**Formulação final:**

Para a função objetivo utilizei uma constante dummy $C_0$

$$
\begin{aligned}
&\text{maximize } C_0 \\
\text{sujeito a } &\sum_{k=1}^{n} z_{ijk} = 1, \quad \forall i, j \quad \text{(somente um valor por célula)} \\
&\sum_{j=1}^{n} z_{ijk} = 1, \quad \forall i, k \quad \text{(cada valor aparece uma vez por linha)} \\
&\sum_{i=1}^{n} z_{ijk} = 1, \quad \forall j, k \quad \text{(cada valor aparece uma vez por coluna)} \\
&x_{ij} = \sum_{k=1}^{n} k \cdot z_{ijk}, \quad \forall i, j \quad \text{(definição do valor final)} \\
&z_{ijk} = 1. \quad \forall (i, j, k) \in \text{(Valores dados das posições fixadas (i,j))}
\end{aligned}
$$

[92]:
```python
import cvxpy as cp
import numpy as np

import warnings
warnings.filterwarnings('ignore')

def make_discrete_decision_variables(array_shape, allowed_values):
    """
    Cria variáveis de decisão binárias para representar valores discretos.
    """
    # Variável binária 3D: z[i,j,k] = 1 se C[i,j] == allowed_values[k]
    z = cp.Variable(array_shape + (len(allowed_values),), boolean=True,␣
 ↪name='z')

    constraints = []
    # Cada célula [i,j] deve ter exatamente um valor selecionado
    for i in range(array_shape[0]):
        for j in range(array_shape[1]):
            constraints += [cp.sum(z[i, j, :]) == 1]  # Somente um valor por␣
 ↪célula

    return z, constraints

def guarantee_one_of_each_in_full_matrix(z_array, allowed_values):
    """
    Garante que cada valor apareça exatamente uma vez em cada linha e coluna.
    """
    constraints = []

    # Para cada linha, cada valor deve aparecer exatamente uma vez
    for row in range(z_array.shape[0]):
        for v_idx, _ in enumerate(allowed_values):
```

```
            constraints += [cp.sum(z_array[row, :, v_idx]) == 1]

    # Para cada coluna, cada valor deve aparecer exatamente uma vez
    for col in range(z_array.shape[1]):
        for v_idx, _ in enumerate(allowed_values):
            constraints += [cp.sum(z_array[:, col, v_idx]) == 1]

    return constraints

def get_final_solution_constraints(decision_variables, allowed_values):
    """
    Converte as variáveis binárias em uma solução final inteira.
    """
    # Variável final que conterá os valores inteiros
    final_result = cp.Variable(decision_variables[:,:,0].shape, integer=True,␣
 ↪name='final')

    # Constrói a matriz resultado combinando as variáveis binárias com os␣
 ↪valores permitidos
    final_result_arr = np.zeros(decision_variables[:,:,0].shape)
    for idx in range(decision_variables.shape[2]):
        final_result_arr = final_result_arr + decision_variables[:,:, idx] *␣
 ↪allowed_values[idx]

    # Restrição que iguala a variável final ao cálculo
    # (ps: Talvez redundante? Mas a lib não estava me ajudando sem fazer isso)
    constraint = [final_result == final_result_arr]

    return final_result, constraint

# Parâmetros do problema
n = 4
allowed_values = [1, 2, 3, 4]  # Valores discretos permitidos

# Cria as variáveis de decisão e restrições iniciais
decision_variables, constraints = make_discrete_decision_variables((n, n),␣
 ↪allowed_values)

# Adiciona restrições de latin square (cada valor uma vez por linha/coluna)
constraints += guarantee_one_of_each_in_full_matrix(decision_variables,␣
 ↪allowed_values)

# Prepara a solução final
final_solution, final_solution_constraints =␣
 ↪get_final_solution_constraints(decision_variables, allowed_values)
constraints += final_solution_constraints
```

```
# Restrições de teste para fixar alguns valores
z = decision_variables
"""
Matriz de Teste:
    [[x. 1. x. x.]
     [x. 3. 2. 1.]
     [x. x. x. 3.]
     [x. 4. x. x.]]
"""

test_constraints = [
    z[1, 1, 2] == 1,  # C[1,1] = 3 (allowed_values[2])
    z[1, 2, 1] == 1,  # C[1,2] = 2
    z[1, 3, 0] == 1,  # C[1,3] = 1
    z[2, 3, 2] == 1,  # C[2,3] = 3
    z[3, 1, 3] == 1,  # C[3,1] = 4
    z[0, 1, 0] == 1   # C[0,1] = 1
]
constraints += test_constraints

# Formula e resolve o problema
objective = cp.Maximize(0)
problem = cp.Problem(objective, constraints)
problem.solve(solver=cp.SCIP)

# Exibe os resultados
print("Status da solução:", problem.status)
print("Solução final:")
print(final_solution.value)
```

```
Status da solução: optimal
Solução final:
[[2. 1. 3. 4.]
 [4. 3. 2. 1.]
 [1. 2. 4. 3.]
 [3. 4. 1. 2.]]
```

**(b) Prédios vistos em uma fileira.** Suponha que uma fileira de n prédios de alturas inteiras e distintas de 1 até n seja descrita pela sequência h1 , · · · , hn das suas alturas. Proponha um modelo de otimização inteira cuja solução forneça o número de prédios que seriam vistos por uma pessoa próxima ao primeiro prédio da fileira. Indique as variáveis adotadas e justifique as restrições utilizadas.

**Resposta:**

Esta condição aqui foi um pouco mais complicada de pensar, minha linha de raciocínio foi utilizar o fato que podemos representar operações lógicas com PLI.

Assim pensei nas seguintes variáveis de decisão $v_i \in \{1, 2, \dots, n\}$ que define se o prédio $i$ é visível da ponta da fileira, para implementar a lógica por trás dessa definição utilizei variáveis auxiliares $c_{ij} \in \mathbb{B}$, que indicam se o prédio $i$ é tampado pelo prédio $j$ (ou seja, se a altura $h_j$ é maior que $h_i$, dado que $j < i$, se não $c_{ij}$ trivialmente igual a 0).

Com estas variáveis em mãos, definir se o número de torres visíveis na ponta da fileira é uma questão de fazer o somatório

$$\text{(Número de torres visíveis na borda da fileira)} = \sum_{i=1}^{n} v_i$$

As restrições ficaram um pouco mais convolutas, tentei fazer o seguinte raciocínio

$$\text{(A torre é visível na borda da fileira)} \Leftrightarrow \text{ (Esta torre não é tampada por nenhum } j \text{ anterior)}$$
$$v_i = 1 \Leftrightarrow \sum_j c_{ij} = (i-1), \ \forall (j < i)$$

E também fiz

$$\text{(A torre i não é tampada por j)} \Leftrightarrow (h_i > h_j) \text{ ou } (j \geq i)$$
$$c_{ij} = 1 \Leftrightarrow, (h_j \leq h_i) \text{ e } (j < i)$$

daí foi uma questão de formular estas operações lógicas em termo dessas variáveis booleanas determinadas, utilizando grande M (também precisei usar um *epsilon* de sensibilidade para evitar problemas no simulador).

para o primeiro "se o somente se" as restrições ficaram:

$$\begin{aligned} &h_i - h_j + \epsilon \leq M.c_{ij} \ \forall i, (j < i) \\ &h_j - h_i - \epsilon \leq M.(1 - c_{ij}) \ \forall i, (j < i) \\ &c_{ij} = 0 \quad \forall i, (j \geq i) \end{aligned}$$

A ideia aqui é que representamos o "se e somente se" ($\Leftrightarrow$) como duas condições, uma suficiente ($\Rightarrow$) e uma necessária ($\Leftarrow$) que no fundo são representado pelas duas restrições, que forçam o $c_{ij}$ a um ou zero dependo de se a altura de $h_i$ é maior que $h_j$ ou não.

Para o "se e somente se" da visibilidade fiz assim

$$\begin{aligned} &(i-1) - \sum_{j=1}^{n} c_{ij} + \epsilon \leq M.v_i \ \forall i, \\ &\sum_{j=1}^{n} c_{ij} - (i-1) - \epsilon \leq M.(1 - v_i) \ \forall i, \end{aligned}$$

Coloquei para função objetivo apenas uma constante dummy $C_0$, pois satisfazendo-se as equações com as variaveis, já temos nossa resposta como a soma dos $v_i$.

**Formulação Final:**

$$
\begin{aligned}
\text{maximize} \quad & C_0 \\
\text{sujeito a} \quad & h_i - h_j + \epsilon \leq M.c_{ij} \quad \forall i, (j < i) \\
& h_j - h_i - \epsilon \leq M.(1 - c_{ij}) \quad \forall i, (j < i) \\
& c_{ij} = 0 \quad \forall i, (j \geq i) \\
& (i-1) - \sum_{j=1}^{n} c_{ij} + \epsilon \leq M.v_i \quad \forall i, \\
& \sum_{j=1}^{n} c_{ij} - (i-1) - \epsilon \leq M.(1 - v_i) \quad \forall i, \\
& c_{ij}, v_i \in \mathbb{B} \quad \forall i, j.
\end{aligned}
$$

[50]:
```python
import cvxpy as cp
import numpy as np
from numpy.typing import NDArray
import random

def get_ordered_comparison_model(line_array: NDArray, epsilon=1e-3, M=1e8):
    """
    Essa função gera um modelo de PLI que podemos utilizar para
    resolver quantas torres são visiveis a partir do primeiro indice
    dado um array de torres.
    """
    # Gera subarrays para comparação
    ordered_list = get_list_of_arrays_less_than_k(line_array)

    # Variável binária que indica as comparações
    Cis = cp.Variable(
        (line_array.shape[0], len(ordered_list)),
        boolean=True,
        name=f'c{random.randrange(1,1000)}'
    ) # Criei um label para as variaveis, para tornar mais fácil debuggar depois

    # Variável que indica se a torre é visível
    tower_visibility = cp.Variable(line_array.shape[0], boolean=True)

    constraints_hi = []

    for idx in range(len(ordered_list)):
        # Restrições do tipo "se e somente se" usando Big-M
        # checando se uma dada torre está sendo tampada por outra anteriormente.
        constraints_hi += [
```

```
            ordered_list[idx][idx] - ordered_list[idx][k] + epsilon <= M *␣
↪(Cis[idx, k])
            for k in range(ordered_list[idx].shape[0])
        ]
        constraints_hi += [
            ordered_list[idx][idx] - ordered_list[idx][k] + epsilon >= -M * (1␣
↪- Cis[idx, k])
            for k in range(ordered_list[idx].shape[0])
        ]

        # Preenche o resto da matriz com zeros
        constraints_hi += [
            Cis[idx, k] == 0
            for k in range(ordered_list[idx].shape[0], len(ordered_list))
        ]

        # Restrições para determinar a visibilidade da torre
        constraints_hi += [
            cp.sum(Cis[idx, :]) - (idx + 1) + epsilon <= M *␣
↪(tower_visibility[idx])
        ]
        constraints_hi += [
            cp.sum(Cis[idx, :]) - (idx + 1) + epsilon >= -M * (1 -␣
↪tower_visibility[idx])
        ]

    return tower_visibility, Cis, constraints_hi

def get_list_of_arrays_less_than_k(line_array: NDArray):
    """
    Gera uma lista de todos os subarrays contendo os primeiros k+1 elementos,
    Para todos os valores menores que o shape do array.
    """
    return [line_array[:k+1] for k in range(line_array.shape[0])]

# Dado de exemplo
A = np.array([3, 2, 4, 1])
print("Subarrays gerados:", get_list_of_arrays_less_than_k(A))

# Parâmetros
epsilon = 1e-5  # Tolerância para desigualdade estrita
M = 1e5         # Valor do Big-M

# Gera o modelo
tower_vis, Cis, constraints = get_ordered_comparison_model(A, epsilon, M)

# Formula o problema de otimização
```

```python
#objective = cp.Maximize(cp.sum(tower_vis))
objective = cp.Maximize(0)

problem = cp.Problem(objective, constraints)

# Resolve o problema
problem.solve(solver=cp.SCIP)

# Imprime resultados
print("Status do problema:", problem.status)

# Resultado final
print(f"Número de torres visíveis: {int(np.sum(tower_vis.value))}")
```

```
Subarrays gerados: [array([3]), array([3, 2]), array([3, 2, 4]), array([3, 2, 4,
1])]
Status do problema: optimal
Número de torres visíveis: 2
```

**(c) Modelo Final.** Adapte as ideias usadas nos itens anteriores para propor o modelo de otimização inteira que resolve o desafio dos Arranha-Céus. Valide o seu modelo com ao menos 3 exemplos gerados na url sugerida.

**Resposta:**

Para modelar o problema utilizando as construções feitas anteriormente, foi uma questão de organizar como eu ia representar o jogo. A parte mais complicada foi definir um formato que fosse interessante para as dicas laterais.

A forma como eu fiz foi de fazer um tensor $s_{i\ell m}$ de tamanho $(n, 2, 2)$ onde a primeira entrada representa a posição relativa no quadrado, a segunda coordenada nos fala se a dica é uma dica de linha ou de coluna, já a terceira entrada indica a orientação da dica "direita/esquerda" se for uma dica de linha e "para baixo / para cima" se for uma dica de coluna. Se naquele quadrado não houverem dicas eu defini $s_{ijk} = 0$

Assim iteramos por todas as dicas. Utilizamos as restrições dos quadrados latinos feita anteriormente para o quadrado todo, depois pegamos e adicionamos as restrições de visualização das torres para cada dica, utilizando o problema anterior.

(Ps O conjunto total das restrições ficou relativamente grande, não consegui achar uma forma notacional de representar a formulação toda que fizesse sentido com a mudança de orientações sem apelar para uns negócios chatos, porém é só uma questão sobre reordenar os indices, então vou explicar com palavras)

Para a função objetivo escolhi uma constante dummy $C_0$. Para cada dica reordenei (por conta da orientação, tipo de dica) as variáveis de entrada nas restrições do problema anterior de acordo, para cada linha eu criei as restrições na ordem normal para todo $x_{ij}$ pertencente aquela linha, para a orientação contrária eu reordenei as variaveis de maneira contrária (e respectivamente na coluna também, só que dessa vez atravessando pelas colunas não linhas) também adicionei a condição de quadrado latino para o quadrado todo (cada $x_{ij}$).

Em especial adicionei a condição de que o número de torrem visiveis tem que ser igual ao valor da dica para cada dica, isto é $\sum_{i=1}^{n} v_{i\ell} = s_\ell,\ \forall \ell$ (denovo, desculpa pela notação confusa, os indices mudar de orientacao me atrapalharam muito).

[87]:
```
def get_side_towers_constraints(final_result_vars, side_towers):
    constraints = []

    # Variável que representa a soma total das torres visíveis dado cada␣
↪posição, tipo e orientação
    # das pistas laterais
    total_sum = cp.Variable(side_towers.shape, integer=True, name='tower_sums')

    # Percorre todas as posições da matriz de pistas laterais
    for position in range(side_towers.shape[0]):
        for pos_type in range(side_towers.shape[1]):
            for orientation in range(side_towers.shape[2]):
                towers_value = side_towers[position, pos_type, orientation]

                if not (towers_value == 0):
                    # Determina se a visualização será por linha ou por coluna␣
↪e em qual direção
                    if pos_type == 0:  # Linha
                        if orientation == 0:  # Da esquerda para a direita
                            tower_vis, _, tower_constraint =␣
↪get_ordered_comparison_model(final_result_vars[position, :])
                        else:  # Da direita para a esquerda
                            tower_vis, _, tower_constraint =␣
↪get_ordered_comparison_model(final_result_vars[position, ::-1])
                    else:  # Coluna
                        if orientation == 0:  # De cima para baixo
                            tower_vis, _, tower_constraint =␣
↪get_ordered_comparison_model(final_result_vars[:, position])
                        else:  # De baixo para cima
                            tower_vis, _, tower_constraint =␣
↪get_ordered_comparison_model(final_result_vars[::-1, position])

                    # Adiciona restrições relacionadas à soma das torres␣
↪visíveis
                    constraints += [total_sum[position, pos_type, orientation]␣
↪== cp.sum(tower_vis)]
                    constraints += tower_constraint
                    constraints += [towers_value == cp.sum(tower_vis)]
                else:
                    # Se não há torre na posição, a soma deve ser zero
                    constraints += [total_sum[position, pos_type, orientation]␣
↪== 0]
```

```
    return total_sum, constraints


def solve_towers_problem(n, side_towers):
    # Lista de restrições inicial da modelagem
    constraints = []

    # Conjunto de valores permitidos
    allowed_values = list(range(1, n+1))  # Conjunto discreto 1..n

    # Cria variáveis de decisão para a matriz do quebra-cabeça
    decision_variables, constraints = make_discrete_decision_variables((n, n),␣
 ↪allowed_values)

    # Garante que cada valor apareça uma vez em cada linha e coluna
    constraints += guarantee_one_of_each_in_full_matrix(decision_variables,␣
 ↪allowed_values)

    # Gera as variáveis finais que representam a solução e suas restrições
    final_solution, final_solution_constraints =␣
 ↪get_final_solution_constraints(decision_variables, allowed_values)
    constraints += final_solution_constraints

    # Aplica as restrições de visualização das torres laterais
    _, tower_constraints = get_side_towers_constraints(final_solution,␣
 ↪side_towers)
    constraints += tower_constraints

    # Define o objetivo da otimização: maximizar a soma das variáveis e da␣
 ↪visualização das torres
    objective = cp.Maximize(0)

    # Cria e resolve o problema de otimização
    problem = cp.Problem(objective, constraints)
    problem.solve(solver=cp.SCIP)

    # Exibe o status da solução e a matriz final encontrada
    print(f"\nStatus da solução: {problem.status}")
    print(f"\nValor do resultado exemplo: \n{final_solution.value}")


# Matriz que define as torres visíveis pelas laterais: [posição, tipo (linha/
 ↪coluna), orientação]
s = np.zeros((n, 2, 2))  # posição, tipo (linha ou coluna), orientação (direita␣
 ↪ou esquerda, baixo ou cima)
```

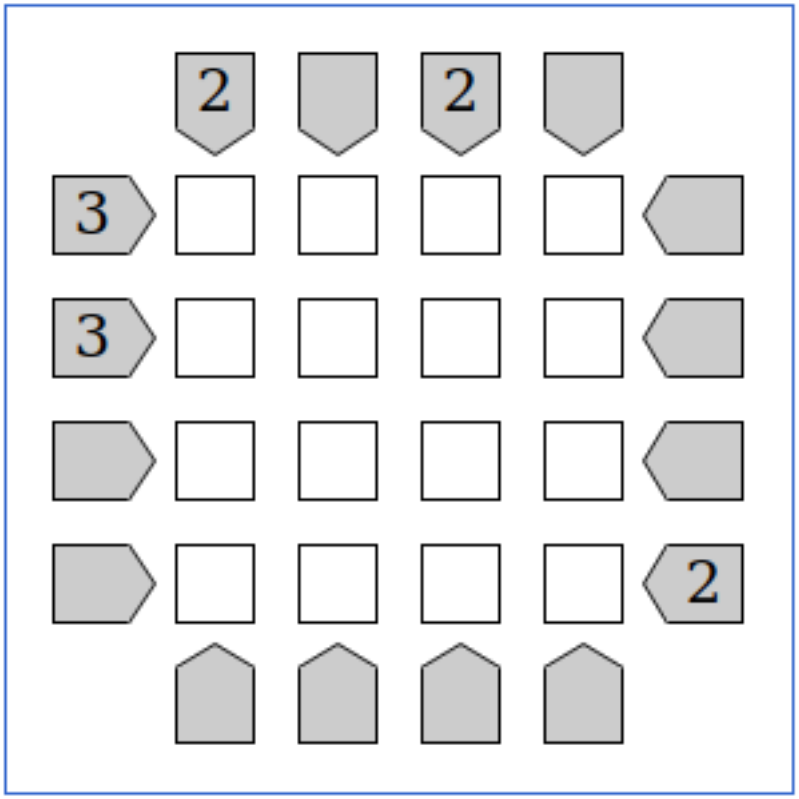
```
## Teste: define algumas pistas nas laterais, exemplo do enunciado
## Linhas
s[1, 0, 0] = 2  # Linha 1, da esquerda para a direita
s[3, 0, 1] = 4  # Linha 3, da direita para a esquerda

## Colunas
s[1, 1, 0] = 1  # Coluna 1, de cima para baixo
s[0, 1, 0] = 3  # Coluna 0, de cima para baixo
s[3, 1, 1] = 2  # Coluna 3, de baixo para cima

# Tamanho da grade
n = 4

solve_towers_problem(n, s)
```

```
Status da solução: optimal

Valor do resultado exemplo:
[[2. 4. 1. 3.]
 [3. 1. 4. 2.]
 [1. 2. 3. 4.]
 [4. 3. 2. 1.]]
```

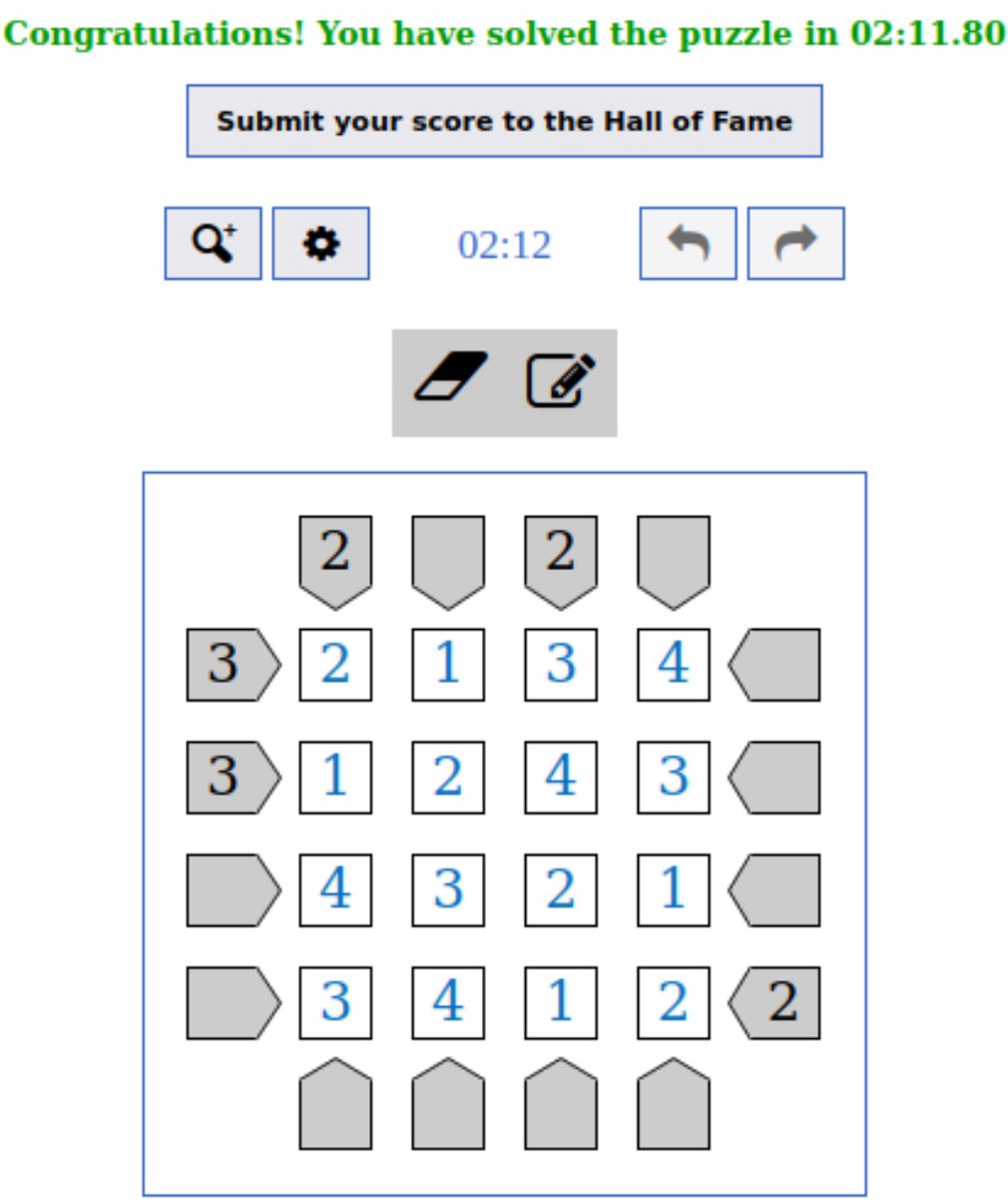
## 1.2 Exemplos da URL

### 1.2.1 Exemplo 1

**Problema:**




**Soluçao:**




4x4 Normal Skyscrapers Puzzle ID: 3,758,873

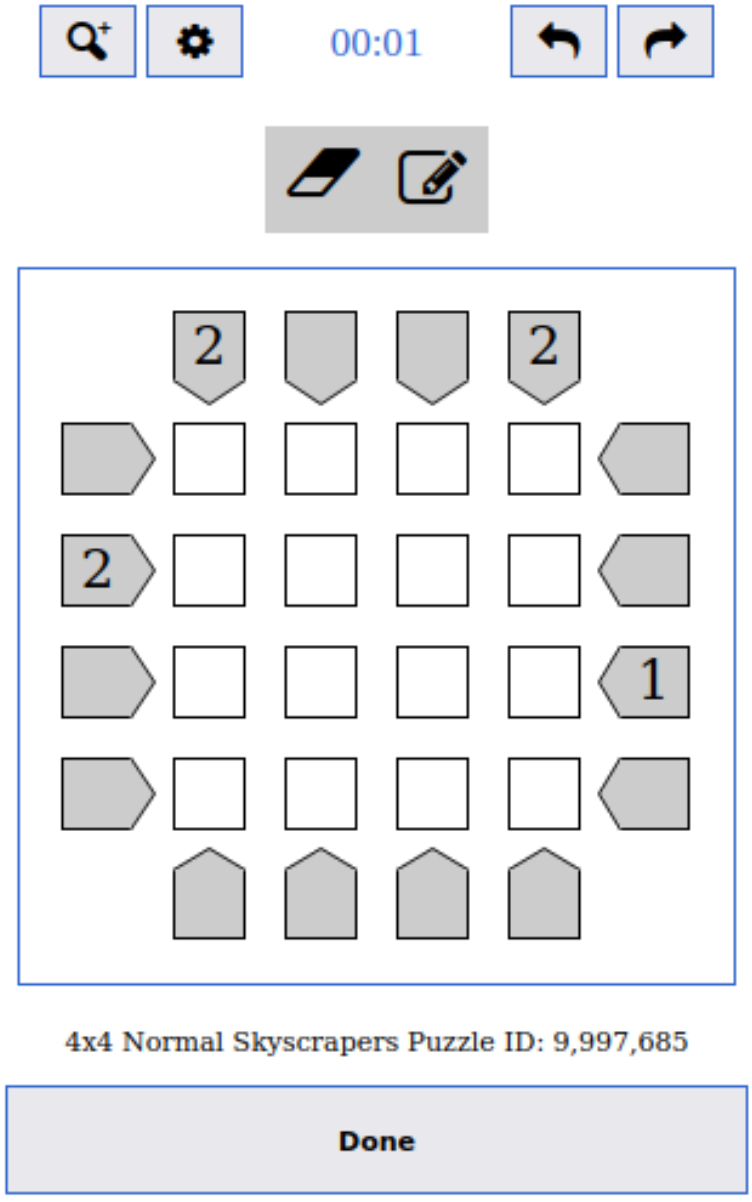
[82]:
```
# Tamanho da grade
n = 4
# Matriz que define as torres visíveis pelas laterais: [posição, tipo (linha/
 ↪coluna), orientação]
exemplo1 = np.zeros((n, 2, 2))  # posição, tipo (linha ou coluna), orientação␣
 ↪(direita ou esquerda, baixo ou cima)

## Teste: define algumas pistas nas laterais, exemplo do enunciado
exemplo1[0, 0, 0] = 3
```

```
exemplo1[1, 0, 0] = 3
exemplo1[0, 1, 0] = 2
exemplo1[2, 1, 0] = 2
exemplo1[3, 0, 1] = 2

solve_towers_problem(n, exemplo1)
```

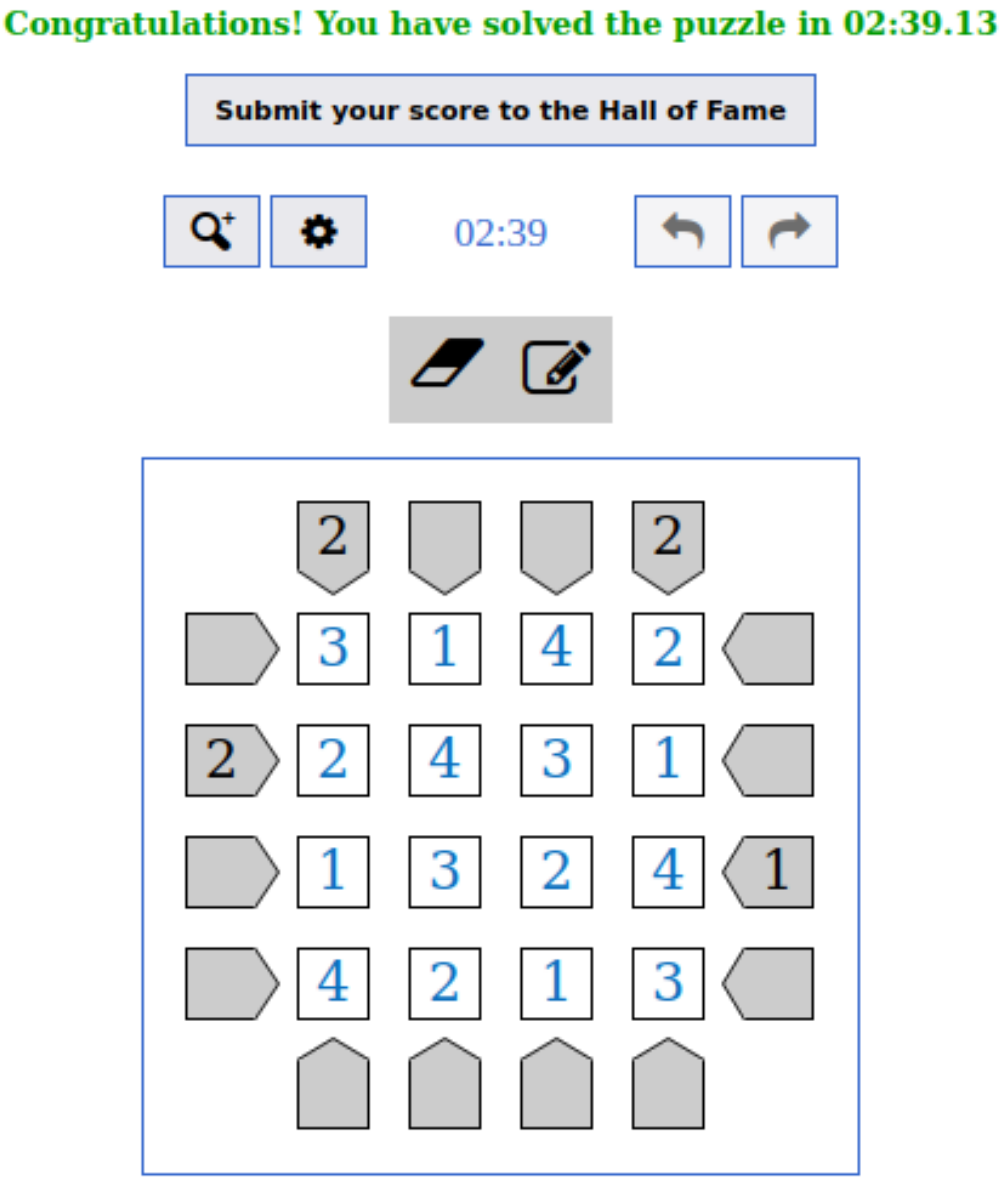
```
Status da solução: optimal

Valor do resultado exemplo:
[[2. 1. 3. 4.]
 [1. 2. 4. 3.]
 [4. 3. 2. 1.]
 [3. 4. 1. 2.]]
```

### 1.2.2 Exemplo 2

**Problema:**




**Soluçao:**



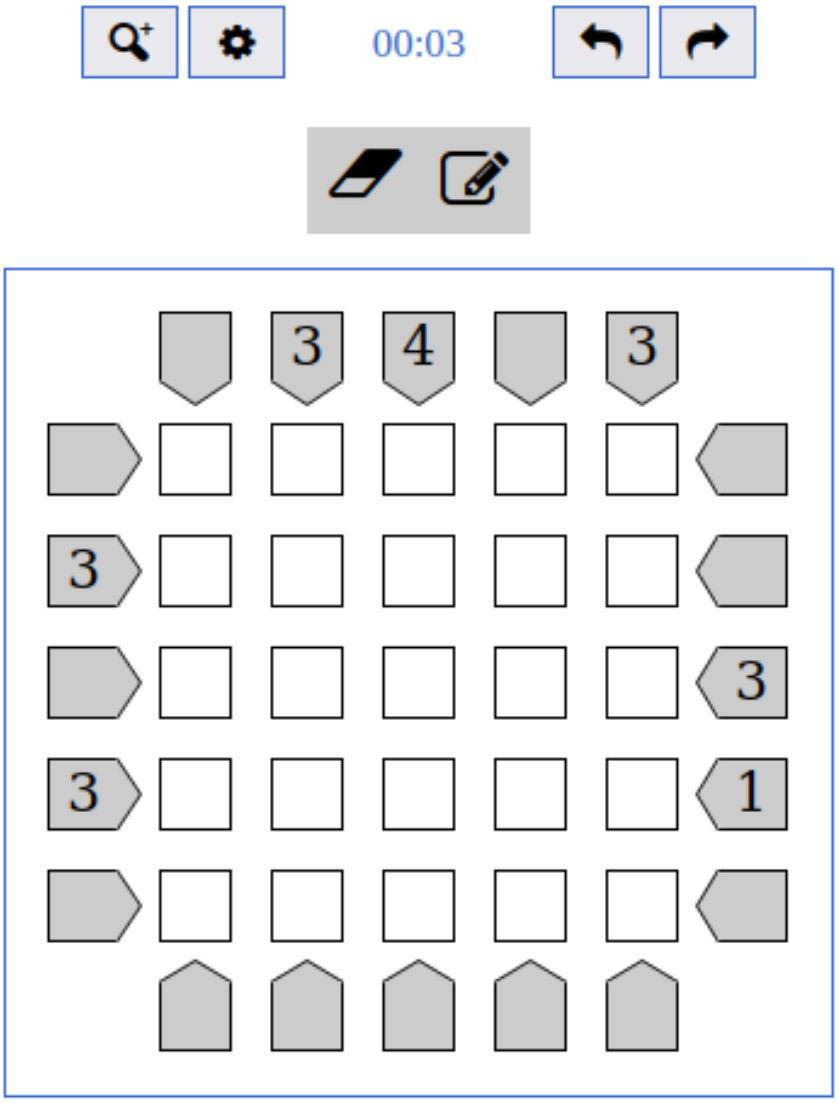

[84]:
```
# Tamanho da grade
n = 4
# Matriz que define as torres visíveis pelas laterais: [posição, tipo (linha/
 ↪coluna), orientação]
exemplo2 = np.zeros((n, 2, 2))  # posição, tipo (linha ou coluna), orientação␣
 ↪(direita ou esquerda, baixo ou cima)

## Teste: define algumas pistas nas laterais, exemplo do enunciado
exemplo2[1, 0, 0] = 2
exemplo2[0, 1, 0] = 2
exemplo2[3, 1, 0] = 2
exemplo2[2, 0, 1] = 1
```

```
solve_towers_problem(n, exemplo2)
```

```
Status da solução: optimal

Valor do resultado exemplo:
[[3. 1. 4. 2.]
 [2. 4. 3. 1.]
 [1. 3. 2. 4.]
 [4. 2. 1. 3.]]
```

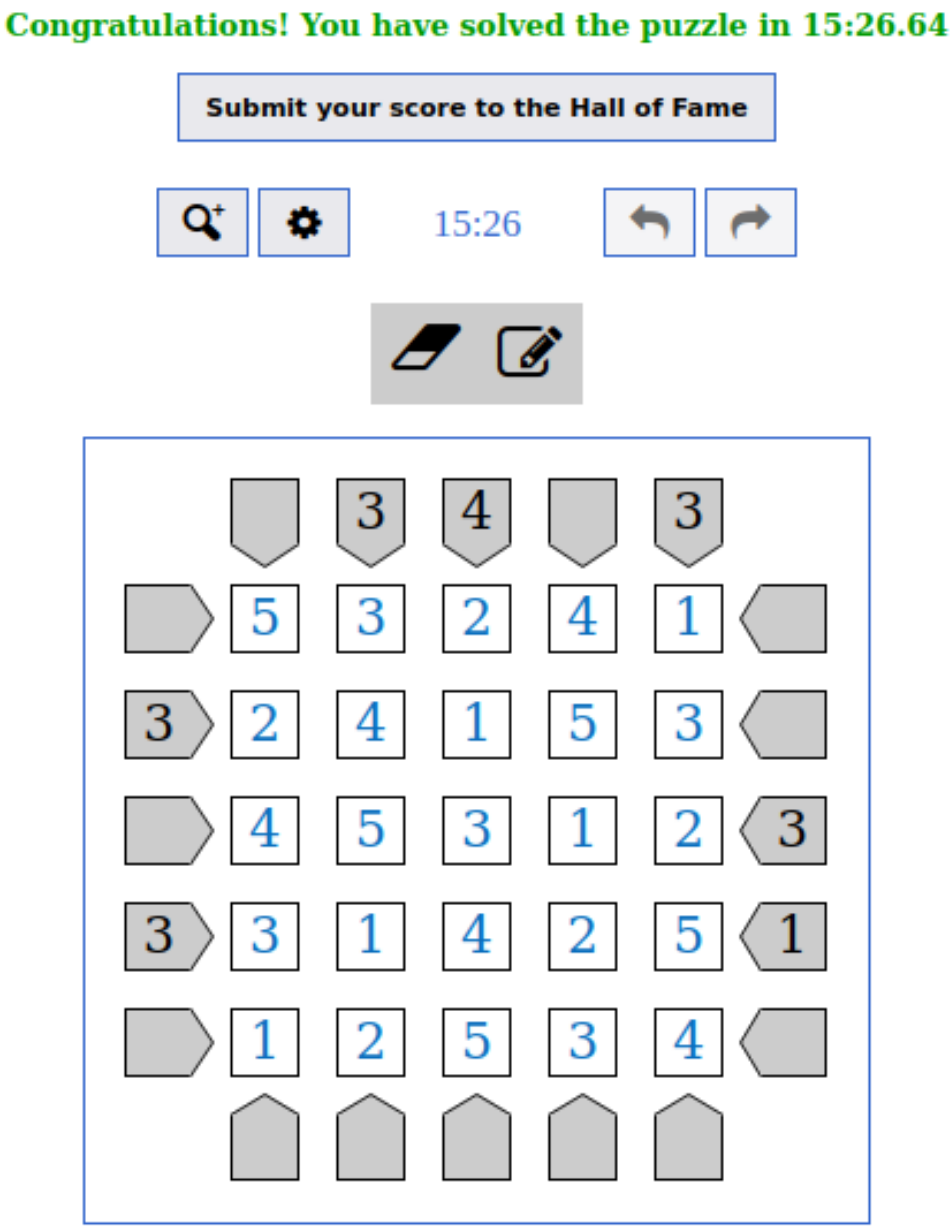
### 1.2.3 Exemplo 3

**Problema:**




5x5 Hard Skyscrapers Puzzle ID: 3,944,585

**Soluçao:**




[80]:
```
# Tamanho da grade
n = 5
# Matriz que define as torres visíveis pelas laterais: [posição, tipo (linha/
 ↪coluna), orientação]
exemplo3 = np.zeros((n, 2, 2))  # posição, tipo (linha ou coluna), orientação␣
 ↪(direita ou esquerda, baixo ou cima)

## Teste: define algumas pistas nas laterais, exemplo do enunciado
exemplo3[1, 0, 0] = 3
exemplo3[3, 0, 0] = 3
```

```
exemplo3[1, 1, 0] = 3
exemplo3[2, 1, 0] = 4
exemplo3[4, 1, 0] = 3
exemplo3[2, 0, 1] = 3
exemplo3[3, 0, 1] = 1

solve_towers_problem(n, exemplo3)
```

```
Status da solução: optimal

Valor do resultado exemplo:
[[5. 3. 2. 4. 1.]
 [2. 4. 1. 5. 3.]
 [4. 5. 3. 1. 2.]
 [3. 1. 4. 2. 5.]
 [1. 2. 5. 3. 4.]]
```

Pedro Henrique Andrade Trindade

## 0.1 Questão 2: (Plantação de tomates!)

### 0.1.1 Item A

**(a) Estratégia gulosa.** Considere a seguinte estratégia gulosa para resolver esse problema: "enquanto houver espaço disponível para plantar um pé de tomate, plante naquele que oferece a maior deliciosidade". Lembre-se de que um espaço só está disponível para plantio se seus vizinhos imediatos não estiverem ocupados. Construa um exemplo em que essa estratégia é ótima e outro em que ela não produz a solução ótima.

**Resposta:**

Um exemplo de vetor T aonde esta estratégia descrita traz a solução ótima é o vetor

$$T = [5, 1, 10, 1, 11]$$

Porque seguindo passo à passo, inicialmente encolhemos o valor 11 na última posição, então partimos para o 10 (como não há nenhum tomate adjacente, podemos plantar lá sem problemas) e depois para o 5 (que também está igualmente espaçado, portanto havendo a possibilidade de plantarmos um tomate na posição respectiva).
Neste processo a deliciosidade total seria D = 11 + 10 + 5 = 26. Que é a deliciosidade máxima possível a ser alcançada (podemos checar por exaustão, neste pequeno exemplo).

Um caso aonde esta estratégia não nos dá a solução ótima é quando temos, por exemplo,

$$T = [6, 10, 8, 1].$$

Neste caso, temos que escolhemos plantar na posição 10, então nossa única alternativa é plantar na posição de valor 1, assim temos a deliciosidade total como D = 10 + 1 = 11. Enquanto pode-se notar que plantar primeiro na posição de valor 8, então plantar na primeira posição de valor 6, encontramos uma deliciosidade total de D = 8 + 6 = 14, maior que a solução gulosa.

### 0.1.2 Item B

**(b) Programação dinâmica.** Construa um algoritmo recursivo que resolva esse problema, isto é, forneça a deliciosidade ótima total e o padrão ótimo de plantio para um dado vetor de deliciosidades T . Explique como a subestrutura ótima é explorada pelo seu algoritmo. A solução pode ser resolvida de forma iterativa e ordenada para melhorar a eficiência? Justifique.

**Resposta:**

O algoritmo planejado para resolver o problema explora a subestrutura ótima através de uma relação de recorrência que considera duas opções em cada posição $i$: plantar (adicionando $T_i$ à

solução ótima até $i-2$) ou não plantar (herdando a solução ótima até $i-1$), maximizando a deliciosidade total.

$$\mathrm{s}[i] = \begin{cases} T_0 & \text{se } i = 0, \\ \max(T_0, T_1) & \text{se } i = 1, \\ \max(T_i + \mathrm{s}[i-2], \mathrm{s}[i-1]) & \text{se } i \geq 2. \end{cases}$$

A reconstrução do padrão de plantio ótimo é feita percorrendo $s$ de trás para frente, identificando as posições onde $s[i] \neq s[i-1]$ (indicando plantio em $i$) e pulando posições adjacentes.

O algoritmo roda em complexidade $\mathcal{O}(2^n)$ pois cada chamada ao problema adiciona mais duas novas chamadas em sequência, assim a árvore de decisão cresce exponencialmente com cada chamada.

Existe a posibilidade de fazer uma versão iterativa desse algoritmo, mais eficiente, preenchendo uma tabela "$s$" de maneira *bottom-up*, começão do inicio $s_0$ que seria o caso base para um array com 1 espaço para plantio (com deliciosidade T0) assim como $s_1$, e então construímos $s_2, \ldots, s_n$ pois cada $s_i$ pois eles depende apenas da solução do problema para dois indices anteriores, este algoritmo possuiria complexidade $\mathcal{O}(n)$, sendo mais eficiente que o algoritmo recursivo ingenuo.

Um detalhe importante de mencionar, é que se utilizarmos as tecnicas de memoização e salvarmos resultados para não repetirmos operações em chamadas repetidas, nosso algoritmo também se torna $\mathcal{O}(n)$ (que foi como eu implementei).

[27]:
```python
def max_deliciosidade(T, i, memo):
    # Caso base
    if i < 0:
        return 0

    # Verifica se já calculamos este subproblema
    if memo[i] != -1:
        return memo[i]

    # Opção 1: Não plantar na posição i
    opcao_nao_plantar = max_deliciosidade(T, i-1, memo)

    # Opção 2: Plantar na posição i (só possível se i-1 não foi plantado)
    opcao_plantar = T[i] + max_deliciosidade(T, i-2, memo)

    # Escolhe a melhor opção
    memo[i] = max(opcao_nao_plantar, opcao_plantar)
    return memo[i]

def plantar_tomates_dp(T):
    n = len(T)
    memo = [-1] * n  # Tabela de memoização
    deliciosidade_total = max_deliciosidade(T, n-1, memo)

    # Reconstruir a solução (padrão de plantio)
    plantio = []
```

```
    i = n - 1
    while i >= 0:
        if i == 0:
            if memo[i] == T[i]:  # Se o valor veio de plantar aqui
                plantio.append(i)
            break
        if memo[i] == memo[i-1]:  # Não plantou em i
            i -= 1
        else:  # Plantou em i
            plantio.append(i)
            i -= 2  # Pula o vizinho

    plantio.reverse()  # Para manter a ordem original
    return deliciosidade_total, plantio

# Exemplo do enunciado
T = [21, 4, 6, 20, 2, 5]

deliciosidade_total, plantio = plantar_tomates_dp(T)

print(f"Deliciosidade total: {deliciosidade_total}")
print(f"Indices do plantio : {plantio}")
print(f"Deliciosidades dos tomates plantados: {[T[i] for i in plantio]}")
```

```
Deliciosidade total: 46
Indices do plantio : [0, 3, 5]
Deliciosidades dos tomates plantados: [21, 20, 5]
```

**(c) Programação linear inteira. Modele esse problema como um problema de otimização linear inteira (PLI). Detalhe a escolha das variáveis de decisão e das restrições.**

**Resposta:**

Nosso problema do plantio será resolvido utilizando a seguinte modelagem com PLI

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} T_i x_i \\ \text{sujeito a} \quad & x_i + x_{i+1} \leq 1, \quad i = 1, \ldots, n-1 \\ & x_i \in \mathbb{B}, \quad i = 1, \ldots, n \end{aligned}$$

Temos nesta formulação que as variáveis de decisão $x_i$ no problema de plantio são variáveis binárias que representam se um pé de tomate é plantado ($x_i = 1$) ou não ($x_i = 0$) na posição $i$. Já o vetor $Ti$, como descrito no enunciado, contém os valores de deliciosidade associados a cada posição possível de plantio. A função objetivo $\sum_{i=1}^{n} T_i x_i$ busca maximizar a deliciosidade total dos pés plantados, somando apenas as deliciosidades das posições onde efetivamente se planta.

As restrições de vizinhança $x_i + x_{i+1} \leq 1$ garantem que não haja plantio em posições adjacentes, se uma posição $i$ for escolhida, ou seja, $x_i = 1$, a sua vizinha é forçada a não ser escolhida por conta

da desigualdade ser com variaveis binárias. Isso assegura que cada pé plantado tenha espaço livre ao redor.

[29]:
```
import cvxpy as cp
import numpy as np

def plantio_tomate_pli(deliciosidade):
    n = len(deliciosidade)

    # Variável de decisão binária
    x = cp.Variable(n, boolean=True)

    # Função objetivo - maximizar a deliciosidade total
    objetivo = cp.Maximize(deliciosidade @ x)

    # Restrições - não pode plantar em posições adjacentes
    restricoes = []
    for i in range(n-1):
        restricoes.append(x[i] + x[i+1] <= 1)

    # Formular e resolver o problema
    prob = cp.Problem(objetivo, restricoes)
    prob.solve()

    # Retornar a deliciosidade total e o padrão de plantio
    deliciosidade_total = prob.value
    padrao_plantio = np.round(x.value).astype(int)

    return deliciosidade_total, padrao_plantio

# exemplo do enunciado
T = np.array([21, 4, 6, 20, 2, 5])
total, padrao_plantio = plantio_tomate_pli(T)

print(f"Deliciosidade total máxima: {int(total)}")
print(f"Padrão de plantio ótimo: {padrao_plantio}")
print(f"Deliciosidades dos tomates plantados: {[T[i].item() for i, result in␣
 ↪enumerate(padrao_plantio) if result == 1]}")
```

```
Deliciosidade total máxima: 46
Padrão de plantio ótimo: [1 0 0 1 0 1]
Deliciosidades dos tomates plantados: [21, 20, 5]
```

### (d) Valide as duas abordagens ótimas desenvolvidas para os vetores

$T1 = [5, 12, 10, 7, 15, 10, 11, 5, 8, 10]$ e $T2 = [10, 12, 5, 12, 20, 18, 5, 3, 2, 8]$.

[28]:
```
T1 = np.array([5, 12, 10, 7, 15, 10, 11, 5, 8, 10])
total_dp, plantio_dp = plantar_tomates_dp(T1)
```

```
total_pli, plantio_pli = plantio_tomate_pli(T1)

print("\nEXEMPLO T1!!!!!")

print("\n CASO COM PROGRAMACAO DINAMICA T1")
print(f"\tDeliciosidade total: {total_dp}")
print(f"\tIndices do plantio : {plantio_dp}")
print(f"\tDeliciosidades dos tomates plantados: {[T1[i].item() for i in␣
 ↪plantio_dp]}")

print("\n CASO COM PLI T1")
print(f"\tDeliciosidade total máxima: {total_pli}")
print(f"\tPadrão de plantio ótimo: {plantio_pli}")
print(f"\tDeliciosidades dos tomates plantados: {[T1[i].item() for i, result in␣
 ↪enumerate(plantio_pli) if result == 1]}")

T2 = np.array([10, 12, 5, 12, 20, 18, 5, 3, 2, 8])
total_dp, plantio_dp = plantar_tomates_dp(T2)
total_pli, plantio_pli = plantio_tomate_pli(T2)

print("\nEXEMPLO T2!!!!!")
print("\n CASO COM PROGRAMACAO DINAMICA T2")
print(f"\tDeliciosidade total: {total_dp}")
print(f"\tIndices do plantio : {plantio_dp}")
print(f"\tDeliciosidades dos tomates plantados: {[T2[i].item() for i in␣
 ↪plantio_dp]}")

print("\n CASO COM PLI T2")
print(f"\tDeliciosidade total máxima: {total_pli}")
print(f"\tPadrão de plantio ótimo: {plantio_pli}")
print(f"\tDeliciosidades dos tomates plantados: {[T2[i].item() for i, result in␣
 ↪enumerate(plantio_pli) if result == 1]}")
```

```
EXEMPLO T1!!!!!

 CASO COM PROGRAMACAO DINAMICA T1
        Deliciosidade total: 51
        Indices do plantio : [0, 2, 4, 6, 9]
        Deliciosidades dos tomates plantados: [5, 10, 15, 11, 10]

 CASO COM PLI T1
        Deliciosidade total máxima: 51.0
        Padrão de plantio ótimo: [1 0 1 0 1 0 1 0 0 1]
        Deliciosidades dos tomates plantados: [5, 10, 15, 11, 10]

EXEMPLO T2!!!!!
```

```
CASO COM PROGRAMACAO DINAMICA T2
       Deliciosidade total: 53
       Indices do plantio : [1, 3, 5, 7, 9]
       Deliciosidades dos tomates plantados: [12, 12, 18, 3, 8]

CASO COM PLI T2
       Deliciosidade total máxima: 53.0
       Padrão de plantio ótimo: [0 1 0 1 0 1 0 1 0 1]
       Deliciosidades dos tomates plantados: [12, 12, 18, 3, 8]
```

Pedro Henrique Andrade Trindade

# 1 Princípio da Casa dos Pombos via Otimização Inteira

**(a) Formule (P) como um problema de otimização linear inteira com dois tipos de restrições:**

1. (r1) aquelas que expressam a condição de que cada pombo deve ser alocado a uma casa;
2. (r2) aquelas que expressam a condição de que cada par de pombos deve estar alocado em casas diferentes. Para tanto, use as variáveis binárias:

$$x_{ij} = \begin{cases} 1, & \text{se o pombo i está alocado à casa j,} \\ 0, & c.c. \end{cases}$$

**Resposta:** Defini o primeiro tipo de restrição **(r1)** como uma igualdade, assim garantindo que cada pombo fique em exatamente uma casa:

$$\sum_{j=1}^{n} x_{ij} = 1 \quad \forall i = 1, \dots, n+1$$

Já para o segundo tipo de restrição **(r2)** expressei-a como uma desigualdade, garantindo que em uma mesma casa não haja dois pombos:

$$x_{ij} + x_{kj} \leq 1 \quad \forall j = 1, \dots, n \text{ e } \forall i \neq k$$

**Formulação completa do problema de otimização**

(ps: a função objetivo sera uma constante $C_0$ dummy)

$$\begin{aligned} \text{minimize} \quad & C_0 \\ \text{sujeito a} \quad & \sum_{j=1}^{n} x_{ij} = 1, \quad \forall i \in \{1, \dots, n+1\} \\ & x_{ij} + x_{kj} \leq 1, \quad \forall j \in \{1, \dots, n\} \text{ e } \forall i \neq k \\ & x_{ij} \in \mathbb{B}, \end{aligned}$$

### 1.0.1 Item (b)

**Desigualdade Tripla** Podemos mostrar que a desigualdade tripla é válida da seguinte forma:

pegue três valores $i$,$k$ e $\ell$, note que para cada dupla vale as desigualdades de **(r2)**:

$$\begin{cases} x_{ij} + x_{kj} \leq 1, \\ x_{ij} + x_{\ell j} \leq 1, \\ x_{kj} + x_{\ell j} \leq 1. \end{cases}$$

Agora, somando as três desigualdades obtemos a desigualdade

$$2(x_{ij} + x_{kj} + x_{\ell j}) \leq 3$$

que podemos manipular levando a

$$x_{ij} + x_{kj} + x_{\ell j} \leq \frac{3}{2} = 1.5$$

note que, como estamos utilizando variáveis binárias (subconjunto dos inteiros), temos que a soma a esquerda será um resultado inteiro positivo, então sabendo que o valor inteiro position mais próximo de 1.5 é 1, podemos substituí-lo na inequação sem perda de generalidade.

Assim $x_{ij} + x_{kj} + x_{\ell j} \leq 1$ é uma desigualdade válida, como queríamos demonstrar.

**Desigualdade tripla => Desigualdade dupla dos pares**

Já para mostrar que a partir da desigualdade tripla podemos obter as outras desigualdades em pares fazemos o seguinte: Por contradição, assuma que a desigualdade tripla seja verdadeira mas para alguma dupla $(i, k)$ a desigualdade dupla não seja válida, isto é

$$\begin{cases} x_{ij} + x_{kj} + x_{\ell j} \leq 1, \\ x_{ij} + x_{kj} > 1, \end{cases}$$

Pela segunda inequação (e pelo fato que $x_{ij}$ e $x_{kj}$) são variáveis binárias, isso acaba forçando que ambos os valores sejam 1, isto é $(x_{ij} = 1)$ e $(x_{kj} = 1)$, o que significa que a nossa primeira desigualdade se torna $x_{\ell j} \leq -1$, o que é uma contradição pois $x_{\ell j} \in 0, 1$. Portante provamos por contradição que se a desigualdade tripla for verdadeira a dupla de cada par também precisa ser.

**Generalizando a solução**

Nós conseguimos utilizar o mesmo truque que utilizamos para provar a inequação tripla para provar o caso geral, podemos fazer isso por indução.

Seja a base de indução o caso mostrado anteriormente, onde adesigualdade dupla implica na tripla.

passo de indução: seja a desigualdade $k$-ésima válida para quaisquer $k$+1 valores em $\{1, 2, \ldots, n{+}1\}$, (sem perda de generalidade, suponha 1 até k+1) onde $(k + 1) < n + 1$, podemos mostrar que conseguimos chegar na desigualdade $(k + 1)$-ésima.

Para isto pegue $k$ desigualdades na seguinte ordem

$$\begin{cases} x_{1j} + x_{1j} \cdots + x_{kj} \leq 1, \\ x_{2j} + x_{3j} \cdots + x_{(k+1)j} \leq 1, \\ \vdots \\ x_{(k+1)j} + x_{1j} \cdots + x_{2j} \leq 1 \end{cases}$$

Agora some todas essas inequações, o resultado é

$$k\sum_{i=1}^{k} x_{ij} \leq k+1 \Leftrightarrow \sum_{i=1}^{k} x_{ij} \leq 1+\frac{1}{k}$$

Usando a mesma retórica do caso anterior, como $\frac{1}{k}$ não é um valor inteiro, podemos arrendondar a inequação para o inteiro mais próximo, no caso 1, resultando em

$$\sum_{i=1}^{n+1} x_{ij} \leq 1$$

Assim terminando a prova por indução e concluindo o resultado que queríamos demonstrar.

**(c) Organize as variáveis de decisão em uma matriz $X = (x_{ij})$, com $(n+1)$ linhas e $n$ colunas. Interprete as restrições (r1) e as restrições válidas construídas ao final do exercício anterior como condições sobre os elementos da matriz X. Conclua que esse conjunto de restrições é incompatível no caso inteiro.**

**Resposta**:

Como descrito no enunciado, temos a matriz

$$X = \begin{pmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{(n+1)1} & \cdots & x_{(n+1)n} \end{pmatrix}$$

Temos os dois tipos de restrição descritos no enunciado anterior, que são 1. Linhas: $\sum_{j=1}^{n} x_{ij} = 1$ (cada pombo em uma casa) 2. Colunas: $\sum_{i=1}^{n+1} x_{ij} \leq 1$ (no máximo um pombo por casa)

Podemos, com estas duas informações, chegar uma prova por absurdo do principio da casa de pombos, pois se somarmos as linhas e colunas em ordens diferentes da matriz chegamos em resultados que são incompatíveis, isto é

$$\begin{aligned} \text{Soma total por linhas} &= \sum_{i=1}^{n+1}\sum_{j=1}^{n} x_{ij} = n+1 \\ \text{Soma total por colunas} &\leq \sum_{j=1}^{n} 1 = n \\ &\Rightarrow n+1 \leq n \quad \text{(contradição)} \end{aligned}$$

Assim concluímos por absurdo que não existe solução inteira, provando o princípio.

Pedro Henrique Andrade Trindade

# 1 Questão 5: (Pudim, o pinguim comilão.)

**(a) Estabeleça uma condição básica para que esse problema admita solução, considerando o vetor F e a distância máxima m que Pudim consegue caminhar sem se alimentar. Essa hipótese será assumida nos itens a seguir.**

**Resposta:**

Para que o problema admita solução, a distância entre quaisquer dois buracos de pesca consecutivos no vetor $F$ não pode exceder $m$. Formalmente:

$$\forall i \in \{1, \dots, n-1\}, \quad F[i+1] - F[i] \leq m.$$

Pois se existir um par de buracos consecutivos $(F[i], F[i+1])$ em que a distância $F[i+1]-F[i] > m$, Pudim não consegue ir de um ao outro sem passar fome, tornando o problema insolúvel. Esta condição garante que pelo menos uma solução exista, que é parar em todos os buracos.

**(b) Modele uma instância genérica desse problema como um problema de otimização linear inteira. Suponha que sejam dados o vetor F e a distância máxima entre paradas m. Descreva as variáveis de decisão e as restrições adotadas.**

**Resposta:**

As variáveis de decisão para este problema serão definidas de forma que para cada buraco de pesca $i \in \{1, \dots, n\}$, teremos $x_i \in \mathbb{B}$, onde:

$$x_i = \begin{cases} 1, & \text{se Pudim para no buraco } i, \\ 0, & \text{caso contrário (c.c.)} \end{cases}$$

Nossa função objetivo será definida de maneira a minimar o número total de paradas, não contabilizando a primeira e a última parada, já que sao fixas. Isto é:

$$\text{minimize} \sum_{i=2}^{n-1} x_i.$$

Para as restrições forçaremos por definição que Pudim pare no começo e no final, isto é, $x_1 = 1$ e $x_n = 1$, assim como faremos que a distância máxima entre paradas consecutivas sejam limitadas a $m$, para garantir que pudim não morra de fome. Esta segunda restrição é um pouco menos trivial, mas conseguimos descrevê-la de maneira linear, para isto seja $B_{>i}(m) = \{k, \text{ tal que } F[k] - F[i] \leq$

$m,\ \forall k > i\}$, isto é, o conjunto de todos os pontos que estão a uma distância menor ou igual a $m$ do ponto $i$ e que estão a frente de $i$. seja $K_i = \max\{B_{>i}(m)\}$.

Podemos então definir as restrição lineares como:

$$\sum_{k=i+1}^{K_i} x_k \geq x_i, \quad \forall i \in \{1, 2, \dots, n-1\}$$

Isto se traduz em português para: se $x_i$ for 1 (fizemos uma parada) então deve haver pelo menos algum $x_k$ maior que 1 (significando outra parada) a uma distância menor que $m$, se $x_i = 0$ note que essa inequação é satisfeita trivialmente.

**Formulação Completa:**

$$\begin{aligned}
\text{minimize} \quad & \sum_{i=2}^{n-2} x_i \\
\text{sujeito a} \quad & \sum_{k=i+1}^{K_i} x_k \geq x_i, \quad \forall i \in \{1, 2, \dots, n-1\} \\
& x_1 = 1, \\
& x_n = 1, \\
& x_i \in \mathbb{B}, \quad i \in \{1, \dots, n\}
\end{aligned}$$

[19]:
```
import cvxpy as cp
import numpy as np

def paradas_pudim_pli(F, m):
    n = len(F)
    x = cp.Variable(n, boolean=True)  # Variáveis binárias (1 = parada, 0 = não parada)

    # Função objetivo: minimizar o número total de paradas
    objective = cp.Minimize(cp.sum(x))

    # Restrições
    constraints = [
        x[0] == 1,   # Obrigatório parar no início
        x[-1] == 1,   # Obrigatório parar no fim
    ]

    # Restrição de distância máxima entre paradas consecutivas
    for i in range(n):
        ball_m = []
        for j in range(i + 1, n):
            if F[j] - F[i] <= m:
                ball_m.append(j)
        if not(ball_m == []):
            constraints.append(cp.sum(x[i+1:np.max(ball_m)+1]) >= x[i])
```

```
    # Resolver o problema
    prob = cp.Problem(objective, constraints)
    prob.solve(solver=cp.SCIP)  # Solver para problemas inteiros

    if prob.status != cp.OPTIMAL:
        return "Não há solução válida"

    # Extrair as paradas selecionadas
    paradas = [F[i] for i in range(n) if np.isclose(x[i].value, 1.0)]
    return paradas

# Exemplo do enunciado
F = [0, 3, 4, 6, 10, 12]
m = 4
print("Paradas para o Pudim com PLI:")
print(paradas_pudim_pli(F, m))
```

```
Paradas para o Pudim com PLI:
[0, 3, 6, 10, 12]
```

**(c) Pudim, além de comilão, é especialista em algoritmos gulosos. A estratégia adotada por Pudim é a seguinte: Pudim irá aguentar o máximo que puder, e só vai parar para pescar se perceber que não conseguirá chegar até o próximo buraco de pesca. Mostre que essa escolha é ótima. Dica: prove por indução.**

**Resposta:**

Vamos utilizar a estratégia gulosa definida em que o Pudim para no buraco mais distante possível dentro do limite $m$, ou seja, a cada parada $i$, escolhemos o maior $j > i$ tal que $F[j] - F[i] <= m$.

Vamos provar que qualquer escolha inicial diferente da gulosa é sub-ótima ou equiparavel a estratégia gulosa. Sejam $g_1, g_2, \ldots, g_n$ as distância dos buracos em que o algoritmo guloso faz parada e $p_1, p_2, \ldots$ para a distância dos buracos em que um algoritmo ótimo faça parada.

Primeiro, escolheremos a base da indução, isso seria tomar $n = 2$, nesse caso o algoritmo guloso é trivialmente ótimo, pois só é possível fazer uma parada, assim como isso é verdade no menor caso em que pode-se haver pelo menos uma parada $n = 3$. (eu fiquei em dúvida se nesse caso eu usava a base ignorando as bordas ou não, por isso coloquei esse segundo pra garantir).

Agora, para o passo indutivo, fazemos o seguinte, suponha que o algoritmo guloso pare em $k$ buracos (não contando o primeiro e o último) e que seja a estratégia ótima, queremos provar que qualquer outra estratégia ótima se equipara para o caso em que o algoritmo guloso faça $k + 1$ paradas.

Note que, pela natureza do problema, temos que $p_i$ tem que estar mais distante do buraco final do que $g_i$, para qualquer $i \leq k$, em especial vale para $p_k$ e $g_k$, o que significa que $p_{k+1}$ atingiria no máximo $g_{k+1}$ começando de $p_k$, então temos que $p_{k+1} \leq g_{k+1}$, portanto, $g_{k+1}$ está mais perto do final, a partir de $p_{k+1}$ teremos que fazer ou mais paradas ou mesmo tanto de paradas do que partindo de $g_{k+1}$, que chega ao final na próxima iteração, provando o resultado esperado.

[20]:
```python
def paradas_pudim_dp(F, m):
    paradas = [F[0]]  # Sempre começa no primeiro ponto
    ultima_parada = 0  # Índice da última parada

    # Enquanto não chegarmos ao final
    while ultima_parada < len(F) - 1:
        # Encontra o buraco mais distante possível dentro do limite m
        prox_parada = ultima_parada
        while (prox_parada + 1 < len(F) and
               F[prox_parada + 1] - F[ultima_parada] <= m):
            prox_parada += 1

        # Se não avançamos, não há solução válida
        if prox_parada == ultima_parada:
            return "Não há solução válida - distância entre buracos muito␣
 ↪grande"

        # Adiciona a parada encontrada
        paradas.append(F[prox_parada])
        ultima_parada = prox_parada

    return paradas

# Exemplo do enunciado
F = [0, 3, 4, 6, 10, 12]
m = 4

print("Paradas para o pudim com programação dinâmica: ")
print(paradas_pudim_dp(F, m))
```

```
Paradas para o pudim com programação dinâmica:
[0, 4, 6, 10, 12]
```