# Resolução Exercícios 🎈

**Tema:** Estruturas Básicas da STL (Vector, Pair, Stack, Queue)
**Equipe:** UberFofas

**A) Sereja and Dima (**https://codeforces.com/problemset/problem/381/A**)**

Código em Python

main.py

```python
# Lê o número de cartas disponíveis no jogo.
numero_cartas = int(input())

# Lê os valores das cartas e os converte para uma lista de inteiros.
cartas = list(map(int, input().split()))

# Inicializa os pontos dos jogadores como 0.
pontos_sereja = 0
pontos_dima = 0

# Define as posições iniciais das extremidades da linha de cartas.
esquerda = 0               # Índice da carta mais à esquerda.
direita = numero_cartas - 1 # Índice da carta mais à direita.

# Variável que indica se é a vez do Sereja. Começa como True, pois ele joga primeiro.
vez_sereja = True

# Este laço continua enquanto ainda houver cartas disponíveis na linha.
while esquerda <= direita:
    # Escolhe a carta com o maior valor entre as extremidades (esquerda e direita).
    if cartas[esquerda] > cartas[direita]:
        carta_escolhida = cartas[esquerda]
        esquerda += 1  # Move o índice da esquerda para a próxima carta.
    else:
        carta_escolhida = cartas[direita]
        direita -= 1  # Move o índice da direita para a próxima carta.

    # Verifica de quem é a vez e atribui os pontos da carta escolhida a ele.
    if vez_sereja:
        pontos_sereja += carta_escolhida  # Pontos acumulados para Sereja.
    else:
        pontos_dima += carta_escolhida    # Pontos acumulados para Dima.

    # Troca a vez para o próximo jogador.
    vez_sereja = not vez_sereja

# Imprime o resultado final: pontos do Sereja e pontos do Dima.
print(pontos_sereja, pontos_dima)

```

Pseudocódigo

```
// Ler o número de cartas
LER numero_cartas
// Ler os valores das cartas e armazenar em uma lista
LER lista_cartas
// Inicializar pontuações dos jogadores
pontos_sereja = 0
pontos_dima = 0
// Inicializar índices das cartas nas extremidades
esquerda = 0
direita = numero_cartas - 1
// Definir que o primeiro jogador é Sereja
vez_sereja = VERDADEIRO

// Enquanto houver cartas na linha
ENQUANTO esquerda <= direita FAÇA

    // Escolher a carta de maior valor entre as extremidades
    SE lista_cartas[esquerda] > lista_cartas[direita] ENTÃO
        carta_escolhida = lista_cartas[esquerda]
        esquerda = esquerda + 1    // Mover para a próxima carta da esquerda
    SENÃO
        carta_escolhida = lista_cartas[direita]
        direita = direita - 1      // Mover para a próxima carta da direita
    FIM-SE

    // Atribuir a carta escolhida ao jogador atual
    SE vez_sereja ENTÃO
        pontos_sereja = pontos_sereja + carta_escolhida
    SENÃO
        pontos_dima = pontos_dima + carta_escolhida
    FIM-SE

    // Alternar a vez do jogador
    vez_sereja = NÃO vez_sereja

FIM-ENQUANTO

// Exibir os pontos finais de ambos os jogadores
IMPRIMIR pontos_sereja, pontos_dima
```

**B) Indian Summer (**https://codeforces.com/problemset/problem/44/A**)**

Código em Python

main.py

```python
# Lê um número inteiro 'n' que representa a quantidade de folhas informadas.
n = int(input())

# Lê 'n' folhas (nomes ou identificadores) fornecidas pelo usuário e armazena em uma lista.
folhas = [input().strip() for _ in range(n)]

# Chama a função para contar as folhas únicas e armazena o resultado na variável 'resultado'.
resultado = contar_folhas_unicas(n, folhas)

# Exibe o número de folhas únicas encontradas.
print(resultado)

```

Pseudocódigo

```
INÍCIO

    // Ler o número total de folhas que serão informadas.
    LER n

    // Criar um conjunto vazio para armazenar folhas únicas.
    folhas_unicas = CONJUNTO_VAZIO

    // Ler cada folha e adicioná-la ao conjunto.
    PARA i DE 1 ATÉ n FAÇA
        LER folha
        ADICIONAR folha AO folhas_unicas   // Conjuntos só armazenam elementos únicos.
    FIM-PARA

    // Contar quantos elementos únicos existem no conjunto.
    resultado = TAMANHO_DE(folhas_unicas)

    // Exibir o número total de folhas únicas.
    IMPRIMIR resultado

FIM
```

**C) Regular Bracket Sequence (**https://codeforces.com/problemset/problem/26/B**)**

Código em Python

main.py

```python
# Lê a sequência de parênteses como uma lista de caracteres.
sequence = list(input())

# Inicializa uma pilha vazia que será usada para rastrear parênteses abertos não fechados.
pilha = []

# Variável que armazena o comprimento máximo da sequência regular de parênteses.
answer = 0

# Percorre cada caractere da sequência de parênteses fornecida.
for elm in sequence:
    # Se o caractere for um parêntese de abertura '(':
    if elm == '(':
        pilha.append(elm)  # Adiciona o parêntese na pilha (indicando que está aberto e esperando ser fechado).
        answer += 1        # Conta esse parêntese como parte de uma possível sequência regular.

    # Se o caractere for um parêntese de fechamento ')':
    else:
        # Verifica se há algum parêntese aberto na pilha que possa ser fechado.
        if len(pilha) > 0:
            pilha.pop()    # Remove o último parêntese aberto da pilha, fazendo um par válido.
            answer += 1    # Conta o par fechado como parte da sequência válida.

# Remover parênteses que não foram fechados corretamente.
while len(pilha) > 0:
    pilha.pop()    # Remove cada parêntese de abertura que não teve fechamento correspondente.
    answer -= 1    # Desconta da contagem final, pois não forma um par válido.

# Exibe o comprimento máximo da sequência regular de parênteses encontrada.
print(answer)
```

Pseudocódigo

```
INÍCIO

    // Ler a sequência de parênteses como uma lista de caracteres.
    LER sequence COMO LISTA_DE_CARACTERES

    // Inicializar uma pilha vazia para armazenar parênteses abertos.
    pilha = CONJUNTO_VAZIO

    // Variável para armazenar o comprimento máximo da sequência válida.
    answer = 0

    // Percorrer cada caractere na sequência fornecida.
    PARA cada elm EM sequence FAÇA
        SE elm É '(' ENTÃO
            ADICIONAR elm NA pilha   // Coloca o parêntese de abertura na pilha.
            answer = answer + 1      // Conta como parte da sequência válida.

        SENÃO  // Se elm é ')'
            SE pilha NÃO ESTÁ VAZIA ENTÃO
                REMOVER ÚLTIMO ELEMENTO DE pilha   // Faz um par válido com o último '(' na pilha.
                answer = answer + 1                // Conta como parte da sequência válida.
            FIM-SE
    FIM-PARA

    // Remover parênteses que não foram fechados corretamente.
    ENQUANTO pilha NÃO ESTÁ VAZIA FAÇA
        REMOVER ÚLTIMO ELEMENTO DE pilha   // Descarta parênteses que não foram fechados.
        answer = answer - 1                // Desconta do comprimento da sequência válida.
    FIM-ENQUANTO

    // Exibir o comprimento máximo da sequência regular encontrada.
    IMPRIMIR answer

FIM
```

**D) Games (**https://codeforces.com/problemset/problem/268/A**)**

Código em Python

main.py

```python
def contar_jogos_uniformes_visitantes(n, uniformes):
    contagem = 0  # Inicializa a contagem de partidas onde o time da casa usa o uniforme de visitante.

    # Duplo loop para comparar cada time com todos os outros times.
    for i in range(n):  # Percorre todos os times como anfitriões.
        for j in range(n):  # Percorre todos os times como visitantes.

            # Verifica se os times são diferentes (i != j)
            # e se o uniforme de casa do time i é igual ao uniforme de visitante do time j.
            if i != j and uniformes[i][0] == uniformes[j][1]:
                contagem += 1  # Incrementa a contagem quando a condição é verdadeira.

    return contagem  # Retorna o número total de jogos onde o time da casa usa uniforme de visitante.

# Lê o número total de times.
n = int(input())

# Lê os uniformes de cada time. Cada uniforme é armazenado como uma tupla (home, guest).
uniformes = [tuple(map(int, input().split())) for _ in range(n)]

# Chama a função para calcular o número de jogos onde o time da casa usa uniforme de visitante.
resultado = contar_jogos_uniformes_visitantes(n, uniformes)

# Exibe o resultado.
print(resultado)

```

Pseudocódigo

```
INICIO

    // Ler o número de times.
    LER n

    // Criar uma lista vazia para armazenar os uniformes dos times.
    uniformes = LISTA_VAZIA

    // Ler os uniformes de cada time e armazenar na lista.
    PARA i DE 1 ATÉ n FAÇA
        LER hi, ai  // hi = Cor do uniforme de casa, ai = Cor do uniforme de visitante
        ADICIONAR (hi, ai) À uniformes
    FIM-PARA

    // Inicializar o contador de partidas onde o time da casa usa uniforme de visitante.
    contagem = 0

    // Comparar cada time com todos os outros times.
    PARA i DE 0 ATÉ n - 1 FAÇA       // i é o time que joga em casa.
        PARA j DE 0 ATÉ n - 1 FAÇA   // j é o time visitante.

            SE i ≠ j E uniformes[i][0] = uniformes[j][1] ENTÃO
                contagem = contagem + 1  // Incrementa a contagem quando os uniformes coincidem.
            FIM-SE

        FIM-PARA
    FIM-PARA

    // Exibir o resultado final.
    IMPRIMIR contagem

FIM
```

**E) Queue (**https://codeforces.com/problemset/problem/545/D**)**

Código em Python

main.py

```python
def max_pessoas(n, tempos):
    # Ordena os tempos de atendimento em ordem crescente para minimizar o tempo de espera.
    tempos.sort()

    # Inicializa variáveis para contar o tempo total de espera acumulado (tempo_total)
    # e o número de pessoas que não ficaram desapontadas (pessoas).
    tempo_total = 0
    pessoas = 0

    # Percorre cada tempo de atendimento na lista ordenada.
    for tempo in tempos:
        # Se o tempo total acumulado até agora for menor ou igual ao tempo necessário para atender a pessoa atual,
        # então essa pessoa não ficará desapontada.
        if tempo_total <= tempo:
            pessoas += 1              # Incrementa o contador de pessoas satisfeitas.
            tempo_total += tempo      # Atualiza o tempo total acumulado somando o tempo da pessoa atual.

    # Retorna o número total de pessoas que não ficaram desapontadas.
    return pessoas

# Lê a quantidade de pessoas na fila.
n = int(input())

# Lê os tempos de atendimento de cada pessoa e os converte para uma lista de inteiros.
tempos = list(map(int, input().split()))

# Calcula o resultado chamando a função max_pessoas() e imprime o resultado.
resultado = max_pessoas(n, tempos)
print(resultado)
```

Pseudocódigo

```
INICIO

    // Ler o número total de pessoas na fila.
    LER n

    // Ler os tempos necessários para atender cada pessoa e armazenar na lista 'tempos'.
    LER tempos COMO UMA LISTA DE INTEIROS

    // Ordenar a lista 'tempos' em ordem crescente.
    ORDENAR(tempos)

    // Inicializar variáveis para contar o tempo total acumulado e o número de pessoas satisfeitas.
    tempo_total = 0
    pessoas = 0

    // Percorrer cada tempo de atendimento na lista ordenada.
    PARA cada tempo EM tempos FAÇA

        // Verificar se a pessoa atual não ficará desapontada.
        SE tempo_total <= tempo ENTÃO
            pessoas = pessoas + 1           // Incrementar o número de pessoas satisfeitas.
            tempo_total = tempo_total + tempo  // Atualizar o tempo total acumulado.
        FIM-SE

    FIM-PARA

    // Exibir o número total de pessoas que não ficaram desapontadas.
    IMPRIMIR pessoas

FIM
```

**F) Rank List (**https://codeforces.com/problemset/problem/166/A**)**

Código em Python

main.py

```python
from collections import Counter

# Função que conta o número de times que compartilham a mesma posição k
def contar_times_na_posicao_k(n, k, times):
    # Ordenar os times por problemas resolvidos (decrescente) e depois por tempo de penalidade (crescente)
    times.sort(key=lambda x: (-x[0], x[1]))

    # Contar a frequência de cada desempenho dos times (problemas resolvidos, tempo de penalidade)
    contador_times = Counter(map(tuple, times))

    # Encontrar o desempenho do time que está na posição k (lembrando que a lista é indexada em 0)
    desempenho_k = times[k - 1]

    # Retornar quantos times têm o mesmo desempenho que o time na posição k
    return contador_times[tuple(desempenho_k)]

# Leitura do número de times e a posição k
n, k = map(int, input().split())

# Leitura do desempenho dos times (problemas resolvidos e tempo de penalidade)
times = [list(map(int, input().split())) for _ in range(n)]

# Calcular o resultado usando a função contar_times_na_posicao_k e imprimir o resultado
resultado = contar_times_na_posicao_k(n, k, times)
print(resultado)
```

Pseudocódigo

```
INÍCIO

    // Ler a quantidade de times e a posição desejada k
    LER n, k

    // Criar uma lista vazia para armazenar os desempenhos dos times
    times = LISTA VAZIA

    // Ler o desempenho dos n times
    PARA i DE 1 ATÉ n FAÇA
        LER problemas_resolvidos, tempo_penalidade
        ADICIONAR (problemas_resolvidos, tempo_penalidade) EM times
    FIM-PARA

    // Ordenar a lista times por:
    // 1. Problemas resolvidos em ordem decrescente
    // 2. Tempo de penalidade em ordem crescente
    ORDENAR times POR (-problemas_resolvidos, tempo_penalidade)

    // Contar quantos times têm cada desempenho específico
    contador_times = DICIONÁRIO VAZIO
    PARA cada time EM times FAÇA
        SE time ESTÁ EM contador_times ENTÃO
            INCREMENTAR contador_times[time] EM 1
        SENÃO
            contador_times[time] = 1
    FIM-PARA

    // Identificar o desempenho do time que está na posição k
    desempenho_k = times[k - 1]

    // Mostrar quantos times têm o mesmo desempenho que o time na posição k
    IMPRIMIR contador_times[desempenho_k]

FIM
```

**G) Sereja and Brackets (**https://codeforces.com/problemset/problem/380/C**)**

Código em Python - Parte 1

main.py

```python
# Importa as bibliotecas necessárias
from collections import namedtuple

# Define a estrutura do nó da árvore com as propriedades:
# 'cs' -> Sequências corretas de parênteses
# 'a' -> Parênteses abertos não emparelhados
# 'f' -> Parênteses fechados não emparelhados
No = namedtuple('No', ['cs', 'a', 'f'])


def combinar(esq, dir):
    # Calcula o número de pares válidos entre os nós da esquerda e direita
    emparelhados = min(esq.a, dir.f)

    # Cria um novo nó combinando os resultados
    return No(
        cs=esq.cs + dir.cs + emparelhados,  # Soma as sequências corretas
        a=esq.a + dir.a - emparelhados,      # Parênteses abertos restantes
        f=esq.f + dir.f - emparelhados       # Parênteses fechados restantes
    )


def construir(indice, inicio, fim):
    if inicio == fim:  # Caso base: folha da árvore
        if sequencia[inicio] == '(':  # Parêntese aberto
            arvore[indice] = No(cs=0, a=1, f=0)
        else:  # Parêntese fechado
            arvore[indice] = No(cs=0, a=0, f=1)
        return

    meio = (inicio + fim) // 2

    # Construção recursiva das subárvores esquerda e direita
    construir(2 * indice, inicio, meio)
    construir(2 * indice + 1, meio + 1, fim)

    # Combina os resultados das subárvores
    arvore[indice] = combinar(arvore[2 * indice], arvore[2 * indice + 1])
```

Código em Python - Parte 2

```python


def consultar(indice, inicio, fim, esquerda, direita):
    if esquerda > fim or direita < inicio:  # Fora do intervalo atual
        return No(cs=0, a=0, f=0)

    if esquerda <= inicio and fim <= direita:  # Intervalo completamente dentro
        return arvore[indice]

    meio = (inicio + fim) // 2

    # Consulta recursiva em ambas as metades
    esquerda_no = consultar(2 * indice, inicio, meio, esquerda, direita)
    direita_no = consultar(2 * indice + 1, meio + 1, fim, esquerda, direita)

    # Combina os resultados
    return combinar(esquerda_no, direita_no)


# Lê a sequência de parênteses
sequencia = input().strip()
n = len(sequencia)

# Inicializa a árvore de segmentação
arvore = [None] * (4 * n)

# Constrói a árvore
construir(1, 0, n - 1)

# Lê o número de consultas
m = int(input())

# Processa cada consulta e imprime o resultado
for _ in range(m):
    esquerda, direita = map(int, input().split())
    resultado = consultar(1, 0, n - 1, esquerda - 1, direita - 1)
    print(resultado.cs * 2)  # Multiplica por 2 porque cada sequência correta é um par de parênteses

```

Pseudocódigo - Parte 1

```
// Estrutura de dados para cada nó da árvore
Estrutura Nó:
    cs = 0  // Quantidade de sequências corretas encontradas
    a = 0   // Número de parênteses abertos sem par
    f = 0   // Número de parênteses fechados sem par

// Combina dois nós da árvore para gerar um nó resultante
Função combinar(nó1, nó2):
    temp = mínimo(nó1.a, nó2.f)  // Calcula o número de pares válidos possíveis
    resultado = novo Nó

    resultado.cs = nó1.cs + nó2.cs + temp  // Incrementa a quantidade de sequências corretas
    resultado.a = nó1.a + nó2.a - temp     // Atualiza o número de parênteses abertos restantes
    resultado.f = nó1.f + nó2.f - temp     // Atualiza o número de parênteses fechados restantes

    Retornar resultado

// Constrói a árvore de segmentos recursivamente
Função construir(nó_atual, início, fim):
    Se início == fim:  // Caso base: nó folha da árvore
        Se sequência[início] é '(':
            árvore[nó_atual] = Nó(cs = 0, a = 1, f = 0)
        Senão:  // Caso seja ')'
            árvore[nó_atual] = Nó(cs = 0, a = 0, f = 1)
        Retornar

    meio = (início + fim) / 2  // Calcula o ponto médio para divisão
    construir(nó_atual * 2, início, meio)  // Constrói a metade esquerda
    construir(nó_atual * 2 + 1, meio + 1, fim)  // Constrói a metade direita

    árvore[nó_atual] = combinar(árvore[nó_atual * 2], árvore[nó_atual * 2 + 1])  // Combina os resultados

// Realiza uma consulta na árvore para um intervalo específico
Função consultar(nó_atual, início, fim, esq, dir):
    Se esq == início e dir == fim:  // Caso perfeito: retorna o nó correspondente
        Retornar árvore[nó_atual]

    meio = (início + fim) / 2  // Calcula o ponto médio para divisão
```

Pseudocódigo - parte 2

```
    Se dir <= meio:  // Se a consulta está totalmente na metade esquerda
        Retornar consultar(nó_atual * 2, início, meio, esq, dir)
    Se esq > meio:  // Se a consulta está totalmente na metade direita
        Retornar consultar(nó_atual * 2 + 1, meio + 1, fim, esq, dir)

    // Caso a consulta cubra ambas as metades, combina os resultados
    resultado_esquerdo = consultar(nó_atual * 2, início, meio, esq, meio)
    resultado_direito = consultar(nó_atual * 2 + 1, meio + 1, fim, meio + 1, dir)

    Retornar combinar(resultado_esquerdo, resultado_direito)

// Programa principal para leitura e processamento das consultas
Ler sequência de parênteses  // Ex.: "()(())(())("
Ler número de consultas m  // Ex.: 7

Construir a árvore de segmentos chamando a função:
    construir(1, 0, comprimento da sequência - 1)

Para cada consulta de 1 até m:
    Ler li e ri  // Índices da consulta (baseados em 1)
    resultado = consultar(1, 0, comprimento da sequência - 1, li - 1, ri - 1)
    Imprimir resultado.cs * 2  // Multiplicado por 2 porque cada sequência correta é composta por parênteses abertos e fechados
```

**H) Journey (**https://codeforces.com/problemset/problem/839/C**)**

Código em Python

main.py

```python
import sys
sys.setrecursionlimit(200000)

def dfs(cidade, pai, adjacencia):
    # Função que calcula o valor esperado para uma determinada cidade.
    # Retorna tanto a soma dos valores esperados das cidades filhas quanto o número de cidades filhas.

    soma_esperada = 0  # Armazena a soma dos valores esperados das cidades filhas
    quantidade_filhos = 0  # Conta quantas cidades filhas a cidade atual possui

    # Percorre todos os vizinhos da cidade atual
    for vizinho in adjacencia[cidade]:
        if vizinho != pai:  # Ignora a cidade de onde veio (pai)
            quantidade_filhos += 1  # Incrementa o número de cidades filhas
            soma_esperada += dfs(vizinho, cidade, adjacencia)  # Soma o valor esperado da cidade filha atual

    if quantidade_filhos == 0:
        # Caso base: se a cidade não tem filhos (é um nó folha)
        return 0  # O percurso termina aqui

    # Retorna o valor esperado calculado como:
    # 1 (a cidade atual conta como um passo na jornada) + média dos valores esperados dos filhos
    return 1 + soma_esperada / quantidade_filhos

def principal():
    n = int(input())  # Lê o número de cidades

    # Cria uma lista de adjacência para representar o grafo (árvore)
    adjacencia = [[] for _ in range(n + 1)]

    # Lê as estradas e preenche a lista de adjacência
    for _ in range(n - 1):
        u, v = map(int, input().split())
        adjacencia[u].append(v)
        adjacencia[v].append(u)

    # Inicia a busca em profundidade (DFS) a partir da cidade 1
    resultado = dfs(1, -1, adjacencia)
    print(f"{resultado:.6f}")  # Imprime o resultado formatado com 6 casas decimais

# Chama a função principal para iniciar o programa
principal()
```

Pseudocódigo

```
FUNÇÃO DFS(cidade, pai, adjacencia)
    INICIAR soma_esperada COMO 0  // Armazena a soma dos valores esperados das cidades filhas
    INICIAR quantidade_filhos COMO 0  // Conta o número de cidades filhas da cidade atual

    PARA CADA vizinho EM adjacencia[cidade] FAÇA
        SE vizinho FOR DIFERENTE DE pai ENTÃO  // Ignora a cidade anterior na jornada
            INCREMENTAR quantidade_filhos EM 1
            ADICIONAR O RETORNO DE DFS(vizinho, cidade, adjacencia) À soma_esperada
        FIM SE
    FIM PARA

    SE quantidade_filhos FOR IGUAL A 0 ENTÃO  // Caso a cidade seja uma folha (sem filhos)
        RETORNAR 0  // O percurso termina aqui
    FIM SE

    // Calcula o valor esperado da jornada para a cidade atual
    RETORNAR 1 + (soma_esperada / quantidade_filhos)
FIM FUNÇÃO


FUNÇÃO PRINCIPAL()
    LER n  // Número de cidades

    // Criar a lista de adjacência para representar o grafo (árvore)
    INICIAR adjacencia COMO lista vazia de tamanho n + 1

    PARA i DE 0 ATÉ n - 2 FAÇA  // Lê todas as conexões entre as cidades
        LER u, v  // Par de cidades conectadas por uma estrada
        ADICIONAR v NA LISTA adjacencia[u]
        ADICIONAR u NA LISTA adjacencia[v]
    FIM PARA

    // Inicia a DFS a partir da cidade 1 e calcula o resultado
    resultado <- DFS(1, -1, adjacencia)
    IMPRIMIR resultado FORMATADO COM 6 CASAS DECIMAIS
FIM FUNÇÃO

// Executa a função principal
PRINCIPAL()
```

**H) Restaurant (https://codeforces.com/problemset/problem/597/B)**

Código em Python

main.py

```python
# Importa o módulo sys para acelerar a leitura de entradas
import sys
input = sys.stdin.read

# Função principal
def restaurante():
    # Lê todas as entradas de uma vez e divide por linhas
    dados = input().strip().split('\n')

    # Lê o número de pedidos
    numero_pedidos = int(dados[0])
    # Lista que armazenará os pedidos como tuplas (inicio, fim, indice)
    pedidos = []

    # Lê cada pedido e adiciona na lista de pedidos
    for i in range(1, numero_pedidos + 1):
        li, ri = map(int, dados[i].split())
        pedidos.append(((li, ri), i - 1))

    # Ordena os pedidos pelo horário de término (ri).
    # Em caso de empate, ordena pelo horário de início (li)
    pedidos.sort(key=lambda x: (x[0][1], x[0][0]))

    # Variável que rastreia o horário de término do último pedido aceito
    ultimo_fim = 0
    # Contador para o número máximo de pedidos aceitos
    numero_aceitos = 0

    # Percorre todos os pedidos ordenados
    for pedido in pedidos:
        inicio, fim = pedido[0]

        # Verifica se o pedido atual não se sobrepõe com o último aceito
        if inicio > ultimo_fim:
            ultimo_fim = fim # Atualiza o último horário de término aceito
            numero_aceitos += 1 # Incrementa o número de pedidos aceitos

    print(numero_aceitos)

# Chama a função principal
restaurante()
```

Pseudocódigo

```
INICIO
    FUNÇÃO Restaurante:

        // Lê todas as entradas de uma vez e divide por linhas
        DADOS <- Ler todas as linhas de entrada

        // Lê o número de pedidos
        NUMERO_PEDIDOS <- Converter DADOS[0] para inteiro

        // Lista para armazenar os pedidos como tuplas (INICIO, FIM, INDICE)
        PEDIDOS <- Lista vazia

        // Ler cada pedido e adicionar na lista PEDIDOS
        PARA i DE 1 ATÉ NUMERO_PEDIDOS FAÇA:
            LI, RI <- Converter DADOS[i] para inteiros
            ADICIONAR ((LI, RI), i - 1) na lista PEDIDOS

        // Ordena os pedidos pelo horário de término (RI).
        // Em caso de empate, ordena pelo horário de início (LI)
        ORDENE PEDIDOS por (RI, LI)

        // Variável que rastreia o horário de término do último pedido aceito
        ULTIMO_FIM <- 0

        // Contador para o número máximo de pedidos aceitos
        NUMERO_ACEITOS <- 0

        // Percorre todos os pedidos ordenados
        PARA CADA PEDIDO EM PEDIDOS FAÇA:
            INICIO, FIM <- PEDIDO[0]

            // Verifica se o pedido atual não se sobrepõe com o último aceito
            SE INICIO > ULTIMO_FIM ENTÃO:
                ULTIMO_FIM <- FIM        // Atualiza o último horário de término aceito
                NUMERO_ACEITOS <- NUMERO_ACEITOS + 1  // Incrementa o número de pedidos aceitos

        // Imprime o número máximo de pedidos aceitos
        IMPRIMA NUMERO_ACEITOS

    FIM_FUNÇÃO
FIM
```