Coding dojo

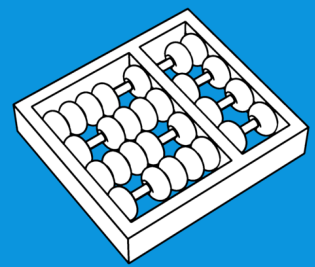
MC102 - Algoritmos e Programação de Computadores

05/25

17

Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br

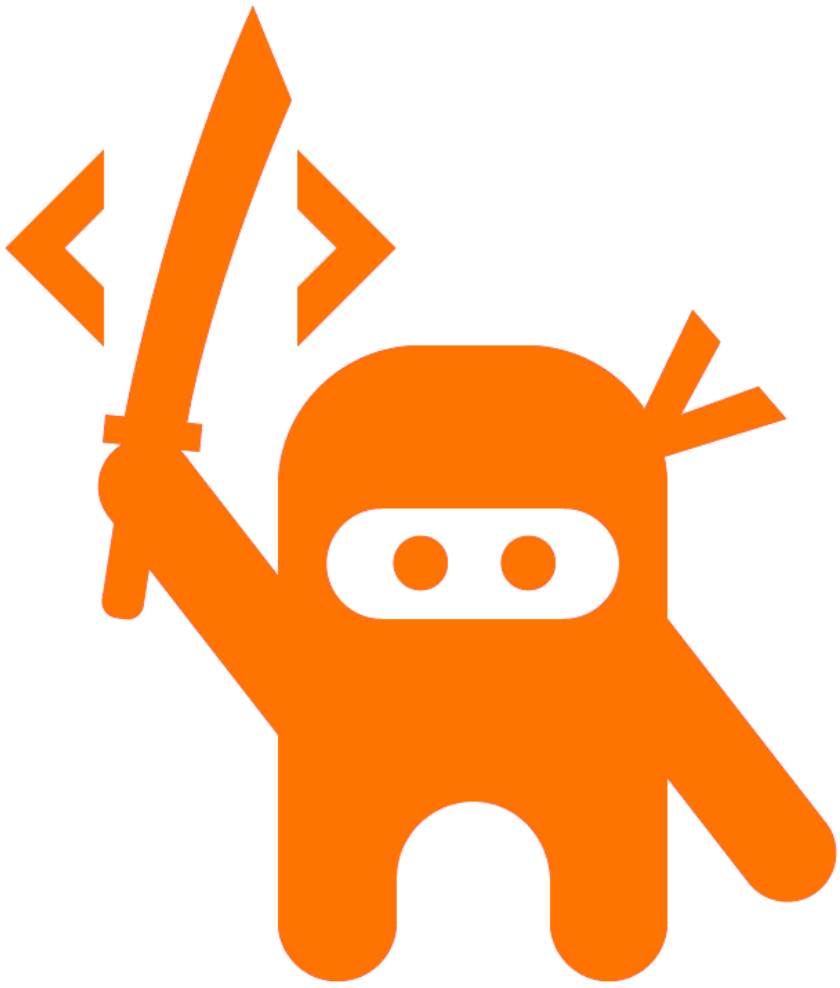
UNICAMP



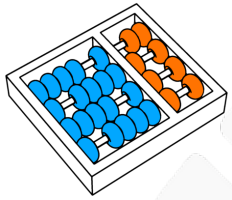
“No tatame, a força bruta se curva diante da técnica.”

Anônimo.

Soluções para os exercícios!

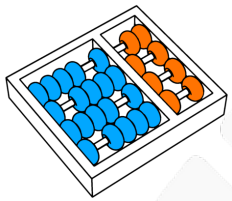


CODE
DOJO



Elemento mais repetido

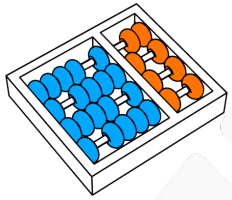
Faça uma função que dada uma matriz de elementos, em que não há repetições em uma mesma linha, retorne o elemento que mais aparece na matriz.



Solução

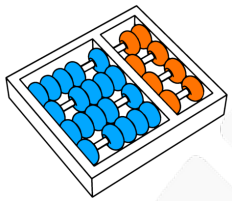
```python
def mais_repetido(matriz):
    repeticoes = {}
    moda = None
    for linha in matriz:
        for elemento in linha:
            if elemento not in repeticoes:
                repeticoes[elemento] = 1
            else:
                repeticoes[elemento] += 1
            if moda is None or repeticoes[moda] < repeticoes[elemento]:
                moda = elemento
    return moda
```

A ideia aqui é criar um dicionário indexado pelos elementos da matriz, onde cada entrada armazena o número de vezes que esse elemento ocorre. Para preencher esse dicionário, percorremos a matriz linha por linha e, em cada linha, elemento por elemento. Se o elemento atual ainda não estiver no dicionário, adicionamos uma nova entrada com ocorrência igual a 1; caso contrário, incrementamos seu contador em 1. Em ambos os casos, verificamos se esse elemento passou a ter mais ocorrências do que o elemento atualmente considerado como o mais frequente (a moda). Se isso ocorrer, atualizamos a moda.



Todos os pares que somam

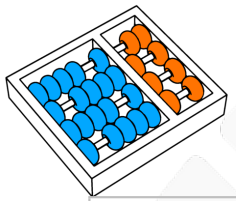
Faça uma função que dada uma lista de inteiros e um número, retorna alguma estrutura que armazena todos os pares da lista que somados são iguais ao número dado.



Solução 1.

```python
def pares_que_somam(lista, numero):
    resultado = set()
    for i in range(len(lista)):
        for j in range (i + 1, len(lista)):
            if lista[i] + lista[j] == numero:
                if lista[i] <= lista[j]:
                    resultado.add((lista[i], lista[j]))
                else:
                    resultado.add((lista[j], lista[i]))
    return resultado
```

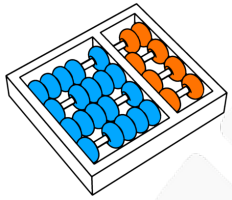
Para evitar repetir pares, podemos utilizar um conjunto como estrutura de resultado, adicionando os pares na forma de tuplas. Para resolver o problema, percorremos todos os pares possíveis: para cada posição ‘i‘ da lista, percorremos cada posição ‘j‘ que vem depois. Verificamos se a soma dos elementos nessas posições é igual ao número alvo; se for, adicionamos o par ao conjunto de resposta. Para garantir que não haja repetições, sempre adicionamos o par ordenado isto é, com o menor elemento primeiro e o maior depois.



Solução 2. Mais eficiente

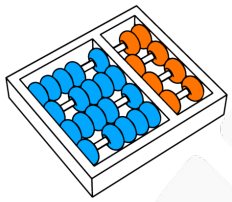
```python
def pares_que_somam(lista, numero):
    ordenada = sorted(lista)
    i = 0
    j = len(ordenada) - 1
    resultado = set()
    while i < j:
        if ordenada[i] + ordenada[j] == numero:
            resultado.add((ordenada[i], ordenada[j]))
            i += 1
            j -= 1
        if ordenada[i] + ordenada[j] < numero:
            i += 1
        if ordenada[i] + ordenada[j] > numero:
            j -= 1
    return resultado
```

A ideia aqui é ordenar primeiro a lista (para realizar menos comparações). Iniciamos com duas variáveis indicando as posições que estamos testando: uma no início da lista (‘i‘) e outra no final (‘j‘), de forma que a primeira apenas aumenta e a segunda apenas diminui. Se a soma dos elementos nas posições ‘i‘ e ‘j‘ for igual ao número alvo, então encontramos um novo par e o adicionamos à resposta. Note que, nesse caso, tanto o elemento em ‘i‘ quanto o em ‘j‘ não podem formar outro par válido com os elementos restantes. Assim, podemos mover os dois índices: incrementamos ‘i‘ e decrementamos ‘j‘. Caso a soma seja menor que o número, o elemento em ‘i‘ não poderá formar um par válido com nenhum outro elemento à esquerda de ‘j‘ (pois todos esses são menores que o atual em ‘j‘), então descartamos o elemento em ‘i‘, incrementando seu índice. Analogamente, se a soma for maior que o número, descartamos o elemento em ‘j‘, decrementando seu índice. O algoritmo termina quando as posições ‘i‘ e ‘j‘ se cruzam.



Anagramas

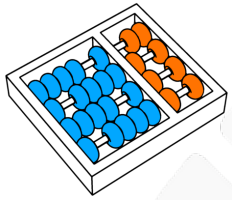
- Os anagramas de uma palavra são outras palavras que se obtém permutando a ordem das letras. Por exemplo “ALMA”, “LAMA” e “MALA” são anagramas.
- Faça uma função que dadas duas strings determine se uma é anagrama de outra.
- Usando a função anterior, faça uma segunda função que recebe um conjunto de strings e retorna uma estrutura em que as strings estejam separadas por grupos de anagramas.



Solução

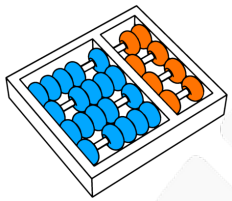
```python
def anagramas(string1, string2):
    return sorted(string1) == sorted(string2)

def agrupa(lista):
    agrupamentos = {}
    for i in range(len(lista)):
        chave = str(sorted(lista[i]))
        if chave not in agrupamentos:
            agrupamentos[chave] = set()
            agrupamentos[chave].add(lista[i])
            for j in range (i + 1, len(lista)):
                if anagramas(lista[i], lista[j]):
                    agrupamentos[chave].add(lista[j])
    return agrupamentos
```

Ao ordenar uma string, obtemos uma lista com os caracteres dispostos em ordem. Dessa forma, podemos verificar se duas strings são anagramas ordenando ambas e comparando os resultados. Na segunda função, a resposta será um dicionário indexado pelo anagrama ordenado (a chave). Em cada posição do dicionário, haverá um conjunto contendo todas as strings que são anagramas entre si, ou seja, que compartilham a mesma chave. Para resolver o problema, percorremos cada posição 'i' da lista de strings e computamos a chave correspondente (o anagrama ordenado da string nessa posição). Se essa chave já estiver no dicionário, não é necessário fazer nada, pois os anagramas dessa string já foram agrupados em uma iteração anterior. Caso contrário, adicionamos uma nova entrada no dicionário com essa chave e, em seguida, percorremos cada posição 'j' (com 'j $>$ i') da lista. Se a string na posição 'j' for anagrama da string em 'i', ela é adicionada ao conjunto associado à chave ordenada de 'i'.



Quadrado máximo

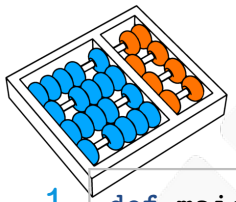
Faça uma função que dada uma matriz binária, retorna o tamanho do maior quadrado de 1's.



Solução 1.

```python
def maior_quadrado(matriz):
    for n in range(len(matriz), 0, -1):
        for i in range(len(matriz) - n + 1):
            for j in range(len(matriz) - n + 1):
                tem_quadrado = True
                for k in range(n):
                    for l in range(n):
                        if matriz[i + k][j + l] == 0:
                            tem_quadrado = False
                if tem_quadrado:
                    return n
    return 0
```

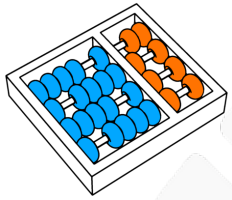
Como o objetivo é encontrar o tamanho da maior submatriz composta apenas por 1s, a ideia é testar cada possível tamanho, do maior para o menor. Dessa forma, no primeiro teste em que se encontrar uma submatriz válida com esse tamanho, o algoritmo pode ser interrompido. Para isso, percorremos cada possível valor de ‘n‘ (tamanho da submatriz). Para cada ‘n‘, verificamos todas as posições iniciais possíveis da submatriz dentro da matriz original. Ou seja, percorremos cada linha ‘i‘ (garantindo que ainda sobrem ao menos ‘n‘ linhas abaixo) e cada coluna ‘j‘ (de modo que sobrem ao menos ‘n‘ colunas à direita). Em seguida, verificamos se a submatriz de tamanho $n \times n$ com canto superior esquerdo em (i, j) é composta apenas por 1s. Essa verificação é feita percorrendo todas as posições da submatriz: linhas ‘i + k‘ e colunas ‘j + l‘, com ‘k‘ e ‘l‘ variando de ‘0‘ a ‘n - 1‘. Se for encontrado algum 0, descartamos essa submatriz. Caso contrário, retornamos imediatamente, indicando que encontramos uma submatriz de 1s de tamanho ‘n‘.



Solução 2. Mais eficiente

```python
def maior_quadrado(matriz):
    soma = sum([sum(linha) for linha in matriz]) # soma todos os elementos da matriz
    if soma == 0: # se não houver pelo menos um 1, então não há submatriz de 1s
        return 0
    submatrizes = [linha[:] for linha in matriz] # copia matriz
    aumento = True
    n = 0
    while aumento:
        aumento = False
        n += 1
        for i in range(len(matriz) - n):
            for j in range(len(matriz) - n):
                if matriz[i][j] == 1 and submatrizes[i][j + 1] == 1 and submatrizes[i + 1][j]
                ↪  == 1 and submatrizes[i + 1][j + 1] == 1:
                    submatrizes[i][j] = 1
                    aumento = True
                else:
                    submatrizes[i][j] = 0
    return n
```

A ideia é basicamente a seguinte: uma submatriz de 1s de tamanho $n \times n$ pode começar na posição (i, j) se a posição (i, j) contém um 1 e se houver submatrizes de tamanho $(n-1) \times (n-1)$ começando nas posições (i, j + 1), (i + 1, j) e (i + 1, j + 1). Dessa forma, em vez de percorrer explicitamente todas as submatrizes, podemos apenas fazer quatro verificações para cada posição. Mas para isso, precisamos calcular os resultados de forma incremental, começando com n = 1, depois n = 2, e assim por diante, até que não haja mais submatrizes válidas para um determinado 'n'. Para n = 1, o resultado é simplesmente a própria matriz original (assumindo que não seja composta apenas por zeros). A partir disso, podemos calcular os valores para n > 1 usando os resultados anteriores. Todos esses valores intermediários podem ser armazenados em uma única matriz auxiliar, atualizada a cada passo.

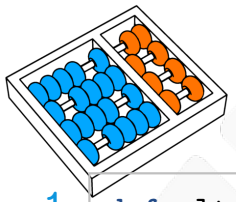


Maximizando a soma

- Faça uma função que recebe uma matriz de inteiros e duas posições adjacentes na matriz (numa mesma linha ou coluna). A sua função deve multiplicar o valor de cada uma das posições por -1 e substituir, nas respectivas posições da matriz da entrada.

- Faça uma função que recebe uma matriz de inteiros e retorna a soma dos elementos.

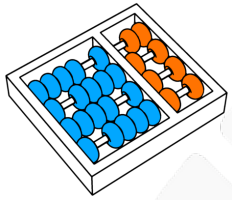
- Faça uma função que recebe uma matriz de inteiros e retorna o máximo valor da segunda função, supondo que seja possível aplicar a primeira função tantas vezes quanto necessário.



Solução

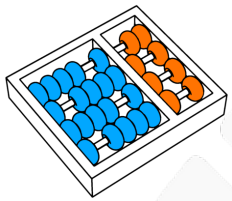
```python
def altera(matriz, linha1, coluna1, linha2, coluna2):
    if abs(linha1 - linha2) + abs(coluna1 - coluna2) == 1:
        matriz[linha1][coluna1] *= -1
        matriz[linha2][coluna2] *= -1
def soma(matriz):
    return sum([sum(linha) for linha in matriz])
def maximo(matriz):
    soma = 0
    negativos = 0
    menor = None
    for linha in matriz:
        for elemento in linha:
            if elemento <= 0:
                negativos += 1
            valor = abs(elemento)
            if menor = None or valor < menor:
                menor = valor
            soma += valor
    if negativos % 2 == 1:
        soma -= 2 * menor
    return soma
```

Neste problema, a ideia principal é perceber que basta contar a quantidade de números menores ou iguais a zero. Se essa quantidade for par, aplicando a primeira função descrita, sempre será possível tornar todos os elementos maiores ou iguais a zero. Caso contrário (quantidade ímpar), será inevitável que restem números negativos, mas sempre podemos fazer com que seja exatamente só um e com menor valor absoluto possível. Tendo isso em vista, basta somar os valores absolutos de todos os elementos da matriz; contar quantos desses elementos são negativos; identificar o elemento com menor valor absoluto. Se a contagem de negativos for ímpar, subtraímos duas vezes o menor valor absoluto da soma total. Assim, obtemos a soma máxima possível após aplicar a operação descrita.



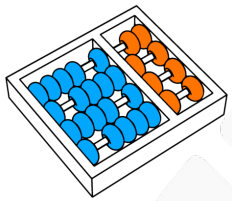
Mínimo número de alterações

Faça uma função que, dadas duas palavras, retorna o menor número de inserções, remoções ou trocas de letras que a segunda precisa fazer para ficar igual à primeira.



Solução

```python
def minimo_alteracoes(palavra1, palavra2):
    alteracoes = [[0] * len(palavra2 + 1) for i in range(len(palavra1) + 1)]
    for i in range(len(palavra1) + 1):
        alteracoes[i][0] = i
    for j in range(len(palavra2) + 1):
        alteracoes[0][j] = j
    for i in range(1, len(palavra1) + 1):
        for j in range(1, len(palavra2) + 1):
            if palavra1[i - 1] == palavra2[j - 1]: # não precisa alterar esse caractere
                alteracoes[i][j] = alteracoes[i - 1][j - 1]
            elif alteracoes[i - 1][j - 1] < alteracoes[i - 1][j] and alteracoes[i - 1][j - 1]
            ↪  < alteracoes[i][j - 1]: # compensa mais trocar de caractere
                alteracoes[i][j] = 1 + alteracoes[i - 1][j - 1]
            elif alteracoes[i - 1][j] < alteracoes[i][j - 1]: # compensa mais inserir
                alteracoes[i][j] = 1 + alteracoes[i - 1][j]
            else: # compensa mais remover
                alteracoes[i][j] = 1 + alteracoes[i][j - 1]
    return alteracoes[len(palavra1)][len(palavra2)]
```



Solução. Explicação

Vamos começar resolvendo o seguinte subproblema: para cada posição ‘i‘ da primeira palavra (‘p1‘) e cada posição ‘j‘ da segunda (‘p2‘), qual é o menor número de alterações necessárias para transformar a substring ‘p2[:j]‘ (isto é, os ‘j‘ primeiros caracteres de ‘p2‘) em ‘p1[:i]‘ (os ‘i‘ primeiros caracteres de ‘p1‘)?
Ao resolver esse subproblema para todos os valores de ‘i‘ e ‘j‘, teremos a resposta do problema original ao calcular o valor para ‘i = len(p1)‘ e ‘j = len(p2)‘.
Nos casos mais simples: se ‘j = 0‘ (ou seja, ‘p2[:j]‘ é a string vazia), basta inserir todos os caracteres de ‘p1[:i]‘, totalizando ‘i‘ operações; analogamente, se ‘i = 0‘, devemos remover todos os caracteres de ‘p2[:j]‘, totalizando ‘j‘ operações.
Nos demais casos, há duas situações principais:
Uma, se ‘p1[i1] == p2[j1]‘, os últimos caracteres coincidem, e nenhuma operação é necessária nesse passo. O problema se reduz a transformar ‘p2[:j1]‘ em ‘p1[:i1]‘.
Se os últimos caracteres forem diferentes, precisamos escolher entre três operações:

- **Substituição**: trocar o último caractere de ‘p2[:j]‘ para que fique igual ao de ‘p1[:i]‘. Isso equivale a transformar ‘p2[:j1]‘ em ‘p1[:i1]‘ e somar 1 (pela troca).
- **Inserção**: inserir o caractere final de ‘p1[:i]‘ ao final de ‘p2[:j]‘. Isso equivale a transformar ‘p2[:j]‘ em ‘p1[:i1]‘ e somar 1 (pela inserção).
- **Remoção**: remover o último caractere de ‘p2[:j]‘. Isso equivale a transformar ‘p2[:j1]‘ em ‘p1[:i]‘ e somar 1 (pela remoção).

Se usarmos uma matriz indexada por ‘i‘ e ‘j‘ para armazenar o custo mínimo de cada subproblema, podemos preencher a matriz baseando-nos nos valores já computados, até chegar à solução completa.

Coding dojo

MC102 - Algoritmos e Programação de Computadores

05/25

17

Santiago Valdés Ravelo
https://ic.unicamp.br/~santiago/
ravelo@unicamp.br

UNICAMP