Coding dojo

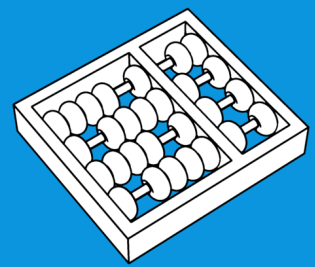
MC102 - Algoritmos e Programação de Computadores

06/25

26

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

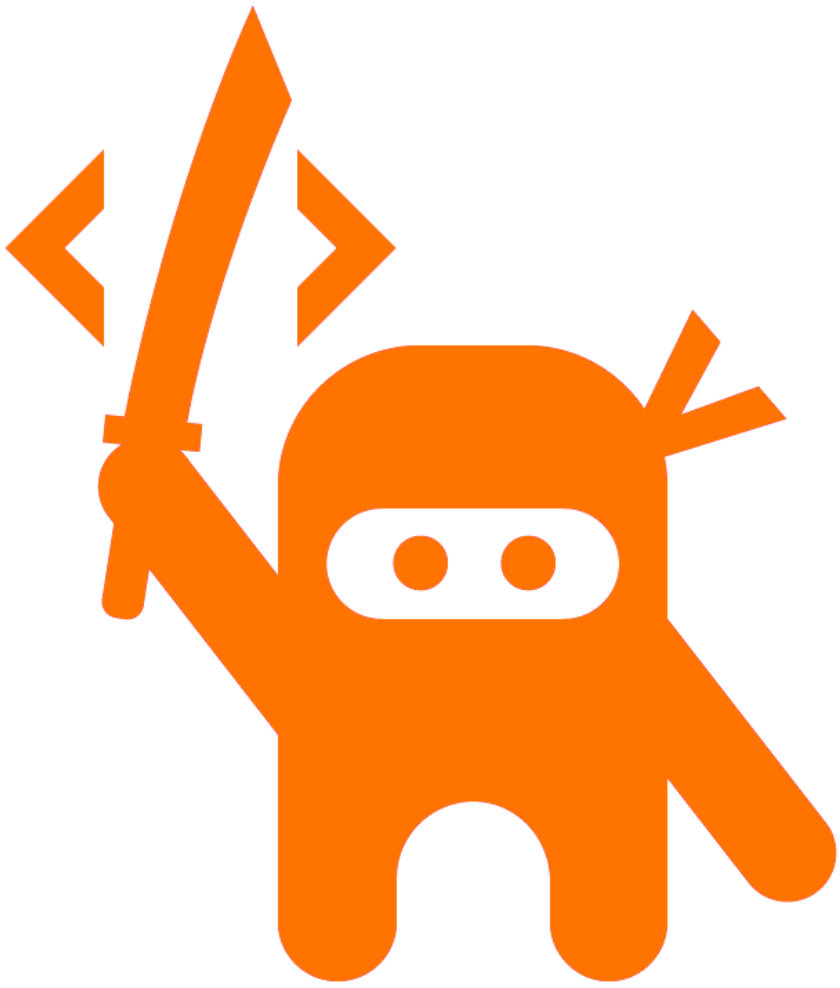
UNICAMP



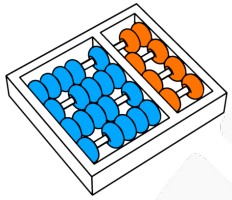
“Se houver esforço, sempre haverá conquista.”

Jigoro Kano.

Soluções para os exercícios!

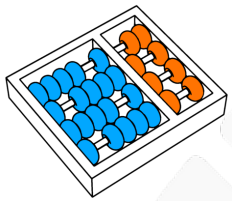


CODE
DOJO



Palíndromo

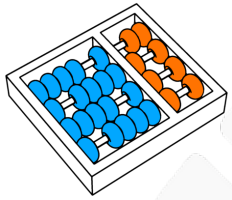
Faça uma função recursiva que dada uma string, determina se é um palíndromo ou não.



Solução

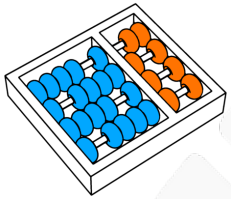
```
1  def palindromo_rec(palavra, e, d):
2      if e >= d:
3          return True
4      return palavra[e] == palavra[d] and palindromo_rec(palavra, e + 1, d - 1)
5
6  def palindromo(palavra):
7      return palindromo_rec(palavra, 0, len(palavra) - 1)
```

Para que uma palavra seja um palíndromo, o primeiro e o último caracteres devem ser iguais, e a substring que vai do segundo ao penúltimo caractere também deve ser um palíndromo. Essa é exatamente a ideia que define a recursão acima.



Produto

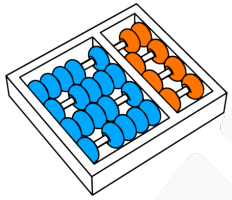
Dados dois inteiros a e b, faça uma função recursiva que calcule $a \cdot b$, sem usar multiplicação.



Solução

```
1  def produto(a, b):
2      if b == 1:
3          return a
4      return a + produto(a, b - 1)
```

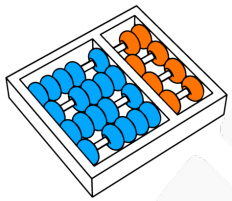
Note que: $a \cdot b = a + a \cdot (b - 1)$. Isso define a recursão acima.



Potência

Dados a e n, faça uma função recursiva que calcule a^n.

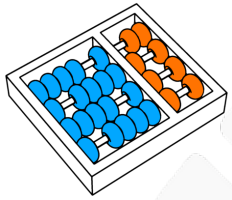
- Consegue fazer com aproximadamente $\log_2 n$ multiplicações?



Solução

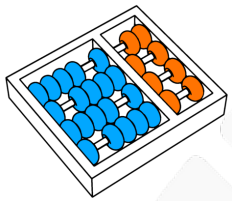
```python
def potencia(a, n):
    if n == 0:
        return 1
    a_n = potencia(a, n // 2)
    a_n *= a_n
    if n % 2 == 1:
        a_n *= a
    return a_n
```

Note que se n for par: $a^n = a^{\frac{n}{2}} \cdot a^{\frac{n}{2}}$; e se n for ímpar: $a^n = a^{\lfloor \frac{n}{2} \rfloor} \cdot a^{\lfloor \frac{n}{2} \rfloor} \cdot a$. Em qualquer caso, precisamos calcular $a^{\lfloor \frac{n}{2} \rfloor}$ e multiplicar pelo mesmo valor; depois em caso de ser ímpar, multiplicamos o resultado por a.



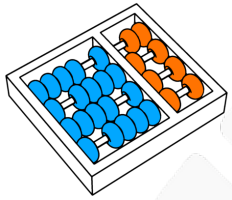
Busca binária

Faça uma função que implemente a busca binária.



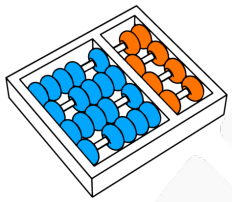
Solução

```
1  def busca_binaria(lista, e, d, x):
2      if e > d: # caso base quando não está na lista
3          return -1
4      m = (e + d) // 2
5      if lista[m] == x: # encontramos o elemento
6          return m
7      if lista[m] > x: # se o da metade é maior, está do lado esquerdo
8          return busca_binaria(lista, e, m - 1, x)
9      return busca_binaria(lista, m + 1, d, x) # senão, no lado direito
```



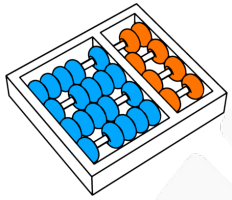
Eliminando duplicatas

Faça uma função recursiva que, dada uma string s, retorna uma string que representa s após remover todos os caracteres iguais consecutivos.



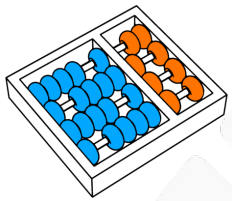
Solução

```
1  def elimina_duplicatas_rec(mem, palavra, n):
2      if n == len(palavra): # chegamos no final da palavra
3          return mem
4      if mem[-1] == palavra[n]: # a letra atual é igual à última que colocamos (ignorar)
5          return elimina_duplicatas_rec(mem, palavra, n + 1)
6      # senão, devemos adicionar a letra na resposta
7      return elimina_duplicatas_rec(mem + palavra[n], palavra, n + 1)
8
9  def elimina_duplicatas(palavra):
10     return elimina_duplicatas_rec(palavra[0], palavra, 1)
```



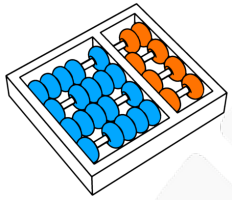
Anagramas

Faça uma função recursiva que imprima todos os anagramas de uma dada string.



Solução

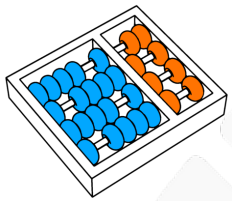
```
1  def anagramas_rec(mem, palavra, usados):
2      if len(mem) == len(palavra): # temos um anagrama
3          print(mem)
4      for i in range(palavra): # tentamos usar cada letra da palavra
5          if not usados[i]: # só colocamos as que não foram usadas
6              usados[i] = True # marcamos como usada
7              mem.append(palavra[i]) # a adicionamos no prefixo do anagrama sendo construído
8              anagramas_rec(mem, palavra, usados) # imprimimos os anagramas com esse prefixo
9              mem.pop() # removemos o caractere do prefixo
10             usados[i] = False # indicamos que deixamos de usar o caractere
11
12 def anagramas(palavra):
13     anagramas_rec([], palavra, [False] * len(palavra))
```



Todos os caminhos

Dada uma matriz binária, um caminho do extremo superior esquerdo ao extremos inferior direito é uma sequência (sem repetição) de casas adjacentes (na horizontal ou vertical), todas com valor 1 (verdadeiro).

- Faça uma função recursiva que recebe uma matriz binária e imprime todos os caminhos válidos do extremos superior esquerdo ao inferior direito.



Solução

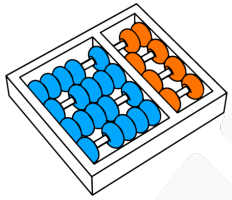
```python
MOVS = ((-1, 0), (1, 0), (0, -1), (0, 1))

def pode_ir(matriz, l, c):
    return 0 <= l < len(matriz) and 0 <= c < len(matriz[l]) and matriz[l][c] == 1

def todos_caminhos_rec(mem, matriz, l, c):
    if l == len(matriz) - 1 and c == len(matriz[l]) - 1: # chegamos no destino
        print(mem)
    for mov in MOVS: # tentamos com cada movimento
        next_l = l + mov[0]
        next_c = c + mov[1]
        if pode_ir(matriz, next_l, next_c): # vai na posição, se for válida
            matriz[next_l][next_c] = 2 # marca que passou por essa casa
            mem.append((next_l, next_c))) # adicionamos a casa no caminho
            todos_caminhos_rec(mem, matriz, next_l, next_c)
            mem.pop() # removemos a casa do caminhos
            matriz[next_l][next_c] = 1 # marca que pode passar pela casa

def todos_caminhos(matriz):
    todos_caminhos_rec([(0,0)], matriz, 0, 0)
```

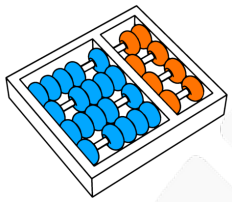
Nessa solução, só podemos usar caminhos formados por casas com valor '1'. Para evitar repetir uma casa já visitada no caminho atual, marcamos essa casa temporariamente com '2'. Assim, garantimos que não nos movimentaremos para uma casa pela qual já passamos.



Colorindo

Dada uma matriz inteira, onde os números representam cores, casas adjacentes (na horizontal, vertical ou diagonal) com uma mesma cor formam uma região dessa cor.

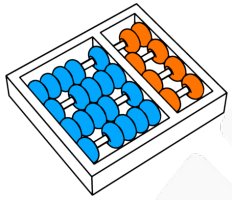
- Faça uma função recursiva que recebe uma matriz inteira, uma posição (i, j) e um número, e modifica todas as casas da região que contém (i, j) para serem iguais ao número dado.



Solução

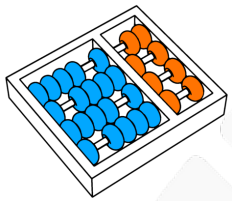
```
1  MOVS = ((-1, 0), (-1, -1), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1))
2
3  def pode_ir(matriz, l, c, x):
4      return 0 <= l < len(matriz) and 0 <= c < len(matriz[l]) and matriz[l][c] == x
5
6  def colorir_rec(matriz, l, c, x, k):
7      if pode_ir(matriz, l, c, x): # podemos colorir
8          matriz[l][c] = k
9          for mov in MOVS: # tentamos colorir em cada adjacente
10             next_l = l + mov[0]
11             next_c = c + mov[1]
12             colorir_rec(matriz, next_l, next_c, x, k)
13
14 def colorir(matriz, l, c, k):
15     if matriz[l][c] != k
16         colorir_rec(matriz, l, c, matriz[l][c], k)
```

Nessa solução, para verificar se uma casa é adjacente na mesma região, ela precisa ser uma casa vizinha e ter a mesma cor 'x'. Nesse caso, colorimos essa casa com 'k'. O caso base ocorre quando chegamos a uma casa fora da matriz ou com uma cor diferente de 'x' (ou seja, de outra região). Para evitar "recursão infinita", é fundamental garantir que 'x' e 'k' não sejam iguais; caso contrário, ficaríamos colorindo repetidamente a mesma casa. No entanto, isso já está garantido na verificação feita na linha 15: repare que, se a cor atual já for igual à que queremos usar, não precisamos fazer nada.



Maior palíndromo

Faça uma função recursiva que, dada uma string, imprima o tamanho do maior palíndromo que é substring dela.



Solução

```
1
2  def maior_pal_rec(palavra, e, d):
3      if palavra[e] == palavra[d] and palindromo_rec(palavra, e + 1, d - 1):
4          return d - e + 1
5      return max(maior_pal_rec(palavra, e + 1, d), maior_pal_rec(palavra, e, d - 1))
6
7
8  def maior_palindromo(palavra):
9      return maior_pal_rec(palavra, 0, len(palavra) - )
```

Aqui, a ideia é a seguinte: se o caractere da esquerda for igual ao da direita e o que estiver entre eles for um palíndromo, então todo o conjunto é um palíndromo (note que usamos aqui a função recursiva de palíndromo feita no primeiro desafio). Caso contrário, o maior palíndromo possível é aquele que ou não usa o caractere da direita ou não usa o caractere da esquerda. Por isso, testamos recursivamente os dois casos e ficamos com o maior.
É a solução mais eficiente? Certamente não. Podemos melhorá-la e, veremos como, em Projeto e Análise de Algoritmos com Programação Dinâmica!

Coding dojo

MC102 - Algoritmos e Programação de Computadores

06/25

26

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

UNICAMP