MATA52 - Exercícios da Semana 09

- Grupo: Claude Monet
- Autores:
    - Bruno de Lucas Santos Barbosa (responsável)
        - Eita segunda questão difícil....
    - Elis Marcela de Souza Alcantara
        - Resolvi a terceira questão.
    - Lucas dos Santos Lima
        - Resolvi a quarta questão.
    - Monique Santos da Silva
        - Resolvi a primeira questão.

Instruções (não apagar)

1. **Responsável**: Após copiar este notebook, altere o nome do notebook/documentação incluindo o nome do seu grupo. Por exemplo, se você é do grupo Ouro, altere o nome do notebook para "MATA53-Semana02-Ouro.ipynb"
2. **Responsável**: Compartilhe este documento com todos os membros do grupo (para participarem da elaboração deste documento). É importante que o notebook utilizado seja o mesmo compartilhado para que os registros de participação e colaboração fiquem salvos no histórico. Sugira uma divisão justa e defina um prazo aceitável para a inserção das soluções no Colab.
3. **Responsável**: Ao concluir a atividade, compartilhe o notebook com januario.ufba@gmail.com (dando permisão para edição) e deixando o aviso de notificação marcado, para que eu receba o seu e-mail. Identificar o nome do grupo na mensagem de compartilhamento.
4. **Cada membro**: Incluia o *seu próprio nome completo* na lista de autores que auxiliaram na elaboração deste notebook. Relate brevemente a sua contribuição na solução desta lista. O responsável aparece como sendo o(a) primeiro(a) autor(a).
5. **Cada membro**: Utilize os recursos de blocos de texto e de código para adicionar as suas respostas, sem alterar os blocos de texto e código existente. Não economize, esses blocos são de graça.

Exercícios

Para cada um dos seguintes problemas (se aplicável):

- Defina os subproblemas
- Reconheça e resolva os casos base
- Apresente a recorrência que resolva os subproblemas
- Apresente uma implementação ingênua
- Apresente uma solução utilizando PD "top-down" ou "bottom-up"
- Analise a complexidade da solução utilizando PD

1. Dado um valor monetário de n reais obtidos de forma ilícita, de quantas formas podemos trocar esse valor dado que temos um suprimento infinito de cada uma das cédulas $S = \{1, 2, 5, 10, 20, 50, 100\}$?

Subproblemas

A quantidade de subproblemas depende do valor monetário escolhido, apresenta-se no formato $S * n$, ou seja, sendo o conjunto das cédulas $S = [1, 2, 5, 10, 20, 50, 100]$ e o valor monetário n for 23, então serão 54 subproblemas.

Nota-se que o valor se torna exponencial, quanto maior o valor de n a quantidade de subproblemas irá se multiplicar. Por exemplo, se escolhermos um valor que fizesse uso de todas as cédulas, como 101, teriamos 4710 subproblemas.

Casos base

$n = 12$

$S = [1, 2, 5, 10, 20, 50, 100]$

	0	1	2	3	4	5	6	7	8	9	10	11	12
[ ]	1	0	0	0	0	0	0	0	0	0	0	0	0
[1]	1	1	1	1	1	1	1	1	1	1	1	1	1
[1, 2]	1	1	2	2	3	3	4	4	5	5	6	6	7
[1, 2, 5]	1	1	2	2	3	4	5	6	7	8	10	11	13
[1, 2, 5, 10]	1	1	2	2	3	4	5	6	7	8	11	12	15

O tamanho da tabela é igual a (número de cédulas + 1) * (n +1). Cada campo da tabela armazena a solução de um subproblema.

Com essa demostração, para o valor $n = 12$, foram analisados os elementos $[1, 2, 5, 10]$ do conjunto S. Não há necessidade de prosseguir com os demais elementos, pois são maiores do

que o valor 12. A conclusão encontra-se na última linha e coluna da tabela, 15 é a quantidade total de maneiras que podemos dar o troco utilizando as cédulas 1, 2, 5 e 10.

Recorrência que resolve os subproblemas

Recorrência:

`tabela[linha][coluna] = tabela[linha - 1][coluna] + tabela[linha][coluna - n[linha-1]]`

Existem duas formas para resolver os subproblemas:

1) Usando Recursividade:

Complexidade de tempo: $O(n^2)$

Complexidade de espaço: $O(1)$

2) Usando programação dinâmica:

Complexidade de tempo: $O(S * n)$

Complexidade de espaço: $O(n)$

Implementação ingênua

Implementação ingênua dos exemplos abordados anteriormente com n sendo representado por: 12, 23 e 101

A implementação ingênua usa recursividade para resolver os subproblemas. Nesta abordagem, temos que iterar sobre todas as combinações possíveis de moedas que igualam a soma dada toda vez que atualizar o número mínimo de moedas necessário para criar essa soma.

Tempo de Complexidade: $O(n^2)$ - Toda chamada recursiva está fazendo n chamadas recursivas

Espaço: $O(1)$ - Nenhum espaço extra está sendo usado.

```
def combinacoes(S, indice, n):
  if n == 0: return 1
  resultado = 0
  for i in range(indice, len(S)):
    if n - S[i] >= 0:
      resultado += combinacoes(S, i, n-S[i])

  return resultado

S = [1, 2, 5, 10, 20, 50, 100]
n = 12

print("Cédulas: {} \nValor monetário = {} \nTotal de combinações = {}" .format(S, n
print()
print("Cédulas: {} \nValor monetário = {} \nTotal de combinações = {}" .format(S, 1
```

```
print()
print("Cédulas: {} \nValor monetário = {} \nTotal de combinações = {}" .format(S,
```

```
Cédulas: [1, 2, 5, 10, 20, 50, 100]
Valor monetário = 12
Total de combinações = 15

Cédulas: [1, 2, 5, 10, 20, 50, 100]
Valor monetário = 23
Total de combinações = 54

Cédulas: [1, 2, 5, 10, 20, 50, 100]
Valor monetário = 101
Total de combinações = 4710
```

Solução utilizando PD "bottom-up"

Podemos otimizar a complexidade de tempo de nossa abordagem anterior mantendo uma matriz `PD`, onde `PD[i]` armazena o máximo de maneiras possíveis para cada quantidade. Como a abordagem recursiva tinha muitos subproblemas, a matriz `PD` evitará repetir operações.

```
def combinacoes(S, n):
    pd = [0] * (n + 1)
    pd[0] = 1

    for i in range(len(S)):
        for j in range(1, len(pd)):
            if j - S[i] >= 0:
                pd[j] += pd[j- S[i]]
            else: continue

    return pd[len(pd) - 1]

S = [1, 2, 5, 10, 20, 50, 100]
n = 12

print("Cédulas: {} \nValor monetário = {} \nTotal de combinações = {}" .format(S, n
print()
print("Cédulas: {} \nValor monetário = {} \nTotal de combinações = {}" .format(S,
print()
print("Cédulas: {} \nValor monetário = {} \nTotal de combinações = {}" .format(S,
```

```
Cédulas: [1, 2, 5, 10, 20, 50, 100]
Valor monetário = 12
Total de combinações = 15

Cédulas: [1, 2, 5, 10, 20, 50, 100]
Valor monetário = 23
Total de combinações = 54

Cédulas: [1, 2, 5, 10, 20, 50, 100]
```

```
Valor monetário = 101
Total de combinações = 4710
```

Complexidade da solução utilizando PD

Complexidade de tempo: $O(S * n)$ para cada combinação, calculamos o número de maneiras de formar todos os valores.

Complexidade do Espaço: $O(n)$ como espaço extra para o máximo de maneiras possíveis para cada quantidade de armazenamento que foi usada.

2. Dados números $p_1, p_2, \ldots, p_n, v_1, v_2, \ldots, v_n$ e um subconjunto X de $1, 2, \ldots, n$, denotaremos por p(X) e v(X) as somas $\sum_{i \in X} pi$ e $\sum_{i \in X} vi$ respectivamente. Considere o seguinte problema: Dados números naturais $p_1, p_2, \ldots, p_n, v_1, v_2, \ldots, v_n$ e c, encontrar um subconjunto X de $1, 2, \ldots, n$ que maximize $v(X)$ sob a restrição $p(X) \leq c$.

3. Seja um bastão de metal de n metros e um vetor de preços que contém os preços para todas as pedaços de tamanho até n. Determine o valor máximo que pode ser obtido com o corte do bastão e venda de seus pedaços. Dica: Capítulo 15 do Cormen.

Trabalharemos em cima de um pedaço para vender inteiro + seu restante, primeiro resolveremos de forma recursiva.
Tamanho n: o primeiro pedaço tem tamanho n - 1 e o segundo tem tamanho n - 2 e assim por diante.
O custo de venda do primeiro pedaço é o preço do pedaço + o custo de venda do restante, e assim sucessivamente.
Temos a seguinte solução ingênua utilizando recursividade:

```
def problema_bastao_metal(preco, tamanho):
    if tamanho == 0:
        return 0

    preco_max = -1

    for i in range(tamanho):
        preco_max = max(preco_max, preco[i] + problema_bastao_metal(preco, tamanho

    return preco_max

print(problema_bastao_metal([1, 3, 4, 5, 7, 9, 10, 11], 8))
```

```
12
```

Utilizando memoization, podemos ter uma tabela parar armazenar os valores que já calculamos e assim evitar a recursão, consequentemente deixando o programa mais rápido, basicamente é a mesma implementação da recursividade, porém, adicionando a tabela:

```
def problema_bastao_metal_memoization(preco, tamanho, memo):
    if memo[tamanho] != -1:
        return memo[tamanho]

    preco_max = -1

    for i in range(tamanho):
        preco_max = max(preco_max, preco[i] + problema_bastao_metal_memoization(pr

    memo[tamanho] = preco_max
    return preco_max

def memoization(tamanho, preco):
    memo = [-1] * (tamanho + 1)
    memo[0] = 0
    problema_bastao_metal_memoization(preco, tamanho, memo)

    return memo[tamanho]

print(memoization(8, [1, 3, 4, 5, 7, 9, 10, 11]))
```

```
12
```

Como cada subproblema é resolvido uma vez por conta da memoization, porém, para resolver um subproblema de tamanho n temos que percorrer n iterações no `for`, e por conta disso todas essas iterações do `for` nas chamadas recursivas criam uma série aritmética que resulta em $O(n^2)$.

4. Seja a seguinte recorrência sobre o conjunto dos números naturais:

$$T(n) = \begin{cases} T(n-1) + T(n-3) & \text{se n} \geq 3. \\ 7 & \text{caso contrário} \end{cases}$$

Escreva um algoritmo eficiente, utilizando seus conhecimentos de programação dinâmica, que dado n, calcule o valor do n-ésimo termo da recorrência.

- Subproblemas: Encontrar o valor de $T(n-1)$ e $T(n-3)$ e somá-los.
- Caso base: Caso $n < 3$ então $T(n) = 7$
- A recorrência é $T(n-1)$ e $T(n-3)$ na qual o número n é sempre subtraído, até atingir o caso base e resolver o subproblema.

- Solução Ingênua:

```
def t(n):
  if n < 3: return 7
  return t(n-1) + t(n-3)

print(t(50))
```

```
854742679
```

```
# Solução utilizando PD "Top-Down"
memo = {}
def t(n):
  if n in memo: return memo[n]
  res = 7
  if n >= 3: res = t(n-1) + t(n-3)
  memo[n] = res
  return res

print(t(50))
```

```
854742679
```

Para analisar a complexidade de maneira mais eficaz percebendo a importância da programação dinâmica e principalmente da alternativa utilizada - top-down, primeiro precisamos perceber a assintoticidade da aplicação ingênua e recursiva, na qual temos $O(2^n)$ e $\Theta(1.6180^n)$, nesse último sendo exatamente e proporção áurea.

Considerando que cada número n só será chamado uma única vez em cada função, é razoável dizer que a complexidade se torna, com a abordagem top-down, $O(n)$, visto que cada valor será chamado uma vez e seu resultado armazenado no dicionário memo, e posteriormente só consultado, não calculado novamente.