CP 4 - Dynamic Programming

Questão 1 (2,5 pontos)

Implemente o algoritmo **Heap Sort** em Python para ordenar uma lista de números inteiros em **ordem crescente**.

Exemplo de uso:

```
entrada = [4, 10, 3, 5, 1]

heap_sort(entrada)

print(entrada)

# Saída esperada: [1, 3, 4, 5, 10]
```

Explique a complexidade assintótica desse algoritmo.

Questão 2 (2,5 pontos)

Para cada um dos cenários abaixo, indique qual estrutura de dados é **mais adequada** e **justifique sua escolha**, levando em consideração **tempo de inserção**, **remoção**, **busca** e **complexidade de memória**, quando aplicável.

Cenários:

a) Um sistema que precisa **inserir elementos constantemente no meio** de uma lista com milhares de registros.
b) Uma aplicação que realiza **buscas frequentes e rápidas por chaves específicas**, como nomes de usuários.
c) Uma situação em que é necessário **inserir elementos rapidamente e buscá-los com rapidez**, mas a ordem dos elementos não importa.

d) Um sistema que precisa **manter os elementos sempre ordenados** após inserções e permitir **acesso rápido ao menor elemento**.

e) Um aplicativo que **acessa elementos em posições específicas por índice** com frequência, mas raramente faz inserções ou remoções.

Possíveis estruturas para escolher (mas não se limite a elas):

- Lista (array/vetor)
- Lista encadeada (simples ou duplamente encadeada)
- Tabela Hash
- Heap (min-heap / max-heap)
- Árvore Binária de Busca (BST)
- Fila
- Pilha
- Deque

Questão 3 (2 pontos)

Observe o código abaixo e responda às perguntas:

```
class TabelaHash:
    def __init__(self, tamanho):
        self.tamanho = tamanho
        self.tabela = [None] * tamanho

    def hash(self, chave):
        return chave % self.tamanho

    def inserir(self, chave, valor):
        i = 0
        posicao = self.hash(chave)
        nova_posicao = posicao

        while self.tabela[nova_posicao] is not None and self.tabela[nova_posicao][0] !=
chave:
            i += 1
            nova_posicao = (posicao + i ** 2) % self.tamanho
```

```
            if i > self.tamanho:
                raise Exception("Tabela Hash cheia")

        self.tabela[nova_posicao] = (chave, valor)

    def buscar(self, chave):
        i = 0
        posicao = self.hash(chave)
        nova_posicao = posicao

        while self.tabela[nova_posicao] is not None:
            if self.tabela[nova_posicao][0] == chave:
                return self.tabela[nova_posicao][1]
            i += 1
            nova_posicao = (posicao + i ** 2) % self.tamanho
            if i > self.tamanho:
                break
        return None
```

Qual problema esse código resolve?

a) Ordenação de dados em memória

b) Busca binária em listas encadeadas

c) Gerenciamento de colisões em tabelas hash

d) Compressão de strings

Qual técnica é utilizada no código?

a) Encadeamento (Chaining)

b) Sondagem Linear (Linear Probing)

c) Rehashing com Heap

d) Sondagem Quadrática (Quadratic Probing)

Questão 4 (3 pontos)

Crie um programa em Python que leia uma frase do usuário e utilize uma **tabela hash** para contar quantas vezes **cada letra do alfabeto** aparece na frase.

Ignore espaços, acentuação e **considere apenas letras de A a Z** (não diferencia maiúsculas de minúsculas).

Exemplo:

Entrada:

"Hash é rápido e eficiente"

Saída esperada:

a: 2

c: 1

d: 1

e: 4

f: 1

h: 2

i: 2

p: 1

q: 1

r: 2

s: 1

t: 1

Objetivos da questão:

- Aplicar hash para mapeamento de chaves (letras) para valores (contagens)
- Trabalhar com lower() para uniformização
- Ignorar caracteres irrelevantes usando isalpha()