```
"""
1. Bubble Sort

Implemente o algoritmo Bubble Sort para ordenar
a lista abaixo em ordem crescente: [5, 3, 8, 4, 2]
"""

def bubble_sort(lista):
    n = len(lista)
    for i in range(n-1):
        for j in range(n-1-i):
            if lista[j] > lista[j+1]:
                lista[j], lista[j+1] = lista[j+1], lista[j]  # troca
    return lista

valores = [5, 3, 8, 4, 2]
print("Lista ordenada:", bubble_sort(valores))

"""
2. Selection Sort

Enunciado: Ordene a lista [29, 10, 14, 37, 13] utilizando Selection Sort.
"""

def selection_sort(lista):
    n = len(lista)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if lista[j] < lista[min_idx]:
                min_idx = j
        lista[i], lista[min_idx] = lista[min_idx], lista[i]
    return lista

valores = [29, 10, 14, 37, 13]
print("Lista ordenada:", selection_sort(valores))

"""
3. Insertion Sort

Enunciado: Ordene a lista [12, 11, 13, 5, 6] utilizando Insertion Sort.

Resolução:
"""

def insertion_sort(lista):
    for i in range(1, len(lista)):
        chave = lista[i]
        j = i - 1
        while j >= 0 and chave < lista[j]:
            lista[j+1] = lista[j]
            j -= 1
        lista[j+1] = chave
    return lista

valores = [12, 11, 13, 5, 6]
print("Lista ordenada:", insertion_sort(valores))

"""
4. Merge Sort

Implemente o algoritmo Merge Sort para ordenar a lista
[38, 27, 43, 3, 9, 82, 10].
"""
def merge_sort(lista):
    if len(lista) > 1:
        meio = len(lista)//2
        esquerda = lista[:meio]
        direita = lista[meio:]

        merge_sort(esquerda)
        merge_sort(direita)

        i = j = k = 0
        while i < len(esquerda) and j < len(direita):
            if esquerda[i] < direita[j]:
                lista[k] = esquerda[i]
                i += 1
            else:
                lista[k] = direita[j]
                j += 1
            k += 1

        while i < len(esquerda):
```

```
            lista[k] = esquerda[i]
            i += 1
            k += 1

        while j < len(direita):
            lista[k] = direita[j]
            j += 1
            k += 1
    return lista

valores = [38, 27, 43, 3, 9, 82, 10]
print("Lista ordenada:", merge_sort(valores))

"""
5. Quick Sort

Ordene a lista [10, 7, 8, 9, 1, 5] utilizando Quick Sort.
"""
def quick_sort(lista):
    if len(lista) <= 1:
        return lista
    else:
        pivo = lista[0]
        menores = [x for x in lista[1:] if x <= pivo]
        maiores = [x for x in lista[1:] if x > pivo]
        return quick_sort(menores) + [pivo] + quick_sort(maiores)

valores = [10, 7, 8, 9, 1, 5]
print("Lista ordenada:", quick_sort(valores))

"""
6. Análise Comparativa
Compare a complexidade dos algoritmos estudados em termos de tempo de execução.
"""

| Algoritmo      | Melhor Caso | Caso Médio | Pior Caso  | Observações                                                          |
| -------------- | ----------- | ---------- | ---------- | -------------------------------------------------------------------- |
| Bubble Sort    | O(n)        | O(n²)      | O(n²)      | Simples, mas ineficiente em listas grandes.                          |
| Selection Sort | O(n²)       | O(n²)      | O(n²)      | Independe da ordem inicial; poucas trocas.                           |
| Insertion Sort | O(n)        | O(n²)      | O(n²)      | Muito eficiente para listas quase ordenadas.                         |
| Merge Sort     | O(n log n)  | O(n log n) | O(n log n) | Estável, divide e conquista; requer espaço extra.                    |
| Quick Sort     | O(n log n)  | O(n log n) | O(n²)      | Muito rápido na prática; pior caso raro (pivot mal escolhido).       |

"""
### 1.

O Bubble Sort é classificado como:
A) Algoritmo de divisão e conquista
B) Algoritmo de ordenação simples baseado em trocas adjacentes
C) Algoritmo recursivo eficiente
D) Algoritmo de busca

### 2.

No pior caso, a complexidade de tempo do Bubble Sort é:
A) O(n)
B) O(log n)
C) O(n²)
D) O(n log n)

### 3.

O Selection Sort encontra em cada iteração:
A) O maior elemento e coloca no final
B) O menor elemento e coloca na posição correta
C) O elemento central e divide o vetor
D) Um pivô aleatório

### 4.

Qual algoritmo é mais eficiente para listas já quase ordenadas?
A) Bubble Sort
B) Selection Sort
C) Insertion Sort
D) Merge Sort

### 5.

O Merge Sort utiliza qual técnica principal?
A) Programação dinâmica
B) Divisão e conquista
C) Trocas adjacentes
D) Escolha de pivô

### 6.
```

O Quick Sort pode apresentar pior caso de:
A) O(n log n)
B) O(log n)
C) O(n²)
D) O(1)

7.

O Insertion Sort possui melhor caso de complexidade:
A) O(n)
B) O(n²)
C) O(log n)
D) O(n log n)

8.

O Selection Sort realiza quantas trocas no total, aproximadamente?
A) Sempre O(n)
B) Sempre O(n²)
C) Pode variar de O(n) a O(n²)
D) Zero

9.

O Bubble Sort realiza quantas comparações no pior caso?
A) n
B) n²
C) log n
D) n log n

10.

O Merge Sort precisa de espaço adicional de:
A) Constante (O(1))
B) O(log n)
C) O(n)
D) Nenhum

11.

O Quick Sort funciona escolhendo:
A) Um elemento chamado pivô
B) Sempre o menor elemento
C) Sempre o maior elemento
D) O elemento central do vetor

12.

Qual desses algoritmos é considerado estável?
A) Selection Sort
B) Quick Sort
C) Merge Sort
D) Heap Sort

13.

Um algoritmo de ordenação é estável quando:
A) Ordena mais rápido que O(n log n)
B) Mantém a ordem relativa de elementos iguais
C) Usa menos memória
D) Sempre usa pivô fixo

14.

O algoritmo de ordenação mais simples, mas também um dos mais lentos, é:
A) Merge Sort
B) Quick Sort
C) Bubble Sort
D) Insertion Sort

15.

No melhor caso, a complexidade do Insertion Sort é:
A) O(n²)
B) O(n log n)
C) O(n)
D) O(1)

16.

O Quick Sort atinge seu pior caso quando:
A) O pivô é sempre o menor ou o maior elemento
B) A lista está já ordenada
C) A lista tem apenas um elemento

D) O pivô é sempre o elemento central

17.

O Merge Sort sempre divide a lista em:
A) Metades iguais
B) Dois elementos aleatórios
C) Três partes
D) Elementos repetidos

18.

O Selection Sort é ineficiente porque:
A) Requer recursão
B) Requer memória extra
C) Faz muitas trocas
D) Faz muitas comparações

19.

Entre os algoritmos abaixo, qual geralmente é mais rápido na prática para grandes conjuntos de dados?
A) Bubble Sort
B) Insertion Sort
C) Merge Sort
D) Quick Sort

20.

A complexidade média do Quick Sort é:
A) O(n)
B) O(n^2)
C) O(n log n)
D) O(log n)

21.

A complexidade de tempo do Merge Sort é:
A) O(n^2) em todos os casos
B) O(n log n) em todos os casos
C) O(n) no pior caso
D) O(log n) no melhor caso

22.

Qual algoritmo é mais indicado para listas pequenas e quase ordenadas?
A) Merge Sort
B) Quick Sort
C) Insertion Sort
D) Heap Sort

23.

Em termos de trocas, qual algoritmo realiza menos operações em geral?
A) Bubble Sort
B) Selection Sort
C) Insertion Sort
D) Quick Sort

24.

Qual algoritmo possui pior caso **igual ao caso médio**?
A) Quick Sort
B) Merge Sort
C) Insertion Sort
D) Bubble Sort

25.

Um ponto fraco do Quick Sort é:
A) Sempre requer memória extra proporcional a O(n)
B) É instável em algumas implementações
C) É muito lento em listas pequenas
D) Não funciona com números negativos

26.

1) Explique o funcionamento do algoritmo Bubble Sort e discuta sua eficiência em relação a listas pequenas e grandes.
R:

27.

Compare o Selection Sort e o Insertion Sort quanto ao número de comparações e trocas realizadas.
R:

```
### 28.

Explique a técnica de divisão e conquista utilizada pelo Merge Sort e analise sua
complexidade.
R:

### 29.
O Quick Sort é geralmente mais rápido que o Merge Sort, mas pode ser menos confiável
em alguns casos. Explique por quê.
R:

### 30.
Elabore uma análise comparativa entre os algoritmos Bubble Sort, Selection Sort,
Insertion Sort, Merge Sort e Quick Sort, destacando em que situações cada um pode ser mais indicado.
"""
```