Taller

Integrantes:

192250: Andrés Gómez

192324: Arley Santiago

1. Contar pares de una lista.

   a.

```
def contar_pares_lista(arr):
    count = 0  # 1 operación
    for i in range(len(arr)):  # n iteraciones
        for j in range(i + 1, len(arr)):  # ~n/2 iteraciones en promedio
            if arr[i] + arr[j] == 0:  # 1 operación
                count += 1  # 1 operación
    return count  # 1 operación
# Peso total: ~ n*(n-1)/2 → O(n^2)
```

   b.

```
from collections import Counter

def contar_pares_lista_opt(arr):
    count = 0  # 1 operación
    freq = Counter(arr)  # O(n)
    for num in arr:  # n iteraciones
        complement = -num  # 1 operación
        if complement in freq:  # O(1) promedio
            count += freq[complement]  # 1 operación
            if complement == num:  # 1 operación
                count -= 1  # 1 operación
    return count // 2  # 1 operación
# Peso total: O(n)
```

2. Búsqueda lineal.

a.

```
def busqueda_lineal(arr, target):
    for i in range(len(arr)):  # n iteraciones
        if arr[i] == target:  # 1 operación por iteración
            return i  # 1 operación
    return -1  # 1 operación
# Peso total: O(n)
```

b.

```
def busqueda_lineal_v2(arr, target):
    encontrado = False  # 1 operación
    for i in range(len(arr)):  # n iteraciones
        if arr[i] == target:  # 1 operación
            encontrado = True  # 1 operación
            return i  # 1 operación
    if not encontrado:  # 1 operación
        return -1  # 1 operación
# Peso total: O(n)
```

3. ¿Hay duplicados?

a.

```
def hay_duplicados(arr):
    for i in range(len(arr)):  # n iteraciones
        for j in range(i + 1, len(arr)):  # ~n/2 iteraciones promedio
            if arr[i] == arr[j]:  # 1 operación
                return True  # 1 operación
    return False  # 1 operación
# Peso total: ~ n*(n-1)/2 → O(n^2)
```

b.

```
def hay_duplicados_opt(arr):
    seen = set()  # 1 operación
    for num in arr:  # n iteraciones
        if num in seen:  # O(1) promedio
```

```
            return True  # 1 operación
        seen.add(num)  # O(1) promedio
    return False  # 1 operación
# Peso total: O(n)
```

4. Sumar todos los elementos de una matriz mxn

    a.

```
def suma_matriz(matriz):
    total = 0  # 1 operación
    for fila in matriz:  # n iteraciones (filas)
        for valor in fila:  # m iteraciones (columnas)
            total += valor  # 1 operación por elemento
    return total  # 1 operación
# Peso total: n*m → O(n*m)
```

    b.

```
def suma_matriz_opt(matriz):
    return sum(sum(fila) for fila in matriz)  # n*m operaciones (suma
s)
# Peso total: O(n*m)
```

5. Suma de submatrices con preprocesamiento -> 2D.

    a.

```
def suma_submatrices(matriz):
    n = len(matriz)  # 1 operación
    m = len(matriz[0])  # 1 operación
    preprocesado = [[0] * (m + 1) for _ in range(n + 1)]  # (n+1)*(m+1)
operaciones

    for i in range(1, n + 1):  # n iteraciones
        for j in range(1, m + 1):  # m iteraciones
            preprocesado[i][j] = matriz[i-1][j-1] + preprocesado[i-1][j] + p
reprocesado[i][j-1] - preprocesado[i-1][j-1]  # 4 operaciones
```

```
    return preprocesado  # 1 operación
# Peso total: O(n*m)
```

b.

```
def consulta_suma(preprocesado, x1, y1, x2, y2):
    return preprocesado[x2+1][y2+1] - preprocesado[x1][y2+1] - prep
rocesado[x2+1][y1] + preprocesado[x1][y1]  # 4 accesos + 3 restas
+ 1 suma
# Peso total: O(1)
```

6. ¿Son anagramas?

a.

```
def son_anagramas(cadena1, cadena2):
    return sorted(cadena1) == sorted(cadena2)  # O(n log n) para cad
a sorted + O(n) comparación
# Peso total: O(n log n)
```

b.

```
from collections import Counter

def son_anagramas_v2(cadena1, cadena2):
    return Counter(cadena1) == Counter(cadena2)  # O(n) para contar
+ O(n) comparación
# Peso total: O(n)
```

7. Ordenar cadenas.

a.

```
def ordenar_cadena(cadena):
    return ''.join(sorted(cadena))  # O(n log n) + O(n) para join
# Peso total: O(n log n)
```

b.

```
import heapq

def ordenar_cadena_v2(cadena):
    return ''.join(heapq.nsmallest(len(cadena), cadena))  # O(n log n)
# Peso total: O(n log n)
```

8. Máxima suma de subarreglos (Kadane).

a.

```
def kadane(arr):
    max_ending_here = max_so_far = arr[0]  # 2 operaciones
    for x in arr[1:]:  # n-1 iteraciones
        max_ending_here = max(x, max_ending_here + x)  # 2 operacio
nes por iteración
        max_so_far = max(max_so_far, max_ending_here)  # 1 operació
n por iteración
    return max_so_far  # 1 operación
# Peso total: O(n)
```

b.

```
def kadane_subarray(arr):
    max_ending_here = max_so_far = arr[0]  # 2 operaciones
    start = end = s = 0  # 3 operaciones
    for i in range(1, len(arr)):  # n-1 iteraciones
        if arr[i] > max_ending_here + arr[i]:  # 1 operación
            max_ending_here = arr[i]  # 1 operación
            s = i  # 1 operación
        else:
            max_ending_here += arr[i]  # 1 operación

        if max_ending_here > max_so_far:  # 1 operación
            max_so_far = max_ending_here  # 1 operación
            start = s  # 1 operación
            end = i  # 1 operación
    return max_so_far, arr[start:end+1]  # 2 operaciones
# Peso total: O(n)
```

9. Quickselect.

a.

```
def quickselect(arr, k):
    if len(arr) == 1:  # 1 operación
        return arr[0]  # 1 operación

    pivot = arr[len(arr) // 2]  # 1 operación
    left = [x for x in arr if x < pivot]  # n operaciones
    right = [x for x in arr if x > pivot]  # n operaciones
    pivot_list = [x for x in arr if x == pivot]  # n operaciones

    if k < len(left):  # 1 operación
        return quickselect(left, k)  # recursión
    elif k < len(left) + len(pivot_list):  # 1 operación
        return pivot  # 1 operación
    else:
        return quickselect(right, k - len(left) - len(pivot_list))  # recursió
n
# Peso total: Peor caso O(n^2), Promedio O(n)
```

b.

```
def partition(arr, low, high):
    pivot = arr[high]  # 1 operación
    i = low - 1  # 1 operación
    for j in range(low, high):  # n iteraciones
        if arr[j] < pivot:  # 1 operación
            i += 1  # 1 operación
            arr[i], arr[j] = arr[j], arr[i]  # 3 operaciones (swap)
    arr[i+1], arr[high] = arr[high], arr[i+1]  # 3 operaciones
    return i + 1  # 1 operación
# Peso total: O(n)

def quickselect_v2(arr, low, high, k):
    if low == high:  # 1 operación
        return arr[low]  # 1 operación
```

```
    pivot_index = partition(arr, low, high)  # O(n)
    if k == pivot_index:  # 1 operación
        return arr[k]  # 1 operación
    elif k < pivot_index:  # 1 operación
        return quickselect_v2(arr, low, pivot_index - 1, k)  # recursión
    else:
        return quickselect_v2(arr, pivot_index + 1, high, k)  # recursión
# Peso total: Peor caso O(n^2), Promedio O(n)
```

10. BFS en grafo (listas adyacentes).

    a.

```
from collections import deque

def bfs(grafo, inicio):
    visitado = set()  # 1 operación
    cola = deque([inicio])  # 1 operación
    while cola:  # hasta recorrer todos los nodos → V iteraciones
        nodo = cola.popleft()  # 1 operación
        if nodo not in visitado:  # O(1) promedio
            visitado.add(nodo)  # O(1)
            for vecino in grafo[nodo]:  # iterar todos los vecinos → E itera
ciones en total
                if vecino not in visitado:  # O(1)
                    cola.append(vecino)  # O(1)
    return visitado  # 1 operación
# Peso total: O(V + E)
```

    b.

```
from collections import deque

def bfs_niveles(grafo, inicio):
    visitado = set()  # 1 operación
    cola = deque([(inicio, 0)])  # 1 operación
    niveles = {}  # 1 operación

    while cola:  # V iteraciones
```

```
        nodo, nivel = cola.popleft()  # 1 operación
        if nodo not in visitado:  # O(1)
            visitado.add(nodo)  # O(1)
            niveles[nodo] = nivel  # 1 operación
            for vecino in grafo[nodo]:  # iterar todos los vecinos → E itera
ciones
                if vecino not in visitado:  # O(1)
                    cola.append((vecino, nivel + 1))  # 1 operación
    return niveles  # 1 operación
# Peso total: O(V + E)
```