CÓDIGOS POR INSTANCIA DESARROLLADOS EN EL IDE SPYDER Y LENGUAJE DE PROGRAMACIÓN PYTHON

INSTANCIA 1

```
import random
import time

def generar_vector_indices(columnas, min_valor, max_valor):
    """
    Genera un vector de índices aleatorios entre min_valor y
max_valor,
    sin permitir valores repetidos en posiciones
consecutivas.
    """
    vector = []

    for _ in range(columnas):
        while True:
            nuevo_indice = random.randint(min_valor,
max_valor)
            # Verifica si el nuevo índice es diferente del
último índice agregado
            if not vector or nuevo_indice != vector[-1]:
                vector.append(nuevo_indice)
                break

    return vector

def evaluar_funcion_objetivo(matriz_datos, vector_indices):
    """
    Evalúa la función objetivo, que es la sumatoria de la
segunda columna de la matriz de datos.
    Aplica penalización si los índices consecutivos son
iguales o si no hay un valor igual a 5 en el vector.
    """
    suma_segunda_columna = sum(matriz_datos[indice - 1][1]
for indice in vector_indices)

    # Penalización si hay índices consecutivos iguales
    penalizacion_consecutivos = 50000000  # Un valor alto
para penalizar
    for i in range(1, len(vector_indices)):
        if vector_indices[i] == vector_indices[i - 1]:
            suma_segunda_columna -= penalizacion_consecutivos
```

```
    # Penalización si no hay al menos un valor igual a 5 en
el vector
    penalizacion_sin_cinco = 1000000  # Otro valor alto para
penalizar
    if 5 not in vector_indices:
        suma_segunda_columna -= penalizacion_sin_cinco

    return suma_segunda_columna

def verificar_restriccion(matriz_datos, vector_indices,
limite):
    """
    Verifica si la sumatoria de la primera columna de la
matriz de datos según los índices seleccionados
    cumple con la restricción de ser menor o igual a un
límite.
    """
    return sum(matriz_datos[indice - 1][0] for indice in
vector_indices) <= limite

def generar_memoria_armonica(tamano_memoria, matriz_datos,
columnas, limite):
    """
    Genera la memoria armónica inicial que cumple con las
restricciones.
    """
    memoria_armonica = []

    while len(memoria_armonica) < tamano_memoria:
        vector_indices = generar_vector_indices(columnas, 1,
5)
        if verificar_restriccion(matriz_datos,
vector_indices, limite):
            memoria_armonica.append(vector_indices)

    return memoria_armonica

def generar_nueva_armonia(memoria_armonica, matriz_datos,
HMCR, PAR, columnas, limite):
    """
    Genera una nueva armonía basada en las reglas de HMCR y
PAR.
    """
    nueva_armonia = []

    for i in range(columnas):
```

```
        if random.random() < HMCR:
            # Selecciona un valor de la memoria armónica
            valor_seleccionado =
random.choice(memoria_armonica)[i]
            if random.random() < PAR:
                # Ajuste de tono (cambiar ligeramente el
valor seleccionado)
                nuevo_valor = random.randint(1, 5)
                while nuevo_valor == valor_seleccionado or
(nueva_armonia and nuevo_valor == nueva_armonia[-1]):
                    nuevo_valor = random.randint(1, 5)
                nueva_armonia.append(nuevo_valor)
            else:
                # No se ajusta el tono, se mantiene el valor
seleccionado
                nueva_armonia.append(valor_seleccionado)
        else:
            # Genera un valor completamente aleatorio
            nuevo_valor = random.randint(1, 5)
            while nueva_armonia and nuevo_valor ==
nueva_armonia[-1]:
                nuevo_valor = random.randint(1, 5)
            nueva_armonia.append(nuevo_valor)

    # Verificar si la nueva armonía cumple con la restricción
    if verificar_restriccion(matriz_datos, nueva_armonia,
limite):
        return nueva_armonia
    else:
        return None

def busqueda_armonica(matriz_datos, HMS, iteraciones, HMCR,
PAR, columnas, limite):
    """
    Algoritmo principal de búsqueda armónica.
    """
    # Generar la memoria armónica inicial
    memoria_armonica = generar_memoria_armonica(HMS,
matriz_datos, columnas, limite)

    # Imprimir la memoria armónica inicial
    print("Memoria Armónica Inicial:")
    for vector in memoria_armonica:
        print(vector)
    print()  # Línea en blanco para separación

    # Iniciar el temporizador
```

```
    inicio_tiempo = time.time()

    for _ in range(iteraciones):
        nueva_armonia =
generar_nueva_armonia(memoria_armonica, matriz_datos, HMCR,
PAR, columnas, limite)
        if nueva_armonia:
            # Evaluar la nueva armonía
            valor_nueva_armonia =
evaluar_funcion_objetivo(matriz_datos, nueva_armonia)
            # Ordenar la memoria armónica por valor de la
función objetivo
            memoria_armonica.sort(key=lambda armonia:
evaluar_funcion_objetivo(matriz_datos, armonia))
            # Reemplazar la peor armonía si la nueva es mejor
            if valor_nueva_armonia >
evaluar_funcion_objetivo(matriz_datos, memoria_armonica[0]):
                memoria_armonica[0] = nueva_armonia

    # Detener el temporizador
    fin_tiempo = time.time()
    tiempo_total = fin_tiempo - inicio_tiempo

    # Ordenar la memoria armónica al final de las iteraciones
y seleccionar las 10 mejores
    memoria_armonica.sort(key=lambda armonia:
evaluar_funcion_objetivo(matriz_datos, armonia),
reverse=True)
    mejores_armonias = memoria_armonica[:10]

    # Imprimir las 10 mejores armonías con sus valores
objetivos
    print("Las 10 Mejores Armonías:")
    for i, armonia in enumerate(mejores_armonias, 1):
        valor_objetivo =
evaluar_funcion_objetivo(matriz_datos, armonia)
        print(f"{i}. Armonía: {armonia} - Valor Objetivo:
{valor_objetivo}")

    # Imprimir el tiempo total de computación
    print("\nEstadísticas del Tiempo de Computación:")
    print(f"Tiempo total de ejecución del proceso iterativo:
{tiempo_total:.4f} segundos")

    return mejores_armonias

# Matriz de datos proporcionada
```

```
matriz_datos = [
    [3, 1218346],
    [4, 1137315],
    [4, 10954047],
    [4, 5882428],
    [1, 0]
]

# Parámetros
HMS = 1000
iteraciones = 5000
HMCR = 0.9
PAR = 0.2
columnas = 4
limite = 12

# Ejecutar el algoritmo de búsqueda armónica
mejores_armonias = busqueda_armonica(matriz_datos, HMS,
iteraciones, HMCR, PAR, columnas, limite)
```

INSTANCIA 2

```
import random
import time

def generar_vector_indices(columnas, min_valor, max_valor):
    """
    Genera un vector de tamaño fijo 'columnas' con índices
aleatorios entre min_valor y max_valor,
    asegurando que el valor '5' esté presente exactamente una
vez y no haya repetidos consecutivos,
    excepto para el valor '5', que sí puede ser consecutivo.
    """
    while True:
        vector = []
        contiene_cinco = False  # Bandera para asegurar que
hay un '5' en el vector

        for i in range(columnas):
            intentos = 0
            while True:
```

```
                nuevo_indice = random.randint(min_valor,
max_valor)
                # Verifica si el nuevo índice es diferente
del último índice agregado, a menos que sea '5'
                if not vector or nuevo_indice == 5 or
nuevo_indice != vector[-1]:
                    # Asegura que solo haya un '5' si ya
existe en el vector
                    if nuevo_indice == 5 and contiene_cinco
and vector[-1] != 5:
                        continue
                    vector.append(nuevo_indice)
                    if nuevo_indice == 5:
                        contiene_cinco = True
                    break

                intentos += 1
                # Si hay muchos intentos fallidos, reiniciar
la generación del vector completo
                if intentos > 10:
                    break

            # Si se superan los intentos, reiniciar la
generación del vector
            if intentos > 10:
                break

        # Verifica si el vector generado es válido y contiene
exactamente un '5'
        if len(vector) == columnas and vector.count(5) >= 1:
            return vector

def evaluar_funcion_objetivo(matriz_datos, vector_indices):
    """
    Evalúa la función objetivo, que es la sumatoria de la
segunda columna de la matriz de datos.
    Aplica penalización si los índices consecutivos son
iguales, excepto para el valor '5'.
    """
    suma_segunda_columna = sum(matriz_datos[indice - 1][1]
for indice in vector_indices)

    # Penalización si hay índices consecutivos iguales,
excepto para el valor '5'
```

```
    penalizacion_consecutivos = 5000
    for i in range(1, len(vector_indices)):
        if vector_indices[i] == vector_indices[i - 1] and
vector_indices[i] != 5:
            suma_segunda_columna -= penalizacion_consecutivos

    # Penalización si no hay al menos un valor igual a 5 en
el vector
    penalizacion_cinco = 5000
    if vector_indices.count(5) < 1:
        suma_segunda_columna -= penalizacion_cinco

    # Evitar que la función objetivo sea negativa
    return max(suma_segunda_columna, 0)

def verificar_restriccion(matriz_datos, vector_indices,
limite):
    """
    Verifica si la sumatoria de la primera columna de la
matriz de datos según los índices seleccionados
    cumple con la restricción de ser menor o igual a un
límite.
    """
    return sum(matriz_datos[indice - 1][0] for indice in
vector_indices) <= limite

def generar_memoria_armonica(tamano_memoria, matriz_datos,
columnas, limite):
    """
    Genera la memoria armónica inicial que cumple con las
restricciones.
    """
    memoria_armonica = []

    while len(memoria_armonica) < tamano_memoria:
        vector_indices = generar_vector_indices(columnas, 1,
len(matriz_datos))
        if vector_indices and
verificar_restriccion(matriz_datos, vector_indices, limite):
            memoria_armonica.append(vector_indices)

    return memoria_armonica
```

```
def generar_nueva_armonia(memoria_armonica, matriz_datos,
HMCR, PAR, columnas, limite):
    """
    Genera una nueva armonía basada en las reglas de HMCR y
PAR.
    """
    nueva_armonia = []
    contiene_cinco = False  # Bandera para asegurar que hay
un '5' en la armonía

    for i in range(columnas):
        if random.random() < HMCR and memoria_armonica:
            valor_seleccionado =
random.choice(memoria_armonica)[i]
            if random.random() < PAR:
                nuevo_valor = random.randint(1,
len(matriz_datos))
                while nuevo_valor == valor_seleccionado or
(nueva_armonia and nuevo_valor == nueva_armonia[-1] and
nuevo_valor != 5):
                    nuevo_valor = random.randint(1,
len(matriz_datos))
                nueva_armonia.append(nuevo_valor)
                if nuevo_valor == 5:
                    contiene_cinco = True
            else:
                nueva_armonia.append(valor_seleccionado)
                if valor_seleccionado == 5:
                    contiene_cinco = True
        else:
            nuevo_valor = random.randint(1,
len(matriz_datos))
            while nueva_armonia and nuevo_valor ==
nueva_armonia[-1] and nuevo_valor != 5:
                nuevo_valor = random.randint(1,
len(matriz_datos))
            nueva_armonia.append(nuevo_valor)
            if nuevo_valor == 5:
                contiene_cinco = True

    if verificar_restriccion(matriz_datos, nueva_armonia,
limite) and nueva_armonia.count(5) >= 1:
        return nueva_armonia
    else:
```

```
        return None

def busqueda_armonica(matriz_datos, HMS, iteraciones, HMCR,
PAR, columnas, limite):
    """
    Algoritmo principal de búsqueda armónica.
    """
    memoria_armonica = generar_memoria_armonica(HMS,
matriz_datos, columnas, limite)

    if not memoria_armonica:
        print("No se pudo generar una memoria armónica
inicial válida.")
        return None, None

    # Iniciar el temporizador
    inicio_tiempo = time.time()

    for _ in range(iteraciones):
        nueva_armonia =
generar_nueva_armonia(memoria_armonica, matriz_datos, HMCR,
PAR, columnas, limite)
        if nueva_armonia:
            valor_nueva_armonia =
evaluar_funcion_objetivo(matriz_datos, nueva_armonia)
            memoria_armonica.sort(key=lambda armonia:
evaluar_funcion_objetivo(matriz_datos, armonia))
            if valor_nueva_armonia >
evaluar_funcion_objetivo(matriz_datos, memoria_armonica[0]):
                memoria_armonica[0] = nueva_armonia

    # Detener el temporizador
    fin_tiempo = time.time()
    tiempo_total = fin_tiempo - inicio_tiempo

    memoria_armonica.sort(key=lambda armonia:
evaluar_funcion_objetivo(matriz_datos, armonia),
reverse=True)
    mejores_armonias = memoria_armonica[:10]
    mejores_valores = [evaluar_funcion_objetivo(matriz_datos,
armonia) for armonia in mejores_armonias]

    # Imprimir el tiempo total de computación
    print("\nEstadísticas del Tiempo de Computación:")
```

```
    print(f"Tiempo total de ejecución del proceso iterativo:
{tiempo_total:.4f} segundos")

    return mejores_armonias, mejores_valores

# Matriz de datos proporcionada
matriz_datos = [
    [3, 1218346],
    [4, 1137315],
    [4, 10954047],
    [4, 5882428],
    [1, 0],
    [6, 18238800]
]

# Parámetros
HMS = 1000
iteraciones = 5000
HMCR = 0.9
PAR = 0.2
columnas = 4
limite = 12

# Ejecutar el algoritmo de búsqueda armónica
mejores_armonias, mejores_valores =
busqueda_armonica(matriz_datos, HMS, iteraciones, HMCR, PAR,
columnas, limite)

# Mostrar las 10 mejores soluciones con sus valores de
función objetivo
if mejores_armonias and mejores_valores:
    print("\nLas 10 mejores soluciones y sus valores de
función objetivo:")
    for i, (armonia, valor) in
enumerate(zip(mejores_armonias, mejores_valores), start=1):
        print(f"Solución {i}: {armonia}, Valor Objetivo:
{valor}")
else:
    print("No se pudo encontrar ninguna solución válida.")
```

INSTANCIA 3

```
import random
import time

def generar_vector_indices(columnas, min_valor, max_valor):
    """
    Genera un vector de tamaño fijo 'columnas' con índices
aleatorios entre min_valor y max_valor,
    asegurando que los valores '5', '6' y '7' estén presentes
en el vector y que no haya repetidos consecutivos.
    """
    while True:
        vector = []
        contiene_valores = {5: False, 6: False, 7: False}  #
Bandera para asegurar que hay un '5', '6', y '7' en el vector

        for i in range(columnas):
            intentos = 0
            while True:
                nuevo_indice = random.randint(min_valor,
max_valor)
                # Verifica si el nuevo índice es diferente
del último índice agregado
                if not vector or nuevo_indice != vector[-1]:
                    vector.append(nuevo_indice)
                    if nuevo_indice in contiene_valores:
                        contiene_valores[nuevo_indice] = True
                    break

                intentos += 1
                # Si hay muchos intentos fallidos, reiniciar
la generación del vector completo
                if intentos > 10:
                    break

            # Si se superan los intentos, reiniciar la
generación del vector
            if intentos > 10:
                break

        # Verifica si el vector generado es válido, contiene
los valores 5, 6 y 7, y no tiene repetidos consecutivos
        if len(vector) == columnas and
all(contiene_valores.values()) and all(vector[i] != vector[i
+ 1] for i in range(len(vector) - 1)):
```

```
            return vector

def evaluar_funcion_objetivo(matriz_datos, vector_indices):
    """
    Evalúa la función objetivo, que es la sumatoria de la
segunda columna de la matriz de datos.
    Aplica penalización si los índices consecutivos son
iguales y si no están presentes los valores '5', '6' y '7'.
    """
    suma_segunda_columna = sum(matriz_datos[indice - 1][1]
for indice in vector_indices)

    # Penalización si hay índices consecutivos iguales
    penalizacion_consecutivos = 5000
    for i in range(1, len(vector_indices)):
        if vector_indices[i] == vector_indices[i - 1]:
            suma_segunda_columna -= penalizacion_consecutivos

    # Penalización si no están los valores 5, 6, y 7 en el
vector
    penalizacion_valores = 5000
    if not all(valor in vector_indices for valor in [5, 6,
7]):
        suma_segunda_columna -= penalizacion_valores

    # Evitar que la función objetivo sea negativa
    return max(suma_segunda_columna, 0)

def verificar_restriccion(matriz_datos, vector_indices,
limite):
    """
    Verifica si la sumatoria de la primera columna de la
matriz de datos según los índices seleccionados
    cumple con la restricción de ser menor o igual a un
límite y que no haya valores consecutivos iguales.
    """
    suma_primera_columna = sum(matriz_datos[indice - 1][0]
for indice in vector_indices)
    sin_consecutivos = all(vector_indices[i] !=
vector_indices[i + 1] for i in range(len(vector_indices) -
1))
    return suma_primera_columna <= limite and
sin_consecutivos
```

```
def generar_memoria_armonica(tamano_memoria, matriz_datos,
columnas, limite):
    """
    Genera la memoria armónica inicial que cumple con las
restricciones.
    """
    memoria_armonica = []

    while len(memoria_armonica) < tamano_memoria:
        vector_indices = generar_vector_indices(columnas, 1,
len(matriz_datos))
        if vector_indices and
verificar_restriccion(matriz_datos, vector_indices, limite):
            memoria_armonica.append(vector_indices)

    return memoria_armonica

def generar_nueva_armonia(memoria_armonica, matriz_datos,
HMCR, PAR, columnas, limite):
    """
    Genera una nueva armonía basada en las reglas de HMCR y
PAR.
    """
    nueva_armonia = []
    contiene_valores = {5: False, 6: False, 7: False}  #
Bandera para asegurar que hay un '5', '6', y '7' en la
armonía

    for i in range(columnas):
        if random.random() < HMCR and memoria_armonica:
            valor_seleccionado =
random.choice(memoria_armonica)[i]
            if random.random() < PAR:
                nuevo_valor = random.randint(1,
len(matriz_datos))
                while nuevo_valor == valor_seleccionado or
(nueva_armonia and nuevo_valor == nueva_armonia[-1]):
                    nuevo_valor = random.randint(1,
len(matriz_datos))
                nueva_armonia.append(nuevo_valor)
                if nuevo_valor in contiene_valores:
                    contiene_valores[nuevo_valor] = True
            else:
                nueva_armonia.append(valor_seleccionado)
```

```
                if valor_seleccionado in contiene_valores:
                    contiene_valores[valor_seleccionado] =
True
        else:
            nuevo_valor = random.randint(1,
len(matriz_datos))
            while nueva_armonia and nuevo_valor ==
nueva_armonia[-1]:
                nuevo_valor = random.randint(1,
len(matriz_datos))
            nueva_armonia.append(nuevo_valor)
            if nuevo_valor in contiene_valores:
                contiene_valores[nuevo_valor] = True

    if verificar_restriccion(matriz_datos, nueva_armonia,
limite) and all(contiene_valores.values()):
        return nueva_armonia
    else:
        return None

def busqueda_armonica(matriz_datos, HMS, iteraciones, HMCR,
PAR, columnas, limite):
    """
    Algoritmo principal de búsqueda armónica.
    """
    memoria_armonica = generar_memoria_armonica(HMS,
matriz_datos, columnas, limite)

    if not memoria_armonica:
        print("No se pudo generar una memoria armónica
inicial válida.")
        return None, None

    # Iniciar el temporizador
    inicio_tiempo = time.time()

    for _ in range(iteraciones):
        nueva_armonia =
generar_nueva_armonia(memoria_armonica, matriz_datos, HMCR,
PAR, columnas, limite)
        if nueva_armonia:
            valor_nueva_armonia =
evaluar_funcion_objetivo(matriz_datos, nueva_armonia)
```

```
            memoria_armonica.sort(key=lambda armonia:
evaluar_funcion_objetivo(matriz_datos, armonia))
            if valor_nueva_armonia >
evaluar_funcion_objetivo(matriz_datos, memoria_armonica[0]):
                memoria_armonica[0] = nueva_armonia

    # Detener el temporizador
    fin_tiempo = time.time()
    tiempo_total = fin_tiempo - inicio_tiempo

    memoria_armonica.sort(key=lambda armonia:
evaluar_funcion_objetivo(matriz_datos, armonia),
reverse=True)
    mejores_armonias = memoria_armonica[:10]
    mejores_valores = [evaluar_funcion_objetivo(matriz_datos,
armonia) for armonia in mejores_armonias]

    # Imprimir el tiempo total de computación
    print("\nEstadísticas del Tiempo de Computación:")
    print(f"Tiempo total de ejecución del proceso iterativo:
{tiempo_total:.4f} segundos")

    return mejores_armonias, mejores_valores

# Nueva matriz de datos proporcionada
matriz_datos = [
    [3, 1218346],
    [4, 1137315],
    [4, 10954047],
    [4, 5882428],
    [1, 0],
    [1, 0],
    [1, 0],
]

# Parámetros
HMS = 1000
iteraciones = 5000
HMCR = 0.9
PAR = 0.2
columnas = 8
limite = 24

# Ejecutar el algoritmo de búsqueda armónica
```

```
mejores_armonias, mejores_valores =
busqueda_armonica(matriz_datos, HMS, iteraciones, HMCR, PAR,
columnas, limite)

# Mostrar las 10 mejores soluciones con sus valores de
función objetivo
if mejores_armonias and mejores_valores:
    print("\nLas 10 mejores armonias y sus valores de función
objetivo:")
    for i, (armonia, valor) in
enumerate(zip(mejores_armonias, mejores_valores), start=1):
        print(f"Armonía {i}: {armonia}, Valor Objetivo:
{valor}")
else:
    print("No se pudo encontrar ninguna solución válida.")
```

INSTANCIA 4

```
import random
import time

def generar_vector_indices(columnas, min_valor, max_valor):
    """
    Genera un vector de tamaño fijo 'columnas' con índices
aleatorios entre min_valor y max_valor,
    asegurando que los valores '6', '7' y '8' estén presentes
en el vector y que no haya repetidos consecutivos.
    """
    while True:
        vector = []
        contiene_valores = {6: False, 7: False, 8: False}  #
Bandera para asegurar que hay un '6', '7', y '8' en el vector

        for i in range(columnas):
            intentos = 0
            while True:
                nuevo_indice = random.randint(min_valor,
max_valor)
                # Verifica si el nuevo índice es diferente
del último índice agregado
                if not vector or nuevo_indice != vector[-1]:
```

```
                    vector.append(nuevo_indice)
                    if nuevo_indice in contiene_valores:
                        contiene_valores[nuevo_indice] = True
                    break

                intentos += 1
                # Si hay muchos intentos fallidos, reiniciar
la generación del vector completo
                if intentos > 10:
                    break

            # Si se superan los intentos, reiniciar la
generación del vector
            if intentos > 10:
                break

        # Verifica si el vector generado es válido y contiene
los valores 6, 7 y 8
        if len(vector) == columnas and
all(contiene_valores.values()):
            return vector

def evaluar_funcion_objetivo(matriz_datos, vector_indices):
    """
    Evalúa la función objetivo, que es la sumatoria de la
segunda columna de la matriz de datos.
    Aplica penalización si los índices consecutivos son
iguales y si no están presentes los valores '6', '7' y '8'.
    """
    suma_segunda_columna = sum(matriz_datos[indice - 1][1]
for indice in vector_indices)

    # Penalización si hay índices consecutivos iguales
    penalizacion_consecutivos = 5000
    for i in range(1, len(vector_indices)):
        if vector_indices[i] == vector_indices[i - 1]:
            suma_segunda_columna -= penalizacion_consecutivos

    # Penalización si no están los valores 6, 7, y 8 en el
vector
    penalizacion_valores = 5000
    if not all(valor in vector_indices for valor in [6, 7,
8]):
        suma_segunda_columna -= penalizacion_valores
```

```
    # Evitar que la función objetivo sea negativa
    return max(suma_segunda_columna, 0)

def verificar_restriccion(matriz_datos, vector_indices,
limite):
    """
    Verifica si la sumatoria de la primera columna de la
matriz de datos según los índices seleccionados
    cumple con la restricción de ser menor o igual a un
límite.
    """
    return sum(matriz_datos[indice - 1][0] for indice in
vector_indices) <= limite

def generar_memoria_armonica(tamano_memoria, matriz_datos,
columnas, limite):
    """
    Genera la memoria armónica inicial que cumple con las
restricciones.
    """
    memoria_armonica = []

    while len(memoria_armonica) < tamano_memoria:
        vector_indices = generar_vector_indices(columnas, 1,
len(matriz_datos))
        if vector_indices and
verificar_restriccion(matriz_datos, vector_indices, limite):
            memoria_armonica.append(vector_indices)

    return memoria_armonica

def generar_nueva_armonia(memoria_armonica, matriz_datos,
HMCR, PAR, columnas, limite):
    """
    Genera una nueva armonía basada en las reglas de HMCR y
PAR.
    """
    nueva_armonia = []
    contiene_valores = {6: False, 7: False, 8: False}  #
Bandera para asegurar que hay un '6', '7', y '8' en la
armonía

    for i in range(columnas):
```

```
        if random.random() < HMCR and memoria_armonica:
            valor_seleccionado =
random.choice(memoria_armonica)[i]
            if random.random() < PAR:
                nuevo_valor = random.randint(1,
len(matriz_datos))
                while nuevo_valor == valor_seleccionado or
(nueva_armonia and nuevo_valor == nueva_armonia[-1]):
                    nuevo_valor = random.randint(1,
len(matriz_datos))
                nueva_armonia.append(nuevo_valor)
                if nuevo_valor in contiene_valores:
                    contiene_valores[nuevo_valor] = True
            else:
                nueva_armonia.append(valor_seleccionado)
                if valor_seleccionado in contiene_valores:
                    contiene_valores[valor_seleccionado] =
True
        else:
            nuevo_valor = random.randint(1,
len(matriz_datos))
            while nueva_armonia and nuevo_valor ==
nueva_armonia[-1]:
                nuevo_valor = random.randint(1,
len(matriz_datos))
            nueva_armonia.append(nuevo_valor)
            if nuevo_valor in contiene_valores:
                contiene_valores[nuevo_valor] = True

    if verificar_restriccion(matriz_datos, nueva_armonia,
limite) and all(contiene_valores.values()):
        return nueva_armonia
    else:
        return None

def busqueda_armonica(matriz_datos, HMS, iteraciones, HMCR,
PAR, columnas, limite):
    """
    Algoritmo principal de búsqueda armónica.
    """
    memoria_armonica = generar_memoria_armonica(HMS,
matriz_datos, columnas, limite)

    if not memoria_armonica:
```

```
        print("No se pudo generar una memoria armónica
inicial válida.")
        return None, None

    # Iniciar el temporizador
    inicio_tiempo = time.time()

    for _ in range(iteraciones):
        nueva_armonia =
generar_nueva_armonia(memoria_armonica, matriz_datos, HMCR,
PAR, columnas, limite)
        if nueva_armonia:
            valor_nueva_armonia =
evaluar_funcion_objetivo(matriz_datos, nueva_armonia)
            memoria_armonica.sort(key=lambda armonia:
evaluar_funcion_objetivo(matriz_datos, armonia))
            if valor_nueva_armonia >
evaluar_funcion_objetivo(matriz_datos, memoria_armonica[0]):
                memoria_armonica[0] = nueva_armonia

    # Detener el temporizador
    fin_tiempo = time.time()
    tiempo_total = fin_tiempo - inicio_tiempo

    memoria_armonica.sort(key=lambda armonia:
evaluar_funcion_objetivo(matriz_datos, armonia),
reverse=True)
    mejores_armonias = memoria_armonica[:10]
    mejores_valores = [evaluar_funcion_objetivo(matriz_datos,
armonia) for armonia in mejores_armonias]

    # Imprimir el tiempo total de computación
    print("\nEstadísticas del Tiempo de Computación:")
    print(f"Tiempo total de ejecución del proceso iterativo:
{tiempo_total:.4f} segundos")

    return mejores_armonias, mejores_valores

# Matriz de datos proporcionada
matriz_datos = [
    [3, 1218346],
    [4, 1137315],
    [4, 10954047],
    [4, 5882428],
```

```
    [6, 18238800],
    [1, 0],
    [1, 0],
    [1, 0],
]

# Parámetros
HMS = 1000
iteraciones = 5000
HMCR = 0.9
PAR = 0.2
columnas = 7
limite = 24

# Ejecutar el algoritmo de búsqueda armónica
mejores_armonias, mejores_valores =
busqueda_armonica(matriz_datos, HMS, iteraciones, HMCR, PAR,
columnas, limite)

# Mostrar las 10 mejores soluciones con sus valores de
función objetivo
if mejores_armonias and mejores_valores:
    print("\nLas 10 mejores armonias y sus valores de función
objetivo:")
    for i, (armonia, valor) in
enumerate(zip(mejores_armonias, mejores_valores), start=1):
        print(f"Solución {i}: {armonia}, Valor Objetivo:
{valor}")
else:
    print("No se pudo encontrar ninguna solución válida.")
```

INSTANCIA 5

```
import random
import time

def generar_vector_indices(columnas, min_valor, max_valor):
    """
    Genera un vector de tamaño fijo 'columnas' con índices
aleatorios entre min_valor y max_valor,
    asegurando que los valores '7', '8' y '9' estén presentes
en el vector y que no haya repetidos consecutivos.
```

```
    """
    while True:
        vector = []
        contiene_valores = {7: False, 8: False, 9: False}  # 
Bandera para asegurar que hay un '7', '8', y '9' en el vector

        for i in range(columnas):
            intentos = 0
            while True:
                nuevo_indice = random.randint(min_valor, 
max_valor)
                # Verifica si el nuevo índice es diferente 
del último índice agregado
                if not vector or nuevo_indice != vector[-1]:
                    vector.append(nuevo_indice)
                    if nuevo_indice in contiene_valores:
                        contiene_valores[nuevo_indice] = True
                    break

                intentos += 1
                # Si hay muchos intentos fallidos, reiniciar 
la generación del vector completo
                if intentos > 10:
                    break

            # Si se superan los intentos, reiniciar la 
generación del vector
            if intentos > 10:
                break

        # Verifica si el vector generado es válido, contiene 
los valores 7, 8 y 9, y no tiene repetidos consecutivos
        if len(vector) == columnas and 
all(contiene_valores.values()) and all(vector[i] != vector[i 
+ 1] for i in range(len(vector) - 1)):
            return vector

def evaluar_funcion_objetivo(matriz_datos, vector_indices):
    """
    Evalúa la función objetivo, que es la sumatoria de la 
segunda columna de la matriz de datos.
    Aplica penalización si los índices consecutivos son 
iguales y si no están presentes los valores '7', '8' y '9'.
    """
```

```
    suma_segunda_columna = sum(matriz_datos[indice - 1][1]
for indice in vector_indices)

    # Penalización si hay índices consecutivos iguales
    penalizacion_consecutivos = 5000
    for i in range(1, len(vector_indices)):
        if vector_indices[i] == vector_indices[i - 1]:
            suma_segunda_columna -= penalizacion_consecutivos

    # Penalización si no están los valores 7, 8, y 9 en el
vector
    penalizacion_valores = 5000
    if not all(valor in vector_indices for valor in [7, 8,
9]):
        suma_segunda_columna -= penalizacion_valores

    # Evitar que la función objetivo sea negativa
    return max(suma_segunda_columna, 0)

def verificar_restriccion(matriz_datos, vector_indices,
limite):
    """
    Verifica si la sumatoria de la primera columna de la
matriz de datos según los índices seleccionados
    cumple con la restricción de ser menor o igual a un
límite y que no haya valores consecutivos iguales.
    """
    # Verificar la suma de la primera columna
    suma_primera_columna = sum(matriz_datos[indice - 1][0]
for indice in vector_indices)
    # Verificar que no haya valores consecutivos iguales
    sin_consecutivos = all(vector_indices[i] !=
vector_indices[i + 1] for i in range(len(vector_indices) -
1))
    return suma_primera_columna <= limite and
sin_consecutivos

def generar_memoria_armonica(tamano_memoria, matriz_datos,
columnas, limite):
    """
    Genera la memoria armónica inicial que cumple con las
restricciones.
    """
    memoria_armonica = []
```

```
    while len(memoria_armonica) < tamano_memoria:
        vector_indices = generar_vector_indices(columnas, 1,
len(matriz_datos))
        if vector_indices and
verificar_restriccion(matriz_datos, vector_indices, limite):
            memoria_armonica.append(vector_indices)

    return memoria_armonica

def generar_nueva_armonia(memoria_armonica, matriz_datos,
HMCR, PAR, columnas, limite):
    """
    Genera una nueva armonía basada en las reglas de HMCR y
PAR.
    """
    nueva_armonia = []
    contiene_valores = {7: False, 8: False, 9: False}  #
Bandera para asegurar que hay un '7', '8', y '9' en la
armonía

    for i in range(columnas):
        if random.random() < HMCR and memoria_armonica:
            valor_seleccionado =
random.choice(memoria_armonica)[i]
            if random.random() < PAR:
                nuevo_valor = random.randint(1,
len(matriz_datos))
                while nuevo_valor == valor_seleccionado or
(nueva_armonia and nuevo_valor == nueva_armonia[-1]):
                    nuevo_valor = random.randint(1,
len(matriz_datos))
                nueva_armonia.append(nuevo_valor)
                if nuevo_valor in contiene_valores:
                    contiene_valores[nuevo_valor] = True
            else:
                nueva_armonia.append(valor_seleccionado)
                if valor_seleccionado in contiene_valores:
                    contiene_valores[valor_seleccionado] =
True
        else:
            nuevo_valor = random.randint(1,
len(matriz_datos))
```

```
            while nueva_armonia and nuevo_valor ==
nueva_armonia[-1]:
                nuevo_valor = random.randint(1,
len(matriz_datos))
            nueva_armonia.append(nuevo_valor)
            if nuevo_valor in contiene_valores:
                contiene_valores[nuevo_valor] = True

    if verificar_restriccion(matriz_datos, nueva_armonia,
limite) and all(contiene_valores.values()):
        return nueva_armonia
    else:
        return None

def busqueda_armonica(matriz_datos, HMS, iteraciones, HMCR,
PAR, columnas, limite):
    """
    Algoritmo principal de búsqueda armónica.
    """
    memoria_armonica = generar_memoria_armonica(HMS,
matriz_datos, columnas, limite)

    if not memoria_armonica:
        print("No se pudo generar una memoria armónica
inicial válida.")
        return None, None

    # Iniciar el temporizador
    inicio_tiempo = time.time()

    for _ in range(iteraciones):
        nueva_armonia =
generar_nueva_armonia(memoria_armonica, matriz_datos, HMCR,
PAR, columnas, limite)
        if nueva_armonia:
            valor_nueva_armonia =
evaluar_funcion_objetivo(matriz_datos, nueva_armonia)
            memoria_armonica.sort(key=lambda armonia:
evaluar_funcion_objetivo(matriz_datos, armonia))
            if valor_nueva_armonia >
evaluar_funcion_objetivo(matriz_datos, memoria_armonica[0]):
                memoria_armonica[0] = nueva_armonia

    # Detener el temporizador
```

```
    fin_tiempo = time.time()
    tiempo_total = fin_tiempo - inicio_tiempo

    memoria_armonica.sort(key=lambda armonia:
evaluar_funcion_objetivo(matriz_datos, armonia),
reverse=True)
    mejores_armonias = memoria_armonica[:10]
    mejores_valores = [evaluar_funcion_objetivo(matriz_datos,
armonia) for armonia in mejores_armonias]

    # Imprimir el tiempo total de computación
    print("\nEstadísticas del Tiempo de Computación:")
    print(f"Tiempo total de ejecución del proceso iterativo:
{tiempo_total:.4f} segundos")

    return mejores_armonias, mejores_valores

# Nueva matriz de datos proporcionada
matriz_datos = [
    [3, 1218346],
    [4, 1137315],
    [4, 10954047],
    [4, 5882428],
    [6, 18238800],
    [6, 34436943],
    [1, 0],
    [1, 0],
    [1, 0],
]

# Parámetros
HMS = 1000
iteraciones = 5000
HMCR = 0.9
PAR = 0.2
columnas = 7
limite = 24

# Ejecutar el algoritmo de búsqueda armónica
mejores_armonias, mejores_valores =
busqueda_armonica(matriz_datos, HMS, iteraciones, HMCR, PAR,
columnas, limite)
# Mostrar las 10 mejores soluciones con sus valores de
función objetivo
```

```
if mejores_armonias and mejores_valores:
    print("\nLas 10 mejores soluciones y sus valores de
función objetivo:")
    for i, (armonia, valor) in
enumerate(zip(mejores_armonias, mejores_valores), start=1):
        print(f"Solución {i}: {armonia}, Valor Objetivo:
{valor}")
else:
    print("No se pudo encontrar ninguna solución válida.")
```

INSTANCIA 6

```
import random
import time

def generar_vector_indices(columnas, min_valor, max_valor):
    """
    Genera un vector de tamaño fijo 'columnas' con índices
aleatorios entre min_valor y max_valor,
    asegurando que los valores '9', '10' y '11' estén
presentes en el vector y que no haya repetidos consecutivos.
    """
    while True:
        vector = []
        contiene_valores = {9: False, 10: False, 11: False}
# Bandera para asegurar que hay un '9', '10', y '11' en el
vector

        for i in range(columnas):
            intentos = 0
            while True:
                nuevo_indice = random.randint(min_valor,
max_valor)
                # Verifica si el nuevo índice es diferente
del último índice agregado
                if not vector or nuevo_indice != vector[-1]:
                    vector.append(nuevo_indice)
                    if nuevo_indice in contiene_valores:
                        contiene_valores[nuevo_indice] = True
                    break

                intentos += 1
```

```
                # Si hay muchos intentos fallidos, reiniciar
la generación del vector completo
                if intentos > 10:
                    break

            # Si se superan los intentos, reiniciar la
generación del vector
            if intentos > 10:
                break

        # Verifica si el vector generado es válido, contiene
los valores 9, 10 y 11, y no tiene repetidos consecutivos
        if len(vector) == columnas and
all(contiene_valores.values()) and all(vector[i] != vector[i
+ 1] for i in range(len(vector) - 1)):
            return vector

def evaluar_funcion_objetivo(matriz_datos, vector_indices):
    """
    Evalúa la función objetivo, que es la sumatoria de la
segunda columna de la matriz de datos.
    Aplica penalización si los índices consecutivos son
iguales y si no están presentes los valores '9', '10' y '11'.
    """
    suma_segunda_columna = sum(matriz_datos[indice - 1][1]
for indice in vector_indices)

    # Penalización si hay índices consecutivos iguales
    penalizacion_consecutivos = 50000000
    for i in range(1, len(vector_indices)):
        if vector_indices[i] == vector_indices[i - 1]:
            suma_segunda_columna -= penalizacion_consecutivos

    # Penalización si no están los valores 9, 10, y 11 en el
vector
    penalizacion_valores = 50000000
    if not all(valor in vector_indices for valor in [9, 10,
11]):
        suma_segunda_columna -= penalizacion_valores

    # Evitar que la función objetivo sea negativa
    return max(suma_segunda_columna, 0)
```

```
def verificar_restriccion(matriz_datos, vector_indices,
limite):
    """
    Verifica si la sumatoria de la primera columna de la
matriz de datos según los índices seleccionados
    cumple con la restricción de ser menor o igual a un
límite y que no haya valores consecutivos iguales.
    """
    # Verificar la suma de la primera columna
    suma_primera_columna = sum(matriz_datos[indice - 1][0]
for indice in vector_indices)
    # Verificar que no haya valores consecutivos iguales
    sin_consecutivos = all(vector_indices[i] !=
vector_indices[i + 1] for i in range(len(vector_indices) -
1))
    return suma_primera_columna <= limite and
sin_consecutivos

def generar_memoria_armonica(tamano_memoria, matriz_datos,
columnas, limite):
    """
    Genera la memoria armónica inicial que cumple con las
restricciones.
    """
    memoria_armonica = []

    while len(memoria_armonica) < tamano_memoria:
        vector_indices = generar_vector_indices(columnas, 1,
len(matriz_datos))
        if vector_indices and
verificar_restriccion(matriz_datos, vector_indices, limite):
            memoria_armonica.append(vector_indices)

    return memoria_armonica

def generar_nueva_armonia(memoria_armonica, matriz_datos,
HMCR, PAR, columnas, limite):
    """
    Genera una nueva armonía basada en las reglas de HMCR y
PAR.
    """
    nueva_armonia = []
```

```
    contiene_valores = {9: False, 10: False, 11: False}  #
Bandera para asegurar que hay un '9', '10', y '11' en la
armonía

    for i in range(columnas):
        if random.random() < HMCR and memoria_armonica:
            valor_seleccionado =
random.choice(memoria_armonica)[i]
            if random.random() < PAR:
                nuevo_valor = random.randint(1,
len(matriz_datos))
                while nuevo_valor == valor_seleccionado or
(nueva_armonia and nuevo_valor == nueva_armonia[-1]):
                    nuevo_valor = random.randint(1,
len(matriz_datos))
                nueva_armonia.append(nuevo_valor)
                if nuevo_valor in contiene_valores:
                    contiene_valores[nuevo_valor] = True
            else:
                nueva_armonia.append(valor_seleccionado)
                if valor_seleccionado in contiene_valores:
                    contiene_valores[valor_seleccionado] =
True
        else:
            nuevo_valor = random.randint(1,
len(matriz_datos))
            while nueva_armonia and nuevo_valor ==
nueva_armonia[-1]:
                nuevo_valor = random.randint(1,
len(matriz_datos))
            nueva_armonia.append(nuevo_valor)
            if nuevo_valor in contiene_valores:
                contiene_valores[nuevo_valor] = True

    if verificar_restriccion(matriz_datos, nueva_armonia,
limite) and all(contiene_valores.values()):
        return nueva_armonia
    else:
        return None

def busqueda_armonica(matriz_datos, HMS, iteraciones, HMCR,
PAR, columnas, limite):
    """
    Algoritmo principal de búsqueda armónica.
```

```
    """
    memoria_armonica = generar_memoria_armonica(HMS,
matriz_datos, columnas, limite)

    if not memoria_armonica:
        print("No se pudo generar una memoria armónica
inicial válida.")
        return None, None

    # Iniciar el temporizador
    inicio_tiempo = time.time()

    for _ in range(iteraciones):
        nueva_armonia =
generar_nueva_armonia(memoria_armonica, matriz_datos, HMCR,
PAR, columnas, limite)
        if nueva_armonia:
            valor_nueva_armonia =
evaluar_funcion_objetivo(matriz_datos, nueva_armonia)
            memoria_armonica.sort(key=lambda armonia:
evaluar_funcion_objetivo(matriz_datos, armonia))
            if valor_nueva_armonia >
evaluar_funcion_objetivo(matriz_datos, memoria_armonica[0]):
                memoria_armonica[0] = nueva_armonia

    # Detener el temporizador
    fin_tiempo = time.time()
    tiempo_total = fin_tiempo - inicio_tiempo

    memoria_armonica.sort(key=lambda armonia:
evaluar_funcion_objetivo(matriz_datos, armonia),
reverse=True)
    mejores_armonias = memoria_armonica[:10]
    mejores_valores = [evaluar_funcion_objetivo(matriz_datos,
armonia) for armonia in mejores_armonias]

    # Imprimir el tiempo total de computación
    print("\nEstadísticas del Tiempo de Computación:")
    print(f"Tiempo total de ejecución del proceso iterativo:
{tiempo_total:.4f} segundos")

    return mejores_armonias, mejores_valores

# Nueva matriz de datos proporcionada
```

```
matriz_datos = [
    [3, 1218346],
    [4, 1137315],
    [4, 10954047],
    [4, 5882428],
    [6, 18238800],
    [6, 34436943],
    [5, 910497],
    [5, 861891],
    [1, 0],
    [1, 0],
    [1, 0],
]

# Parámetros
HMS = 1000
iteraciones = 5000
HMCR = 0.9
PAR = 0.2
columnas = 7
limite = 24

# Ejecutar el algoritmo de búsqueda armónica
mejores_armonias, mejores_valores =
busqueda_armonica(matriz_datos, HMS, iteraciones, HMCR, PAR,
columnas, limite)

# Mostrar las 10 mejores soluciones con sus valores de
función objetivo
if mejores_armonias and mejores_valores:
    print("\nLas 10 mejores soluciones y sus valores de
función objetivo:")
    for i, (armonia, valor) in
enumerate(zip(mejores_armonias, mejores_valores), start=1):
        print(f"Solución {i}: {armonia}, Valor Objetivo:
{valor}")
else:
    print("No se pudo encontrar ninguna solución válida.")
```