Open in Colab

AG2 - Actividad Guiada 2

Nombre: Johnny Andres Illescas Fernandez
Link: https://colab.research.google.com/drive/1FpirdSakwX4LYAat3PTOvICdzzUoaVq8?usp=sharing
Github: https://github.com/AndresGin/VIU_Algoritmos_de_optimizacion/tree/main/AG2

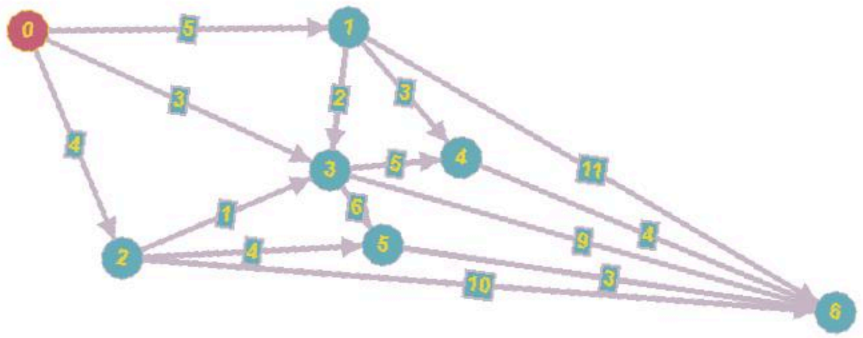
```
import math
```

Programación Dinámica. Viaje por el rio

- **Definición**: Es posible dividir el problema en subproblemas más pequeños, guardando las soluciones para ser utilizadas más adelante.
- **Características** que permiten identificar problemas aplicables:
  -Es posible almacenar soluciones de los subproblemas para ser utilizados más adelante
  -Debe verificar el principio de optimalidad de Bellman: "en una secuencia optima de decisiones, toda sub-secuencia también es óptima" (*)
  -La necesidad de guardar la información acerca de las soluciones parciales unido a la recursividad provoca la necesidad de preocuparnos por la complejidad espacial (cuantos recursos de espacio usaremos)

Problema

En un río hay **n** embarcaderos y debemos desplazarnos río abajo desde un embarcadero a otro. Cada embarcadero tiene precios diferentes para ir de un embarcadero a otro situado más abajo. Para ir del embarcadero i al j, puede ocurrir que sea más barato hacer un trasbordo por un embarcadero intermedio k. El problema consiste en determinar la combinación más barata.



*Consideramos una tabla TARIFAS(i,j) para almacenar todos los precios que nos ofrecen los embarcaderos.
*Si no es posible ir desde i a j daremos un valor alto para garantizar que ese trayecto no se va a elegir en la ruta óptima(modelado habitual para restricciones)

```
#Viaje por el rio - Programación dinámica
################################################################################

TARIFAS = [
[0,5,4,3,float("inf"),999,999],   #desde nodo 0
[999,0,999,2,3,999,11],  #desde nodo 1
[999,999, 0,1,999,4,10], #desde nodo 2
[999,999,999, 0,5,6,9],
[999,999, 999,999,0,999,4],
[999,999, 999,999,999,0,3],
[999,999,999,999,999,999,0]
]

#999 se puede sustituir por float("inf") del modulo math
TARIFAS
```

```
[[0, 5, 4, 3, inf, 999, 999],
 [999, 0, 999, 2, 3, 999, 11],
 [999, 999, 0, 1, 999, 4, 10],
 [999, 999, 999, 0, 5, 6, 9],
 [999, 999, 999, 999, 0, 999, 4],
```

```
      [999, 999, 999, 999, 999, 0, 3],
      [999, 999, 999, 999, 999, 999, 0]]


#Calculo de la matriz de PRECIOS y RUTAS
#  PRECIOS - contiene la matriz del mejor precio para ir de un nodo a otro
#  RUTAS   - contiene los nodos intermedios para ir de un nodo a otro
################################################################
def Precios(TARIFAS):
################################################################
  #Total de Nodos
  N = len(TARIFAS[0])

  #Inicialización de la tabla de precios
  PRECIOS = [ [9999]*N for i in [9999]*N]  #n x n
  RUTA = [ [""]*N for i in [""]*N]

  #Se recorren todos los nodos con dos bucles(origen - destino)
  #  para ir construyendo la matriz de PRECIOS
  for i in range(N-1):
    for j in range(i+1, N):
      MIN = TARIFAS[i][j]
      RUTA[i][j] = i

      for k in range(i, j):
        if PRECIOS[i][k] + TARIFAS[k][j] < MIN:
            MIN = min(MIN, PRECIOS[i][k] + TARIFAS[k][j] )
            RUTA[i][j] = k
        PRECIOS[i][j] = MIN

  return PRECIOS,RUTA


PRECIOS,RUTA = Precios(TARIFAS)
#print(PRECIOS[0][6])

print("PRECIOS")
for i in range(len(TARIFAS)):
  print(PRECIOS[i])

print("\nRUTA")
for i in range(len(TARIFAS)):
  print(RUTA[i])
```

```
PRECIOS
[9999, 5, 4, 3, 8, 8, 11]
[9999, 9999, 999, 2, 3, 8, 7]
[9999, 9999, 9999, 1, 6, 4, 7]
[9999, 9999, 9999, 9999, 5, 6, 9]
[9999, 9999, 9999, 9999, 9999, 999, 4]
[9999, 9999, 9999, 9999, 9999, 9999, 3]
[9999, 9999, 9999, 9999, 9999, 9999, 9999]

RUTA
['', 0, 0, 0, 1, 2, 5]
['', '', 1, 1, 1, 3, 4]
['', '', '', 2, 3, 2, 5]
['', '', '', '', 3, 3, 3]
['', '', '', '', '', 4, 4]
['', '', '', '', '', '', 5]
['', '', '', '', '', '', '']
```

```
#Calculo de la ruta usando la matriz RUTA
def calcular_ruta(RUTA, desde, hasta):
  if desde == RUTA[desde][hasta]:
  #if desde == hasta:
    #print("Ir a :" + str(desde))
    return desde
  else:
    return str(calcular_ruta(RUTA, desde, RUTA[desde][hasta])) +  ',' + str(RUTA[desde][hasta])

print("\nLa ruta es:")
calcular_ruta(RUTA, 0,6)
```

```
La ruta es:
'0,2,5'
```

Double-click (or enter) to edit

Problema de Asignacion de tarea

```
#Asignacion de tareas - Ramificación y Poda
################################################################################
#    T A R E A
#   A
#   G
#   E
#   N
#   T
#   E

COSTES=[[11,12,18,40],
        [14,15,13,22],
        [11,17,19,23],
        [17,14,20,28]]



#Calculo del valor de una solucion parcial
def valor(S,COSTES):
  VALOR = 0
  for i in range(len(S)):
    VALOR += COSTES[S[i]][i]
  return VALOR



valor((3,2,  ),COSTES)
```

```
34
```

```
#Coste inferior para soluciones parciales
#  (1,3,) Se asigna la tarea 1 al agente 0 y la tarea 3 al agente 1

def CI(S,COSTES):
  VALOR = 0
  #Valores establecidos
  for i in range(len(S)):
    VALOR += COSTES[i][S[i]]

  #Estimacion
  for i in range( len(S), len(COSTES)   ):
    VALOR += min( [ COSTES[j][i] for j in range(len(S), len(COSTES))  ])
  return VALOR

def CS(S,COSTES):
  VALOR = 0
  #Valores establecidos
  for i in range(len(S)):
    VALOR += COSTES[i][S[i]]

  #Estimacion
  for i in range( len(S), len(COSTES)   ):
    VALOR += max( [ COSTES[j][i] for j in range(len(S), len(COSTES))  ])
  return VALOR

CI((0,1),COSTES)
```

```
68
```

```
#Genera tantos hijos como como posibilidades haya para la siguiente elemento de la tupla
#(0,) -> (0,1), (0,2), (0,3)
def crear_hijos(NODO, N):
  HIJOS = []
  for i in range(N ):
    if i not in NODO:
      HIJOS.append({'s':NODO +(i,)    })
  return HIJOS


crear_hijos((0,) , 4)
```

```
[{'s': (0, 1)}, {'s': (0, 2)}, {'s': (0, 3)}]
```

```
def ramificacion_y_poda(COSTES):
#Construccion iterativa de soluciones(arbol). En cada etapa asignamos un agente(ramas).
#Nodos del grafo  { s:(1,2),CI:3,CS:5  }
  #print(COSTES)
  DIMENSION = len(COSTES)
  MEJOR_SOLUCION=tuple( i for i in range(len(COSTES)) )
  CotaSup = valor(MEJOR_SOLUCION,COSTES)
```

```
  #print("Cota Superior:", CotaSup)

  NODOS=[]
  NODOS.append({'s':(), 'ci':CI((),COSTES)    } )

  iteracion = 0

  while( len(NODOS) > 0):
    iteracion +=1

    nodo_prometedor = [ min(NODOS, key=lambda x:x['ci']) ][0]['s']
    #print("Nodo prometedor:", nodo_prometedor)

    #Ramificacion
    #Se generan los hijos
    HIJOS =[ {'s':x['s'], 'ci':CI(x['s'], COSTES)   } for x in crear_hijos(nodo_prometedor, DIMENSION) ]

    #Revisamos la cota superior y nos quedamos con la mejor solucion si llegamos a una solucion final
    NODO_FINAL = [x for x in HIJOS if len(x['s']) == DIMENSION  ]
    if len(NODO_FINAL ) >0:
      #print("\n********Soluciones:",  [x for x in HIJOS if len(x['s']) == DIMENSION  ] )
      if NODO_FINAL[0]['ci'] < CotaSup:
        CotaSup = NODO_FINAL[0]['ci']
        MEJOR_SOLUCION = NODO_FINAL

    #Poda
    HIJOS = [x for x in HIJOS if x['ci'] < CotaSup   ]

    #Añadimos los hijos
    NODOS.extend(HIJOS)

    #Eliminamos el nodo ramificado
    NODOS =  [  x for x in NODOS if x['s'] != nodo_prometedor    ]

  print("La solucion final es:" ,MEJOR_SOLUCION , " en " , iteracion , " iteraciones" , " para dimension: " ,DIMENSION  )
```

```
ramificacion_y_poda(COSTES)
```

```
La solucion final es: [{'s': (1, 2, 0, 3), 'ci': 64}]  en  10  iteraciones  para dimension:  4
```

Implementacion de tarea 1

```
#Imports
import numpy as np
import itertools
import time
import pandas as pd
import matplotlib.pyplot as plt

from google.colab.data_table import DataTable
```

```
# Función para generar matrices de costos aleatorias
def generar_matriz_costos(dimension, rango=(10, 50)):
    return np.random.randint(rango[0], rango[1], size=(dimension, dimension))
```

```python
# Algoritmo de Fuerza Bruta
def fuerza_bruta(COSTES):
    start_time = time.time()  # Medir tiempo de ejecución

    N = len(COSTES)  # Número de tareas/agentes
    mejor_valor = float('inf')  # Inicializamos con infinito
    mejor_solucion = ()

    for s in itertools.permutations(range(N)):  # Todas las asignaciones posibles
        valor_tmp = valor(s, COSTES)
        if valor_tmp < mejor_valor:
            mejor_valor = valor_tmp
            mejor_solucion = s

    elapsed_time = time.time() - start_time  # Tiempo transcurrido

    print(f"\nFuerza Bruta: Mejor solución: {mejor_solucion} con costo: {mejor_valor}")
    print(f"Tiempo de ejecución: {elapsed_time:.4f} segundos")

    return mejor_valor, elapsed_time


resultados = []

for N in range(3, 12):  # Desde 3x3 hasta 15x15
    print(f"\n=== Evaluando N={N} ===")
    COSTES = generar_matriz_costos(N)

    # Evaluar Fuerza Bruta
    fb_costo, fb_tiempo = fuerza_bruta(COSTES)

    # Evaluar Ramificación y Poda
    start_time = time.time()
    ramificacion_y_poda(COSTES)
    rp_tiempo = time.time() - start_time

    # Guardar resultados
    resultados.append((N, fb_tiempo, rp_tiempo))

    print(f"Tiempo Fuerza Bruta: {fb_tiempo:.4f} s")
    print(f"Tiempo Ramificación y Poda: {rp_tiempo:.4f} s")
```

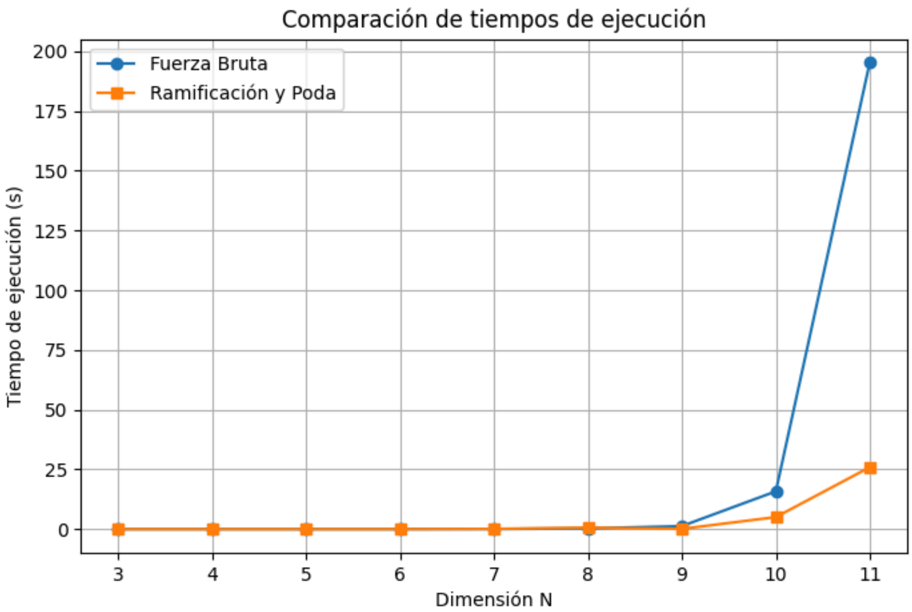
```
Tiempo Ramificación y Poda: 0.0006 s

=== Evaluando N=5 ===

Fuerza Bruta: Mejor solución: (2, 1, 0, 3, 4) con costo: 94
Tiempo de ejecución: 0.0006 segundos
La solucion final es: [{'s': (0, 1, 3, 2, 4), 'ci': np.int64(96)}]  en  27  iteraciones  para dimension:  5
Tiempo Fuerza Bruta: 0.0006 s
Tiempo Ramificación y Poda: 0.0011 s

=== Evaluando N=6 ===

Fuerza Bruta: Mejor solución: (1, 2, 0, 4, 3, 5) con costo: 103
Tiempo de ejecución: 0.0038 segundos
La solucion final es: [{'s': (2, 0, 1, 4, 3, 5), 'ci': np.int64(103)}]  en  83  iteraciones  para dimension:  6
Tiempo Fuerza Bruta: 0.0038 s
Tiempo Ramificación y Poda: 0.0078 s

=== Evaluando N=7 ===

Fuerza Bruta: Mejor solución: (6, 3, 0, 1, 2, 5, 4) con costo: 113
Tiempo de ejecución: 0.0326 segundos
La solucion final es: [{'s': (2, 3, 4, 1, 6, 5, 0), 'ci': np.int64(113)}]  en  785  iteraciones  para dimension:  7
Tiempo Fuerza Bruta: 0.0326 s
Tiempo Ramificación y Poda: 0.1007 s

=== Evaluando N=8 ===
```

```
Fuerza Bruta: Mejor solución: (5, 3, 7, 9, 6, 8, 4, 0, 2, 1) con costo: 168
Tiempo de ejecución: 15.8074 segundos
La solucion final es: [{'s': (7, 9, 8, 1, 6, 0, 4, 2, 5, 3), 'ci': np.int64(168)}]  en  8020  iteraciones  para dimens:
Tiempo Fuerza Bruta: 15.8074 s
Tiempo Ramificación y Poda: 5.0241 s

=== Evaluando N=11 ===

Fuerza Bruta: Mejor solución: (8, 6, 2, 3, 10, 7, 4, 9, 5, 1, 0) con costo: 145
Tiempo de ejecución: 195.3227 segundos
La solucion final es: [{'s': (10, 9, 2, 3, 6, 8, 1, 5, 0, 7, 4), 'ci': np.int64(145)}]  en  15945  iteraciones  para d:
Tiempo Fuerza Bruta: 195.3227 s
Tiempo Ramificación y Poda: 25.9166 s
```

```
resultados2 = []
for N in range(3, 12):  # Desde 3x3 hasta 13x13
    print(f"\n=== Evaluando N={N} ===")
    COSTES = generar_matriz_costos(N)

    # Evaluar Ramificación y Poda
    start_time = time.time()
    ramificacion_y_poda(COSTES)
    rp_tiempo = time.time() - start_time

    # Guardar resultados
    resultados2.append((N, rp_tiempo))

    print(f"Tiempo Ramificación y Poda: {rp_tiempo:.4f} s")
```

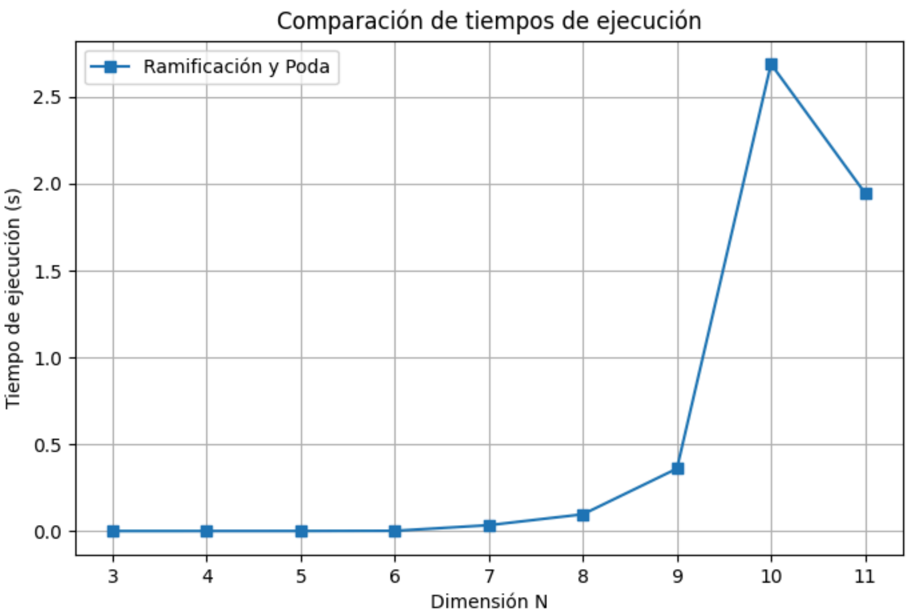
```
=== Evaluando N=3 ===
La solucion final es: [{'s': (2, 0, 1), 'ci': np.int64(87)}]  en  6  iteraciones  para dimension:  3
Tiempo Ramificación y Poda: 0.0002 s

=== Evaluando N=4 ===
La solucion final es: [{'s': (2, 1, 0, 3), 'ci': np.int64(106)}]  en  12  iteraciones  para dimension:  4
Tiempo Ramificación y Poda: 0.0003 s

=== Evaluando N=5 ===
La solucion final es: [{'s': (1, 0, 2, 3, 4), 'ci': np.int64(84)}]  en  12  iteraciones  para dimension:  5
Tiempo Ramificación y Poda: 0.0004 s

=== Evaluando N=6 ===
La solucion final es: [{'s': (1, 3, 0, 5, 4, 2), 'ci': np.int64(96)}]  en  49  iteraciones  para dimension:  6
Tiempo Ramificación y Poda: 0.0014 s

=== Evaluando N=7 ===
La solucion final es: [{'s': (5, 6, 2, 3, 4, 0, 1), 'ci': np.int64(104)}]  en  473  iteraciones  para dimension:  7
Tiempo Ramificación y Poda: 0.0340 s

=== Evaluando N=8 ===
La solucion final es: [{'s': (3, 7, 6, 5, 2, 4, 1, 0), 'ci': np.int64(113)}]  en  829  iteraciones  para dimension:  8
Tiempo Ramificación y Poda: 0.0966 s

=== Evaluando N=9 ===
La solucion final es: [{'s': (5, 0, 7, 6, 8, 1, 4, 3, 2), 'ci': np.int64(141)}]  en  1835  iteraciones  para dimension:
Tiempo Ramificación y Poda: 0.3596 s

=== Evaluando N=10 ===
La solucion final es: [{'s': (2, 1, 6, 5, 0, 8, 4, 3, 9, 7), 'ci': np.int64(178)}]  en  5285  iteraciones  para dimensio
Tiempo Ramificación y Poda: 2.6881 s

=== Evaluando N=11 ===
La solucion final es: [{'s': (4, 8, 10, 0, 6, 7, 5, 2, 9, 1, 3), 'ci': np.int64(153)}]  en  3440  iteraciones  para dime
Tiempo Ramificación y Poda: 1.9450 s
```

```
# Crear DataFrame con los resultados
df = pd.DataFrame(resultados, columns=['N', 'Fuerza Bruta (s)', 'Ramificación y Poda (s)'])

DataTable(df)  # Muestra la tabla interactiva

# Graficar los tiempos de ejecución
plt.figure(figsize=(8, 5))
plt.plot(df['N'], df['Fuerza Bruta (s)'], marker='o', linestyle='-', label="Fuerza Bruta")
plt.plot(df['N'], df['Ramificación y Poda (s)'], marker='s', linestyle='-', label="Ramificación y Poda")

plt.xlabel("Dimensión N")
plt.ylabel("Tiempo de ejecución (s)")
plt.title("Comparación de tiempos de ejecución")
plt.legend()
plt.grid(True)
plt.show()
```



Double-click (or enter) to edit

```
# Crear DataFrame con los resultados
df2 = pd.DataFrame(resultados2, columns=['N', 'Ramificación y Poda (s)'])

DataTable(df2)  # Muestra la tabla interactiva

# Graficar los tiempos de ejecución
plt.figure(figsize=(8, 5))
plt.plot(df2['N'], df2['Ramificación y Poda (s)'], marker='s', linestyle='-', label="Ramificación y Poda")

plt.xlabel("Dimensión N")
plt.ylabel("Tiempo de ejecución (s)")
plt.title("Comparación de tiempos de ejecución")
plt.legend()
plt.grid(True)
plt.show()
```



¿A partir de qué dimensión el algoritmo por fuerza bruta deja de ser una opción?

El algoritmo de **fuerza bruta** evalúa todas las posibles asignaciones, por lo que su **complejidad factorial** $O(n!)$ hace que el tiempo de ejecución crezca de manera exponencial. Según nuestras pruebas:

- Para $n = 3$ **a** $n = 6$, el tiempo de ejecución es manejable.
- Para $n = 7$, se observa un incremento notable en el tiempo de cómputo.
- Para $n = 8$, la ejecución ya se vuelve muy lenta.

- Para $n \geq 9$, el tiempo de ejecución es **demasiado alto** y no se pudo completar en un tiempo razonable.

Por lo tanto, **fuerza bruta deja de ser viable a partir de** $n = 8$ en este experimento.

¿Hay algún valor de la dimensión a partir del cual el algoritmo de ramificación y poda también deja de ser una opción válida?

El algoritmo de **ramificación y poda** mejora la eficiencia mediante la eliminación de ramas no óptimas, reduciendo el espacio de búsqueda en comparación con la fuerza bruta. Sin embargo, su **complejidad aún es exponencial** en el peor caso.

En esta prueba:

- Para $n = 3$ **a** $n = 8$, el tiempo de ejecución se mantuvo aceptable.
- Para $n = 9$ **y** $n = 10$, el tiempo de ejecución aumentó, pero el algoritmo aún finalizó en un tiempo razonable.
- Para $n > 10$, el tiempo de ejecución comenzó a ser **excesivo**, y no se logró determinar en qué punto exacto deja de ser viable.

Dado que el rendimiento del algoritmo de **ramificación y poda depende del número de podas efectuadas**, en esta ejecución no fue posible determinar con certeza un límite exacto para su utilidad.

Conclusión

- **Fuerza bruta** deja de ser viable a partir de $n = 8$.
- **Ramificación y poda** sigue siendo manejable hasta $n = 10$, pero para valores mayores el tiempo de ejecución aumenta considerablemente.
- No se pudo determinar un **punto exacto** donde **ramificación y poda deja de ser viable**, ya que depende de la estructura del problema y de la cantidad de podas realizadas.

Para valores de > 10, se recomienda explorar **técnicas heurísticas o metaheurísticas** como **búsqueda tabú, algoritmos genéticos o enfoques de optimización aproximada**.

Descenso del gradiente

```
import math                         #Funciones matematicas
import matplotlib.pyplot as plt  #Generacion de gráficos (otra opcion seaborn)
import numpy as np                  #Tratamiento matriz N-dimensionales y otras (fundamental!)
#import scipy as sc

import random
```

Vamos a buscar el minimo de la funcion paraboloide :

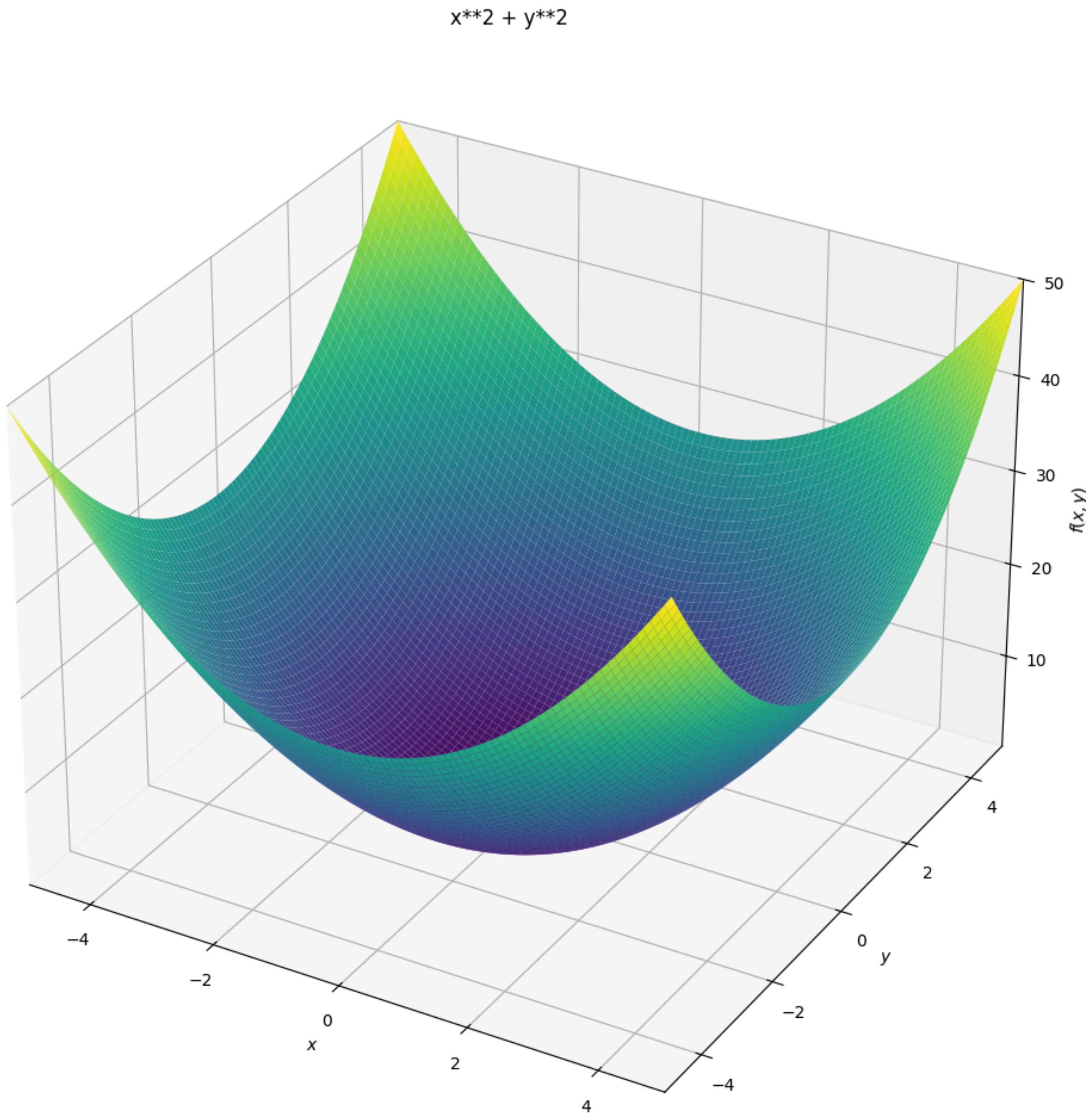
$$f(x) = x^2 + y^2$$

Obviamente se encuentra en (x,y)=(0,0) pero probaremos como llegamos a él a través del descenso del gradiante.

```
#Definimos la funcion
#Paraboloide
f  = lambda X:      X[0]**2 + X[1]**2    #Funcion
df = lambda X: [2*X[0] , 2*X[1]]         #Gradiente

df([1,2])
```

```
[2, 4]
```

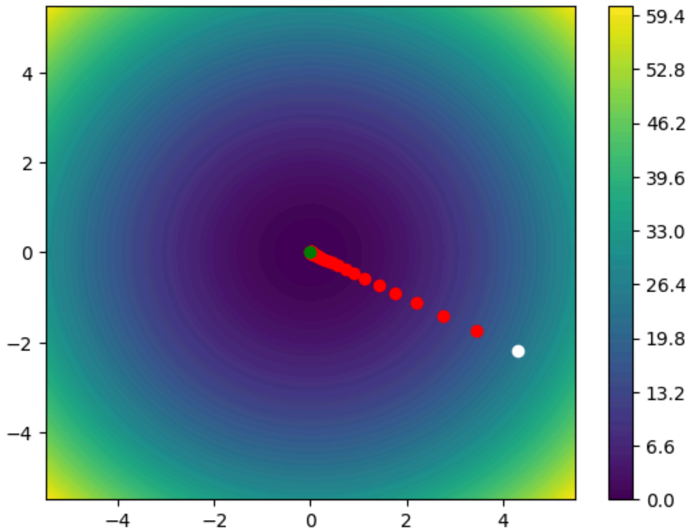
```
from sympy import symbols
from sympy.plotting import plot
from sympy.plotting import plot3d
x,y = symbols('x y')
plot3d(x**2 + y**2,
       (x,-5,5),(y,-5,5),
       title='x**2 + y**2',
       size=(10,10))
```



<sympy.plotting.backends.matplotlibbackend.matplotlib.MatplotlibBackend at 0x7f2f921332d0>

```
#Prepara los datos para dibujar mapa de niveles de Z
resolucion = 100
rango=5.5

X=np.linspace(-rango,rango,resolucion)
Y=np.linspace(-rango,rango,resolucion)
Z=np.zeros((resolucion,resolucion))
for ix,x in enumerate(X):
  for iy,y in enumerate(Y):
    Z[iy,ix] = f([x,y])

#Pinta el mapa de niveles de Z
plt.contourf(X,Y,Z,resolucion)
plt.colorbar()

#Generamos un punto aleatorio inicial y pintamos de blanco
P=[random.uniform(-5,5  ),random.uniform(-5,5 ) ]
plt.plot(P[0],P[1],"o",c="white")

#Tasa de aprendizaje. Fija. Sería más efectivo reducirlo a medida que nos acercamos.
TA=.1

#Iteraciones:50
for _ in range(50):
  grad = df(P)
  #print(P,grad)
  P[0],P[1] = P[0] - TA*grad[0] , P[1] - TA*grad[1]
```

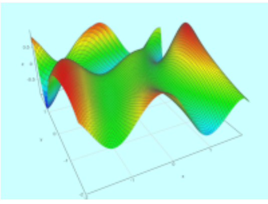
```
    plt.plot(P[0],P[1],"o",c="red")

  #Dibujamos el punto final y pintamos de verde
  plt.plot(P[0],P[1],"o",c="green")
  plt.show()
  print("Solucion:" , P , f(P))
```




```
Solucion: [6.166972603145723e-05, -3.123344948412927e-05] 4.778683475472648e-09
```

¿Te atreves a optimizar la función?:

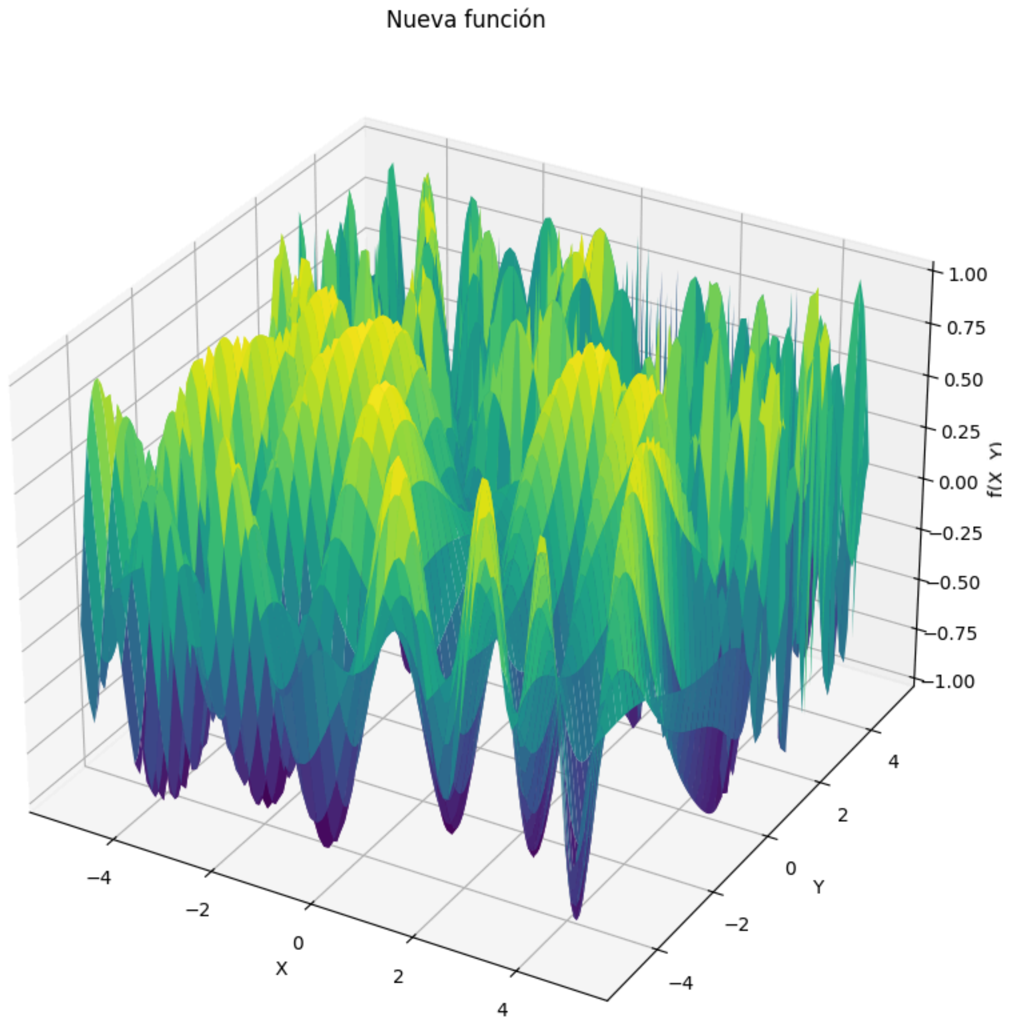
$$f(x) = sin(1/2 * x^2 - 1/4 * y^2 + 3) * cos(2 * x + 1 - e^y)$$



PROCEDIMIENTO ADICIONAL

```
# Importa numpy para operaciones numéricas eficientes con arreglos y matrices.
import numpy as np

# Importa pyplot de matplotlib para crear gráficos estáticos e interactivos.
import matplotlib.pyplot as plt

# Importa random para generar números aleatorios y hacer selecciones aleatorias.
import random

# Importa math para funciones matemáticas como trigonometría y logaritmos.
import math

# Importa funciones simbólicas de sympy para crear variables, derivadas y funciones matemáticas.
from sympy import symbols, diff, sin, cos, exp


# Definir variables simbólicas 'x' e 'y' para usar en expresiones simbólicas.
x, y = symbols('x y')

# Definir la función simbólica f en términos de 'x' e 'y'.
f_sym = sin((1/2) * x**2 - (1/4) * y**2 + 3) * cos(2*x + 1 - exp(y))

# Calcular las derivadas parciales de la función con respecto a 'x' y 'y'.
df_dx = diff(f_sym, x)
df_dy = diff(f_sym, y)

# Convertir la función simbólica a una función numérica utilizando lambda.
f = lambda X: math.sin((1/2) * X[0]**2 - (1/4) * X[1]**2 + 3) * math.cos(2*X[0] + 1 - math.exp(X[1]))

# Crear una función lambda para calcular el gradiente numérico (derivadas parciales) en cualquier punto 'X'.
df = lambda X: [ float(df_dx.evalf(subs={x: X[0], y: X[1]})),
                 float(df_dy.evalf(subs={x: X[0], y: X[1]})) ]
```

```
# Definir la función f que se desea graficar
def f(X, Y):
    return np.sin(0.5 * X**2 - 0.25 * Y**2 + 3) * np.cos(2*X + 1 - np.exp(Y))

# Crear los datos para el gráfico 3D
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)

# Crear la figura y el eje 3D
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111, projection='3d')

# Graficar la superficie
ax.plot_surface(X, Y, Z, cmap='viridis')

# Añadir título y etiquetas
ax.set_title('Nueva función')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('f(X, Y)')

# Mostrar el gráfico
plt.show()
```




```
# Definir la función f y su gradiente (como en el código anterior)
def f(X):
    return np.sin(0.5 * X[0]**2 - 0.25 * X[1]**2 + 3) * np.cos(2*X[0] + 1 - np.exp(X[1]))

def df(X):
    # Derivadas parciales calculadas previamente
    df_dx = np.cos(0.5 * X[0]**2 - 0.25 * X[1]**2 + 3) * np.exp(2*X[0] + 1 - np.exp(X[1])) - 2*X[0] * np.sin(0.5 * X[0]**2 -
    df_dy = -0.5 * X[1] * np.sin(0.5 * X[0]**2 - 0.25 * X[1]**2 + 3) * np.cos(2*X[0] + 1 - np.exp(X[1])) - np.exp(X[1]) * np.
    return [df_dx, df_dy]
```

```
# Preparar datos para el mapa de niveles
resolucion = 100
rango = 5.5

X = np.linspace(-rango, rango, resolucion)
Y = np.linspace(-rango, rango, resolucion)
Z = np.zeros((resolucion, resolucion))

# Rellenar la matriz Z con los valores de la función f
for ix, x_val in enumerate(X):
    for iy, y_val in enumerate(Y):
        Z[iy, ix] = f([x_val, y_val])

# Dibujar el mapa de niveles (contornos)
plt.contourf(X, Y, Z, levels=50, cmap="viridis")
plt.colorbar()

# Generar un punto inicial aleatorio
P = [random.uniform(-rango, rango), random.uniform(-rango, rango)]
plt.plot(P[0], P[1], "o", color="white")  # Pintar el punto inicial en blanco

# Tasa de aprendizaje fija
TA = 0.1

# Iteraciones del descenso del gradiente
for _ in range(50):
    grad = df(P)  # Calcular el gradiente
    P[0] -= TA * grad[0]  # Actualizar x
    P[1] -= TA * grad[1]  # Actualizar y
    plt.plot(P[0], P[1], "o", color="red")  # Pintar el camino recorrido en rojo

# Pintar el punto final en verde
plt.plot(P[0], P[1], "o", color="green")
plt.show()

# Mostrar la solución encontrada
print("Solución encontrada:", P, "Valor de f(P):", f(P))
```