practica1

November 24, 2024

1 Ejercicio 1

Se cuenta con un arreglo de tamaño “n” (n<=10) cuyos elementos son números enteros positivos de 1 o 2 cifras, todos diferentes. Escriba un programa que reciba el arreglo y mediante una función que recibe el tamaño n y el arreglo, muestre las sumas que pueden ser obtenidas con los subconjuntos formados por dichos números. Ejemplo: si n=4 y los valores son 1, 2, 5 y 10 Entonces las sumas son 1, 2, 3, 5, 6, 7, 8, 10, 11, 12, 13, 15, 16, 17 y 18 Sugerencia: si el arreglo contiene n valores entonces puede utilizar números binarios de n cifras para encontrar todos los subconjuntos

```
0001 10
0010 5
0100 2
1000 1
0011 5 + 10
0110 2 + 5
…
1001 1 + 10
1011 1 + 5 + 10
1111 1 + 2+ 5 + 10
```

[1]:
```python
def calcular_sumas_subconjuntos(n, arreglo):
    """
    Calcula las sumas posibles de los subconjuntos de un arreglo de n elementos.
    """
    sumas = set()
    for i in range(1, 2**n):
        suma = 0
        for j in range(n):
            if i & (1 << j):
                suma += arreglo[j]
        sumas.add(suma)
    sumas = sorted(sumas)
    return sumas
```

1.0.1 Probando metodo

[2]:
```
n = 4
arreglo = [1, 2, 5, 10]
```

[3]:
```
sumas = calcular_sumas_subconjuntos(n, arreglo)
print("Las sumas posibles son:", sumas)
```

```
Las sumas posibles son: [1, 2, 3, 5, 6, 7, 8, 10, 11, 12, 13, 15, 16, 17, 18]
```

---

2 Ejercicio 2

Escriba un algoritmo que dados 2 números enteros x y n donde 0<n<25 determine el valor de la siguiente serie:

$$S = \frac{x^{(2n+1)}}{3!} - \frac{x^{(2n-1)}}{5!} + \frac{x^{(2n-3)}}{7!} - \ldots$$

Considera que no existen operadores ni funciones de potencia ni de factorial.

[4]:
```
# Funciones auxiliares para potencia y factorial
potencia = lambda base, exponente: 1 if exponente == 0 else base *␣
 ↪potencia(base, exponente - 1)
factorial = lambda num: 1 if num == 0 else num * factorial(num - 1)
```

[5]:
```
def calcular_serie(x, n):
    """
    Calcula el valor de la serie para los valores dados de x y n:
    S = x^(2n+1)/3! - x^(2n-1)/5! + x^(2n-3)/7! - ...
    """

    signo = 1  # Controla el signo de cada término (+ o -)
    suma = 0

    # Inicializamos los valores de la potencia y el factorial
    potencia_actual = potencia(x, 2 * n + 1)  # x^(2n+1)
    factorial_actual = factorial(3)  # 3!

    # Generación de la serie
    for k in range(n):
        # Añadir el término actual
        suma += signo * (potencia_actual / factorial_actual)

        # Actualizamos la potencia y el factorial para el siguiente término
        potencia_actual = potencia_actual / (x * x)
```

```
        # Actualizamos el factorial de manera incremental
        factorial_actual *= (2 * k + 4) * (2 * k + 5)

        # Cambiar el signo para el siguiente término
        signo *= -1

    return suma
```

2.0.1 Probando metodo

[6]:
```
x = 2
n = 5
```

[7]:
```
resultado = calcular_serie(x, n)
print(f"El resultado de la serie para x = {x} y n = {n} es: {resultado}")
```

```
El resultado de la serie para x = 2 y n = 5 es: 337.0919755090589
```

---

3 Ejercicio 3

Un tablero “n-diagonal” (o también conocido como “Tablero Diagonal Cíclico”) es un tablero con n x (n + 1) casillas. Nos referiremos a las casillas de este tablero mediante las coordenadas (i, j), en donde i es la fila (contadas desde 1 y de arriba a abajo) y j es la columna (contadas desde 1 y de izquierda a derecha). Como ejemplo:

Una propiedad interesante es que se pueden visitar todas sus casillas siguiendo un patrón diagonal que abarca todo el tablero de manera cíclica. El recorrido comienza en la casilla (1,1), avanzando hacia la derecha y hacia abajo a lo largo de la diagonal principal. Cuando se sobrepasa el límite del tablero, ya sea por la derecha o por abajo, el recorrido continúa desde la primera fila o la primera columna, respectivamente. Este proceso se repite hasta que todas las casillas han sido visitadas. El recorrido asegura que, tras alcanzar una casilla límite, se salta de vuelta a la fila o columna inicial. Como ejemplo este tablero de 4 x 5 casillas, rellenas con un número que indica en qué momento se visitó:

Haz un programa que dado un número “n” (n<10) muestre en pantalla: - Un tablero “n-diagonal” con sus casillas rellenas con el número correspondiente al momento en que se visitan. - La matriz cuadrada resultante de eliminar el menor elemento de cada fila del tablero “n- diagonal”.

[8]:
```
def crear_tablero_n_diagonal(n):
    """
    Crea un tablero 'n-diagonal' de tamaño n x (n+1) siguiendo el patrón␣
 ↪cíclico diagonal.
    """
    tablero = [[0] * (n + 1) for _ in range(n)]  # Crear una matriz vacía
    numero = 1  # Número inicial para rellenar
    i, j = 0, 0  # Coordenadas iniciales
```

```
    # Rellenar el tablero siguiendo el patrón
    while numero <= n * (n + 1):
        tablero[i][j] = numero
        numero += 1
        # Moverse a la siguiente casilla diagonal
        i += 1
        j += 1
        if i >= n:  # Si excede el límite inferior, regresar al principio
            i = 0
        if j >= n + 1:  # Si excede el límite derecho, regresar al principio
            j = 0

    return tablero
```

[9]:
```
def eliminar_menor_por_fila(tablero):
    """
    Elimina el menor elemento de cada fila del tablero.
    """
    nuevo_tablero = []
    for fila in tablero:
        menor = min(fila)
        nueva_fila = [x for x in fila if x != menor]  # Crear fila sin el menor␣
 ↪elemento
        nuevo_tablero.append(nueva_fila)
    return nuevo_tablero
```

3.0.1 Probando metodo

[10]:
```
import pandas as pd
```

[11]:
```
n = 4
```

[12]:
```
tablero = crear_tablero_n_diagonal(n)
tablero_df = pd.DataFrame(tablero)
tablero_df
```

[12]:

	0	1	2	3	4
0	1	17	13	9	5
1	6	2	18	14	10
2	11	7	3	19	15
3	16	12	8	4	20

[13]:
```
nuevo_tablero = eliminar_menor_por_fila(tablero)
tablero_df = pd.DataFrame(nuevo_tablero)
tablero_df
```

[13]:

	0	1	2	3
0	17	13	9	5
1	6	18	14	10
2	11	7	19	15
3	16	12	8	20

4 Ejercicio 4

Diseñe un algoritmo que permita resolver el siguiente problema: El alcalde de la ciudad contrató a Bob el constructor para que tape los agujeros que hay en una carretera. La carretera tiene un largo N y un ancho M. Bob dividió esta carretera en cuadrados tal que el resultado fue una rejilla de N x M.

Ayuda a Bob a saber cuántos huecos tendrá que tapar, un hueco es un conjunto de cuadrados imperfectos conectados, se dice que dos cuadros o casillas están conectadas si comparten un lado en común. (Ver el ejemplo).

Entrada **N**, **M**, el largo y ancho de la carretera. Seguidamente de **N** filas cada una con **M** caracteres. Si el carácter es 'X' significa que la sección está bien. Si el carácter es 'O' significa que hay un problema en la sección.

Salida La cantidad de huecos que tendrá que tapar Bob el constructor. #### Restricciones 1 N, M 100

Ejemplo **Entrada** 6 10

Salida 4 Huecos

[14]:
```python
def dfs(tablero, i, j, N, M):
    # Comprobar si estamos dentro de los límites
    if i < 0 or i >= N or j < 0 or j >= M:
        return
    # Si la casilla ya fue visitada o no es un hueco ('O'), terminamos
    if tablero[i][j] != 'O':
        return

    # Marcar la casilla como visitada (la cambiamos a 'X' para evitar␣
↪revisitarla)
    tablero[i][j] = 'X'

    # Realizar DFS en las 4 direcciones posibles (arriba, abajo, izquierda,␣
↪derecha)
    dfs(tablero, i-1, j, N, M)  # Arriba
    dfs(tablero, i+1, j, N, M)  # Abajo
    dfs(tablero, i, j-1, N, M)  # Izquierda
    dfs(tablero, i, j+1, N, M)  # Derecha
```

[15]:
```python
def contar_huecos(N, M, tablero):
    huecos = 0

    for i in range(N):
        for j in range(M):
            if tablero[i][j] == '0':
                huecos += 1
                dfs(tablero, i, j, N, M)
    return huecos
```

4.0.1 Probando metodo

[16]:
```python
N = 6
M = 10
tablero = [
    ['X', 'X', 'X', 'X', 'X', '0', '0', '0', '0', 'X'],
    ['X', '0', '0', '0', 'X', '0', 'X', '0', '0', 'X'],
    ['X', '0', '0', 'X', '0', 'X', '0', '0', '0', '0'],
    ['X', '0', '0', 'X', '0', '0', 'X', '0', '0', '0'],
    ['0', '0', '0', 'X', 'X', 'X', 'X', 'X', '0', 'X'],
    ['0', '0', '0', '0', '0', 'X', 'X', 'X', 'X', 'X']
]
tablero_df = pd.DataFrame(tablero)
tablero_df
```

[16]:

	0	1	2	3	4	5	6	7	8	9
0	X	X	X	X	X	0	0	0	0	X
1	X	0	0	0	X	0	X	0	0	X
2	X	0	0	X	0	X	0	0	0	0
3	X	0	0	X	0	0	X	0	0	0
4	0	0	0	X	X	X	X	X	0	X
5	0	0	0	0	0	X	X	X	X	X

[17]:
```python
huecos = contar_huecos(N, M, tablero)
print(f"Cantidad de huecos: {huecos}")
```

```
Cantidad de huecos: 3
```

4.1 Ejercicio 5

Dados dos conjuntos y de y elementos enteros, se define la distancia entre los conjuntos y como el número de elementos de la diferencia simétrica de dichos conjuntos con sus elementos indexados.

Ejemplo: - **Conjunto :** {1; 3; 5; 7} - **Conjunto A con sus elementos indexado:** = {11; 32; 53; 74} - **Conjunto :** {1; 2; 4; 5} - **Conjunto B con sus elementos indexado:** = {11; 22; 43; 54} - **Diferencia simétrica de y :** (−) (−) = {22; 32; 43; 53; 54; 74} - **La distancia entre y es:** 6

Nota: Validar que los elementos de un conjunto no se repitan

[18]:
```python
def calcular_distancia(A, B):
    A_ind = {f"{a}_{i+1}" for i, a in enumerate(A)}
    B_ind = {f"{b}_{i+1}" for i, b in enumerate(B)}

    # Calcular la diferencia simétrica
    diferencia_A_B = A_ind - B_ind
    diferencia_B_A = B_ind - A_ind

    # La diferencia simétrica es la unión de las dos diferencias
    diferencia_simetrica = diferencia_A_B | diferencia_B_A

    # La distancia es el tamaño de la diferencia simétrica
    return len(diferencia_simetrica), diferencia_simetrica
```

4.1.1 Probando metodo

[19]:
```python
A = [1, 3, 5, 7]
B = [1, 2, 4, 5]

distancia, diferencia = calcular_distancia(A, B)

print(f"La distancia entre A y B es: {distancia}")
print(f"La diferencia simétrica es: {diferencia}")
```

```
La distancia entre A y B es: 6
La diferencia simétrica es: {'7_4', '4_3', '3_2', '5_3', '5_4', '2_2'}
```

4.2 Ejercicio 6

Desarrolle un programa que reciba por consola un texto que representa a un matriz de 9 x 9 que representa a un SUDOKU, donde lo ceros representan a los casilleros vacíos. Su programa debe determinar para cada casillero vacío la lista de valores posibles. Ejemplo: si se ingresa la siguiente matriz:

Entrada

530070000
600195000
098000060
800060003
400803001
700020006
060000280
000419005
000080079

Representa

Salida (solo se muestra algunos ejemplos de la salida): - Casillero (2,2) valores posibles: 2, 4, 7 - Casillero (4,7) valores posibles: 4, 5, 7, 8, 9

> **Nota:** Recordar que no puede haber valores repetidos en las filas, ni en las columnas, ni en los recuadros de 3 x 3

[20]:
```python
def obtener_posibles(tablero, fila, columna):
    """
    Dado un tablero de Sudoku y una posición (fila, columna), determina los valores posibles
    para esa casilla vacía (representada por un 0), teniendo en cuenta las reglas del Sudoku.
    """
    # Valores posibles del 1 al 9
    posibles = {1, 2, 3, 4, 5, 6, 7, 8, 9}

    # Eliminar los valores ya presentes en la fila
    posibles -= set(tablero[fila])

    # Eliminar los valores ya presentes en la columna
    posibles -= set(tablero[i][columna] for i in range(9))

    # Eliminar los valores ya presentes en el subcuadro 3x3
    bloque_fila = (fila // 3) * 3  # Calcula la fila inicial del subcuadro 3x3
    bloque_columna = (columna // 3) * 3  # Calcula la columna inicial del subcuadro 3x3

    for i in range(bloque_fila, bloque_fila + 3):
        for j in range(bloque_columna, bloque_columna + 3):
            posibles.discard(tablero[i][j])
    return sorted(list(posibles))
```

[21]:
```python
def mostrar_posibles(tablero):
    """
    Recorre el tablero de Sudoku e imprime los valores posibles para cada casillero vacío (0).
    """
    for i in range(9):
        for j in range(9):
            if tablero[i][j] == 0:  # Casillero vacío
                posibles = obtener_posibles(tablero, i, j)
                print(f"Casillero ({i+1},{j+1}) valores posibles: {posibles}")
```

4.2.1 Probando metodo

[22]:
```
tablero = [
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 9, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [8, 0, 0, 6, 0, 0, 0, 0, 3],
    [4, 0, 0, 8, 3, 0, 0, 0, 1],
    [7, 0, 0, 0, 2, 0, 0, 0, 6],
    [0, 6, 0, 0, 0, 0, 2, 8, 0],
    [0, 0, 0, 4, 1, 9, 0, 0, 5],
    [0, 0, 0, 0, 8, 0, 0, 7, 9]
]
tablero_df = pd.DataFrame(tablero)
tablero_df
```

[22]:

	0	1	2	3	4	5	6	7	8
0	5	3	0	0	7	0	0	0	0
1	6	0	0	1	9	5	0	0	0
2	0	9	8	0	0	0	0	6	0
3	8	0	0	6	0	0	0	0	3
4	4	0	0	8	3	0	0	0	1
5	7	0	0	0	2	0	0	0	6
6	0	6	0	0	0	0	2	8	0
7	0	0	0	4	1	9	0	0	5
8	0	0	0	0	8	0	0	7	9

[23]:
```
mostrar_posibles(tablero)
```

```
Casillero (1,3) valores posibles: [1, 2, 4]
Casillero (1,4) valores posibles: [2]
Casillero (1,6) valores posibles: [2, 4, 6, 8]
Casillero (1,7) valores posibles: [1, 4, 8, 9]
Casillero (1,8) valores posibles: [1, 2, 4, 9]
Casillero (1,9) valores posibles: [2, 4, 8]
Casillero (2,2) valores posibles: [2, 4, 7]
Casillero (2,3) valores posibles: [2, 4, 7]
Casillero (2,7) valores posibles: [3, 4, 7, 8]
Casillero (2,8) valores posibles: [2, 3, 4]
Casillero (2,9) valores posibles: [2, 4, 7, 8]
Casillero (3,1) valores posibles: [1, 2]
Casillero (3,4) valores posibles: [2, 3]
Casillero (3,5) valores posibles: [4]
Casillero (3,6) valores posibles: [2, 3, 4]
Casillero (3,7) valores posibles: [1, 3, 4, 5, 7]
Casillero (3,9) valores posibles: [2, 4, 7]
Casillero (4,2) valores posibles: [1, 2, 5]
Casillero (4,3) valores posibles: [1, 2, 5, 9]
```

```
Casillero (4,5) valores posibles: [4, 5]
Casillero (4,6) valores posibles: [1, 4, 7]
Casillero (4,7) valores posibles: [4, 5, 7, 9]
Casillero (4,8) valores posibles: [2, 4, 5, 9]
Casillero (5,2) valores posibles: [2, 5]
Casillero (5,3) valores posibles: [2, 5, 6, 9]
Casillero (5,6) valores posibles: [7]
Casillero (5,7) valores posibles: [5, 7, 9]
Casillero (5,8) valores posibles: [2, 5, 9]
Casillero (6,2) valores posibles: [1, 5]
Casillero (6,3) valores posibles: [1, 3, 5, 9]
Casillero (6,4) valores posibles: [5, 9]
Casillero (6,6) valores posibles: [1, 4]
Casillero (6,7) valores posibles: [4, 5, 8, 9]
Casillero (6,8) valores posibles: [4, 5, 9]
Casillero (7,1) valores posibles: [1, 3, 9]
Casillero (7,3) valores posibles: [1, 3, 4, 5, 7, 9]
Casillero (7,4) valores posibles: [3, 5, 7]
Casillero (7,5) valores posibles: [5]
Casillero (7,6) valores posibles: [3, 7]
Casillero (7,9) valores posibles: [4]
Casillero (8,1) valores posibles: [2, 3]
Casillero (8,2) valores posibles: [2, 7, 8]
Casillero (8,3) valores posibles: [2, 3, 7]
Casillero (8,7) valores posibles: [3, 6]
Casillero (8,8) valores posibles: [3]
Casillero (9,1) valores posibles: [1, 2, 3]
Casillero (9,2) valores posibles: [1, 2, 4, 5]
Casillero (9,3) valores posibles: [1, 2, 3, 4, 5]
Casillero (9,4) valores posibles: [2, 3, 5]
Casillero (9,6) valores posibles: [2, 3, 6]
Casillero (9,7) valores posibles: [1, 3, 4, 6]
```

4.3 Ejercicio 7

Diseñe una función que reciba una oración, de no más de 200 caracteres, que contiene tanto palabras como números enteros de hasta 3 cifras. Asuma que las palabras no contienen dígitos numéricos, ni letras tildadas ni caracteres especiales, ni signos de puntuación.

Luego ejecute lo siguiente:

- Calcule la suma de los números enteros en dicha oración (s).
- Obtenga el resto (r) de la división entera entre dicha suma (s) y el número 5. Luego a dicho resto (r) le sume 1 y obtenga un nuevo número (n).

Luego determine cuantas palabras en la oración tienen “n” vocales diferentes. Debe imprimir en pantalla un mensaje indicando cuantas palabras tienen “n” vocales diferentes y la función deber retornar la cantidad de palabras calculadas.

Ejemplo: Se recibe la siguiente oración: - Mis primos tienen casi 40 anios yo tengo 34 y tenemos

que jugar pelota con gente de 18 esta bien dificil ganar Debe retornar el número 2 (que indica que hay 2 palabras con 3 vocales diferentes) El programa principal debe imprimir Hay 2 palabras con 3 vocales diferentes > Nota: s = 40+34+18 = 92; r = 92 mod 5 = 2; n = r+1 = 3;

[24]:
```
import re
```

[25]:
```
def sumar_numeros(oracion):
    """
    Extrae los números enteros de la oración y devuelve su suma.
    """
    numeros = re.findall(r'\b\d{1,3}\b', oracion)  # Buscar números de 1 a 3␣
 ↪dígitos
    numeros = list(map(int, numeros))
    return sum(numeros)
```

[26]:
```
def calcular_n(suma):
    """
    Calcula el resto entre la suma y el numero 5 y al dicho resto sumarle 1.
    """
    return suma % 5 + 1
```

[27]:
```
def contar_vocales_diferentes(palabra):
    """
    Cuenta el número de vocales diferentes en una palabra.
    """
    vocales = set('aeiou')  # Conjunto de vocales
    return len(set([letra for letra in palabra.lower() if letra in vocales]))
```

[28]:
```
def contar_palabras_con_n_vocales_diferentes(oracion, n):
    """
    Cuenta cuántas palabras en la oración tienen exactamente n vocales␣
 ↪diferentes.
    """
    return sum(1 for palabra in oracion.split(' ') if␣
 ↪contar_vocales_diferentes(palabra) == n)
```

4.3.1 Probando metodo

[29]:
```
oracion = "Mis primos tienen casi 40 anios yo tengo 34 y tenemos que jugar␣
 ↪pelota con gente de 18 esta bien dificil ganar"
suma_enteros_en_oracion = sumar_numeros(oracion)

print(f"La suma de enteros en la oración es: {suma_enteros_en_oracion}")
```

```
La suma de enteros en la oración es: 92
```

[30]:
```
n = calcular_n(suma_enteros_en_oracion)
```

```
print(f"El resultado entre la division entera de la suma y 5, incrementado en␣
 ↪1, se obtiene: {n}")
```

```
El resultado entre la division entera de la suma y 5, incrementado en 1, se
obtiene: 3
```

[31]:
```
palabras_con_n_letras = contar_palabras_con_n_vocales_diferentes(oracion, n)
print(f"Hay {palabras_con_n_letras} palabras con {n} vocales")
```

```
Hay 2 palabras con 3 vocales
```

4.4 Ejercicio 8

Sergei ha diseñado una nueva línea de juguetes anidados: "babushkas generalizadas" (en adelante BG), inspirados en las tradicionales babushkas rusas (una babushkas es una muñeca que puede abrirse en dos mitades, de modo que se encuentra otra muñeca dentro, la misma que puede tener otra muñeca dentro y así seguir varias veces, hasta llegar a una muñeca final que no puede abrirse). En estas BG, cada juguete puede contener varios juguetes (no sólo 1) más pequeños dentro de él.

Vladimir ha desarrollado una notación para describir cómo deben construirse los juguetes anidados. Un juguete se representa con un número entero positivo, según su tamaño. Más concretamente: si al abrir el juguete representado por encontramos los juguetes representados por 1 , 2, … , , debe ser cierto que 1 , 2, … , < . Y si esto es así, decimos que el juguete contiene directamente a los juguetes 1 , 2, … , (ojo: cualquier juguete que pueda estar contenidos en cualquiera de los juguetes 1 , 2, … , no se consideran directamente contenido en el juguete).

Una babushka generalizada (BG) se denota con una secuencia no vacía de enteros no nulos de la forma:

$a_1, a_2, \ldots, a_N$

tal que el juguete se representa en la secuencia con dos enteros − y , ubicándose el negativo antes que el positivo.

Por ejemplo, la secuencia:

-9 -7 -2 2 -3 -2 -1 1 2 3 7 9

representa una BG formada por seis juguetes, a saber, 1, 2 (dos veces), 3, 7 y 9. Obsérvese que el juguete 7 contiene directamente los juguetes 2 y 3. Obsérvese que la primera copia del juguete 2 se produce a la izquierda de la segunda y que la segunda copia contiene directamente al juguete 1. Sería erróneo entender que el primer -2 y el último 2 deberían estar emparejados.

Por otro lado, las siguientes secuencias no describen babushkas generalizadas:

−9 −7 −2 2 −3 −1 −2 2 1 3 7 9

porque el juguete 2 es mayor que el juguete 1 y no puede asignarse dentro de él.

-9 -7 -2 2 -3 -2 -1 1 2 3 7 -2 2 9

porque 7 y 2 no pueden asignarse juntos dentro de 9.

Tu tarea es escribir un programa que determine si una secuencia leída representa una babushkas generalizada válida.

Entrada La primera línea constará de un entero “l” que indicará la cantidad de líneas a leer (l<20) Cada caso de prueba es una línea de enteros no nulos de 3 cifras o menos.

Salida Por cada línea, imprime “: -) babushkas!” si es válida o “:-(intenta de nuevo.” si no lo es.

Ejemplo de Entrada 2 -9 -7 -2 2 -3 -2 -1 1 2 3 7 9 -9 -7 -2 2 -3 -1 -2 2 1 3 7 9

Ejemplo de Salida - :-) babushkas! - :-(intenta de nuevo

[32]:
```
def verificar_babushka(secuencia):
    pila = []

    for num in secuencia:
        if num < 0:  # Si es un número negativo, representa un juguete que abre
            pila.append(-num)  # Guardamos el valor positivo del juguete en la␣
 ↪pila
        else:  # Si es un número positivo, representa un juguete que se cierra
            if not pila or pila[-1] != num:  # Si la pila está vacía o no␣
 ↪coincide el juguete
                return ":-( intenta de nuevo."
            pila.pop()  # Si todo está bien, cerramos el juguete, sacamos el␣
 ↪número de la pila

    # Si la pila está vacía, significa que todos los juguetes fueron␣
 ↪correctamente cerrados
    if not pila:
        return ":-) babushkas!"
    else:
        return ":-( intenta de nuevo."
```

4.4.1 Probando metodo

[33]:
```
casos_de_prueba = [
    [-9, -7, -2, 2, -3, -2, 1, 2, 3, 7, 9],
    [-9, -7, -2, 2, -3, -1, -2, 2, 1, 3, 7, 9]
]

for secuencia in casos_de_prueba:
    print(verificar_babushka(secuencia))
```

```
:-( intenta de nuevo.
:-) babushkas!
```

4.5 Ejercicio 9

Una abeja muy organizada ha numerado las celdas hexagonales de su – potencialmente infinito – panal de abeja en la forma indicada en la figura 2. Luego, la abeja se pregunta: ¿cuántas celdas deben ser recorridas para llegar desde la celda con numeración (x, y) hasta la celda con numeración (u, v)?

Desarrolle un programa que lea 4 enteros x, y, u, v, en el rango [-1000, 1000] e imprima la cantidad de celdas que deben ser recorridas para llegar desde la celda (x, y) hasta la celda (u, v) del panal de abeja (incluyendo las celdas inicial y final)

[34]:
```python
from collections import deque

directions = [(-1, 0), (-1, -1), (0, 1), (0, -1), (1, 1), (1, 0)]

def bfs(x, y, u, v):
    """
    Encuentra el camino más corto entre (x, y) y (u, v) utilizando el algoritmo␣
 ↪BFS
    en un panal hexagonal con las direcciones especificadas.
    """

    queue = deque([(x, y, 0)])
    visited = set()
    parent = {}
    visited.add((x, y))

    while queue:
        cx, cy, dist = queue.popleft()

        # Si alcanzamos la celda destino, devolvemos la distancia + 1␣
 ↪(incluyendo la celda de destino)

        # Agregamos
        if (cx, cy) == (u, v):
            # Reconstruir el camino desde (u, v) hasta (x, y) siguiendo los␣
 ↪padres
            path = []
            current = (u, v)
            while current != (x, y):
                path.append(current)
                current = parent[current]
            path.append((x, y))
            path.reverse()  # Invertimos el camino
            return dist + 1, path  # Devolvemos la distancia y la ruta completa

        # Explorar los vecinos en las 6 direcciones
        for dx, dy in directions:
            nx, ny = cx + dx, cy + dy
            if (nx, ny) not in visited:
                visited.add((nx, ny))
                parent[(nx, ny)] = (cx, cy)  # Guardamos el padre del nodo
                queue.append((nx, ny, dist + 1))
```

```
    return -1, []
```

[ ]:
```
```

4.5.1 Probando metodo

[35]:
```
casos_de_prueba = [
    (0, 0, 1, 1), #2
    (1, 0, 3, 3), #4
    (2, 1, 2, 1)  #1
]

for caso in casos_de_prueba:
    x, y, u, v = caso
    # Calcular la distancia
    resultado = bfs(x, y, u, v)
    # Imprimir el resultado
    print(resultado)
```

```
(2, [(0, 0), (1, 1)])
(4, [(1, 0), (1, 1), (2, 2), (3, 3)])
(1, [(2, 1)])
```

[36]:
```
import numpy as np
```

4.6 Ejercicio 10

Nuestro mundo ha sido invadido por extraterrestres que cambian de forma y secuestran personas y roban sus identidades. Eres un inspector de un grupo de trabajo dedicado a detectarlos y capturarlos. Como tal, se te proporcionaron herramientas especiales para detectar extraterrestres y diferenciarlos de los humanos reales. Tu misión actual es visitar una ciudad que se sospecha que ha sido invadida, inspeccionar en secreto a todas las personas allí para saber quiénes son extraterrestres y quienes no, e informarlo todo al Cuartel General. Luego podrían enviar fuerzas a la ciudad por sorpresa y capturar a todos los alienígenas a la vez.

Los extraterrestres conocen el trabajo de inspectores como usted y están monitoreando todos los canales de radio para detectar la transmisión de dichos informes, con el fin de anticipar cualquier represalia. Por lo tanto, se han realizado varios esfuerzos para cifrar los informes y el método más reciente utiliza polinomios.

La ciudad que debes visitar tiene N ciudadanos, cada uno identificado por un número entero par distinto de 2 a 2N. Desea encontrar un polinomio P tal que, para cada ciudadano i, $P(i) > 0$ si el ciudadano i es un ser humano, y $P(i) < 0$ en caso contrario. Este polinomio será transmitido a la sede. Con el objetivo de minimizar el ancho de banda, el polinomio tiene algunos requisitos adicionales: cada raíz y coeficiente debe ser un número entero, el coeficiente de su término de mayor grado debe ser 1 o -1, y su grado debe ser el más bajo posible. Para cada ciudadano, sabes si es humano o no. Dada esta información, debes encontrar un polinomio que satisfaga las restricciones descritas.

Entrada La entrada consta de una sola línea que contiene una cadena S de longitud N (1 N 104), donde N es la población de la ciudad. Para i = 1, 2, ..., N, el i-ésimo carácter de S es la letra mayúscula "H" o la letra mayúscula "A", lo que indica respectivamente que el ciudadano 2i es un humano o un extraterrestre.

```
caso 1
HHH
caso 2
AHHA
caso 3
AHHHAH
```

Salida La primera línea debe contener un número entero D que indique el grado de un polinomio que satisface las restricciones descritas. La segunda línea debe contener D + 1 números enteros que representen los coeficientes del polinomio, en orden decreciente de los términos correspondientes. Se garantiza que existe al menos una solución tal que el valor absoluto de cada coeficiente sea menor que 263.

```
caso 1
0 1
caso 2
2-
1 10 -21
caso 3
3
1 -23 159 -297
```

[37]:
```python
def evaluar_polinomio(coeficientes, x):
    """
    Evalúa el polinomio dado en el valor x utilizando los coeficientes␣
 ↪proporcionados.
    """
    return sum(coef * (x ** (len(coeficientes) - i - 1)) for i, coef in␣
 ↪enumerate(coeficientes))
```

[38]:
```python
def verificar_signos(S, coeficientes):
    """
    Verifica si el polinomio cumple con las condiciones de signo.
    """
    puntos = []
    for i, c in enumerate(S):
        if c == 'H':
            puntos.append((2 * (i + 1), 1))  # Humanos -> (2i, 1)
        elif c == 'A':
            puntos.append((2 * (i + 1), -1))  # Extraterrestres -> (2i, -1)

    # Comprobar los valores del polinomio en los puntos
    for (x, esperado) in puntos:
        valor = evaluar_polinomio(coeficientes, x)
```

```
        if (esperado == 1 and valor <= 0) or (esperado == -1 and valor >= 0):
            return False
    return True
```

[39]:
```
def generar_polinomio(S):
    """
    Genera el polinomio mínimo que cumple las condiciones especificadas para␣
↪humanos y extraterrestres,
    comenzando con grado 0 y aumentando hasta encontrar el polinomio adecuado,␣
↪incluyendo hasta grado 4.
    """
    grado = 0
    while grado <= 4:
        print(f"\nProbando polinomio de grado {grado}")

        for a in [1, -1]:
            for b in range(-50, 51, 10):
                for c in range(-50, 51, 10):
                    for d in range(-50, 51, 10):
                        for e in range(-50, 51, 10):
                            if grado == 0:
                                coeficientes = [a]
                            elif grado == 1:
                                coeficientes = [a, b]
                            elif grado == 2:
                                coeficientes = [a, b, c]
                            elif grado == 3:
                                coeficientes = [a, b, c, d]
                            else:
                                coeficientes = [a, b, c, d, e]

                            if any(abs(coef) > 263 for coef in coeficientes):
                                continue

                            for constante in range(-500, 501, 10):
                                coeficientes[-1] = constante

                                if verificar_signos(S, coeficientes):
                                    print(f"Polinomio de grado {grado}␣
↪encontrado:")
                                    print("Coeficientes:", coeficientes)
                                    # Imprimir el polinomio en formato legible
                                    polinomio = []
                                    for i, coef in enumerate(coeficientes):
                                        exponente = grado - i
                                        if coef == 0:
                                            continue
```

```
                                        elif exponente == 0:
                                            polinomio.append(f"{coef}")
                                        elif exponente == 1:
                                            polinomio.append(f"{coef}x")
                                        else:
                                            polinomio.
↪append(f"{coef}x^{exponente}")

                                    polinomio_str = " + ".join(polinomio).
↪replace(" + -", " - ")
                                    print(f"Polinomio: P(x) = {polinomio_str}")
                                    return coeficientes
            grado += 1  # Incrementar el grado y probar con el siguiente

    print("No se encontró un polinomio válido usando la búsqueda.")
    return None
```

4.6.1 Probando metodo

[40]:
```
S1 = "HHH"
S2 = "AHHA"
S3 = "AHHHAH"

generar_polinomio(S1)
generar_polinomio(S2)
generar_polinomio(S3)
```

```
Probando polinomio de grado 0
Polinomio de grado 0 encontrado:
Coeficientes: [10]
Polinomio: P(x) = 10

Probando polinomio de grado 0

Probando polinomio de grado 1

Probando polinomio de grado 2
Polinomio de grado 2 encontrado:
Coeficientes: [-1, 10, -20]
Polinomio: P(x) = -1x^2 + 10x - 20

Probando polinomio de grado 0

Probando polinomio de grado 1

Probando polinomio de grado 2
```

```
Probando polinomio de grado 3

Probando polinomio de grado 4
No se encontró un polinomio válido usando la búsqueda.
```

[ ]: