Algoritmos de optimización - Seminario

Nombre y Apellidos: Luis Enrique Sanchez Zamora
Url: https://github.com/sanieni6/03MIAR---Algoritmos-de-Optimizacion
Problema:

> 1. Combinar cifras y operaciones

Descripción del problema:

- El problema consiste en analizar el siguiente problema y diseñar un algoritmo que lo resuelva.
- Disponemos de las 9 cifras del 1 al 9 (excluimos el cero) y de los 4 signos básicos de las operaciones fundamentales: suma(+), resta(-), multiplicación(*) y división(/)
- Debemos combinarlos alternativamente sin repetir ninguno de ellos para obtener una cantidad dada. Un ejemplo sería para obtener el 4: 4+2-6/3*1 = 4

• Debe analizarse el problema para encontrar todos los valores enteros posibles planteando las siguientes cuestiones:

- ¿Qué valor máximo y mínimo se pueden obtener según las condiciones del problema?
- ¿Es posible encontrar todos los valores enteros posibles entre dicho mínimo y máximo ? • Nota: Es posible usar la función de python “eval” para evaluar una expresión

(*) La respuesta es obligatoria

¿Qué valor máximo y mínimo se pueden obtener según las condiciones del problema?

Mínimo: el valor mínimo posible que se puede obtener es -69

Máximo: el valor máximo posible es 77

¿Es posible encontrar todos los valores enteros posibles entre dicho mínimo y máximo ?

Si es posible encontrar todos los valores enteros posibles entre dicho mínimo y máximo.

Objetivo.

Usando las cifras del 1 al 9 (una sola vez cada una de ellas) y los signos de operaciones basicas +, -, *, / (una sola vez cada uno de ellos) se debe obtener expresiones que den como resultado un numero entero.

Restricciones.

1. No se repite ninguna cifra.
2. No se repite ningun signo de operacion.
3. No se permite concatenar cifras.
4. No se permite usar agrupaciones '()'
5. El resultado de la expresion debe ser un numero entero.

Ejemplo.

4+2-6/3*1 = 4

Desarrollo.

Nos encontramos ante un problema de combinatoria, especificamente de permutacion de elementos sin repeticion (de cifras y de signos de operacion).

Que se esta combinando?

- **9 cifras (1,2,3,4,5,6,7,8,9) Se usan sin repetir, pero solo se seleccionan 5 de ellas para formar una expresion de la siguiente forma:**

c1 op1 c2 op2 c3 op3 c4 c5

5 digitos y 4 signos de operacion intercalados.

- **4 signos de operacion (+, -, *, /) Se usan todos sin repetir.**

Expresion Matematica combinatoria.

1. **Selección y permutación de 5 cifras de 9:**

$$P(9,5) = \frac{9!}{(9-5)!} = 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 = 15{,}120$$

1. **Permutación de los 4 operadores:**

Hay 4 operadores distintos, se usan todos sin repetir:

$$P(4,4) = 4! = 24$$

1. **Total de combinaciones posibles (sin considerar evaluación o validez):**

$$\text{Total expresiones posibles} = P(9,5) \cdot P(4,4) = 15{,}120 \cdot 24 = \boxed{362{,}880}$$

(*)¿Cuantas posibilidades hay sin tener en cuenta las restricciones?

Sin tomar en cuenta las restricciones, se tiene que:

1. **Permitimos repetir cifras y operadores, por ende,**
2. **Un total de 9 opciones para cada número hasta llegar a 5 números. Esto nos da un total de**

$$P(9) = 9^5 = 59049$$

3. **hay 4 operadores posibles (suma, resta, multiplicación y división) para cada posición, (se permite repetir operadores)**

$$P(4) = 4^4 = 256$$

4. **por ende, el total de posibilidades es:**

$$P(9) \cdot P(4) = 362{,}880 \cdot 256 = \boxed{15{,}116{,}544}$$

¿Cuantas posibilidades hay teniendo en cuenta todas las restricciones.

Teniendo en cuenta las restricciones, se tiene que:

1. **Selección y permutación de 5 cifras de 9:**

$$
\begin{aligned}
&P(9,5) \\
&= \frac{9!}{(9-5)!} = 9 \\
&\cdot 8 \cdot 7 \cdot 6 \cdot 5 \\
&= 15{,}120
\end{aligned}
$$

1. **Permutación de los 4 operadores:**

Hay 4 operadores distintos, se usan todos sin repetir:

$$
\begin{aligned}
P(4,4) &= 4! \\
&= 24
\end{aligned}
$$

1. **Total de combinaciones posibles (sin considerar evaluación o validez):**

$$\text{Total expresiones posibles} = P(9,5) \cdot P(4,4) = 15{,}120 \cdot 24 = \boxed{362{,}880}$$

Modelo para el espacio de soluciones

(*) ¿Cual es la estructura de datos que mejor se adapta al problema? Argumentalo.(Es posible que hayas elegido una al principio y veas la necesidad de cambiar, arguentalo)

La estructura de datos que mejor se adapta al problema son las listas, ya que se pueden almacenar los operadores y cifras de manera ordenada y se pueden acceder a ellos de manera eficiente. De esta manera tenemos que se uso dos listas, una para los operadores y otra para las cifras.

Si bien el ejercicio no restringe si se aceptan resultados repetidos, se puede usar un conjunto para almacenar los resultados y así evitar duplicados.

Según el modelo para el espacio de soluciones

(*)¿Cual es la función objetivo?

La función objetivo no busca optimizar una cantidad (como maximizar valor o minimizar coste), sino verificar si una expresión construida con dígitos y operadores da como resultado un número entero. Por tanto, la función objetivo es:

Evaluar la expresión generada y comprobar si su resultado es un número entero (dado)

(*)¿Es un problema de maximización o minimización?

No. Este no es un problema de optimización clásica , donde se busca un máximo o mínimo de una función, sino un problema de búsqueda y enumeración:

- **Es un problema de búsqueda sobre un espacio finito de soluciones posibles (expresiones).**
- **Se desea explorar todo el espacio de soluciones posibles y recolectar los resultados que sean números enteros.**
- **Ademas, dado un valor objetivo, se desea encontrar una expresion que lo evalue.**

El objetivo final no es encontrar el mejor resultado, sino:

1. **Hallar todos los resultados enteros posibles.**
2. **Identificar el mínimo y máximo a posteriori, no como meta de optimización, sino como analisis dentro de un conjunto de resultados.**

Diseña un algoritmo para resolver el problema por fuerza bruta

Pseudocódigo

```
INICIO

  definir CONJUNTO_RESULTADOS ← conjunto vacío

  para cada PERMUTACION_CIFRAS en PERMUTACIONES(1..9, tomar 5):
    para cada PERMUTACION_OPERADORES en PERMUTACIONES(['+', '-', '*', '/']):

      EXPRESION ← lista vacía

      para i desde 0 hasta 4:
          añadir PERMUTACION_CIFRAS[i] a EXPRESION
          si i < 4:
              añadir PERMUTACION_OPERADORES[i] a EXPRESION

      RESULTADO ← evaluar(EXPRESION)

      si RESULTADO es un número entero:
          añadir RESULTADO al CONJUNTO_RESULTADOS

  MIN ← mínimo(CONJUNTO_RESULTADOS)
  MAX ← máximo(CONJUNTO_RESULTADOS)

  TODOS_PRESENTES ← (CONJUNTO_RESULTADOS == conjunto de enteros entre MIN y MAX)

  imprimir "Valor mínimo:", MIN
  imprimir "Valor máximo:", MAX
  imprimir "¿Todos los valores enteros presentes?:", TODOS_PRESENTES

FIN
```

In [112]:

```
# Importamos algunas librerias que usaremos a lo largo de todo el notebook
# Importamos la libreria time para medir el tiempo de ejecución
import time

# Importamos la libreria itertools para generar permutaciones (usado para la compobrar el resultado de fuerza bruta)
import itertools

# importamos numpy ya que lo usaremos en el ultimo apartado para generar numeros aleatorios
import numpy as np
```

In [ ]:

```
# Definimos los digitos y operadores
digitos = ['1','2','3','4','5','6','7','8','9']
operadores = ['+','-','*','/']

# Numeros objetivo usados para comprobar el algoritmo de backtracking con poda
n1 = 4
n2 = 10
n3 = -65
n4 = 76
```

Solucion por fuerza bruta sin Librerias.

In [ ]:

In [ ]:

```
# Generar permutaciones sin repetición
def permutaciones(arr, r):
    """
    Genera todas las permutaciones de r elementos de un arreglo arr.
    """
    if r == 0:
        return [[]]
    perms = []
    for i in range(len(arr)): #iteramos sobre todos los elementos del arreglo
        elem = arr[i]
        restantes = arr[:i] + arr[i+1:]
        for subperm in permutaciones(restantes, r - 1): #generamos todas las permutacion
es de los elementos restantes
            perms.append([elem] + subperm) #agregamos el elemento actual a la permutació
n
    return perms #devolvemos todas las permutaciones
```

In [118]:

```
# Evaluar expresión respetando jerarquía de operadores
def evaluar_expresion(expresion):
    """
    Evalúa una expresión matemática representada como una lista de tokens.
    """
    try:
        i = 0
        while i < len(expresion):
            if expresion[i] == '*':
                res = int(expresion[i-1]) * int(expresion[i+1])
                expresion = expresion[:i-1] + [str(res)] + expresion[i+2:]
                i = 0
            elif expresion[i] == '/':
                if int(expresion[i+1]) == 0:
                    return None
                res = int(expresion[i-1]) / int(expresion[i+1])
                if res != int(res):
                    return None
                expresion = expresion[:i-1] + [str(int(res))] + expresion[i+2:]
                i = 0
            else:
                i += 1

        i = 0
        while i < len(expresion):
            if expresion[i] == '+':
                res = int(expresion[i-1]) + int(expresion[i+1])
                expresion = expresion[:i-1] + [str(res)] + expresion[i+2:]
                i = 0
            elif expresion[i] == '-':
                res = int(expresion[i-1]) - int(expresion[i+1])
                expresion = expresion[:i-1] + [str(res)] + expresion[i+2:]
                i = 0
            else:
                i += 1

        return int(expresion[0])
    except:
        return None
```

In [ ]:

```
def generar_permutaciones(digitos, operadores, truncar_output=50):
    """
    Genera todas las permutaciones de 5 cifras y 4 operadores.
    """
    resultados_set = set()
    expresiones = []

    cifras_perm = permutaciones(digitos, 5) # Generamos todas las permutaciones de 5 cif
ras
```

```
    operadores_perm = permutaciones(operadores, 4) # Generamos todas las permutaciones de 4 operadores

    for cifras in cifras_perm: #iteramos sobre todas las permutaciones de 5 cifras
        for ops in operadores_perm: #iteramos sobre todas las permutaciones de 4 operadores
            expresion = []
            for i in range(4): #iteramos sobre los 4 operadores
                expresion.append(cifras[i])
                expresion.append(ops[i])
            expresion.append(cifras[4])

            expresiones.append(''.join(expresion))  # Guardar siempre la expresión

            resultado = evaluar_expresion(expresion) #evaluamos la expresión
            if resultado is not None: #si el resultado no es None, lo agregamos al conjunto de resultados
                resultados_set.add(int(resultado)) #agregamos el resultado al conjunto de resultados

    if not resultados_set:
        print("⚠ No se encontraron resultados.")
        return

    min_val = min(resultados_set) #valor minimo
    max_val = max(resultados_set) #valor maximo
    todos = set(range(min_val, max_val + 1)) #conjunto de todos los valores posibles

    print(f"\n✔ Valores posibles (sin repeticiones): {sorted(resultados_set)}")
    print(f"📉 Valor mínimo: {min_val}")
    print(f"📈 Valor máximo: {max_val}")
    print(f"🔍 ¿Todos los enteros entre min y max están presentes?: {resultados_set == todos}")
    print(f"🔢 Total de valores únicos: {len(resultados_set)}")

    print(f"\n📋 Ejemplos de expresiones evaluadas (primeros {truncar_output} de {len(expresiones)}):\n")
    for expr in expresiones[:truncar_output]:
        print(expr)

    print(f"\n📦 Total de expresiones evaluadas (con repetición): {len(expresiones)}")
```

Evaluar todas las expresiones posibles sin un valor objetivo

In [ ]:

```
# ejecución del algoritmo
inicio = time.time()

generar_permutaciones(digitos, operadores) #generamos todas las permutaciones de 5 cifras y 4 operadores

fin = time.time()
print(f"Tiempo de ejecución: {fin - inicio:.3f} segundos")
```

```
✔ Valores posibles (sin repeticiones): [-69, -68, -67, -66, -65, -64, -63, -62, -61, -60, -59, -58, -57, -56, -55, -54, -53, -52, -51, -50, -49, -48, -47, -46, -45, -44, -43, -42, -41, -40, -39, -38, -37, -36, -35, -34, -33, -32, -31, -30, -29, -28, -27, -26, -25, -24, -23, -22, -21, -20, -19, -18, -17, -16, -15, -14, -13, -12, -11, -10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77]
📉 Valor mínimo: -69
📈 Valor máximo: 77
🔍 ¿Todos los enteros entre min y max están presentes?: True
🔢 Total de valores únicos: 147

📋 Ejemplos de expresiones evaluadas (primeros 50 de 362880):

1+2-3*4/5
```

```
1+2-3/4*5
1+2*3-4/5
1+2*3/4-5
1+2/3-4*5
1+2/3*4-5
1-2+3*4/5
1-2+3/4*5
1-2*3+4/5
1-2*3/4+5
1-2/3+4*5
1-2/3*4+5
1*2+3-4/5
1*2+3/4-5
1*2-3+4/5
1*2-3/4+5
1*2/3+4-5
1*2/3-4+5
1/2+3-4*5
1/2+3*4-5
1/2-3+4*5
1/2-3*4+5
1/2*3+4-5
1/2*3-4+5
1+2-3*4/6
1+2-3/4*6
1+2*3-4/6
1+2*3/4-6
1+2/3-4*6
1+2/3*4-6
1-2+3*4/6
1-2+3/4*6
1-2*3+4/6
1-2*3/4+6
1-2/3+4*6
1-2/3*4+6
1*2+3-4/6
1*2+3/4-6
1*2-3+4/6
1*2-3/4+6
1*2/3+4-6
1*2/3-4+6
1/2+3-4*6
1/2+3*4-6
1/2-3+4*6
1/2-3*4+6
1/2*3+4-6
1/2*3-4+6
1+2-3*4/7
1+2-3/4*7

 Total de expresiones evaluadas (con repetición): 362880
Tiempo de ejecución: 1.118 segundos
```

Busqueda de todas las expresiones que evalúen a un objetivo

In [ ]:

```
def buscar_expresiones_para_valor(objetivo):
    """
    Busca todas las expresiones que evalúan a un objetivo dado.
    """

    cifras_perm = permutaciones(digitos, 5) #generamos todas las permutaciones de 5 cifr
as
    operadores_perm = permutaciones(operadores, 4) #generamos todas las permutaciones de
4 operadores

    soluciones = [] #lista para almacenar las soluciones

    for cifras in cifras_perm: #iteramos sobre todas las permutaciones de 5 cifras
```

```
        for ops in operadores_perm: #iteramos sobre todas las permutaciones de 4 operadores
            expresion = []
            for i in range(4): #iteramos sobre los 4 operadores
                expresion.append(cifras[i]) #agregamos la cifra actual a la expresión
                expresion.append(ops[i]) #agregamos el operador actual a la expresión
            expresion.append(cifras[4]) #agregamos la ultima cifra a la expresión

            resultado = evaluar_expresion(expresion) #evaluamos la expresión
            if resultado == objetivo: #si el resultado es igual al objetivo, agregamos la expresión a la lista de soluciones
                soluciones.append(''.join(expresion)) #agregamos la expresión a la lista de soluciones

    return soluciones
```

In [62]:

```
objetivo = 4
inicio = time.time()

soluciones = buscar_expresiones_para_valor(objetivo)

fin = time.time()

print(f"\n Soluciones que evalúan a {objetivo}: ({len(soluciones)} encontradas)\n")
for expr in soluciones:
    print(expr)

print(f"\n Tiempo de ejecución: {fin - inicio:.3f} segundos")
```

```
 Soluciones que evalúan a 4: (2112 encontradas)

1-2*3/6+4
1-2/3*6+7
1/2*4-3+5
1/2*4+5-3
1/2*4-5+7
1*2+4-6/3
1/2*4-6+8
1/2*4+7-5
1/2*4-7+9
1/2*4+8-6
1-2/4*8+7
1/2*4+9-7
1*2+5-9/3
1-2/6*3+4
1*2-6/3+4
1/2*6-3+4
1-2*6/3+7
1/2*6+4-3
1/2*6-4+5
1/2*6+5-4
1/2*6-7+8
1/2*6+8-7
1/2*6-8+9
1/2*6+9-8
1-2*8/4+7
1*2-9/3+5
1-3/2*4+9
1-3*2/6+4
1+3/2*8-9
1-3*4/2+9
1*3+4-6/2
1-3*4/6+5
1-3+4/6*9
1-3/4*8+9
1-3+4*9/6
1*3+5-8/2
1-3/6*2+4
1*3-6/2+4
1/3*6-2+4
```

1/3*6+4-2
1-3/6*4+5
1/3*6-5+7
1/3*6+7-5
1/3*6-7+9
1-3/6*8+7
1-3*6/9+5
1/3*6+9-7
1*3-8/2+5
1+3*8/2-9
1-3*8/4+9
1-3*8/6+7
1/3*9-4+5
1-3+9*4/6
1/3*9+5-4
1/3*9-5+6
1-3+9/6*4
1-3/9*6+5
1/3*9+6-5
1/3*9-6+7
1/3*9+7-6
1/3*9-7+8
1/3*9+8-7
1*4/2-3+5
1+4-2*3/6
1-4/2*3+9
1*4/2+5-3
1+4/2*5-7
1*4/2-5+7
1+4-2/6*3
1*4+2-6/3
1*4-2+6/3
1*4/2-6+8
1+4/2*6-9
1*4/2+7-5
1*4/2-7+9
1*4/2+8-6
1*4/2+9-7
1+4-3*2/6
1-4*3/2+9
1+4-3/6*2
1*4+3-6/2
1*4-3+6/2
1+4/3*6-5
1-4*3/6+5
1+4*5/2-7
1*4+6/2-3
1*4-6/2+3
1+4*6/2-9
1*4+6/3-2
1*4-6/3+2
1+4*6/3-5
1-4/6*3+5
1+4/6*9-3
1/4*8-3+5
1/4*8+5-3
1/4*8-5+7
1/4*8+7-5
1/4*8-7+9
1/4*8+9-7
1+4*9/6-3
1+5/2*4-7
1*5+2-9/3
1*5-3+4/2
1+5-3*4/6
1+5-3/6*4
1+5/3*6-7
1+5-3*6/9
1*5+3-8/2
1*5-3+8/4
1+5-3/9*6
1*5+4/2-3

1+5*4/2-7
1+5-4*3/6
1-5+4/3*6
1*5-4+6/2
1+5-4/6*3
1-5+4*6/3
1+5/4*8-7
1*5-4+9/3
1*5+6/2-4
1-5+6/3*4
1+5*6/3-7
1+5-6*3/9
1-5+6*4/3
1+5-6/9*3
1*5-8/2+3
1*5+8/4-3
1+5*8/4-7
1*5-9/3+2
1*5+9/3-4
1*6/2-3+4
1-6*2/3+7
1*6/2+4-3
1*6/2-4+5
1+6/2*4-9
1*6/2+5-4
1*6/2-7+8
1*6/2+8-7
1*6/2-8+9
1*6/2+9-8
1*6/3-2+4
1-6/3*2+7
1*6/3+4-2
1+6/3*4-5
1+6/3*5-7
1*6/3-5+7
1*6/3+7-5
1*6/3-7+9
1-6*3/9+5
1*6/3+9-7
1+6*4/2-9
1+6*4/3-5
1+6/4*8-9
1+6*5/3-7
1*6-5+9/3
1+6*8/4-9
1-6/9*3+5
1*6+9/3-5
1+7-2/3*6
1+7-2/4*8
1+7-2*6/3
1+7-2*8/4
1+7-3/6*8
1+7-3*8/6
1-7+4/2*5
1*7+4/2-5
1-7+4*5/2
1-7+5/2*4
1-7+5/3*6
1-7+5*4/2
1*7-5+4/2
1-7+5/4*8
1-7+5*6/3
1*7-5+6/3
1-7+5*8/4
1*7-5+8/4
1+7-6*2/3
1+7-6/3*2
1-7+6/3*5
1*7+6/3-5
1-7+6*5/3
1*7-6+9/3
1+7-8*2/4

1+7-8*3/6
1+7-8/4*2
1-7+8/4*5
1*7+8/4-5
1-7+8*5/4
1+7-8/6*3
1*7+9/3-6
1+8/2*3-9
1-8*2/4+7
1+8*3/2-9
1-8*3/4+9
1-8*3/6+7
1*8+4/2-6
1-8/4*2+7
1*8/4-3+5
1-8/4*3+9
1*8/4+5-3
1+8/4*5-7
1*8/4-5+7
1+8/4*6-9
1*8/4+7-5
1*8/4-7+9
1*8/4+9-7
1+8*5/4-7
1*8+6/2-7
1-8/6*3+7
1*8-6+4/2
1+8*6/4-9
1*8-7+6/2
1*8-7+9/3
1*8+9/3-7
1+9-3/2*4
1-9+3/2*8
1+9-3*4/2
1*9/3-4+5
1+9-3/4*8
1*9/3+5-4
1*9/3-5+6
1*9/3+6-5
1*9/3-6+7
1*9/3+7-6
1*9/3-7+8
1-9+3*8/2
1+9-3*8/4
1*9/3+8-7
1+9-4/2*3
1-9+4/2*6
1*9+4/2-7
1+9-4*3/2
1-9+4*6/2
1+9*4/6-3
1-9+6/2*4
1*9+6/2-8
1*9+6/3-7
1-9+6*4/2
1+9/6*4-3
1-9+6/4*8
1-9+6*8/4
1*9-7+4/2
1*9-7+6/3
1*9-7+8/4
1-9+8/2*3
1-9+8*3/2
1+9-8*3/4
1+9-8/4*3
1-9+8/4*6
1*9+8/4-7
1*9-8+6/2
1-9+8*6/4
2/1*3+4-6
2/1*3+5-7
2-1/3*6+4

2/1*3-6+4
2/1*3+6-8
2/1*3-7+5
2/1*3+7-9
2/1*3-8+6
2-1/3*9+5
2/1*3-9+7
2/1*4+3-7
2/1*4+5-9
2+1*4-6/3
2*1+4-6/3
2-1+4*6/8
2/1*4-7+3
2-1+4/8*6
2/1*4-9+5
2/1*5+3-9
2+1*5-9/3
2*1+5-9/3
2/1*5-9+3
2-1*6/3+4
2*1-6/3+4
2-1+6*4/8
2-1+6/8*4
2-1*9/3+5
2*1-9/3+5
2*3/1+4-6
2*3/1+5-7
2*3/1-6+4
2*3/1+6-8
2*3/1-7+5
2*3/1+7-9
2*3/1-8+6
2*3/1-9+7
2*3+4/1-6
2*3+4-6/1
2*3+5/1-7
2*3+5-7/1
2*3-6/1+4
2*3/6-1+4
2*3+6/1-8
2*3-6+4/1
2*3/6+4-1
2*3/6-4+7
2*3/6-5+8
2*3/6+7-4
2*3+6-8/1
2*3/6+8-5
2-3*6/9+4
2*3-7/1+5
2*3+7/1-9
2*3-7+5/1
2*3+7-9/1
2*3-8/1+6
2*3-8+6/1
2*3-9/1+7
2/3*9+4-6
2/3*9+5-7
2-3/9*6+4
2/3*9-6+4
2/3*9+6-8
2*3-9+7/1
2/3*9-7+5
2/3*9-8+6
2+4-1/3*6
2*4/1+3-7
2*4/1+5-9
2+4-1*6/3
2+4*1-6/3
2*4/1-7+3
2*4/1-9+5
2*4+3/1-7
2+4-3*6/9

2*4+3-7/1
2+4-3/9*6
2*4+5/1-9
2*4+5-9/1
2+4-6*1/3
2+4-6/3*1
2+4-6*3/9
2/4*6-7+8
2+4*6/8-1
2-4*6/8+5
2/4*6+8-7
2/4*6-8+9
2+4-6/9*3
2-4/6*9+8
2/4*6+9-8
2*4-7/1+3
2*4-7+3/1
2*4-7+9/3
2*4/8-3+6
2+4/8*6-1
2*4/8+6-3
2-4/8*6+5
2*4/8-6+9
2*4/8+9-6
2*4-9/1+5
2*4+9/3-7
2*4-9+5/1
2-4*9/6+8
2+5-1/3*9
2*5/1+3-9
2+5-1*9/3
2+5*1-9/3
2*5/1-9+3
2*5+3/1-9
2+5/3*6-8
2*5+3-9/1
2+5-4*6/8
2+5-4/8*6
2+5*6/3-8
2*5+6/3-8
2+5-6*4/8
2+5-6/8*4
2*5-8+6/3
2+5-9*1/3
2*5-9/1+3
2+5-9/3*1
2*5-9+3/1
2-6*1/3+4
2-6/3+1*4
2-6/3*1+4
2/6*3-1+4
2-6/3+4*1
2/6*3+4-1
2/6*3-4+7
2+6/3*5-8
2/6*3-5+8
2/6*3+7-4
2/6*3+8-5
2-6*3/9+4
2*6/4-7+8
2+6*4/8-1
2-6*4/8+5
2*6/4+8-7
2*6/4-8+9
2*6/4+9-8
2+6*5/3-8
2+6/8*4-1
2-6/8*4+5
2-6/9*3+4
2/6*9-3+4
2/6*9+4-3
2/6*9-4+5

2/6*9+5-4
2/6*9-7+8
2/6*9+8-7
2/8*4-3+6
2/8*4+6-3
2+8-4/6*9
2/8*4-6+9
2+8-4*9/6
2/8*4+9-6
2-8+5/3*6
2-8+5*6/3
2-8+6/3*5
2-8+6*5/3
2+8-9*4/6
2+8-9/6*4
2-9*1/3+5
2-9/3+1*5
2-9/3*1+5
2*9/3+4-6
2-9/3+5*1
2*9/3+5-7
2*9/3-6+4
2*9/3+6-8
2*9/3-7+5
2*9/3-8+6
2-9*4/6+8
2*9/6-3+4
2*9/6+4-3
2*9/6-4+5
2-9/6*4+8
2*9/6+5-4
2*9/6-7+8
2*9/6+8-7
3/1*2+4-6
3/1+2*4-7
3/1-2*4+9
3/1*2+5-7
3/1+2*5-9
3-1/2*6+4
3/1*2-6+4
3/1*2+6-8
3/1*2-7+5
3/1*2+7-9
3-1/2*8+5
3/1*2-8+6
3/1*2-9+7
3/1+4*2-7
3/1-4*2+9
3+1*4-6/2
3*1+4-6/2
3/1+5*2-9
3+1*5-8/2
3*1+5-8/2
3-1*6/2+4
3*1-6/2+4
3/1-7+2*4
3/1-7+4*2
3-1*8/2+5
3*1-8/2+5
3/1+9-2*4
3/1-9+2*5
3/1+9-4*2
3/1-9+5*2
3*2/1+4-6
3+2/1*4-7
3-2/1*4+9
3*2/1+5-7
3+2/1*5-9
3*2/1-6+4
3*2/1+6-8
3*2/1-7+5
3*2/1+7-9

3*2/1-8+6
3*2/1-9+7
3*2+4/1-6
3+2*4/1-7
3-2*4/1+9
3/2*4+5-7
3*2+4-6/1
3-2+4*6/8
3/2*4+6-8
3+2*4-7/1
3/2*4-7+5
3/2*4+7-9
3-2/4*8+5
3-2+4/8*6
3/2*4-8+6
3-2*4+9/1
3/2*4-9+7
3*2+5/1-7
3+2*5/1-9
3*2+5-7/1
3+2*5-9/1
3*2-6/1+4
3*2/6-1+4
3*2+6/1-8
3*2-6+4/1
3*2/6+4-1
3*2/6-4+7
3-2+6*4/8
3/2*6+4-9
3*2/6-5+8
3*2/6+7-4
3*2+6-8/1
3-2+6/8*4
3*2/6+8-5
3-2/6*9+4
3/2*6-9+4
3*2-7/1+5
3*2+7/1-9
3*2-7+5/1
3*2+7-9/1
3*2-8/1+6
3/2*8+1-9
3-2*8/4+5
3*2-8+6/1
3/2*8-9+1
3*2-9/1+7
3-2*9/6+4
3*2-9+7/1
3+4-1/2*6
3+4/1*2-7
3-4/1*2+9
3+4-1*6/2
3+4*1-6/2
3+4*2/1-7
3-4*2/1+9
3*4/2+5-7
3+4/2*5-9
3*4/2+6-8
3+4-2/6*9
3+4*2-7/1
3*4/2-7+5
3*4/2+7-9
3*4/2-8+6
3-4*2+9/1
3+4-2*9/6
3*4/2-9+7
3+4*5/2-9
3+4-6*1/2
3+4-6/2*1
3*4/6-5+7
3*4/6+7-5
3*4/6-7+9

3+4*6/8-2
3+4/6*9-5
3-4/6*9+7
3*4/6+9-7
3/4*8+5-7
3+4/8*6-2
3/4*8-7+5
3/4*8+7-9
3/4*8-9+7
3+4-9*2/6
3+4-9/6*2
3+4*9/6-5
3-4*9/6+7
3+5-1/2*8
3+5/1*2-9
3+5-1*8/2
3+5*1-8/2
3+5*2/1-9
3+5-2/4*8
3+5/2*4-9
3+5-2*8/4
3+5*2-9/1
3+5*4/2-9
3-5+4/6*9
3+5/4*8-9
3-5+4*9/6
3+5-8*1/2
3+5-8/2*1
3+5-8*2/4
3+5-8/4*2
3+5*8/4-9
3-5+9*4/6
3-5+9/6*4
3-6*1/2+4
3-6/2+1*4
3-6/2*1+4
3/6*2-1+4
3-6/2+4*1
3/6*2+4-1
3/6*2-4+7
3*6/2+4-9
3/6*2-5+8
3/6*2+7-4
3/6*2+8-5
3*6/2-9+4
3/6*4-5+7
3/6*4+7-5
3/6*4-7+9
3+6*4/8-2
3/6*4+9-7
3+6/8*4-2
3*6/9-2+4
3*6/9+4-2
3*6/9-5+7
3*6/9+7-5
3-7/1+2*4
3-7/1+4*2
3-7+2/1*4
3-7+2*4/1
3-7+4/1*2
3-7+4*2/1
3+7-4/6*9
3+7-4*9/6
3+7-9*4/6
3+7-9/6*4
3-8*1/2+5
3-8/2+1*5
3-8/2*1+5
3*8/2+1-9
3-8*2/4+5
3-8/2+5*1
3*8/2-9+1

3-8/4*2+5
3*8/4+5-7
3+8/4*5-9
3*8/4-7+5
3*8/4+7-9
3*8/4-9+7
3+8*5/4-9
3+9/1-2*4
3-9/1+2*5
3+9/1-4*2
3-9/1+5*2
3+9-2/1*4
3-9+2/1*5
3+9-2*4/1
3-9+2*5/1
3-9*2/6+4
3+9-4/1*2
3+9-4*2/1
3-9+4/2*5
3-9+4*5/2
3+9*4/6-5
3-9*4/6+7
3-9+5/1*2
3-9+5*2/1
3-9+5/2*4
3-9+5*4/2
3-9+5/4*8
3-9+5*8/4
3-9/6*2+4
3/9*6-2+4
3/9*6+4-2
3+9/6*4-5
3-9/6*4+7
3/9*6-5+7
3/9*6+7-5
3-9+8/4*5
3-9+8*5/4
4*1/2-3+5
4+1-2*3/6
4-1+2*3/6
4/1+2*3-6
4/1-2*3+6
4/1*2+3-7
4*1/2+5-3
4*1/2-5+7
4/1*2+5-9
4+1-2/6*3
4+1*2-6/3
4+1/2*6-3
4-1+2/6*3
4-1*2+6/3
4-1/2*6+3
4*1+2-6/3
4*1-2+6/3
4*1/2-6+8
4/1*2-7+3
4*1/2+7-5
4*1/2-7+9
4*1/2+8-6
4/1*2-9+5
4*1/2+9-7
4+1-3*2/6
4-1+3*2/6
4/1+3*2-6
4/1-3*2+6
4+1-3/6*2
4+1*3-6/2
4+1/3*6-2
4-1+3/6*2
4-1*3+6/2
4-1/3*6+2
4*1+3-6/2

4*1-3+6/2
4+1*6/2-3
4-1*6/2+3
4*1+6/2-3
4*1-6/2+3
4/1+6-2*3
4/1-6+2*3
4+1*6/3-2
4-1*6/3+2
4*1+6/3-2
4*1-6/3+2
4/1+6-3*2
4/1-6+3*2
4/2-1*3+5
4/2*1-3+5
4+2-1/3*6
4+2/1*3-6
4-2+1/3*6
4-2/1*3+6
4*2/1+3-7
4/2+1*5-3
4/2*1+5-3
4/2-1*5+7
4/2*1-5+7
4*2/1+5-9
4+2-1*6/3
4+2*1-6/3
4-2+1*6/3
4-2*1+6/3
4/2-1*6+8
4/2*1-6+8
4*2/1-7+3
4/2+1*7-5
4/2*1+7-5
4/2-1*7+9
4/2*1-7+9
4/2+1*8-6
4/2*1+8-6
4*2/1-9+5
4/2+1*9-7
4/2*1+9-7
4/2-3+1*5
4/2-3*1+5
4+2*3/1-6
4-2*3/1+6
4*2+3/1-7
4/2-3+5*1
4/2*3+5-7
4+2*3-6/1
4+2*3/6-1
4-2*3+6/1
4-2*3/6+1
4/2*3+6-8
4+2-3*6/9
4-2+3*6/9
4*2+3-7/1
4/2*3-7+5
4/2*3+7-9
4/2*3-8+6
4+2-3/9*6
4+2/3*9-6
4-2+3/9*6
4-2/3*9+6
4/2*3-9+7
4/2+5-1*3
4/2+5*1-3
4/2-5+1*7
4/2-5*1+7
4/2*5+1-7
4*2+5/1-9
4/2+5-3*1
4/2*5+3-9

4/2-5+7*1
4/2*5-7+1
4*2+5-9/1
4/2*5-9+3
4+2-6*1/3
4-2+6*1/3
4/2-6+1*8
4/2-6*1+8
4/2*6+1-9
4+2-6/3*1
4+2/6*3-1
4-2+6/3*1
4-2/6*3+1
4+2-6*3/9
4-2+6*3/9
4/2-6+8*1
4/2*6-9+1
4+2-6/9*3
4+2/6*9-3
4-2+6/9*3
4-2/6*9+3
4*2-7/1+3
4/2+7-1*5
4/2+7*1-5
4/2-7+1*9
4/2-7*1+9
4*2-7+3/1
4/2+7-5*1
4/2-7+9*1
4*2-7+9/3
4/2+8-1*6
4/2+8*1-6
4*2/8-3+6
4/2+8-6*1
4*2/8+6-3
4*2/8-6+9
4*2/8+9-6
4*2-9/1+5
4/2+9-1*7
4/2+9*1-7
4+2*9/3-6
4-2*9/3+6
4*2+9/3-7
4*2-9+5/1
4+2*9/6-3
4-2*9/6+3
4/2+9-7*1
4+3-1/2*6
4+3/1*2-6
4-3+1/2*6
4-3/1*2+6
4+3-1*6/2
4+3*1-6/2
4-3+1*6/2
4-3*1+6/2
4+3*2/1-6
4-3*2/1+6
4*3/2+5-7
4+3*2-6/1
4+3*2/6-1
4-3*2+6/1
4-3*2/6+1
4*3/2+6-8
4+3-2/6*9
4+3/2*6-9
4-3+2/6*9
4-3/2*6+9
4*3/2-7+5
4*3/2+7-9
4*3/2-8+6
4+3-2*9/6
4-3+2*9/6

4*3/2-9+7
4+3-6*1/2
4-3+6*1/2
4/3*6+1-5
4+3-6/2*1
4+3/6*2-1
4-3+6/2*1
4-3/6*2+1
4+3*6/2-9
4-3*6/2+9
4/3*6-5+1
4*3/6-5+7
4/3*6+5-9
4*3/6+7-5
4*3/6-7+9
4+3*6/9-2
4-3*6/9+2
4/3*6-9+5
4*3/6+9-7
4+3-9*2/6
4-3+9*2/6
4+3-9/6*2
4+3/9*6-2
4-3+9/6*2
4-3/9*6+2
4*5/2+1-7
4*5/2+3-9
4*5/2-7+1
4*5/2-9+3
4+6*1/2-3
4+6/1-2*3
4-6*1/2+3
4-6/1+2*3
4+6*1/3-2
4+6/1-3*2
4-6*1/3+2
4-6/1+3*2
4+6-2/1*3
4+6/2-1*3
4+6/2*1-3
4-6+2/1*3
4-6/2+1*3
4-6/2*1+3
4*6/2+1-9
4+6-2*3/1
4+6/2-3*1
4-6+2*3/1
4-6/2+3*1
4+6-2/3*9
4+6/2*3-9
4-6+2/3*9
4-6/2*3+9
4*6/2-9+1
4+6-2*9/3
4-6+2*9/3
4+6-3/1*2
4+6/3-1*2
4+6/3*1-2
4-6+3/1*2
4-6/3+1*2
4-6/3*1+2
4*6/3+1-5
4+6-3*2/1
4+6/3-2*1
4-6+3*2/1
4-6/3+2*1
4+6*3/2-9
4-6*3/2+9
4*6/3-5+1
4/6*3-5+7
4*6/3+5-9
4/6*3+7-5

4/6*3-7+9
4+6*3/9-2
4-6*3/9+2
4*6/3-9+5
4/6*3+9-7
4*6/8-1+2
4*6/8+2-1
4*6/8-2+3
4*6/8+3-2
4/6*9+1-3
4+6-9*2/3
4-6+9*2/3
4/6*9-3+1
4+6-9/3*2
4+6/9*3-2
4-6+9/3*2
4-6/9*3+2
4/6*9+3-5
4/6*9-5+3
4/6*9+5-7
4/6*9-7+5
4/8*2-3+6
4/8*2+6-3
4/8*2-6+9
4/8*2+9-6
4/8*6-1+2
4/8*6+2-1
4/8*6-2+3
4/8*6+3-2
4+9*2/3-6
4-9*2/3+6
4+9*2/6-3
4-9*2/6+3
4+9-3/2*6
4+9/3*2-6
4-9+3/2*6
4-9/3*2+6
4+9-3*6/2
4-9+3*6/2
4*9/6+1-3
4+9-6/2*3
4+9/6*2-3
4-9+6/2*3
4-9/6*2+3
4*9/6-3+1
4+9-6*3/2
4-9+6*3/2
4*9/6+3-5
4*9/6-5+3
4*9/6+5-7
4*9/6-7+5
5/1+2*3-7
5/1*2+3-9
5+1/2*4-3
5/1-2*4+7
5/1+2*4-9
5+1/2*6-4
5-1/2*8+3
5+1*2-9/3
5*1+2-9/3
5/1*2-9+3
5/1+3*2-7
5-1*3+4/2
5*1-3+4/2
5+1-3*4/6
5+1-3/6*4
5+1-3*6/9
5+1*3-8/2
5*1+3-8/2
5-1*3+8/4
5*1-3+8/4
5-1/3*9+2

5+1/3*9-4
5+1-3/9*6
5+1*4/2-3
5*1+4/2-3
5/1-4*2+7
5/1+4*2-9
5+1-4*3/6
5-1*4+6/2
5*1-4+6/2
5+1-4/6*3
5+1/4*8-3
5-1*4+9/3
5*1-4+9/3
5+1*6/2-4
5*1+6/2-4
5+1-6*3/9
5+1-6/9*3
5/1-7+2*3
5/1+7-2*4
5/1-7+3*2
5/1+7-4*2
5-1*8/2+3
5*1-8/2+3
5+1*8/4-3
5*1+8/4-3
5/1-9+2*4
5-1*9/3+2
5*1-9/3+2
5+1*9/3-4
5*1+9/3-4
5/1-9+4*2
5+2/1*3-7
5+2-1/3*9
5*2/1+3-9
5-2/1*4+7
5+2/1*4-9
5+2-1*9/3
5+2*1-9/3
5*2/1-9+3
5+2*3/1-7
5*2+3/1-9
5+2*3-7/1
5*2+3-9/1
5+2/3*9-7
5-2*4/1+7
5/2*4+1-7
5+2*4/1-9
5/2*4+3-9
5+2-4*6/8
5-2*4+7/1
5/2*4-7+1
5-2/4*8+3
5+2-4/8*6
5+2*4-9/1
5/2*4-9+3
5*2+6/3-8
5+2-6*4/8
5+2-6/8*4
5+2/6*9-4
5-2*8/4+3
5*2-8+6/3
5+2-9*1/3
5*2-9/1+3
5+2-9/3*1
5*2-9+3/1
5+2*9/3-7
5+2*9/6-4
5-3+1/2*4
5+3/1*2-7
5+3-1/2*8
5-3+1*4/2
5-3*1+4/2

5-3+1/4*8
5+3-1*8/2
5+3*1-8/2
5-3+1*8/4
5-3*1+8/4
5+3*2/1-7
5+3/2*4-7
5+3-2/4*8
5-3/2*6+8
5+3*2-7/1
5+3-2*8/4
5-3+4*1/2
5-3+4/2*1
5+3*4/2-7
5-3*4/6+1
5+3/4*8-7
5/3*6+1-7
5-3*6/2+8
5/3*6+2-8
5-3/6*4+1
5/3*6-7+1
5/3*6-8+2
5-3*6/9+1
5+3-8*1/2
5-3+8*1/4
5+3-8/2*1
5+3-8*2/4
5-3+8/4*1
5+3-8/4*2
5+3*8/4-7
5-3/9*6+1
5+4*1/2-3
5-4+1/2*6
5-4/1*2+7
5+4/1*2-9
5-4+1/3*9
5-4+1*6/2
5-4*1+6/2
5-4+1*9/3
5-4*1+9/3
5+4/2-1*3
5+4/2*1-3
5-4*2/1+7
5*4/2+1-7
5+4*2/1-9
5+4/2-3*1
5+4/2*3-7
5*4/2+3-9
5-4+2/6*9
5-4*2+7/1
5*4/2-7+1
5+4*2-9/1
5*4/2-9+3
5-4+2*9/6
5+4*3/2-7
5-4*3/6+1
5-4/3*6+7
5+4/3*6-9
5-4+6*1/2
5-4+6/2*1
5-4/6*3+1
5-4*6/3+7
5+4*6/3-9
5-4*6/8+2
5+4/6*9-7
5/4*8+1-7
5/4*8+3-9
5-4/8*6+2
5/4*8-7+1
5/4*8-9+3
5-4+9*1/3
5-4+9*2/6

5-4+9/3*1
5-4+9/6*2
5+4*9/6-7
5+6*1/2-4
5+6/2-1*4
5+6/2*1-4
5-6/2*3+8
5+6/2-4*1
5*6/3+1-7
5-6*3/2+8
5*6/3+2-8
5-6/3*4+7
5+6/3*4-9
5*6/3-7+1
5*6/3-8+2
5-6*3/9+1
5-6*4/3+7
5+6*4/3-9
5-6*4/8+2
5-6/8*4+2
5-6/9*3+1
5-7/1+2*3
5+7/1-2*4
5-7/1+3*2
5+7/1-4*2
5-7+2/1*3
5+7-2/1*4
5-7+2*3/1
5-7+2/3*9
5+7-2*4/1
5-7+2*9/3
5-7+3/1*2
5-7+3*2/1
5-7+3/2*4
5-7+3*4/2
5-7+3/4*8
5-7+3*8/4
5+7-4/1*2
5+7-4*2/1
5-7+4/2*3
5-7+4*3/2
5+7-4/3*6
5+7-4*6/3
5-7+4/6*9
5-7+4*9/6
5+7-6/3*4
5+7-6*4/3
5-7+8*3/4
5-7+8/4*3
5-7+9*2/3
5-7+9/3*2
5-7+9*4/6
5-7+9/6*4
5-8*1/2+3
5+8*1/4-3
5-8/2+1*3
5-8/2*1+3
5-8/2+3*1
5-8*2/4+3
5+8-3/2*6
5+8*3/4-7
5+8-3*6/2
5+8/4-1*3
5+8/4*1-3
5*8/4+1-7
5-8/4*2+3
5+8/4-3*1
5+8/4*3-7
5*8/4+3-9
5*8/4-7+1
5*8/4-9+3
5+8-6/2*3

5+8-6*3/2
5-9/1+2*4
5-9*1/3+2
5+9*1/3-4
5-9/1+4*2
5-9+2/1*4
5+9*2/3-7
5-9+2*4/1
5+9*2/6-4
5-9/3+1*2
5-9/3*1+2
5+9/3-1*4
5+9/3*1-4
5-9/3+2*1
5+9/3*2-7
5+9/3-4*1
5-9+4/1*2
5-9+4*2/1
5-9+4/3*6
5-9+4*6/3
5+9*4/6-7
5+9/6*2-4
5-9+6/3*4
5-9+6*4/3
5+9/6*4-7
6*1/2-3+4
6/1-2*3+4
6/1+2*3-8
6*1/2+4-3
6*1/2-4+5
6*1/2+5-4
6/1-2*5+8
6*1/2-7+8
6*1/2+8-7
6*1/2-8+9
6*1/2+9-8
6*1/3-2+4
6/1-3*2+4
6/1+3*2-8
6*1/3+4-2
6*1/3-5+7
6*1/3+7-5
6*1/3-7+9
6+1/3*9-5
6*1/3+9-7
6/1+4-2*3
6/1+4-3*2
6/1-5*2+8
6-1*5+9/3
6*1-5+9/3
6/1-8+2*3
6/1+8-2*5
6/1-8+3*2
6/1+8-5*2
6+1*9/3-5
6*1+9/3-5
6-2/1*3+4
6/2-1*3+4
6/2*1-3+4
6+2/1*3-8
6/2+1*4-3
6/2*1+4-3
6/2-1*4+5
6/2*1-4+5
6/2+1*5-4
6/2*1+5-4
6-2/1*5+8
6/2-1*7+8
6/2*1-7+8
6/2+1*8-7
6/2*1+8-7
6/2-1*8+9

6/2*1-8+9
6/2+1*9-8
6/2*1+9-8
6-2*3/1+4
6/2-3+1*4
6/2-3*1+4
6+2*3/1-8
6-2*3+4/1
6/2-3+4*1
6/2*3+4-9
6+2*3-8/1
6-2/3*9+4
6/2*3-9+4
6+2/3*9-8
6/2+4-1*3
6/2+4*1-3
6/2-4+1*5
6/2-4*1+5
6/2*4+1-9
6/2+4-3*1
6/2-4+5*1
6*2/4-7+8
6+2*4/8-3
6*2/4+8-7
6*2/4-8+9
6/2*4-9+1
6*2/4+9-8
6/2+5-1*4
6/2+5*1-4
6-2*5/1+8
6/2+5-4*1
6-2*5+8/1
6/2-7+1*8
6/2-7*1+8
6/2-7+8*1
6/2+8-1*7
6/2+8*1-7
6/2-8+1*9
6/2-8*1+9
6+2/8*4-3
6/2+8-7*1
6/2-8+9*1
6/2+9-1*8
6/2+9*1-8
6-2*9/3+4
6+2*9/3-8
6/2+9-8*1
6-3/1*2+4
6/3-1*2+4
6/3*1-2+4
6+3/1*2-8
6/3+1*4-2
6/3*1+4-2
6/3-1*5+7
6/3*1-5+7
6/3+1*7-5
6/3*1+7-5
6/3-1*7+9
6/3*1-7+9
6/3+1*9-7
6/3*1+9-7
6-3*2/1+4
6/3-2+1*4
6/3-2*1+4
6+3*2/1-8
6-3*2+4/1
6/3-2+4*1
6+3/2*4-8
6-3+2*4/8
6*3/2+4-9
6/3+2*5-8
6+3*2-8/1

6-3+2/8*4
6*3/2-9+4
6/3+4-1*2
6/3+4*1-2
6/3*4+1-5
6/3+4-2*1
6+3*4/2-8
6-3+4*2/8
6/3*4-5+1
6/3*4+5-9
6-3+4/8*2
6/3*4-9+5
6/3-5+1*7
6/3-5*1+7
6/3*5+1-7
6/3+5*2-8
6/3*5+2-8
6/3-5+7*1
6/3*5-7+1
6/3*5-8+2
6/3+7-1*5
6/3+7*1-5
6/3-7+1*9
6/3-7*1+9
6/3+7-5*1
6/3-7+9*1
6/3-8+2*5
6/3-8+5*2
6/3+9-1*7
6/3+9*1-7
6*3/9-2+4
6*3/9+4-2
6*3/9-5+7
6/3+9-7*1
6*3/9+7-5
6+4/1-2*3
6+4/1-3*2
6+4-2/1*3
6*4/2+1-9
6+4-2*3/1
6+4/2*3-8
6+4-2/3*9
6-4/2*5+8
6/4*2-7+8
6+4*2/8-3
6/4*2+8-7
6/4*2-8+9
6*4/2-9+1
6+4-2*9/3
6/4*2+9-8
6+4-3/1*2
6*4/3+1-5
6+4-3*2/1
6+4*3/2-8
6*4/3-5+1
6*4/3+5-9
6*4/3-9+5
6-4*5/2+8
6*4/8-1+2
6/4*8+1-9
6*4/8+2-1
6+4/8*2-3
6*4/8-2+3
6*4/8+3-2
6/4*8-9+1
6+4-9*2/3
6+4-9/3*2
6-5/1*2+8
6-5+1/3*9
6-5+1*9/3
6-5*1+9/3
6-5*2/1+8

6-5/2*4+8
6-5*2+8/1
6*5/3+1-7
6*5/3+2-8
6*5/3-7+1
6*5/3-8+2
6-5*4/2+8
6-5+9*1/3
6-5+9/3*1
6-8/1+2*3
6+8/1-2*5
6-8/1+3*2
6+8/1-5*2
6-8+2/1*3
6+8-2/1*5
6-8+2*3/1
6-8+2/3*9
6+8-2*5/1
6-8+2*9/3
6-8+3/1*2
6-8+3*2/1
6-8+3/2*4
6-8+3*4/2
6/8*4-1+2
6*8/4+1-9
6/8*4+2-1
6-8+4/2*3
6/8*4-2+3
6+8-4/2*5
6-8+4*3/2
6/8*4+3-2
6+8-4*5/2
6*8/4-9+1
6+8-5/1*2
6+8-5*2/1
6+8-5/2*4
6+8-5*4/2
6-8+9*2/3
6-8+9/3*2
6+9*1/3-5
6-9*2/3+4
6+9*2/3-8
6+9/3-1*5
6+9/3*1-5
6-9/3*2+4
6/9*3-2+4
6+9/3*2-8
6/9*3+4-2
6+9/3-5*1
6/9*3-5+7
6/9*3+7-5
7+1-2/3*6
7/1+2*3-9
7+1/2*4-5
7/1-2*4+5
7+1-2/4*8
7+1-2*6/3
7/1-2*6+9
7+1-2*8/4
7/1+3*2-9
7/1-3*4+9
7+1/3*6-5
7+1-3/6*8
7+1-3*8/6
7+1/3*9-6
7+1*4/2-5
7*1+4/2-5
7/1-4*2+5
7/1-4*3+9
7+1/4*8-5
7/1+5-2*4
7-1*5+4/2

7*1-5+4/2
7/1+5-4*2
7-1*5+6/3
7*1-5+6/3
7-1*5+8/4
7*1-5+8/4
7+1-6*2/3
7/1-6*2+9
7+1-6/3*2
7+1*6/3-5
7*1+6/3-5
7-1*6+9/3
7*1-6+9/3
7+1-8*2/4
7+1-8*3/6
7+1-8/4*2
7+1*8/4-5
7*1+8/4-5
7+1-8/6*3
7/1-9+2*3
7/1+9-2*6
7/1-9+3*2
7/1+9-3*4
7+1*9/3-6
7*1+9/3-6
7/1+9-4*3
7/1+9-6*2
7+2/1*3-9
7-2/1*4+5
7-2/1*6+9
7+2*3/1-9
7-2/3*6+1
7+2*3/6-4
7+2*3-9/1
7-2*4/1+5
7-2*4+5/1
7-2/4*8+1
7-2*6/1+9
7-2*6/3+1
7+2/6*3-4
7-2*6+9/1
7-2*8/4+1
7+3/1*2-9
7-3/1*4+9
7+3*2/1-9
7+3/2*4-9
7+3*2/6-4
7-3/2*8+9
7+3*2-9/1
7-3*4/1+9
7+3*4/2-9
7+3*4/6-5
7+3-4/6*9
7+3/4*8-9
7-3*4+9/1
7+3-4*9/6
7+3/6*2-4
7+3/6*4-5
7-3/6*8+1
7+3*6/9-5
7-3*8/2+9
7+3*8/4-9
7-3*8/6+1
7+3-9*4/6
7+3-9/6*4
7+3/9*6-5
7+4*1/2-5
7-4/1*2+5
7-4/1*3+9
7+4/2-1*5
7+4/2*1-5
7-4*2/1+5

7-4+2*3/6
7+4/2*3-9
7+4/2-5*1
7-4*2+5/1
7-4+2/6*3
7-4/2*6+9
7-4*3/1+9
7-4+3*2/6
7+4*3/2-9
7-4+3/6*2
7+4*3/6-5
7-4/3*6+5
7-4*3+9/1
7-4*6/2+9
7+4/6*3-5
7-4*6/3+5
7-4/6*9+3
7-4*9/6+3
7+5/1-2*4
7-5+1/2*4
7-5+1/3*6
7+5/1-4*2
7-5+1*4/2
7-5*1+4/2
7-5+1/4*8
7-5+1*6/3
7-5*1+6/3
7-5+1*8/4
7-5*1+8/4
7+5-2/1*4
7+5-2*4/1
7-5+3*4/6
7-5+3/6*4
7-5+3*6/9
7-5+3/9*6
7+5-4/1*2
7-5+4*1/2
7+5-4*2/1
7-5+4/2*1
7+5-4/3*6
7-5+4*3/6
7+5-4*6/3
7-5+4/6*3
7-5+6*1/3
7-5+6/3*1
7+5-6/3*4
7-5+6*3/9
7+5-6*4/3
7-5+6/9*3
7-5+8*1/4
7-5+8/4*1
7-6/1*2+9
7+6*1/3-5
7-6+1/3*9
7-6+1*9/3
7-6*1+9/3
7-6*2/1+9
7-6*2/3+1
7-6/2*4+9
7-6*2+9/1
7+6/3-1*5
7+6/3*1-5
7-6/3*2+1
7-6/3*4+5
7+6/3-5*1
7+6*3/9-5
7-6*4/2+9
7-6*4/3+5
7-6/4*8+9
7-6*8/4+9
7-6+9*1/3
7-6+9/3*1

7+6/9*3-5
7+8*1/4-5
7-8/2*3+9
7-8*2/4+1
7-8*3/2+9
7+8*3/4-9
7-8*3/6+1
7+8/4-1*5
7+8/4*1-5
7-8/4*2+1
7+8/4*3-9
7+8/4-5*1
7-8/4*6+9
7-8/6*3+1
7-8*6/4+9
7-9/1+2*3
7+9/1-2*6
7-9/1+3*2
7+9/1-3*4
7+9*1/3-6
7+9/1-4*3
7+9/1-6*2
7-9+2/1*3
7+9-2/1*6
7-9+2*3/1
7+9-2*6/1
7-9+3/1*2
7+9-3/1*4
7+9/3-1*6
7+9/3*1-6
7-9+3*2/1
7-9+3/2*4
7+9-3/2*8
7+9-3*4/1
7-9+3*4/2
7-9+3/4*8
7+9/3-6*1
7+9-3*8/2
7-9+3*8/4
7+9-4/1*3
7-9+4/2*3
7+9-4/2*6
7+9-4*3/1
7-9+4*3/2
7+9-4*6/2
7-9*4/6+3
7+9-6/1*2
7+9-6*2/1
7+9-6/2*4
7+9-6*4/2
7-9/6*4+3
7+9-6/4*8
7+9-6*8/4
7+9-8/2*3
7+9-8*3/2
7-9+8*3/4
7-9+8/4*3
7+9-8/4*6
7+9-8*6/4
8+1/2*4-6
8/1-2*5+6
8+1/2*6-7
8+1/3*9-7
8+1*4/2-6
8*1+4/2-6
8*1/4-3+5
8*1/4+5-3
8*1/4-5+7
8*1/4+7-5
8*1/4-7+9
8*1/4+9-7
8/1-5*2+6

8/1+6-2*5
8+1*6/2-7
8*1+6/2-7
8-1*6+4/2
8*1-6+4/2
8/1+6-5*2
8-1*7+6/2
8*1-7+6/2
8-1*7+9/3
8*1-7+9/3
8+1*9/3-7
8*1+9/3-7
8-2/1*5+6
8/2*3+1-9
8+2*3/6-5
8/2*3-9+1
8+2/4*6-7
8+2-4/6*9
8+2-4*9/6
8-2*5/1+6
8-2*5+6/1
8+2/6*3-5
8+2*6/4-7
8+2/6*9-7
8+2-9*4/6
8+2-9/6*4
8+2*9/6-7
8*3/2+1-9
8+3*2/6-5
8-3/2*6+5
8*3/2-9+1
8*3/4+5-7
8*3/4-7+5
8*3/4+7-9
8*3/4-9+7
8+3/6*2-5
8-3*6/2+5
8+4*1/2-6
8/4-1*3+5
8/4*1-3+5
8/4+1*5-3
8/4*1+5-3
8/4-1*5+7
8/4*1-5+7
8/4+1*7-5
8/4*1+7-5
8/4-1*7+9
8/4*1-7+9
8/4+1*9-7
8/4*1+9-7
8+4/2-1*6
8+4/2*1-6
8-4/2*5+6
8+4/2-6*1
8/4-3+1*5
8/4-3*1+5
8/4-3+5*1
8/4*3+5-7
8/4*3-7+5
8/4*3+7-9
8/4*3-9+7
8/4+5-1*3
8/4+5*1-3
8/4-5+1*7
8/4-5*1+7
8/4*5+1-7
8-4*5/2+6
8/4+5-3*1
8/4*5+3-9
8/4-5+7*1
8/4*5-7+1
8/4*5-9+3

8/4*6+1-9
8/4*6-9+1
8-4/6*9+2
8/4+7-1*5
8/4+7*1-5
8/4-7+1*9
8/4-7*1+9
8/4+7-5*1
8/4-7+9*1
8/4+9-1*7
8/4+9*1-7
8-4*9/6+2
8/4+9-7*1
8-5/1*2+6
8-5*2/1+6
8-5+2*3/6
8-5/2*4+6
8-5*2+6/1
8-5+2/6*3
8+5-3/2*6
8-5+3*2/6
8+5-3*6/2
8-5+3/6*2
8*5/4+1-7
8-5*4/2+6
8*5/4+3-9
8*5/4-7+1
8*5/4-9+3
8+5-6/2*3
8+5-6*3/2
8-6+1/2*4
8+6/1-2*5
8+6*1/2-7
8-6+1*4/2
8-6*1+4/2
8+6/1-5*2
8+6-2/1*5
8+6/2-1*7
8+6/2*1-7
8-6/2*3+5
8+6*2/4-7
8+6-2*5/1
8+6/2-7*1
8-6*3/2+5
8-6+4*1/2
8*6/4+1-9
8-6+4/2*1
8+6-4/2*5
8+6/4*2-7
8+6-4*5/2
8*6/4-9+1
8+6-5/1*2
8+6-5*2/1
8+6-5/2*4
8+6-5*4/2
8-7+1/2*6
8-7+1/3*9
8-7+1*6/2
8-7*1+6/2
8-7+1*9/3
8-7*1+9/3
8-7+2/4*6
8-7+2*6/4
8-7+2/6*9
8-7+2*9/6
8-7+6*1/2
8-7+6/2*1
8-7+6*2/4
8-7+6/4*2
8-7+9*1/3
8-7+9*2/6
8-7+9/3*1

8-7+9/6*2
8+9*1/3-7
8+9*2/6-7
8+9/3-1*7
8+9/3*1-7
8+9/3-7*1
8-9*4/6+2
8+9/6*2-7
8-9/6*4+2
9/1-2*4+3
9+1/2*4-7
9/1-2*6+7
9+1/2*6-8
9+1-3/2*4
9/1+3-2*4
9+1-3*4/2
9/1+3-4*2
9*1/3-4+5
9/1-3*4+7
9+1-3/4*8
9*1/3+5-4
9*1/3-5+6
9*1/3+6-5
9+1/3*6-7
9*1/3-6+7
9*1/3+7-6
9*1/3-7+8
9+1-3*8/4
9*1/3+8-7
9+1-4/2*3
9/1-4*2+3
9+1*4/2-7
9*1+4/2-7
9+1-4*3/2
9/1-4*3+7
9+1/4*8-7
9/1-6*2+7
9+1*6/2-8
9*1+6/2-8
9+1*6/3-7
9*1+6/3-7
9/1+7-2*6
9/1+7-3*4
9-1*7+4/2
9*1-7+4/2
9/1+7-4*3
9/1+7-6*2
9-1*7+6/3
9*1-7+6/3
9-1*7+8/4
9*1-7+8/4
9+1-8*3/4
9+1-8/4*3
9+1*8/4-7
9*1+8/4-7
9-1*8+6/2
9*1-8+6/2
9-2/1*4+3
9-2/1*6+7
9*2/3+4-6
9*2/3+5-7
9*2/3-6+4
9*2/3+6-8
9*2/3-7+5
9*2/3-8+6
9-2*4/1+3
9-2*4+3/1
9+2/4*6-8
9+2*4/8-6
9-2*6/1+7
9*2/6-3+4
9*2/6+4-3

9*2/6-4+5
9+2*6/4-8
9*2/6+5-4
9-2*6+7/1
9*2/6-7+8
9*2/6+8-7
9+2/8*4-6
9+3/1-2*4
9+3/1-4*2
9/3-1*4+5
9/3*1-4+5
9-3/1*4+7
9/3+1*5-4
9/3*1+5-4
9/3-1*5+6
9/3*1-5+6
9/3+1*6-5
9/3*1+6-5
9/3-1*6+7
9/3*1-6+7
9/3+1*7-6
9/3*1+7-6
9/3-1*7+8
9/3*1-7+8
9/3+1*8-7
9/3*1+8-7
9+3-2/1*4
9+3-2*4/1
9-3/2*4+1
9/3*2+4-6
9/3+2*4-7
9/3*2+5-7
9-3/2*6+4
9/3*2-6+4
9/3*2+6-8
9/3*2-7+5
9/3*2-8+6
9-3/2*8+7
9+3-4/1*2
9/3-4+1*5
9/3-4*1+5
9-3*4/1+7
9+3-4*2/1
9-3*4/2+1
9/3+4*2-7
9/3-4+5*1
9+3*4/6-7
9-3*4+7/1
9-3/4*8+1
9/3+5-1*4
9/3+5*1-4
9/3-5+1*6
9/3-5*1+6
9/3+5-4*1
9/3-5+6*1
9/3+6-1*5
9/3+6*1-5
9/3-6+1*7
9/3-6*1+7
9-3*6/2+4
9+3/6*4-7
9/3+6-5*1
9/3-6+7*1
9/3+7-1*6
9/3+7*1-6
9/3-7+1*8
9/3-7*1+8
9/3-7+2*4
9/3-7+4*2
9/3+7-6*1
9/3-7+8*1
9/3+8-1*7

9/3+8*1-7
9-3*8/2+7
9-3*8/4+1
9/3+8-7*1
9-4/1*2+3
9+4*1/2-7
9-4/1*3+7
9-4*2/1+3
9+4/2-1*7
9+4/2*1-7
9-4*2+3/1
9-4/2*3+1
9-4/2*6+7
9+4/2-7*1
9+4*2/8-6
9-4*3/1+7
9-4*3/2+1
9+4-3/2*6
9+4-3*6/2
9+4*3/6-7
9-4*3+7/1
9*4/6+1-3
9+4-6/2*3
9-4*6/2+7
9*4/6-3+1
9+4-6*3/2
9*4/6+3-5
9+4/6*3-7
9*4/6-5+3
9*4/6+5-7
9*4/6-7+5
9+4/8*2-6
9-6/1*2+7
9+6*1/2-8
9+6*1/3-7
9-6*2/1+7
9+6/2-1*8
9+6/2*1-8
9-6/2*3+4
9/6*2-3+4
9/6*2+4-3
9/6*2-4+5
9-6/2*4+7
9+6*2/4-8
9-6+2*4/8
9/6*2+5-4
9-6*2+7/1
9/6*2-7+8
9+6/2-8*1
9-6+2/8*4
9/6*2+8-7
9+6/3-1*7
9+6/3*1-7
9-6*3/2+4
9+6/3-7*1
9/6*4+1-3
9-6*4/2+7
9+6/4*2-8
9-6+4*2/8
9/6*4-3+1
9/6*4+3-5
9/6*4-5+3
9/6*4+5-7
9/6*4-7+5
9-6+4/8*2
9-6/4*8+7
9-6*8/4+7
9-7+1/2*4
9+7/1-2*6
9+7/1-3*4
9-7+1/3*6
9-7+1*4/2

```
9-7*1+4/2
9+7/1-4*3
9-7+1/4*8
9+7/1-6*2
9-7+1*6/3
9-7*1+6/3
9-7+1*8/4
9-7*1+8/4
9+7-2/1*6
9+7-2*6/1
9+7-3/1*4
9+7-3/2*8
9+7-3*4/1
9-7+3*4/6
9-7+3/6*4
9+7-3*8/2
9-7+4*1/2
9+7-4/1*3
9-7+4/2*1
9+7-4/2*6
9+7-4*3/1
9-7+4*3/6
9+7-4*6/2
9-7+4/6*3
9+7-6/1*2
9-7+6*1/3
9+7-6*2/1
9+7-6/2*4
9-7+6/3*1
9+7-6*4/2
9+7-6/4*8
9+7-6*8/4
9-7+8*1/4
9+7-8/2*3
9+7-8*3/2
9-7+8/4*1
9+7-8/4*6
9+7-8*6/4
9-8+1/2*6
9+8*1/4-7
9-8+1*6/2
9-8*1+6/2
9-8/2*3+7
9-8+2/4*6
9-8+2*6/4
9-8*3/2+7
9-8*3/4+1
9+8/4-1*7
9+8/4*1-7
9-8/4*3+1
9-8/4*6+7
9+8/4-7*1
9-8+6*1/2
9-8+6/2*1
9-8+6*2/4
9-8+6/4*2
9-8*6/4+7
```

⏱ Tiempo de ejecución: 1.087 segundos

Busqueda de una expresión que evalúe a unobjetivo

In [ ]:

```
def buscar_expresion_para_valor(objetivo):
    """
    Busca una expresión que evalúe a un objetivo dado.
    """
    cifras_perm = permutaciones(digitos, 5) #generamos todas las permutaciones de 5 cifr
as
```

```
    operadores_perm = permutaciones(operadores, 4) #generamos todas las permutaciones de
4 operadores

    for cifras in cifras_perm: #iteramos sobre todas las permutaciones de 5 cifras
        for ops in operadores_perm:
            expresion = []
            for i in range(4): #iteramos sobre los 4 operadores
                expresion.append(cifras[i]) #agregamos la cifra actual a la expresión
                expresion.append(ops[i]) #agregamos el operador actual a la expresión
            expresion.append(cifras[4]) #agregamos la ultima cifra a la expresión

            resultado = evaluar_expresion(expresion) #evaluamos la expresión
            if resultado == objetivo: #si el resultado es igual al objetivo, devolvemos
la expresión
                return ''.join(expresion)  # Primera coincidencia

    return None  # Si no se encuentra ninguna
```

In [55]:

```
objetivo = 4
inicio = time.time()
expresion = buscar_expresion_para_valor(objetivo)
fin = time.time()

if expresion:
    print(f"Expresión que evalúa a {objetivo}: {expresion}")
    print(f"El resultado de la expresion es igual a {objetivo}: {eval(expresion) == objet
ivo}")
else:
    print(f" No se encontró ninguna expresión que evalúe a {objetivo}")

print(f" Tiempo de ejecución: {fin - inicio:.3f} segundos")
```

```
Expresión que evalúa a 4: 1-2*3/6+4
El resultado de la expresion es igual a 4: True
 Tiempo de ejecución: 0.016 segundos
```

Solución por fuerza bruta con la libreria itertools

In [ ]:

```
def evaluar_expresion_itertools(tokens):
    """
    Evalúa una expresión matemática representada como una lista de tokens.
    """
    try:
        expr = ''.join(tokens) #unimos los tokens para formar la expresión

        result = eval(expr) #evaluamos la expresión
        return result if result == int(result) else None #si el resultado es un número e
ntero, lo devolvemos, sino devolvemos None
    except ZeroDivisionError: #si hay una división por cero, devolvemos None
        return None
```

In [ ]:

```
def generar_permutaciones_itertools(digitos, operadores, truncar_output=50):
    """
    Genera todas las permutaciones de 5 cifras y 4 operadores.
    """
    resultados = set()
    expresiones = []  # Para almacenar TODAS las expresiones evaluadas

    # Generamos permutaciones de 5 dígitos de los 9 disponibles
    for nums in itertools.permutations(digitos, 5):
        for ops in itertools.permutations(operadores):
            tokens = []
            for i in range(4):
                tokens.append(nums[i]) #agregamos la cifra actual a la expresión
```

```
                tokens.append(ops[i]) #agregamos el operador actual a la expresión
            tokens.append(nums[4]) #agregamos la ultima cifra a la expresión

            resultado = evaluar_expresion(tokens)
            expresiones.append(''.join(tokens))  # Guardamos la expresión
            if resultado is not None: #si el resultado no es None, lo agregamos al conjun
to de resultados
                resultados.add(int(resultado))

    # Mostrar resultados
    min_val = min(resultados) #valor minimo
    max_val = max(resultados) #valor maximo
    todos = set(range(min_val, max_val + 1)) #conjunto de todos los valores posibles

    print(f"\n✔ Valores posibles (sin repeticiones): {sorted(resultados)}")
    print(f"□ Valor mínimo: {min_val}")
    print(f"□ Valor máximo: {max_val}")
    print(f"□ ¿Todos los enteros entre min y max están?: {resultados == todos}")
    print(f"□ Total de valores únicos: {len(resultados)}")

    # Mostrar expresiones (truncadas)
    print(f"\n□ Ejemplos de expresiones evaluadas (primeros {truncar_output} de {len(expr
esiones)}):\n")
    for expr in expresiones[:truncar_output]:
        print(expr)

    print(f"\n□ Total de expresiones evaluadas (con repetición): {len(expresiones)}")
```

Evaluar todas las expresiones posibles sin un valor objetivo

In [ ]:

```
inicio = time.time()
generar_permutaciones_itertools(digitos, operadores)
fin = time.time()
print(f"Tiempo de ejecución: {fin - inicio:.3f} segundos")
```

```
✔ Valores posibles (sin repeticiones): [-69, -68, -67, -66, -65, -64, -63, -62, -61, -60,
-59, -58, -57, -56, -55, -54, -53, -52, -51, -50, -49, -48, -47, -46, -45, -44, -43, -42,
-41, -40, -39, -38, -37, -36, -35, -34, -33, -32, -31, -30, -29, -28, -27, -26, -25, -24,
-23, -22, -21, -20, -19, -18, -17, -16, -15, -14, -13, -12, -11, -10, -9, -8, -7, -6, -5,
-4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43
, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65,
66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77]
□ Valor mínimo: -69
□ Valor máximo: 77
□ ¿Todos los enteros entre min y max están?: True
□ Total de valores únicos: 147

□ Ejemplos de expresiones evaluadas (primeros 50 de 362880):

1+2-3*4/5
1+2-3/4*5
1+2*3-4/5
1+2*3/4-5
1+2/3-4*5
1+2/3*4-5
1-2+3*4/5
1-2+3/4*5
1-2*3+4/5
1-2*3/4+5
1-2/3+4*5
1-2/3*4+5
1*2+3-4/5
1*2+3/4-5
1*2-3+4/5
1*2-3/4+5
1*2/3+4-5
1*2/3-4+5
1/2+3-4*5
```

```
1/2+3*4-5
1/2-3+4*5
1/2-3*4+5
1/2*3+4-5
1/2*3-4+5
1+2-3*4/6
1+2-3/4*6
1+2*3-4/6
1+2*3/4-6
1+2/3-4*6
1+2/3*4-6
1-2+3*4/6
1-2+3/4*6
1-2*3+4/6
1-2*3/4+6
1-2/3+4*6
1-2/3*4+6
1*2+3-4/6
1*2+3/4-6
1*2-3+4/6
1*2-3/4+6
1*2/3+4-6
1*2/3-4+6
1/2+3-4*6
1/2+3*4-6
1/2-3+4*6
1/2-3*4+6
1/2*3+4-6
1/2*3-4+6
1+2-3*4/7
1+2-3/4*7

 Total de expresiones evaluadas (con repetición): 362880
Tiempo de ejecución: 1.118 segundos
```

Busqueda de todas las expresiones que evalúen a un objetivo

In [ ]:

```
def buscar_expresiones_para_valor(digitos, operadores, objetivo, truncar_a=50):
    soluciones = []

    for nums in itertools.permutations(digitos, 5): #iteramos sobre todas las permutacio
nes de 5 cifras
        for ops in itertools.permutations(operadores): #iteramos sobre todas las permuta
ciones de 4 operadores
            tokens = [] #lista para almacenar los tokens
            for i in range(4): #iteramos sobre los 4 operadores
                tokens.append(nums[i])
                tokens.append(ops[i])
            tokens.append(nums[4])

            resultado = evaluar_expresion_itertools(tokens) #evaluamos la expresión
            if resultado == objetivo: #si el resultado es igual al objetivo, agregamos la
expresión a la lista de soluciones
                soluciones.append(''.join(tokens)) #agregamos la expresión a la lista de
soluciones

    # Mostrar soluciones
    print(f"\n Expresiones que evalúan a {objetivo} ({len(soluciones)} encontradas):")
    for expr in soluciones[:truncar_a]:
        print(expr)

    if len(soluciones) > truncar_a:
        print(f"... y {len(soluciones) - truncar_a} más.")

    print(f"\n Total de expresiones evaluadas que dan {objetivo}: {len(soluciones)}")
```

In [49]:

```
objetivo = 4

inicio = time.time()
buscar_expresiones_para_valor(digitos, operadores, objetivo, truncar_a=50)
fin = time.time()

print(f"\n Tiempo de ejecución: {fin - inicio:.3f} segundos")
```

```
 Expresiones que evalúan a 4 (2112 encontradas):
1-2*3/6+4
1-2/3*6+7
1/2*4-3+5
1/2*4+5-3
1/2*4-5+7
1*2+4-6/3
1/2*4-6+8
1/2*4+7-5
1/2*4-7+9
1/2*4+8-6
1-2/4*8+7
1/2*4+9-7
1*2+5-9/3
1-2/6*3+4
1*2-6/3+4
1/2*6-3+4
1-2*6/3+7
1/2*6+4-3
1/2*6-4+5
1/2*6+5-4
1/2*6-7+8
1/2*6+8-7
1/2*6-8+9
1/2*6+9-8
1-2*8/4+7
1*2-9/3+5
1-3/2*4+9
1-3*2/6+4
1+3/2*8-9
1-3*4/2+9
1*3+4-6/2
1-3*4/6+5
1-3+4/6*9
1-3/4*8+9
1-3+4*9/6
1*3+5-8/2
1-3/6*2+4
1*3-6/2+4
1/3*6-2+4
1/3*6+4-2
1-3/6*4+5
1/3*6-5+7
1/3*6+7-5
1/3*6-7+9
1-3/6*8+7
1-3*6/9+5
1/3*6+9-7
1*3-8/2+5
1+3*8/2-9
1-3*8/4+9
... y 2062 más.

 Total de expresiones evaluadas que dan 4: 2112

 Tiempo de ejecución: 1.121 segundos
```

Busqueda de una expresión que evalúe a unobjetivo

In [ ]:

```
import itertools
```

```
def buscar_expresion_para_valor_itertools(objetivo):

    cifras_perm = itertools.permutations(digitos, 5) #generamos todas las permutaciones de 5 cifras
    operadores_perm = list(itertools.permutations(operadores)) #generamos todas las permutaciones de 4 operadores

    for nums in cifras_perm: #iteramos sobre todas las permutaciones de 5 cifras
        for ops in operadores_perm: #iteramos sobre todas las permutaciones de 4 operadores
            tokens = [] #lista para almacenar los tokens
            for i in range(4): #iteramos sobre los 4 operadores
                tokens.append(nums[i]) #agregamos la cifra actual a la expresión
                tokens.append(ops[i]) #agregamos el operador actual a la expresión
            tokens.append(nums[4]) #agregamos la ultima cifra a la expresión

            expr = ''.join(tokens) #unimos los tokens para formar la expresión
            try:
                result = eval(expr) #evaluamos la expresión
                if result == objetivo: #si el resultado es igual al objetivo, devolvemos la expresión
                    return expr #devolvemos la expresión
            except ZeroDivisionError: #si hay una división por cero, continuamos con la siguiente permutación
                continue

    return None  # Si no se encuentra ninguna
```

In [59]:

```
objetivo = 4
inicio = time.time()
expr = buscar_expresion_para_valor_itertools(objetivo)
fin = time.time()

if expr:
    print(f"□ Primera expresión que evalúa a {objetivo}: {expr}")
else:
    print(f"□ No se encontró ninguna expresión que evalúe a {objetivo}")

print(f"□ Tiempo de ejecución: {fin - inicio:.3f} segundos")
```

```
□ Primera expresión que evalúa a 4: 1-2*3/6+4
□ Tiempo de ejecución: 0.003 segundos
```

Calcula la complejidad del algoritmo por fuerza bruta

Método / Funcionalidad	Todas las soluciones (sin filtro)	Todas las soluciones para número dado	Una solución para número dado	Notas
□ **Sin librerías (`perm()`)**	□ ~1.118 s	□ ~1.087 s	□ ~0.016 s	Implementación recursiva propia, más ligera, sin C-optim
⚙ **Con `itertools` (librerías estándar)**	□ ~1.118 s	□ ~1.121 s	□ ~0.003 s	Más rápida por estar optimizada en C

- **La complejidad teórica del algoritmo por fuerza bruta, para ambas soluciones, tanto en la que usa permutaciones como en la que usa itertools, es la misma.**
- **Como ya has visto antes, las combinaciones válidas (sin repetir cifras ni operadores) son:**

$$P(9,5) \cdot P(4,4)$$
$$= 15,120 \cdot 24$$
$$= \boxed{362,880}$$

Para ambas versiones, el costo de la busqueda de una expresion que evalue a un numero dado es de:

- **En el peor caso, se recorren todas las combinaciones:**

$$O(n) \text{ donde } n = 362{,}880$$

o bien,

$$O(n!) \text{ donde } n = 9$$

Por lo tanto, la complejidad es de orden factorial con respecto con respecto al número de combinaciones posibles:

$$\boxed{O(n!)}$$

(*)Diseña un algoritmo que mejore la complejidad del algortimo por fuerza bruta. Argumenta porque crees que mejora el algoritmo por fuerza bruta

En vista de que este es un problema de combinatoria, se propone usar backtracking con poda para mejorar la eficiencia del algoritmo de fuerza bruta.

En lugar de generar todas las combinaciones posibles (como hace la fuerza bruta), el algoritmo:

- **Construye expresiones paso a paso (número – operador – número – ...).**
- **Evalúa parcialmente durante la construcción.**
- **Descarta ramas (poda) que:**
- **Den divisiones no exactas.**
- **Generen resultados no enteros.**
- **Violen las restricciones (dígitos u operadores repetidos).**

In [122]:

```
def evaluar_expresion(expresion):
    """
    Evalúa una expresión matemática representada como una lista de tokens.
    """
    try:
        tokens = expresion[:]
        i = 0
        while i < len(tokens):
            if tokens[i] == '*':
                res = int(tokens[i - 1]) * int(tokens[i + 1])
                tokens = tokens[:i - 1] + [str(res)] + tokens[i + 2:]
                i = 0
            elif tokens[i] == '/':
                if int(tokens[i + 1]) == 0:
                    return None
                res = int(tokens[i - 1]) / int(tokens[i + 1])
                if res != int(res):
                    return None
                tokens = tokens[:i - 1] + [str(int(res))] + tokens[i + 2:]
                i = 0
            else:
                i += 1
        i = 0
        while i < len(tokens):
            if tokens[i] == '+':
                res = int(tokens[i - 1]) + int(tokens[i + 1])
                tokens = tokens[:i - 1] + [str(res)] + tokens[i + 2:]
                i = 0
            elif tokens[i] == '-':
                res = int(tokens[i - 1]) - int(tokens[i + 1])
                tokens = tokens[:i - 1] + [str(res)] + tokens[i + 2:]
                i = 0
            else:
                i += 1
```

```
        return int(tokens[0])
    except:
        return None
```

In [ ]:

```
def buscar_expresiones_para_valor(objetivo):
    """
    Busca todas las expresiones que evalúan a un objetivo dado.
    """
    soluciones = [] #lista para almacenar las soluciones

    def backtrack_con_poda(expresion, digitos_usados, operadores_usados):
        if len(expresion) == 9: #si la expresión tiene 9 tokens, evaluamos la expresión
            resultado = evaluar_expresion(expresion) #evaluamos la expresión
            if resultado == objetivo: #si el resultado es igual al objetivo, agregamos la
expresión a la lista de soluciones
                soluciones.append(''.join(expresion)) #agregamos la expresión a la lista
de soluciones
            return

        if len(expresion) % 2 == 0: #si la expresión tiene un número par de tokens, agre
gamos una cifra
            for d in digitos: #iteramos sobre todas las cifras
                if d not in digitos_usados: #si la cifra no ha sido usada, la agregamos
a la expresión
                    digitos_usados.add(d) #agregamos la cifra a la lista de cifras usada
s
                    expresion.append(d) #agregamos la cifra a la expresión
                    backtrack_con_poda(expresion, digitos_usados, operadores_usados) #ll
amamos a la función recursivamente
                    expresion.pop() #eliminamos la cifra de la expresión
                    digitos_usados.remove(d) #eliminamos la cifra de la lista de cifras
usadas
        else:  # operador
            for o in operadores: #iteramos sobre todos los operadores
                if o not in operadores_usados: #si el operador no ha sido usado, lo agre
gamos a la expresión
                    operadores_usados.add(o) #agregamos el operador a la lista de operad
ores usados
                    expresion.append(o) #agregamos el operador a la expresión
                    backtrack_con_poda(expresion, digitos_usados, operadores_usados) #ll
amamos a la función recursivamente
                    expresion.pop() #eliminamos el operador de la expresión
                    operadores_usados.remove(o) #eliminamos el operador de la lista de o
peradores usados

    backtrack_con_poda([], set(), set()) #llamamos a la función recursivamente
    return soluciones #devolvemos la lista de soluciones
```

Solucion propuesta: Backtracking con poda

In [125]:

```
def buscar_expresion_para_valor(objetivo):
    """
    Busca y retorna una única expresión (la primera encontrada)
    que evalúe exactamente al valor objetivo, usando backtracking con poda.
    """
    resultado = [None]  # Se usa lista mutable para poder modificar dentro de la función
interna

    def backtrack_con_poda(expresion, digitos_usados, usados_operadores):
        # si ya se encontró una solución, no seguir explorando
        if resultado[0] is not None:
            return

        # expresión completa (5 dígitos + 4 operadores = 9 tokens)
        if len(expresion) == 9:
            eval_result = evaluar_expresion(expresion)
```

```
            if eval_result == objetivo:
                resultado[0] = ''.join(expresion)  # Guardamos la primera coincidencia e
ncontrada
                return

        #  construir expresión paso a paso
        if len(expresion) % 2 == 0:
            # stamos en una posición de número
            for d in digitos:
                if d not in digitos_usados:
                    # no reutilizamos dígitos
                    digitos_usados.add(d)
                    expresion.append(d)

                    # seguimos explorando con el nuevo número
                    backtrack_con_poda(expresion, digitos_usados, usados_operadores)

                    #  deshacer cambios
                    expresion.pop()
                    digitos_usados.remove(d)
        else:
            # posición de operador
            for o in operadores:
                if o not in usados_operadores:
                    # no reutilizamos operadores
                    usados_operadores.add(o)
                    expresion.append(o)

                    # seguir explorando con el nuevo operador
                    backtrack_con_poda(expresion, digitos_usados, usados_operadores)

                    #  deshacer cambios
                    expresion.pop()
                    usados_operadores.remove(o)

    # comienza la exploración
    backtrack_con_poda([], set(), set())

    # devolvemos la expresión encontrada (o None si no hay)
    return resultado[0]
```

In [128]:

```
# Buscar todas las expresiones que evalúan a 4
inicio = time.time()
todas = buscar_expresiones_para_valor(4)
fin = time.time()
print(f"Tiempo de ejecución: {fin - inicio:.3f} segundos")
print(f" Total de expresiones para 4: {len(todas)}")
print("50 primeras expresiones:")
print(todas[:50])

# Buscar solo una expresión que evalúe a

inicio = time.time()
una = buscar_expresion_para_valor(n1) # 4
fin = time.time()
print(f"\n Expresión encontrada para 4: {una}")
print(f" Tiempo de ejecución: {fin - inicio:.3f} segundos")
inicio = time.time()
una = buscar_expresion_para_valor(n2) # 10
fin = time.time()
print(f"\n Expresión encontrada para 10: {una}")
print(f" Tiempo de ejecución: {fin - inicio:.3f} segundos")
inicio = time.time()
una = buscar_expresion_para_valor(n3) # 15
fin = time.time()
print(f"\n Expresión encontrada para -65: {una}")
print(f" Tiempo de ejecución: {fin - inicio:.3f} segundos")
inicio = time.time()
una = buscar_expresion_para_valor(n4) # 20
```

```
fin = time.time()
print(f"\n Expresión encontrada para 76: {una}")
print(f" Tiempo de ejecución: {fin - inicio:.3f} segundos")
```

```
Tiempo de ejecución: 0.354 segundos
 Total de expresiones para 4: 1676
50 primeras expresiones:
['1+3*8/2-9', '1+4-2*3/6', '1+4-3*2/6', '1+4*5/2-7', '1+4*6/2-9', '1+4*6/3-5', '1+4*9/6-3
', '1+4/2*5-7', '1+4/2*6-9', '1+5-3*4/6', '1+5-3*6/9', '1+5-4*3/6', '1+5-6*3/9', '1+5*4/2
-7', '1+5*6/3-7', '1+5*8/4-7', '1+6*4/2-9', '1+6*4/3-5', '1+6*5/3-7', '1+6*8/4-9', '1+6/2
*4-9', '1+6/3*4-5', '1+6/3*5-7', '1+7-2*6/3', '1+7-2*8/4', '1+7-3*8/6', '1+7-6*2/3', '1+7
-6/3*2', '1+7-8*2/4', '1+7-8*3/6', '1+7-8/4*2', '1+8*3/2-9', '1+8*5/4-7', '1+8*6/4-9', '1
+8/2*3-9', '1+8/4*5-7', '1+8/4*6-9', '1+9-3*4/2', '1+9-3*8/4', '1+9-4*3/2', '1+9-4/2*3',
'1+9-8*3/4', '1+9-8/4*3', '1+9*4/6-3', '1-2*3/6+4', '1-2*6/3+7', '1-2*8/4+7', '1-3+4*9/6'
, '1-3+9*4/6', '1-3*2/6+4']

 Expresión encontrada para 4: 1+3*8/2-9
 Tiempo de ejecución: 0.002 segundos

 Expresión encontrada para 10: 1+2*6-9/3
 Tiempo de ejecución: 0.001 segundos

 Expresión encontrada para -65: 2+5-8*9/1
 Tiempo de ejecución: 0.036 segundos

 Expresión encontrada para 76: 6+8*9-2/1
 Tiempo de ejecución: 0.191 segundos
```

Comparacion de los algoritmos

Método / Funcionalidad	Todas las soluciones (sin filtro)	Todas las soluciones para número dado (número 4)	Una solución para número dado (número 4)
Sin librerías (`perm()`)	**~1.118 s**	**~1.087 s**	**~0.016 s**
⚙ Con `itertools` (librerías estándar)	**~1.118 s**	**~1.121 s**	**~0.003 s**
Backtracking con poda	—	**~0.35 s**	**~0.002 s**

(*)Calcula la complejidad del algoritmo

La complejidad del algoritmo de backtracking con poda es de:

- **Mejor caso:** $\boxed{O(1)}$ **- solución encontrada en el primer camino válido**
- **Promedio:** $\boxed{O(k)}$**, con** $k \ll n$**, por la poda efectiva**
- **Peor caso:** $\boxed{O(n)}$**, si no hay solución o es la última**

Es de esta forma como podemos llegar a la conclusión de que la busqueda con backtracking y poda es O(n) en el peor caso, pero mucho menor en el promedio gracias a la poda.

Según el problema (y tenga sentido), diseña un juego de datos de entrada aleatorios

In [113]:

```
juego_aleatorio = np.random.randint(-69, 77, size=10)
print(juego_aleatorio)
```

```
[ -6 -35 -35 -56 -40 -28  48  63  56 -39]
```

Aplica el algoritmo al juego de datos generado

In [127]:

```
for numero in juego_aleatorio:
    inicio = time.time()
    resultado = buscar_expresion_para_valor(numero) # 4
    fin = time.time()
    print(f"Expresión encontrada para {numero}: {resultado}")
    print(f"Tiempo de ejecución: {fin - inicio:.3f} segundos")
```

```
Expresión encontrada para -6: 1+2*3/6-8
Tiempo de ejecución: 0.001 segundos
Expresión encontrada para -35: 2+3-5*8/1
Tiempo de ejecución: 0.053 segundos
Expresión encontrada para -35: 2+3-5*8/1
Tiempo de ejecución: 0.037 segundos
Expresión encontrada para -56: 2+5-7*9/1
Tiempo de ejecución: 0.037 segundos
Expresión encontrada para -40: 1+8/2-5*9
Tiempo de ejecución: 0.008 segundos
Expresión encontrada para -28: 1+6/2-4*8
Tiempo de ejecución: 0.005 segundos
Expresión encontrada para 48: 2+6*9-8/1
Tiempo de ejecución: 0.039 segundos
Expresión encontrada para 63: 2+7*9-6/3
Tiempo de ejecución: 0.041 segundos
Expresión encontrada para 56: 2+7*8-6/3
Tiempo de ejecución: 0.041 segundos
Expresión encontrada para -39: 1+4/2-6*7
Tiempo de ejecución: 0.003 segundos
```

Enumera las referencias que has utilizado(si ha sido necesario) para llevar a cabo el trabajo

[1] https://docs.python.org/3/library/itertools.html

[2] https://youtu.be/L0NxT2i-LOY?si=mMigY8is0VKzl1MA

Describe brevemente las lineas de como crees que es posible avanzar en el estudio del problema. Ten en cuenta incluso posibles variaciones del problema y/o variaciones al alza del tamaño

Limitándome al uso del algoritmo de backtracking con poda, es posible profundizar en estrategias de poda más agresivas, como la validación anticipada de subexpresiones o la eliminación temprana de ramas que no pueden alcanzar el valor objetivo. Estas optimizaciones reducirían significativamente el espacio de búsqueda y el tiempo de ejecución.

Por otro lado, si salimos del ámbito de los algoritmos deterministas, podríamos explorar enfoques heurísticos como algoritmos genéticos o búsqueda aleatoria voraz. Sin embargo, para el problema actual —con un espacio de soluciones limitado y bien definido— estos métodos no son necesarios. Solo serían útiles en caso de que se amplíen las restricciones del problema, como permitir más cifras, operadores, paréntesis u otras variantes que aumenten exponencialmente la complejidad o exijan resultados en tiempo más reducido.