Descripción:

¿Recuerdas cómo empezaste tu aventura en esta Olimpiada Nacional? Tal vez no lo recuerdes, pero comenzaste calculando expresiones matemáticas con dos operandos. Luego determinaste si una expresión formada solamente por paréntesis era balanceada. Ambos problemas no resultaron tan complicados, así que ahora vamos a explorar más allá. No me digas que...¿vamos a combinar ambos problemas? ¡Efectivamente! ¿Cómo resolverías expresiones matemáticas con varios paréntesis y un número arbitrario de operandos?

Entrada:

Una única línea, formada de una expresión matemática con un número arbitrario de paréntesis, signos de suma, resta, multiplicación y división y un número arbitrario de operandos enteros. La expresión no contiene espacios.

Salida:

El valor numérico al resolver la expresión.

Límites:

- La expresión de entrada no tendrá más de 200 caracteres.
- El resultado final y los resultados intermedios evaluados en la fórmula siempre serán menores a 2^{30}.

Subtarea 1 [60 puntos]

La expresión dada no contiene ningún paréntesis y consta solamente de signos de suma y multiplicación, con un número arbitrario de operandos.

Subtarea 2 [40 puntos]

- La expresión dada no tiene restricciones: puede contener un número arbitrario de paréntesis, signos de suma, resta, multiplicación y división y un número arbitrario de operandos.
- La expresión dada cumple que siempre existen paréntesis alrededor de operandos compuestos (que contengan operaciones dentro de ellos). Por ejemplo: $(6/3) - 2$ es una expresión válida para esta subtarea, ya que $(6/3)$ es un operando compuesto. Sin embargo, la expresión $1 + 2 + 6/2$ no es válida para esta subtarea, ya que el operando $6/2$ es un operando compuesto pero no contiene paréntesis.

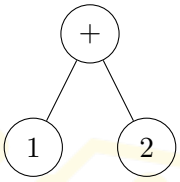
Notas

- Los signos de suma, resta, multiplicación y división están representados por '+','−','∗' y '/', respectivamente.
- La expresión inicial siempre es válida.
- El resultado final y resultados intermedios en el cálculo de la expresión siempre serán enteros.
- Si decides utilizar Python, no tienes permitido usar funciones como "eval", pues es precisamente lo que el problema pide implementar.
- Nótese que los casos de la Subtarea 1 no son válidos según la definición de la Subtarea 2. Por ejemplo: $1*3+2*4+8*2$ no es válido pues $1*3$, $2*4$ y $8*2$ son compuestos y deberían llevar paréntesis para ser válidos según la Subtarea 2. Para que tu solución obtenga 100 puntos, debes considerar tanto los casos de la Subtarea 1 como los de la Subtarea 2 en tu programa.

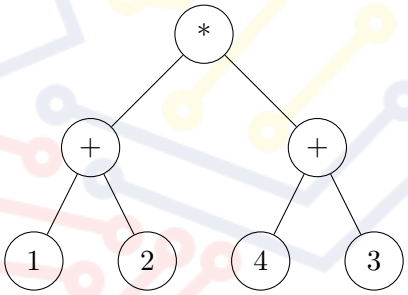
[Subtarea 1] *Solución:*
La clave es darse cuenta que se puede evaluar este tipo de expresiones iterando una sola vez sobre los caracteres de la expresión. Para ello simplemente es necesario ir acumulando el producto actual, mientras se vea un signo de producto. Una vez que se termina el producto actual, al ver un signo de suma, acumulamos el producto

actual en el acumulado total de la suma. Notemos que en caso de no existir un producto, el número se acumula en el total de la sumatoria. En cierta forma, un número es equivalente a un producto de solamente un número. Es necesario saber convertir un caracter a un dígito, e ir acumulando caracteres de dígitos en el número total.

[Subtarea 2] *Solución:*
Podemos visualizar una operación en dos operandos como un nodo padre que representa a la operación y dos nodos hijos que representan a los operandos a la izquierda y a la derecha. Por ejemplo, la operación $1+2$ puede ser representada como:



¿Qué sucedería si en lugar de un número, los operandos son expresiones compuestas? Por ejemplo, si tenemos la expresión $(1+2)\times(4+3)$, podríamos representar la expresión de la misma forma, pero las subexpresiones $(1+2)$ y $(4+3)$ serían árboles, tal como el ejemplo anterior. El grafo para la expresión nos quedaría de la siguiente forma:



¿Ves un patrón? Para expresiones donde los operandos son dos operaciones y no números, podemos poner al operador entre ambas como padre, crear el árbol recursivamente para las operaciones, y "pegar" el árbol de cada expresión como hijo del operador raíz. Este proceso lo podemos repetir recursivamente para cada expresión que encontremos. El resultado es un árbol binario donde todas las hojas son números y todas los nodos interiores son operaciones. Ahora, ¿cómo evaluaríamos la expresión una vez construido el árbol, asegurándonos que estemos respetando los paréntesis? Extrapolando observaciones con los ejemplos anteriores, podemos notar que para evaluar el árbol, podemos evaluar ambos subárboles recursivamente y aplicar la operación padre a ambos operandos. Dicho recorrido se conoce como recorrido "en-orden". Por lo tanto, el problema se puede dividir en dos partes: construir el árbol dada la expresión y evaluar el árbol una vez construido.

Cabe recalcar que representaremos un árbol binario usando memoria dinámica, donde un nodo está representado por una estructura que contiene el tipo de datos, en este caso un string, y dos punteros, uno para el hijo izquierdo y otro para el hijo derecho. Un árbol binario, estaría representado por solamente un nodo raíz y se puede recorrer el árbol usando los punteros de los hijos. La clave para construir el árbol es usar una pila (stack) como estructura de datos auxiliar. La idea inicial es ir apilando operandos en una pila. Una vez que sabemos que la operación termina (con un paréntesis de cierre por ejemplo), desapilamos los dos operandos de arriba de la pila, armamos un árbol con raíz en el operador y operandos como hijos (como ya describimos arriba) y lo apilamos nuevamente. El problema es que pueden existir varios operadores pendientes a realizar. Por lo tanto, podemos usar otra pila para los operadores. Es decir, tendríamos una pila para los operadores y otra pila que guarde nodos (expresiones intermedias). Iteramos sobre la expresión:

- Si vemos un paréntesis de apertura lo apilamos en la pila de operadores.
- Si vemos un número, lo apilamos en la pila de nodos.
- Si vemos un operador, lo apilamos en la pila de operadores.
- Si vemos un paréntesis de cierre, desapilamos los dos operandos de la pila de nodos, armamos el árbol, como describimos, y apilamos el nodo resultante en la pila de nodos.

Al final de la iteración sobre la expresión, nos queda un nodo en la pila de los nodos, que representa la raíz del árbol binario.

Para evaluar el valor final del árbol binario, podemos crear un método recursivo para cada nodo, donde se evalué recursivamente la expresión a la izquierda, a la derecha y aplicar la operación a los valores calculados. El caso base es una hoja que solo contenga un número (los punteros apuntan al puntero nulo, o nullptr). En este caso se regresa el valor del número. En este problema no tomábamos en cuenta la complejidad de tiempo y memoria de las soluciones, pero cabe recalcar que esta solución usa tiempo lineal para construir el árbol de expresiones ya que se itera solamente una vez sobre la expresión. Es fácil ver que el recorrido en orden del árbol binario también usa tiempo lineal, pues cada nodo es procesado solamente una vez. En cuestión de memoria, utilizamos dos pilas y un árbol binario. Cada una de estas estructuras puede llegar a contener un elemento por caracter de la expresión inicial, y por lo tanto usa memoria lineal. En resumen, si $|s|$ es la cantidad de caracteres en la expresión inicial s, el algoritmo usa tiempo y memoria lineal, en el orden de $O(|s|)$.

[Subtarea 2] *Solución:*

Existe otra forma equivalente de resolver el problema, que consiste en resolver las expresiones de adentro hacia afuera iterativamente, respetando la prioridad de las operaciones. Usaremos dos vectores (arreglos adaptables), uno para almacenar los números y otro para los operadores. No modificaremos el string de entrada, los cambios los realizaremos en los vectores creados, navegando el string de entrada varias veces.

Para esto, vamos a dividir el problema en 3 partes:

1. Analizar el string ingresado para separar los números de los operadores $(+, -, *, /)$.
2. Analizar los paréntesis en la expresión para calcular el resultado dentro de los mismos.
3. Encontrar el resultado final de la expresión.

Para realizar las operaciones en este problema, vamos a ir atravesando el string buscando divisiones, multiplicaciones, restas y sumas.

1. Analizar el string: Para esto usamos un simple while loop que atraviesa todo el string de entrada. Si un caracter es operador, se lo añade al vector *operadores*. Si se encuentra con un dígito, se navega hasta el óltimo dígito del número (el número justo antes del siguiente operador) para formar el número que se debe ingresar en el vector *numeros*.
2. Analizar paréntesis: Para esto vamos a navegar a través del string de entrada buscando paréntesis de clausura. Por cada paréntesis de clausura, buscamos el respectivo de apertura y hallamos el resultado de la operación dentro del mismo. Una vez resuelto un paréntesis, continuamos buscando los demás paréntesis de clausura para realizar el mismo procedimiento. Hacemos este proceso hasta que hayamos atravesado todo el string resolviendo las operaciones para cada paréntesis de clausura.
3. Resultado: En este punto ya resolvimos todos los paréntesis, así que solo queda hacer las operaciones fuera de paréntesis. Para esto suponemos que toda la expresión está dentro de un paréntesis gigante y hacemos el mismo procedimiento que hicimos al analizar los paréntesis. Atravesamos el string buscando divisiones,

multiplicaciones, restas y sumas, en ese orden, para respetar la jerarquía de operaciones. Realizamos este proceso hasta que quede un único número que será nuestro resultado final.

La complejidad de tiempo de esta solución es cuadrática, ya que se itera al buscar los paréntesis y se itera dentro para hacer las operaciones. En el peor de los casos, este proceso podría llevar a realizar un total de operaciones en orden cuadrático. La complejidad de memoria usada es lineal, ya que usamos dos arreglos con tamaño máximo la cantidad de caracteres en la expresión de entrada. En resumen, si $|s|$ es la cantidad de caracteres en la expresión inicial s, el algoritmo usa tiempo cuadrático $O(|s|^2)$ y memoria lineal, en el orden de $O(|s|)$.