**Cálculo de Combinaciones Posibles para un valor**

Profesor:

Aschettino, Giovanna

Cuadrado Estrebou, Maria Fernanda

Grupo:

Rios, Mateo – LU: 1120860

Cassi, Daniel Ignacio – LU: 1115907

Von Elm, Lucas– LU: 1133639

**Tabla de Contenidos**

Introducción 3

Descripción del Problema 3

Estrategia de Resolución 3

Pseudocódigo del Algoritmo de Resolución del Problema 3

Análisis de Complejidad Temporal 3

Conclusiones 3

**Introducción**

El objetivo del proyecto es realizar un algoritmo, que dado un listado de operadores y numeros a usar, obtener todas las combinaciones posibles que den un numero “X” dado. Los operadores deben ser todos utilizados y del listado de números se deben solamente utilizar “n” números a la vez.

**Descripción del Problema**

**Estrategia de Resolución**

Para resolver el problema de las combinaciones se utiliza un algoritmo de Backtracking, con el cual iteramos por todas las combinaciones de orden posibles de los operadores, y todas las combinaciones de numeros para cada uno de ellas. Al obtener una combinacion posible, se realiza el cálculo, y si el resultado de este da el valor deseado, se registra esa combinación.

Inicialmente utilizamos una funcion auxiliar que recibira los parámetros iniciales del problema, la lista de numeros, la lista de operadores, la cantidad de numeros a usar como máximo y el número buscado y que inicializara las estructuras necesarias para el llamado de la funcion buscarCombinaciones.

La función buscarCombinaciones generará como primer etapa todas las combinaciones posibles de operadores, y para cada una de ellas todas las combinaciones de numeros. Si el resultado de la expresión matemática obtenida es igual al número buscado, lo agrega a una lista de combinaciones.

En la primera etapa que genera las combinaciones de operadores, si aun no todos fueron utilizados, por cada operador si puede usarse, se marca como usado y se baja un nivel (llamado recursivo) de operadores utilizando el nuevo operador. Al volver de este llamado, se desmarca como usado el operador.

Cuando todos los operadores fueron utilizados, comienza la siguiente etapa en la cual se generan todas las combinaciones de numeros a utilizar para la combinación. Si aún no fueron utilizados todos los numeros necesarios, por cada número, si puede usarse se marca como usado y se baja un nivel de numeros (llamado recursivo) utilizando el nuevo número. Al volver de este llamado, se desmarca como usado el operador.

Cuando todos los numeros necesarios fueron utilizados, si la combinación de numeros y operadores conforman una expresión matemática que resulte igual a el número buscado, se agrega a la lista de combinaciones que forman parte de la solucion.

**A continuación se adjunta en ejemplo en formato de árbol:**



**Estrategia de poda:**

La poda se empieza a calcular en cada llamado después de que se hayan elegido dos números. Se realiza el cálculo del valor actual y dependiendo del resultado:

* Si el valor dado es menor al valor buscado: Se busca si hay operadores que aumenten el valor posible (Multi o Suma). Si no se los encuentra se corta la rama, ya que resultaría imposible alcanzar el valor buscado.
* Si el valor dado es mayor al valor buscado: Se busca si hay operadores que decrementen el valor posible (Div o Resta). Si no se los encuentra se corta la rama, ya que resultaría imposible alcanzar el valor buscado.

**Pseudocódigo del Algoritmo de Resolución del Problema**

**Algoritmo obtenerOperaciones**

**Entrada:** (List<Entero> numeros, List<Operadores> operadores, Entero valorBuscado, Entero numerosUsar)

**Salida: (**List<String> combinacionesObtenidasString)

combinacionesObtenidas ← Inicializar()

combinacionesObtenidasString Inicializar

Si numerosUsar <= numeros.tamaño

Si operadores.tamaño + 1 <= numeros.tamaño

numerosDouble ← Inicializar()

numerosUsados ← Inicializar()

Para i = 0 hasta longitud(numeros)

agregar(numerosDouble, numeros[i])

Para j = 0 hasta longitud(numeros)

agregar(numerosUsados, 0)

ordenNumeros ← Inicializar()

operadoresUsados ← Inicializar()

Para k = 0 hasta longitud(listaOperadores)

agregar(operadoresUsados, 0)

ordenOperadores ← Inicializar()

etapaOperadores ← 0

etapaNumeros ← 0

combinacionesObtenidas ← buscarCombinaciones(combinacionesObtenidas, numerosUsar, valorBuscado, operadores, operadoresUsados, ordenOperadores, numerosDouble, numerosUsados, ordenNumeros, etapaOperadores, etapaNumeros)

combinacionesObtenidasString pasarCombinacionAString(combinacionesObtenidas)

Fin si

Fin si

Devolver combinacionesObtenidas

**Algoritmo pasarCombinacioAString**

**Entrada**: (List<Combinacion> combinaciones)

**Salida:** (List<String> combinacionesString)

List<String> combinacionesString Incializar()

combinacionesString obtenerCadenasDeCombinaciones(combinaciones)

Devolver combinacionesString

**Algoritmo obtenerCadenasDeCombinaciones**

**Entrada**: (List<Combinacion> combinaciones)

**Salida**: (List<String> cadenaDeCombinaciones)

List<String> cadenaDeCombinaciones Inicializar()

Por cada combinacion en combinaciones

Entero i 0

Entero numeroActual 0

Entero operadorActual 0

String cadenaDeOperacion “”

List<Double> números ObtenerNumeros(combinacion)

List<Operadores> operadores ObtenerOperadores(combinacion)

Mientras i < números.tamaño + operadores.tamaño

Si i % 2 = 0

cadenaDeOperacion Concatenar(números[numeroActual])

numeroActual numeroActual + 1

Sino

cadenaDeOperacion Concatenar(obtenerCaracter(operadores[operadorActual]))

operadorActual 1

cadenaDeCombinaciones Agregar(cadenaDeOperacion)

Devolver cadenaDeCombinaciones

**Algoritmo obtenerCaracter**

**Entrada**: (String operador)

**Salida**: String

Si operador = SUMA

Devolver “+”

Si operador = RESTA

Devolver “-“

Si operador = MULTI

Devolver “\*”

Sino

Devolver “/”

**Algoritmo buscarCombinaciones**

**Entrada:** (Entero n, Double x, List<Operadores> listaOperadores, List<Entero> listaOperadoresUsados, List<Operadores> ordenOperadores, List<Double> listaNumeros, List<Entero> listaNumerosUsados, List<Double> ordenNumeros, etapaOperadores, etapaNumeros)

**Salida:** (List<Combinacion> listadoCombinaciones)

Si etapaOperadores = listaOperadores.size

Si etapaNumeros = n

Double valorEncontrado = calcularCombinacion(x, ordenOperadores, ordenNumeros)

Si valorEncontrado = valorBuscado

Combinacion comb Crear (ordenOperadores, ordenNumeros)

listadoCombinaciones Agregar (comb)

Sino

Si no (podaSimple (ordenOperadores, ordenNumeros, etapaNumeros)

Por cada j = 0 hasta listaNumeros.tamaño

Si puedeUsarse(j, listaNumerosUsados, listaNumeros, ordenOperadores)

marcarUsadoNumero(j, listaNumerosUsados, ordenNumeros)

buscarCombinaciones (n, x, listadoCombinaciones, listaOperadores, listaOperadoresUsados, ordenOperadores, listaNumeros, listaNumerosUsados, ordenNumeros, etapaOperadores, etapaNumeros + 1)

desmarcarUsadoNumero(j, listaNumerosUsados, ordenNumeros)

Fin Si

Fin por

Sino

Devolver combinaciones

Fin Sino

Sino

Por cada i = 0 hasta listaOperadores.tamaño

Si puedeUsarse(i, listaOperadoresUsados)

marcarUsadoOperador(i, listaOperadoresUsados)

buscarCombinaciones(n, x, listadoCombinaciones, listaOperadores, listaOperadoresUsados, ordenOperadores, listaNumeros, listaNumerosUsados, ordenNumeros, etapaOperadores + 1, etapaNumeros)

desmarcarUsadoOperador(i, listaOperadoresUsados)

Fin Si

Fin Por

Fin Sino

Devolver listadoCombinaciones

**Algoritmo puedeUsarseNumero**

**Entrada**: (Entero quieroUsar, List<Entero> listaUsados, List<Double> numerosUsar,List<Operadores> ordenOperadores)

**Salida**: Boolean

Si usados[quieroUsar] = 0

Si etapaNumeros = 0

Devolver verdadero

Sino si númerosUsar[quieroUsar] = 0

Devolver ordenOperadores[quieroUsar-1] =/= Div

Sino

Devolver verdadero

Devolver falso

**Algoritmo puedeUsarse**

**Entrada**: (Entero quieroUsar List<Entero> listaUsados)

**Salida**: Boolean

Devolver listaUsados[quieroUsar]= 0

**Algoritmo marcarUsadoNumero**

**Entrada**: (Entero x, List<Double> listaNumeros, List<Entero> listaNumerosUsados, List<Double> ordenNumeros)

numeroUsado listaNumeros[x]

listaNumerosUsados Reemplazar (x, 1)

ordenNumeros Agregar(numeroUsado)

**Algoritmo desmarcarUsadoNumero**

**Entrada**: (Entero x, List<Double> listaNumeros, List<Entero> listaNumerosUsados, List<Double> ordenNumeros)

indiceNumero ordenNumeros.tamaño - 1

listaNumerosUsados Reemplazar (x, 0)

ordenNumeros Quitar(indiceNumero)

**Algoritmo marcarUsadoOperador**

**Entrada**: (Entero x, List<Operadores> listaOperadores, List<Entero> listaOperadoresUsados, List<Operadores> ordenOperadores)

operadorUsado listaOperadores[x]

listaOperadoresUsados Reemplazar (x, 1)

ordenOperadores Agregar(operadorUsado)

**Algoritmo desmarcarUsadoOperador**

**Entrada**: (Entero x, List<Operadores> listaOperadores, List<Entero> listaOperadoresUsados, List<Operadores> ordenOperadores)

indiceOperador ordenOperadores.tamaño - 1

listaOperadoresUsados Reemplazar (x, 0)

ordenOperadores Quitar(indiceOperador)

**Algoritmo calcularCombinacion**

**Entrada**: (Entero x, List<Operadores> operadores, List<Double> numeros)

**Salida**: Double

List<> operadoresAux Copiar(operadores)

List<> numerosAux Copiar(numeros)

Double calculoAux 0

Entero j 0

Por cada i = 0 hasta 1

Mientras j < operadoresAux.tamaño

Si i = 0 //Hago primero las multiplicaciones y divisiones

Si operadorAux[j] = “\*” O operadorAux[j] = “/”

calculoAux calculoSimple (operadoresAux[j], numerosAux[j], numerosAux[j+1])

operadoresAux Quitar(j)

numerosAux Quitar(j+1)

numerosAux Reemplazar(j, calculoAux)

Sino

j j+1 //Busca el siguiente “\*” O “/”

Sino

calculoAux calculoSimple (operadoresAux[j], numerosAux[j], numerosAux[j+1])

operadoresAux Quitar(j)

numerosAux Quitar(j+1)

numerosAux Reemplazar (j, calculoAux)

Fin Sino

Fin por

Fin por

Devolver numerosAux[0]

**Algoritmo calculoSimple**

**Entrada**: (Operador operador, Double numero1, Double numero2)

**Salida**: Double

Si operador = MULTI

Sino Si operador = DIV Devolver numero1 \* numero2

Devolver numero1 / numero2

Sino si operador = SUMA

Devolver numero1 + numero2

Sino

Devolver numero1 – numero2

**Algoritmo podaSimple**

**Entrada**: (Entero valorBuscado, List<Operadores> ordenOperadores, List<Double> ordenNumeros, Entero etapaNumeros)

**Salida**: Boolean

Si (etapaNumeros > 1)

List<Operadores> operadoresAux Copiar (ordenOperadores, 0, etapaNumeros-1)

List<Double> numerosAux Copiar (ordenNumeros, 0, etapaNumeros)

Double valorActual CalcularCombinacion (operadoresAux, numerosAux)

Si valorActual > valorBuscado

Devolver minimizarPodaSimple (operadores)

Sino si valorActual < valorBuscado

Devolver maximizarPodaSimple (operadores)

Devolver falso

**Algoritmo maximizarPodaSimple**

**Entrada**: (List<Operadores> operadores, Entero etapaNumeros)

**Salida**: Boolean

Por cada i = etapaNumeros-1 hasta operadores.tamaño

Si operador[i] == Multi O operador[i] == Suma

Devolver falso

Devolver verdadero

**Algoritmo minimizarPodaSimple**

**Entrada**: (List<Operadores> operadores, Entero etapaNumeros)

**Salida**: Boolean

Por cada i = etapaNumeros-1 hasta operadores.tamaño

Si operador[i] == Div O operador[i] == Resta

Devolver falso

Devolver verdadero

**Análisis de Complejidad Temporal**

**Análisis de Complejidad Teórica:**

**Algoritmo pasarCombinacionAString**

Costo: O(N\*I) Dado por la función obtenerCadenasDeCombinaciones

**Algoritmo obtenerCadenasDeCombinaciones**

Costo: O(N\*I) Siendo N la cantidad de Combinaciones e I la longitud de la combinacion

**Algoritmo obtenerOperaciones**

O(N) Dado por los ciclos que iteran la cantidad de numeros y operadores respectivamente. + O(O\*N) Dado por la funcion buscarCombinaciones()

Costo: O(O\*N)

**Algoritmo buscarCombinaciones**

Costo: O(O\*N) ^ N

A = O \* N B = 1 K = 1

**Algoritmo puedeUsarse**

Costo: 0(1) - Constante

**Algoritmo marcarUsadoNumero**

Costo: 0(n) en promedio – Dado por la función Agregar.

Costo Constante en caso de que la función agregar funcione en complejidad O(1)

**Algoritmo desmarcarUsadoNumero**

Costo: 0(n)– Dado por la función Quitar.

Costo Constante en caso de que la función Quitar funcione en complejidad O(1)

**Algoritmo marcarUsadoOperador**

Costo: 0(n) en promedio – Dado por la función Agregar.

Costo Constante en caso de que la función agregar funcione en complejidad O(1)

**Algoritmo desmarcarUsadoOperador**

Costo: 0(n)– Dado por la función Quitar.

Costo Constante en caso de que la función Quitar funcione en complejidad O(1)

**Algoritmo calcularCombinacion**

Costo: O(2J) Siendo J la cantidad de operadores.

**Algoritmo calculoSimple**

Costo: Constante

**Algoritmo podaSimple**

Costo: O(2J) dado por la función calcularCombinacion

**Algoritmo maximizarPodaSimple**

Costo: O(N) siendo N la cantidad de operadores

**Algoritmo minimizarPodaSimple**

Costo: O(N) siendo N la cantidad de operadores

**Análisis de Complejidad Práctica:**

**Datos de prueba**

**Numeros**:

4 | 3 | 2 | 1 | 10 | 8 | 0 | 3 | 2

**Operadores**:

RESTA – RESTA – MULTI – SUMA – RESTA

**Numeros a usar:** 6

**Numero a buscar**: 25

**Sin poda**:  
**Alocaciones de Memoria**: 3.095.608.224

**CPU Samples**: 225

**Tiempo de ejecución**: 4 segundos

**Con poda Simple**:  
**Alocaciones de memoria**: 2.024.833.712

**CPU Samples**: 151

**Tiempo de ejecución**: 3 segundos

**Conclusiones**

Con estos algoritmos se busca obtener todas las combinaciones posibles para un set de números y operadores a usar, un numero a buscar y una cantidad de números limitada a usar (Se deben usar todos los operadores).

Se decidió utilizar la técnica de backtracking para la creación del algoritmo debido a que a fuerza bruta logra obtener los resultados deseados, sin necesidad de una estrategia especializada para el caso específico.

La resolución fue realizada con un algoritmo que se encarga de realizar todas las combinaciones posibles de orden de los operadores dados, y para cada una de esas combinaciones realizar todas las combinaciones de orden de los números posibles a usar. La combinación es considera válida si el resultado es igual al valor buscado. Las combinaciones válidas son mostradas posteriormente en pantalla.

El algoritmo cuenta con un algoritmo de poda simple, donde se evalúa si es conveniente o no continuar la rama. En caso negativo, se poda y prueba una distinta combinación.

Los resultados demuestran que el algoritmo funciona correctamente, y que la poda llega a reducir a lo sumo en un 25% el tiempo de ejecución del algoritmo, ahorrando en recursos físicos (memoria y ciclos de CPU utilizados) y tiempo.