Trabajo Práctico Obligatorio

Programación III Virtual 2 do Cuatrimestre 2024 Facultad de Ingeniería y Ciencias Exactas  
 **“Back-tracking”**

Profesor/es:

Cuadrado Estrebou, Maria Fernanda

Foglino, Alejandro Luis

Grupo:

Sanchez, Luca – LU: 1115225

Camporro, Ignacio Manuel– LU: 1116829

Gauna, Alejo – LU: 1154607

Buenos Aires, 15 de Noviembre de 2024.-

**Tabla de Contenidos**

[Introducción 3](#_heading=h.30j0zll)

[Descripción del Problema 3](#_heading=h.1fob9te)

[Estrategia de Resolución 3](#_heading=h.3znysh7)

[Pseudocódigo del Algoritmo de Resolución del Problema](#_heading=h.2et92p0) 7

[Análisis de Complejidad Temporal 1](#_heading=h.tyjcwt)2

[Conclusiones 14](#_heading=h.3dy6vkm)

# Introducción

Una blockchain es una estructura de datos distribuida que garantiza la integridad de las transacciones mediante reglas criptográficas y consensos entre los nodos.

El objetivo es diseñar una cadena de bloques compleja, donde no solo deben respetarse las reglas básicas de tamaño y validación de hash, sino también deben gestionarse transacciones que dependen entre sí, firmas múltiples, restricciones de prueba de trabajo, y prioridades en las transacciones.

El objetivo es implementar un algoritmo que construya una blockchain válida, considerando todas las combinaciones posibles de bloques. Cada bloque debe cumplir con un conjunto más amplio de reglas, que incluyen validaciones adicionales como la dependencia entre transacciones, transacciones con firmas múltiples, priorización y restricciones de prueba de trabajo (hash).

Este será el informe en el que se detalla la versión final de la solución junto con el análisis de la complejidad práctica y conclusiones finales basadas en la implementación. Además, se incluyen correcciones que se fueron haciendo de forma grupal con respecto a la optimización del código, su lógica y comportamiento.

# Descripción del ProblemaC:\Users\NACHO-PC\Downloads\image (1).png

## Estrategia de Resolución

El algoritmo se resolverá utilizando técnicas de backtracking.

**Por nivel**: Una transacción

**Opciones a evaluar por nivel:** Se evalúa si la transacción puede ser agregada al bloque.

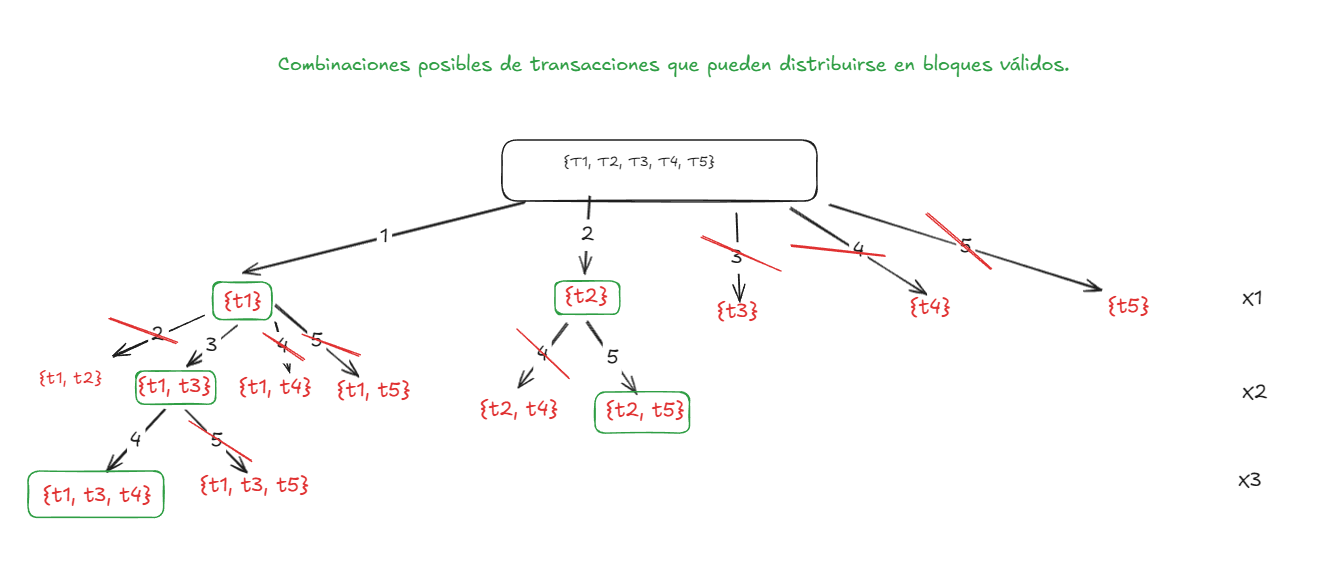
**Poda:**

1. Valor máximo por bloque: La suma de los valores de las transacciones en un bloque no debe superar 100 satoshis.

2. Máximo de transacciones por bloque: Un bloque puede contener como máximo 3 transacciones.

3. Prueba de trabajo: La suma de los valores de las transacciones en un bloque debe ser divisible por 10.

4. Tamaño máximo de bloque: El tamaño de cada bloque sigue siendo de 1 MB.



**Implementación final**

**Explicación de la Implementación:**

Para esto ideamos un algoritmo llamado 'AlgoritmoDeBlockchainIImpl' que sigue la lógica planteada en el pseudocódigo. Luego incorporamos ciertos métodos que nos permiten satisfacer las exigencias que fueron planteadas en la estrategia y en el pseudocódigo.

Método "construirBlockchain": ideamos este método para el flujo principal de la cadena de bloques. En el, se inicializa 'cadenaDeBloques' como una lista de bloques y luego se recorre la lista de 'transacciones', para generar bloques hasta que se cumplan las condiciones de:

* Máximo tamaño.
* valor
* cantidad de bloques.

En cada iteración, hicimos que se llame a 'construirBloque' para intentar formar un nuevo bloque.

Metodo 'construirBloque': este método lo construimos para que se pueda implementar el backtracking para evaluar si las transacciones cumplen con las restricciones específicas que plantea el problema:

* tamaño
* valor
* Prueba de trabajo.

Lo que ideamos en este método fue la acción de añadir cada transacción potencialmente válida al bloque. Luego se verifica si se excede los límites. Si separa alguna restricción, la transacción se va a retirar y se va a continuar buscando otras combinaciones.

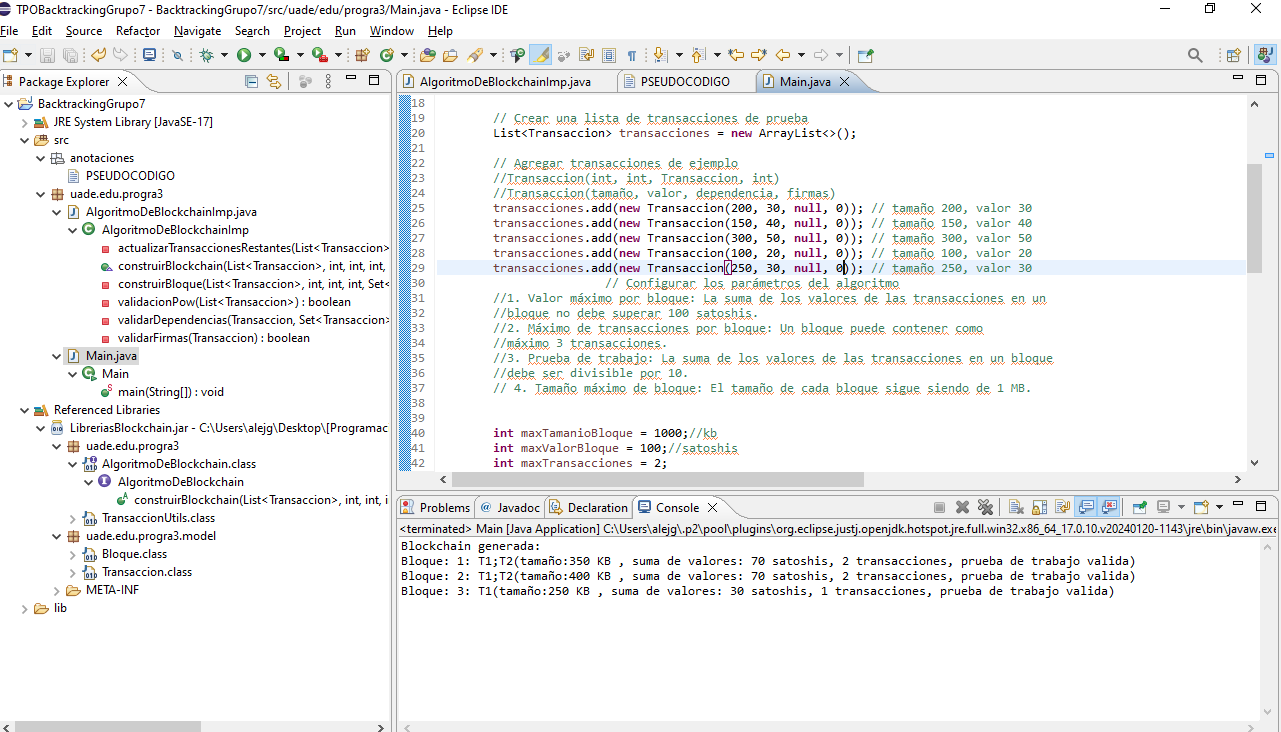
En nuestra implementación de código, también incorporamos las validaciones suficientes para resolver el problema:

* 'validacionPow': esta validación la ideamos para la prueba de trabajo, en él se verifica que la suma total de transacciones en un bloque pueda ser divisible por 10.
* 'validarDependencias': con el afán de verificar que las dependencias de las transacciones estén satisfechas.
* 'validarFirmas': aquí se intenta asegurar que cada transacción tiene el número requerido de firmas.

Es importante destacar, que no tuvimos la necesidad de hacer un método de validación para el valor máximo por bloque y tamaño máximo de bloque, con lo dejamos planteado en el pseudocódigo. Lo que hicimos fue hacer la propia validación dentro del método 'construirBloque'

Diferencias entre nuestra implementación con respecto al pseudocódigo planteado: Realizamos algunas modificaciones para que la implementación final sea más confortable. Para esto añadimos la validación de firmas.

**Prueba de resultado**

****

## Pseudocódigo del Algoritmo de Resolución del Problema

**ALGORITMO Blockchain**

**Entrada**: CadenaDeBloques: Vector<Objeto>,   
 Block: Objeto,   
 Transacciones: Vector<Objeto>   
 //tamaño   
 //valor  
 //dependencia  
 //firma   
 **Salida**: CadenaDeBloques  
  
 <- []  
 TransaccionesYaAgregadas <- {}   
  
 Entero i ← 0  
 **mientras** longitud(Transacciones) > 0 **hacer**   
 CadenaDeBloques[i] ← ConstruirBloque(Transacciones, TransaccionesYaAgregadas)  
 ActualizarTransaccionesRestantes(Transacciones, CadenaDeBloques[i])  
 i <- i + 1  
 **fin mientras**  
  
 **retornar** CadenaDeBloques  
**fin** Blockchain

**ALGORITMO ConstruirBloque**  
 Entrada:   
 Transacciones: Vector<Objeto>  
 ActBloque: Vector <Objeto>,  
 ActTamaño: entero,  
 nroT: entero,  
 TransaccionesYaAgregadas: Conjunto

Salida: Bloque  
  
 Entero i <- 0  
 cantT <- longitud(Transacciones) – 1

**mientras** i ≤ cantT **hacer**  
 ActBloque[i] += Transacciones[nroT]   
 si nroT ≤ 3 entonces // Máximo 3 transacciones por bloque  
 **si** ValidacionSatoshis(ActBloque) y ValidacionPow(ActBloque) Y TamañoBloques(ActTamaño) **entonces**   
 **retornar** copia de ActBloque  
 **fin** si  
 **retornar** ActBloque  
 **fin si**  
   
//SI TODO ESTO DA VERDADERO, ENTONCES MANDO A CONSTRUIR OTRO BLOQUE Y GUARDÓ EL QUE YA TENGO  
 **si** TamañoBloque(ActBloque + Tansacciones[i].tamaño) Y ValidacionSatoshis(ActBloque + Transacciones[i].valor) Y ValidacionPow(ActBloque + Transacciones[i].tamaño) y ValidDependencia(TransaccionesYaAgregadas, Transacciones[i]) **entonces**  
 TransaccionesYaAgregadas <- TransaccionesYaAgregadas + Transacciones[i]

**ConstruirBloque**(Transacciones, ActBloque, ActTamaño += Transacciones[i].tamaño , nroT + 1, TransaccionesYaAgregadas)  
 **fin si**   
//EN CASO DE QUE DE TODO FALSO O ALGUNO DE FALSO, ME VOY A LA SIGUIENTE TRANSACCION  
 ActBloque[i] <- null   
 i <- i + 1   
 nroT <- nroT + 1  
 **fin mientras**  
 **retornar** ActBloque  
**fin** ConstruirBloque

**ALGORITMO TamañoBloque**

Entrada: ActTamaño: Entero,

Salida: Booleano

sumaTotal <- 0  
 **para cada transaccion en bloque**  
 sumaTotal <- sumaTotal + transaccion.tamaño  
 **retornar** sumaTotal ≤ 1000  
 fin TamañoBloque

**ALGORITMO ValidacionSatoshis**

Entrada: Bloque: Objeto

Salida: Booleano

sumaTotal <- 0   
 **para cada transaccion en bloque**  
 sumaTotal<- sumaTotal + transaccion.valor   
 **retornar** sumaTotal ≤ 100//deberia retornar V o F  
 **fin** ValidacionSatoshis

**ALGORITMO ValidacionPow**

Entrada: Bloque: Objeto

Salida: Booleano

sumaTotal <- 0   
 **para cada transaccion en bloque**  
 sumaTotal ← sumaTotal + transaccion.valor   
 **retornar** sumaTotal % 10 = 0//deberia retornar V o F   
 **fin** ValidacionPow

**ALGORITMO ValidDependencia**

Entrada: TransaccionesYaAgregadas: Vector <Objeto>, transaccion: Objeto

Salida: Booleano

dependencia<-transaccion.dependencia

**PARA CADA transac** EN TransaccionesYaAgregadas  
 si transac == dependencia

**retornar true**

**FIN SI**  
 **FIN PARA** **Retornar false**

**ALGORITMO ActualizarTransaccionesRestantes**

Entrada: Transacciones: Vector <Objeto>, Block: Objeto

Salida: Transacciones: Vector <Objeto>,

**PARA CADA transaccionAc EN Block**

Transacciones.sacar(transaccionAc)  
 **FIN PARA**

**Devolver** Transacciones  
 **FIN** ActualizarTransaccionesRestantes

**Modificamos el pseudocódigo para que cumpla con las exigencias del problema y pueda fusionarse con la implementación final, agregando la validación de firmas.**

ALGORITMO ValidarFirmas

Entrada: transaccion: Objeto

Salida: Booleano

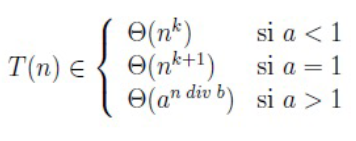
retornar transaccion.firmasActuales >= transaccion.firmasRequeridas

FIN ValidarFirmas

## Análisis de Complejidad Temporal

| **Algoritmo** | **Complejidad Temporal** |  |
| --- | --- | --- |
| Blockhain | O(n3) | O(n . n2 ) = O(n3) |
| ConstruirBloque | O(3n) |  |
| ActualizarTransaccionesRestantes | O(n2) | Se recorre cada transacción máximo n veces y en cada iteración elimina la transacción correspondiente. La operación de eliminar tiene complejidad de n en el peor caso por lo que la complejidad en total sería n2 |
| ValidDependencia | O(n) | Es una iteración, en el peor caso debe recorrer todo el conjunto |
| ValidacionPow | O(n) | Es una iteración, en el peor caso debe recorrer todo el conjunto |
| ValidacionSatoshis | O(n) | Es una iteración, en el peor caso debe recorrer todo el conjunto |
| TamañoBloque | O(n) | Es una iteración, en el peor caso debe recorrer todo el conjunto |

**Análisis de ConstruirBloque (Algoritmo Recursivo)**

a=3, b=1, k=1

Como a=3 se la opción O (a n div b), y al ser k=1, la complejidad es:

**O(3n)**

*‘A’ va a estar determinado por la cantidad de transacciones que puedo ingresar en un bloque, en este caso, el máximo es 3, por lo tanto a = 3.*

***C*omplejidad Temporal Práctica**

Explicaremos el comportamiento de la complejidad temporal practica en función del tiempo de ejecución con datos varios ingresados:

El equipo probó ingresando cien, quinientos y mil datos para comprobar como es el rendimiento de la complejidad:

* + Con el ingreso de cinco datos, vimos que el tiempo de ejecución en milisegundos era de 30, 32, 30, 30 llo que nos da un promedio estimado de *30,5*.
  + Con el ingreso de quince datos, vimos que el tiempo de ejecución en milisegundos era de 24,26,40,36; lo que nos da un promedio estimado de 36.5.
  + Con el ingreso de veinte datos, vimos que el tiempo de ejecución en milisegundos era de 37,39,40,411, lo que nos da un promedio estimado de 39.25.

Con esta información que obtuvimos podemos dejar varios puntos en claro:

* con 100 Transacciones: El tiempo de ejecución es bastante bajo, con un promedio de *30,5* ms. Y con esto podemos dejar por sentado que el algoritmo es eficiente para números pequeños de transacciones.
* con 500 Transacciones. El tiempo de ejecución promedio es de *36.5* ms, lo cual podemos asegurar que el algoritmo aún es bastante eficiente.
* con 1000 Transacciones: El tiempo de ejecución promedio es de *39.25* ms. Con esto ya tenemos un aumento en la complejidad temporal a medida que se aumenta el número de Transacciones.

| ***Transacciones/ejecuciones*** | ***primera ejecución*** | ***segunda ejecución*** | ***tercera ejecución*** | ***cuarta ejecución*** | ***promedio total*** |
| --- | --- | --- | --- | --- | --- |
| ***100 transacciones*** | ***30*** | ***32*** | ***30*** | ***30*** | ***30,5*** |
| ***500 transacciones*** | ***34*** | ***36*** | ***40*** | ***36*** | ***36.5*** |
| ***10000 transacciones*** | ***37*** | ***39*** | ***40*** | ***41*** | ***39.25*** |

# 

# Conclusiones

En conclusión, por lo visto en la prueba práctica, la recursividad en ConstruirBloque determinan la complejidad del algoritmo, teniendo una complejidad del orden O(mn), donde n es la cantidad de transacciones ingresadas y m la cantidad de transacciones que se pueden ingresar en un bloque, en este caso como se puede ingresar un máximo de 3 transacciones por bloque, la complejidad queda en O(3n).

**En conclusión final:**

El pseudocódigo fue modificado agregando la validación de firmas, y a la hora de trasladar la lógica a la implementación en formato código java, nos vimos en la necesidad de hacer cambios sutiles en cuanto a las validaciones de valor máximo y tamaño máximo del bloque.

La construcción del codigo java fue basado en la información del pseudocódigo, implementando la lógica.

En cuanto a la eficiencia, creemos que el código cumple con los requisitos del problema. La complejidad temporal fue calculada nuevamente para todas los algoritmos incluyendo el que se agregó.