**Ejercicio 4**

**Detección de Actividades Fraudulentas en el National Bank**

***Por: Ing. Alex Tumiri Huanca***

**Introducción**

El National Bank tiene una política sencilla para advertir a sus clientes sobre posibles actividades fraudulentas en sus cuentas. Si el monto gastado por un cliente en un día determinado es mayor o igual al doble del gasto medio del cliente durante un número de días anteriores, el banco envía una notificación sobre un posible fraude. Este documento describe la implementación de un sistema que calcula la cantidad de notificaciones enviadas a los clientes basándose en sus gastos diarios, aplicando buenas prácticas de programación y estructuras de datos eficientes.

**Descripción del Problema**

El objetivo es determinar cuántas veces un cliente recibirá una notificación durante un período de días, dado un array de gastos diarios y un número que representa los días anteriores para calcular la mediana. Para ello, debemos tener en cuenta que:

- La mediana de los gastos de los días anteriores se calcula de forma eficiente.

- Las notificaciones se envían solo si se dispone de al menos  días de datos.

**Algoritmo Propuesto**

La solución inicial consiste en recorrer el array de gastos, pero su complejidad puede ser mejorada. En lugar de calcular la mediana ordenando los gastos anteriores cada vez, se utiliza una combinación de dos montones (heaps), que permite mantener la mediana actualizada de forma eficiente.

**Complejidad Algorítmica**

- La inserción y eliminación en un heap tiene una complejidad de .

- La extracción de la mediana se realiza en tiempo constante .

- En total, para días, la complejidad se convierte en , lo que es más eficiente.

**Implementación**

A continuación, se presenta el código Java que implementa la solución:

|  |  |
| --- | --- |
| 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  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132 | **package** com.eldarinterview;  **import** **java.util.Collections**;  **import** **java.util.PriorityQueue**;  /\*\*  \* The com.eldarinterview.NationalBankFraudDetection class is responsible for detecting potential fraudulent activities  \* based on daily spending of customers. It provides a method to count the number of notifications  \* sent to customers when their spending exceeds a certain threshold relative to their historical  \* spending data.  \*/  **public** **class** **NationalBankFraudDetection** {  /\*\*  \* Calculates the number of notifications sent to the customer based on their daily spending.  \*  \* @param expenses An array of daily spending amounts.  \* @param d The number of previous days to consider for calculating the median.  \* @return The number of notifications sent to the customer.  \*/  **public** **static** **int** **countFraudNotifications**(**int**[] expenses, **int** d) {  **if** (d <= **0** || expenses.length <= d) {  **return** **0**; // No notifications if not enough data  }  **int** notifications = **0**;  PriorityQueue<Integer> low = **new** PriorityQueue<>(Collections.reverseOrder()); // Max heap for the lower half  PriorityQueue<Integer> high = **new** PriorityQueue<>(); // Min heap for the upper half  **for** (**int** i = **0**; i < expenses.length; i++) {  // Only start checking for notifications after we have d days of data  **if** (i >= d) {  **double** median = getMedian(low, high);  // Check if the current expense triggers a notification  **if** (expenses[i] >= **2** \* median) {  notifications++;  }  // Remove the expense that is no longer considered  removeExpense(low, high, expenses[i - d]);  }  // Add the current expense to the heaps  addExpense(low, high, expenses[i]);  }  **return** notifications;  }  /\*\*  \* Adds an expense to the appropriate heaps to maintain the median calculation.  \*  \* @param low Max heap containing the lower half of the expenses.  \* @param high Min heap containing the upper half of the expenses.  \* @param expense The expense amount to be added.  \*/  **private** **static** **void** **addExpense**(PriorityQueue<Integer> low, PriorityQueue<Integer> high, **int** expense) {  **if** (low.isEmpty() || expense <= low.peek()) {  low.offer(expense);  } **else** {  high.offer(expense);  }  balanceHeaps(low, high);  }  /\*\*  \* Removes an expense from the heaps and rebalances them if necessary.  \*  \* @param low Max heap containing the lower half of the expenses.  \* @param high Min heap containing the upper half of the expenses.  \* @param expense The expense amount to be removed.  \*/  **private** **static** **void** **removeExpense**(PriorityQueue<Integer> low, PriorityQueue<Integer> high, **int** expense) {  **if** (low.contains(expense)) {  low.remove(expense);  } **else** {  high.remove(expense);  }  balanceHeaps(low, high);  }  /\*\*  \* Balances the two heaps to ensure that the difference in their sizes is at most one.  \* The `low` heap (max-heap) is allowed to have one more element than the `high` heap (min-heap)  \* if the total number of elements is odd. This ensures that the median can be efficiently  \* retrieved from the top of these heaps.  \*  \* @param low Max heap containing the lower half of the expenses.  \* @param high Min heap containing the upper half of the expenses.  \*/  **private** **static** **void** **balanceHeaps**(PriorityQueue<Integer> low, PriorityQueue<Integer> high) {  // If `low` has more than one extra element, move the top of `low` to `high`  **if** (low.size() > high.size() + **1**) {  high.offer(low.poll());  }  // If `high` has more elements than `low`, move the top of `high` to `low`  **else** **if** (high.size() > low.size()) {  low.offer(high.poll());  }  }  /\*\*  \* Calculates the median from the current state of the heaps.  \*  \* @param low Max heap containing the lower half of the expenses.  \* @param high Min heap containing the upper half of the expenses.  \* @return The median value.  \*/  **public** **static** **double** **getMedian**(PriorityQueue<Integer> low, PriorityQueue<Integer> high) {  **if** (low.size() > high.size()) {  **return** low.peek();  } **else** {  **return** (low.peek() + high.peek()) / **2.0**;  }  }  /\*\*  \* Main method for testing the functionality of the fraud detection system.  \*  \* @param args Command line arguments.  \*/  **public** **static** **void** **main**(String[] args) {  **int**[] expenses = {**10**, **20**, **30**, **40**, **50**};  **int** d = **3**;  **int** notifications = countFraudNotifications(expenses, d);  System.out.println("Number of notifications sent: " + notifications);  }  } |

**Tests :**

|  |  |
| --- | --- |
| 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  78  79 | **package** com.eldarinterview;  **import** **org.junit.jupiter.api.Test**;  **import** **static** org.junit.jupiter.api.Assertions.\*;  **public** **class** **NationalBankFraudDetectionTest** {  /\*\*  \* Test for the case where the number of days (`d`) is 0.  \* Expects no notifications because there is no data to analyze.  \*/  **@Test**  **public** **void** **testCountFraudNotifications\_DaysZero**() {  **int**[] expenses = {**10**, **20**, **30**, **40**, **50**};  **int** d = **0**;  **int** expectedNotifications = **0**;  assertEquals(expectedNotifications, NationalBankFraudDetection.countFraudNotifications(expenses, d));  }  /\*\*  \* Test for the case where the number of days (`d`) is greater than the number of expenses.  \* Expects no notifications because there is insufficient data.  \*/  **@Test**  **public** **void** **testCountFraudNotifications\_InsufficientData**() {  **int**[] expenses = {**10**, **20**, **30**};  **int** d = **4**;  **int** expectedNotifications = **0**;  assertEquals(expectedNotifications, NationalBankFraudDetection.countFraudNotifications(expenses, d));  }  /\*\*  \* Test for the standard case with sufficient data to calculate notifications.  \* Expects 2 notifications as spending exceeds twice the median on specific days.  \*/  **@Test**  **public** **void** **testCountFraudNotifications\_MultipleNotifications**() {  **int**[] expenses = {**10**, **20**, **30**, **40**, **50**, **80**, **90**};  **int** d = **3**;  **int** expectedNotifications = **2**; // Days 4,6 trigger notifications  assertEquals(expectedNotifications, NationalBankFraudDetection.countFraudNotifications(expenses, d));  }  /\*\*  \* Test for the case where no notifications should be triggered.  \* Expects 1 notification as spending exceeds twice the median on one day.  \*/  **@Test**  **public** **void** **testCountFraudNotifications\_NoNotifications**() {  **int**[] expenses = {**10**, **20**, **30**, **40**, **50**};  **int** d = **3**;  **int** expectedNotifications = **1**;  assertEquals(expectedNotifications, NationalBankFraudDetection.countFraudNotifications(expenses, d));  }  /\*\*  \* Test for the case with an even number of expenses in the sliding window.  \* Expects 2 notifications as spending exceeds twice the median on specific days.  \*/  **@Test**  **public** **void** **testCountFraudNotifications\_EvenDaysWindow**() {  **int**[] expenses = {**10**, **20**, **30**, **50**, **80**, **100**};  **int** d = **4**;  **int** expectedNotifications = **2**;  assertEquals(expectedNotifications, NationalBankFraudDetection.countFraudNotifications(expenses, d));  }  /\*\*  \* Test for the case where all expenses are equal.  \* Expects no notifications as spending does not exceed twice the median on any day.  \*/  **@Test**  **public** **void** **testCountFraudNotifications\_AllEqualExpenses**() {  **int**[] expenses = {**20**, **20**, **20**, **20**, **20**, **20**};  **int** d = **3**;  **int** expectedNotifications = **0**;  assertEquals(expectedNotifications, NationalBankFraudDetection.countFraudNotifications(expenses, d));  } |

**Explicación del Código**

1. Estructura de Heaps: Se utilizan dos montones (`low` y `high`) para gestionar los gastos. El max heap (`low`) mantiene la mitad inferior de los gastos, mientras que el min heap (`high`) mantiene la mitad superior. Esto permite un acceso rápido a la mediana.

2. Método **addExpense** : Este método se encarga de añadir un nuevo gasto a los montones y de equilibrar los heaps si es necesario.

3. Método **removeExpense**: Elimina el gasto que ya no es relevante y reequilibra los montones si es necesario.

4. Método **getMedian**: Este método devuelve la mediana de los gastos de forma eficiente, dependiendo de los tamaños de los montones.

5. Método **balanceHeaps**: El objetivo de balanceHeaps es asegurarse de que la diferencia en el tamaño de los dos heaps no sea mayor a uno. Esto es crucial porque, para calcular la mediana eficientemente, el max-heap puede tener uno más que el min-heap, pero nunca al revés.

**Conclusión**

La implementación propuesta es eficiente en términos de complejidad algorítmica y utiliza buenas prácticas de programación. Mediante el uso de estructuras de datos adecuadas como los montones, se optimiza el cálculo de la mediana y se facilita la detección de actividades fraudulentas en las cuentas de los clientes. Esta solución es escalable y adecuada para manejar un volumen significativo de datos de gastos.