Díaz Graziano, Adán Matías
Pérez, Lucas

Trabajo práctico 1

Breve explicación de las resoluciones

Problema 1: Palabra más usada
Se desea implementar un sistema que dado un String retorne la palabra más usada. Las palabras son divididas por la existencia de espacios o caracteres diferentes a letras entre las mismas. Solo se considera como palabra a toda secuencia de caracteres con una longitud mayor o igual a N donde N es un parámetro. Finalmente, la igualdad entre las palabras debe ignorar el uso de mayúsculas y minúsculas.

Resolución
Inicialmente se realiza el preprocesamiento de la cadena de entrada mediante el método *split* y una expresión regular que nos permite quedarnos con caracteres alfabéticos. Adicionalmente se utilizó un *Map<String, Integer>* como estructura auxiliar para llevar el conteo de apariciones de cada palabra. Por cada palabra que hayamos obtenido luego de la aplicación del método *split*, se verifica que su longitud sea mayor o igual a la del delimitador recibido por parámetro. Si esta condición se cumple, la palabra es insertada en el *Map* con su contador inicializado en 1. Si la palabra ya está contenida en el mismo, este se incrementa. Se utilizó la palabra en minúsculas como clave de acceso para el mapa, pero se almacenó su valor original en una variable que contabiliza cuál fue la palabra más utilizada hasta el momento.

Problema 2: Fibonacci
Utilizando solo tipos primitivos, implemente un algoritmo que retorne el valor de Fibonacci para un entero mayor o igual a 0 y menor igual a 90.

Resolución
Inicialmente se implementó una versión del algoritmo que solo resolviera de forma recursiva la ecuación dada. Rápidamente notamos que esta solución presentaba problemas para números altos debido a que calculaba múltiples veces los valores de iteraciones anteriores. Como alternativa utilizamos una lista para almacenar los resultados de cálculos previos. Esta implementación eliminó el problema de la recursión infinita pero comenzaron los errores de cálculo por *overflow* de las variables enteras utilizadas. Logramos que el algoritmo funcione correctamente hasta la entrada 90 utilizando variables de tipo *long*. Finalmente, para la versión final del algoritmo dejamos atrás los tipos primitivos para experimentar con la clase BigInteger, que ofrece una representación interna con el estándar BCD y no está acotado a rangos aritméticos.

Problema 3: Árbol de búsqueda
Dado un árbol, determine si dicho árbol es un árbol binario de búsqueda. Se considera que es un árbol binario de búsqueda si:

- El dato de un nodo es mayor a todos los datos en su rama izquierda.
- El dato de un nodo es menor a todos los datos en su rama derecha.

Resolución
Para determinar si el árbol dado es un árbol binario de búsqueda se realiza un recorrido recursivo sobre el mismo. Inicialmente se compara el dato del nodo actual con los límites dados (para la raíz, estos límites estarán dados por los valores extremos del tipo de dato utilizado*). Si se cumple que el dato esté entre ellos, se procede a explorar de forma recursiva sus hijos izquierdo y derecho. Para el hijo izquierdo el dato del nodo actual será el nuevo máximo, y para el derecho este se convertirá en el nuevo mínimo. Si todos los datos cumplen la condición dada, el árbol dado es un árbol binario de búsqueda.
* Nuestra definición de árbol binario de búsqueda no acepta elementos repetidos, por lo que estos valores, al utilizarse como discernibles, producen resultados inesperados en la invocación al método si el árbol en cuestión los incluye.

Problema 4: Fotografía artística
Se está desarrollando un sistema para un estudio fotográfico que desea saber cuántas fotografías artísticas se pueden realizar con un arreglo de fotógrafos, artistas y escenario. La organización de los fotógrafos, artistas y fondos se presenta en un arreglo A de N elementos, donde:

- A[i]=’f’ si en la posición i hay un fotógrafo.
- A[i]=’a’ si en la posición i hay un artista.
- A[i]=’e’ si en la posición i hay un escenario.
- A[i]=’.’ si en la posición i no hay nada.

Resolución
Para resolver el problema se implementaron tres métodos. El método principal es **fotografiasArtisticas**, que recorre todo el arreglo **arr** buscando posiciones donde haya un 'f’. Cuando se encuentra una 'f', se realizan dos búsquedas: una hacia la derecha (dirección positiva) y otra hacia la izquierda (dirección negativa), en busca de una combinación válida de 'a' (artista) y 'e' (escenario).
Estas búsquedas se realizan mediante los métodos recorridoPositivo y recorridoNegativo, que funcionan de forma simétrica pero en direcciones opuestas. Se explicará uno de ellos, dado que ambos comparten la misma lógica pero diferente recorrido del arreglo.
Método recorridoPositivo
En este se verifica que los límites de búsqueda estén dentro del rango válido del arreglo. Después se recorre el arreglo desde pos_i hasta pos_f, buscando la letra ‘target’, que inicialmente será 'a'.
Si se encuentra un 'a', se realiza una llamada recursiva al mismo método, pero ahora con 'e' como nuevo ‘target’. Esta búsqueda se realiza desde la posición del 'a' encontrada, avanzando también entre x e y posiciones.
Si la recursión encuentra un 'e' dentro del rango válido, se suma 1 al contador.
Finalmente, se retorna la cantidad total de combinaciones válidas encontradas desde esa dirección.
Método recorridoNegativo
Es similar a recorridoPositivo, pero realiza la búsqueda en sentido inverso (de derecha a izquierda en el arreglo). La lógica y validaciones son equivalentes, pero se adaptan a las condiciones necesarias para evitar desbordes por índices negativos.
Método fotografiasArtisticas
Este método por cada fotógrafo encontrado ('f'), realiza las dos búsquedas mencionadas y suma sus resultados.

Problema 5: Laberinto mágico
Este ejercicio nos recordó al problema del tablero de ajedrez, que se resolvía mediante un algoritmo de *BFS*. En cada paso, teníamos cuatro movimientos posibles (arriba, abajo, izquierda y derecha), y debíamos evaluar recursivamente cada uno para determinar cuál nos llevaba a la salida. Implementamos un algoritmo que, en cada movimiento, se llamaba a sí mismo para explorar las opciones disponibles, actualizando los índices que indican nuestra posición en la matriz y llevando un conteo acumulado de los movimientos realizados hasta ese momento.
Previamente al ejecutar el algoritmo realizamos un recorrido del mapa para reconocer los portales y la entrada al laberinto.
Estructuras utilizadas:

Matriz del laberinto: *char[][]* laberinto
Guarda el mapa del laberinto con todos sus elementos: entrada, salida, paredes, caminos y portales.

Portal: Clase Portal
Tuvimos que crear la clase Portal para poder tener una estructura que esté compuesta por 4 posiciones, dos del portal y dos de su respectiva salida.

Portales: *Map<Character, Portal>*
Se utilizó un *Map* para poder acceder a la salida del portal directamente con el nombre, con una complejidad O(1). Esto fue porque sabemos el coste computacional que tiene este ejercicio en un mapa muy grande.

Visitados: *boolean[][]* visitados
Se creó otro mapa igual vacío pero de *boolean* para marcar qué celdas ya fueron visitadas en ese camino y evitar ciclos o pasos innecesarios. Al volver recursivamente se desmarcan.