UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE ESTUDIOS SUPERIORES ARAGÓN

PROYECTO 3
"Recursividad(Torres de Hanói), Colas (Caja) y Pilas(Inundación)"

MATERIA: Estructuras de datos.

GRUPO: 1307

PROFESOR: Miguel Ángel Sánchez Hernández.

FECHA DE ENTREGA: martes 26 de enero, 2021.

INTEGRANTES: Gómez Guzmán Ricardo.

Padilla Lozano Silvano Fabian.

Vázquez Saldaña Areli Monserrat.

Índice

Objetivos

- Aplicar los conocimientos adquiridos a lo largo del curso.
- Entender qué es la recursividad dentro de la programación.
- Saber cuáles son las ventajas y desventajas que conlleva el realizar una programación recursiva frente a una iterativa.
- Aprender a realizar un programa muy conocido dentro de la programación, el cual es el juego de las “Torres de Hanoi” mediante una programación recursiva o iterativa.
- Saber en qué momento es más eficiente y factible el utilizar la recursividad dentro de estructuras de datos.
- Entender el funcionamiento de las pilas y colas.
- Entender cómo funciona el programa de colas para su funcionamiento en la vida real.
- Aplicar los métodos correspondientes a los temas correspondientes.

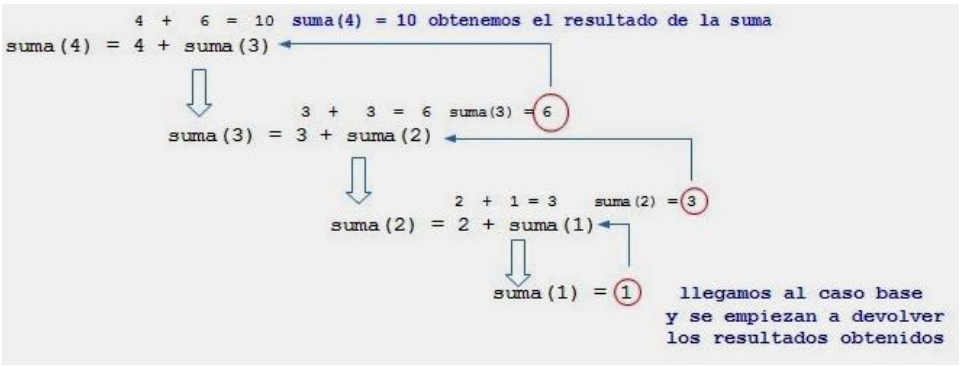
Introducción

Recursividad

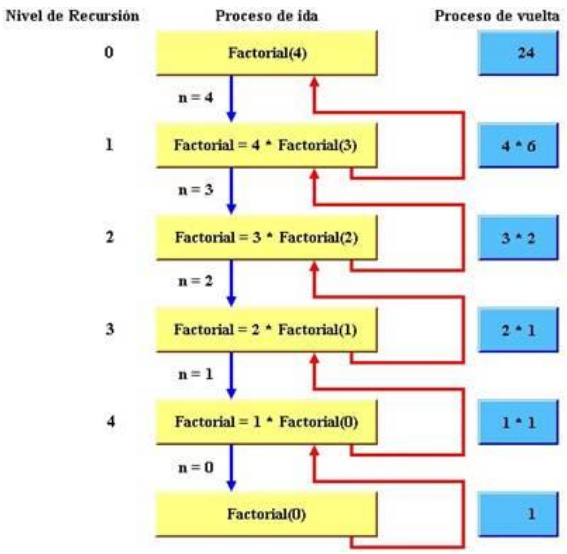
La recursividad es una técnica muy empleada en la programación informática y consiste en que una función se llame a sí misma. Un método usual de simplificación de un problema complejo es la división de este en subproblemas del mismo tipo. Esta técnica de programación se conoce como divide y vencerás y es el núcleo en el diseño de numerosos algoritmos de gran importancia, así como también es parte fundamental de la programación dinámica.

Es una alternativa diferente para implementar estructuras de repetición (ciclos). Los módulos se hacen llamadas recursivas.

Se puede usar en toda situación en la cual la solución pueda ser expresada como una secuencia de movimientos, pasos o transformaciones gobernadas por un conjunto de reglas no ambiguas.



Uno de los ejemplos clásicos dentro de la recursividad es la función que calcula la factorial de un número. Una factorial consiste en multiplicar un número natural por el número anterior, y este a su vez por el anterior, y así sucesivamente hasta llegar al número 1. Por ejemplo, la factorial de 8 sería el resultado de multiplicar 8 por 7, luego por 6 y así sucesivamente hasta llegar a uno.



Implementación en C++:

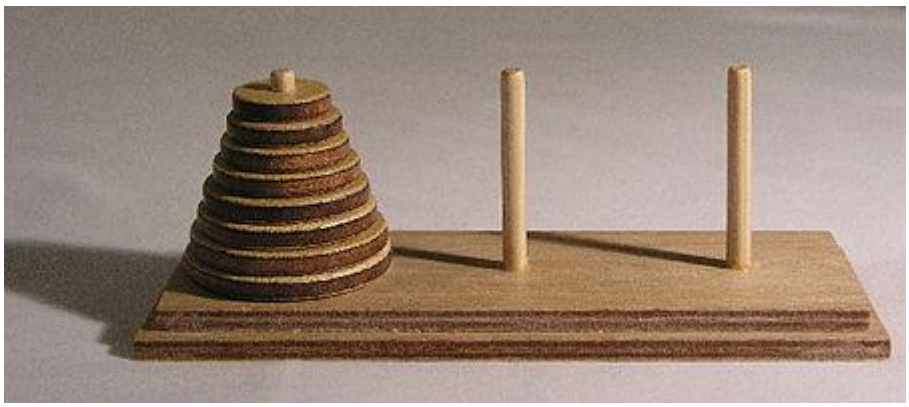
```cpp
int factorial(int x)
{
   if (x > -1 && x < 2) return 1;  // Cuando -1 < x < 2 devolvemos 1 puesto que 0! = 1 y 1! = 1
   else if (x < 0) return 0;       // Error no existe factorial de números negativos
   return x * factorial(x - 1);    // Si x >= 2 devolvemos el producto de x por el factorial de x - 1
}
```

Una programación recursiva dentro de las ventajas que tiene frente a una programación iterativa es que simplifica un problema y lo resuelve desde abajo hacia arriba. Mientras que un proceso iterativo repite un proceso una y otra vez empezando cada nueva iteración con el resultado de la iteración anterior. El objetivo principal de estas técnicas de recursividad es el poder acelerar la ejecución de un programa.

Torres de Hanoi

Las Torres de Hanoi es un rompecabezas o juego matemático inventado en 1883 por el matemático francés Édouard Lucas. Este juego de mesa individual consiste en un número de discos perforados de radio creciente que se apilan insertándose en uno de los tres postes fijados a un tablero.

El problema es muy conocido en la ciencia de la computación y aparece en muchos libros de texto como introducción a la teoría de algoritmos. Este problema se suele plantear a menudo en programación, especialmente para explicar la recursividad.



El problema de las torres de Hanoi puede resolverse fácilmente usando *recursividad*. La estrategia consiste en considerar uno de los pivotes como el origen y otro como destino. El problema de mover n discos al pivote derecho se puede formular recursivamente como sigue:

- Mover los n-1 discos superiores del pivote izquierdo al del centro empleando como sería el pivote de la derecha.
- Mover el n-ésimo disco (el más grande) al pivote de la derecha.
- Mover los n-1 discos del pivote del centro al de la derecha.

Otra manera de resolver el problema, sin utilizar la recursividad (*manera iterativa*), se basa en el hecho de que, para obtener la solución más corta, es necesario mover el disco más pequeño en todos los pasos impares, mientras que en los pasos pares solo existe un movimiento posible que no lo incluye. El problema se reduce a decidir en cada paso impar a cuál de las dos pilas posibles se desplazará el disco pequeño. El algoritmo en cuestión depende del número de discos del problema:

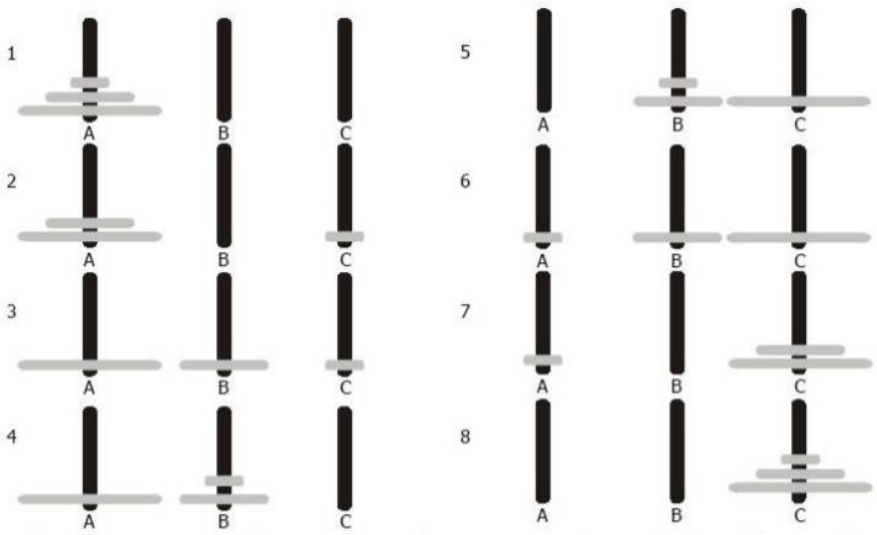
- Si inicialmente se tiene un número impar de discos, el primer movimiento debe ser colocar el disco más pequeño en la pila *destino*, y en cada paso impar se le mueve a la siguiente pila a su izquierda (o a la pila *destino* si está en la pila *origen*).

La secuencia será: *destino*, *auxiliar*, *origen*, *destino*, *auxiliar*, *origen*, etc.

- Si se tiene inicialmente un número par de discos, el primer movimiento debe ser colocar el disco más pequeño en la pila *auxiliar*, y en cada paso impar se le mueve a la siguiente pila a su derecha (o a la pila *origen* si está en la pila *destino*).

La secuencia será: *auxiliar*, *destino*, *origen*, *auxiliar*, *destino*, *origen*, etc.

Una forma equivalente de resolverlo es la siguiente: coloreando los discos pares de un color y los impares de otro, y se resuelve el problema añadiendo la siguiente regla: no colocar juntos dos discos de un mismo color. De esta manera, solo queda un movimiento posible (además del de volver hacia atrás).



Pilas

Estructuras de datos lineales que contienen restricciones que, a diferencia de las colas, se le pueden agregar o quitar datos por un extremo; la cual es la cima. Una característica notable de las pilas es que el último dato que se inserta es el primero en salir (LIFO. Last in, first out).

Las pilas contienen un procesador en el cual dependen de si este elabora arreglos con diferentes tipos de ArrayList, y así los arreglos trabajan con su respectiva lista y su longitud, si la longitud no es establecida ya que puede variar desde su tamaño dependiendo la necesidad del problema o la situación en la que se encuentre y se esté empleando.

Pilas más conocidas en internet como conexión de datos los cuales se pueden acceder mediante un extremo, que se conoce generalmente como tope.

Problemáticas y soluciones.

Torres de Hanoi.

A nuestro usuario le interesa jugar con las torres de Hanói, el decidirá con cuantos discos quiere jugar y se le dará un estimado de movimientos que haya realizado mientras él estaba acomodando los discos en la torre final.

Aquí se necesita implementar el método recursivo ya antes mencionado y así poder tener ayuda con el código que se realizó, ya que por medio de esté se irá guiando la consola para poder sacar el resultado esperado para poder resolver el juego de las torres, irá acomodando uno de los discos en las torres diferentes para que ningún disco pueda

Para realizar este programa utilizamos el programa antes que son el de recursividad y el kit de Herramientas, los cuales van implementados y así poder trabajar con este.

Se crean nuestras variables, las cuales son los discos y las 3 torres en donde se moverán estos discos. Después se crea el método recursivo en el cual calcula cada movimiento factorial para que este saque en consola el resultado del movimiento de discos y diga para que torre se va cada uno de los discos.

Si la recursividad se va cumpliendo se moverá el primer disco de origen a el destino esperado, si va bien el disco, seguirá a mover el disco que estaba debajo de este que es el origen y de este a la torre destino y así sucesivamente se irán moviendo los discos en cada una de las torres, respetando el orden y las reglas del juego.

```
33      //TORRES DE HANOI
34      public static void TorresDeHanoi(int disco, String origen, String medio, String destino) {//Creación del método de torres de hanoi
35
36          if (disco == 1) {//creacion de la condición para que el número minimo de discos sea 1 para jugar o más
37              nMovimientos = nMovimientos +1;//se calcula el número de movimientos y se le da un incremento
38              System.out.println(nMovimientos + "→ Mover disco número:  " + disco +" de "+ origen + " a " + destino);
39          } else {
40              TorresDeHanoi(disco - 1, origen, destino, medio);//Movimientos que dará el disco de origen a al medio
41              nMovimientos = nMovimientos +1;//se hace un incremento de movimientos
42              System.out.println(nMovimientos+ "→ Mover disco número:  " + disco + " de " + origen + " a " + destino);
43              TorresDeHanoi(disco - 1, medio, origen, destino);// Movimiento que dara el disco de en medio al destino.
44
45          }
46      }
```

Ya por último se imprime en consola el nombre “TORRES DE HANOI” y posteriormente el número de discos con el que se utiliza la recursividad para que el programa vaya haciendo el proceso de movimiento y al momento de darle enter, este comienza a trabajar. Ya que termino de leer cada uno de los movimientos y los analizo, en consola aparecerá el resultado con el disco que se fue moviendo de torre a torre y también saldrá el número de movimientos.

```
48      public static void main(String[] args) {
49          System.out.println("Factorial");
50          System.out.println(Recursividad.factorial(3));
51          System.out.println("Fibonacci");
52          System.out.println(Recursividad.fibonacci(5));
53          System.out.println("---------------");
54          System.out.println("TORRES DE HANOI");
55          Scanner sc = new Scanner(System.in);
56          System.out.println("Numero de discos:");// usuario ingresa con cuantos discos desea jugar
57          int nDiscos = sc.nextInt(); // se declará  y se guarda el número de discos que se ingresa
58          System.out.println("");
59          TorresDeHanoi(nDiscos, "origen", "medio", "destino");//llamo a mi método para llamar a las 3 torres y a los discos
60          System.out.println("\nSe resolvio en: " + nMovimientos+ " movimientos");
61      }
62
```

Resultado en consola:

En este punto el usuario deberá darle el número de discos con los que desea jugar, para el ejemplo utilizaremos 7 discos para ver la solución y resultado que arroja el programa.

```
---------------
TORRES DE HANOI
Numero de discos:
```

Así va saliendo el resultado hasta llegar a la torre esperada.

```
1→ Mover disco número:  1 de origen a destino
2→ Mover disco número:  2 de origen a medio
3→ Mover disco número:  1 de destino a medio
4→ Mover disco número:  3 de origen a destino
5→ Mover disco número:  1 de medio a origen
6→ Mover disco número:  2 de medio a destino
7→ Mover disco número:  1 de origen a destino
8→ Mover disco número:  4 de origen a medio
9→ Mover disco número:  1 de destino a medio
```

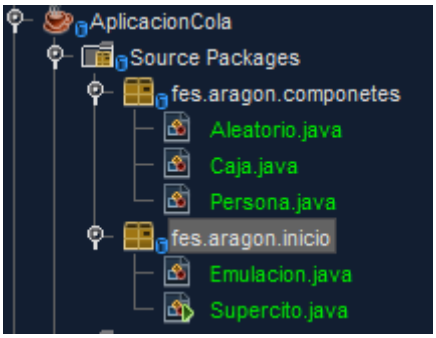
Como se puede ver en total fueron 127 movimientos que hizo el juego y llegó a la torre destino y así se ha completa el juego.

```
126→ Mover disco número:  2 de medio a destino
127→ Mover disco número:  1 de origen a destino

Se resolvio en: 127 movimientos
BUILD SUCCESSFUL (total time: 52 seconds)
```

Aplicación de cajas (supercito).

Para este programa se necesita implementar el kit de estructuras ya antes realizado y ocupado en otros programas, se ocupa cola y lista simple, ya que se haya implementado estas clases, llegó el momento de crear nuestro proyecto base que se llamará AplicaciónCola, en esta se va a ir creando un paquete de componentes y ahí se creará cada una de las clases a ocupar las cuales son clase aleatorio, clase caja y la clase persona; en otro paquete de inicio se crearán las clases emulación y la clase supercito que será nuestro main.



La clase “aleatorio” donde traerá a la persona formada en la fila y el tiempo en atender en la caja, aquí nos brindará números aleatorios de persona y del tiempo en que serán atendidos cada una de nuestras personas y el tiempo que se tarda respectivamente cada caja.

```java
public class Aleatorio {

    public static boolean personaEnFila() {
        Random rd = new Random();
        return rd.nextBoolean();
    }

    public static int tiempoEnAtender() {
        Random rd = new Random();
        return ((int) (rd.nextDouble()* 10 + 5));
    }
}

```

La clase caja en la cual llevará el tiempo en atender , tiempo restante, la persona y el semáforo que indicará cuando la persona pueda pasar a la caja. Se crea el constructor con las variables que se crearon. Esta clase es la encargada de llevar los tiempos correspondientes.

```java
public class Caja {

    private boolean semaforo = true;
    private int tiempoAntender;
    private int tiempoRestante;
    private Persona persona;

    public Caja(int tiempoParaAntender, int tiempoRestante, Persona persona) {
        this.tiempoAntender = tiempoParaAntender;
        this.tiempoRestante = tiempoRestante;
        this.persona = persona;
    }
```

En la clase persona se tienen las variables de tiempo en fila, tiempo atendido, tiempo total, los cuales empiezan desde 0 ya que empezará su conteo desde ahí, esto nos ayudará a tener un mejor control del tiempo en que se va registrando en las cajas.

```java
public class Persona {
    private int tiempoFila = 0;
    private int tiempoAtendido=0;
    private int tiempoTotal = 0;

    public Persona(int tiempoFila, int tiempoTotal) {
        this.tiempoFila = tiempoFila;
        this.tiempoTotal = tiempoTotal;
    }

```

En el otro paquete tenemos la clase emulación donde será la implementación de todos los métodos y así poder llevar a cabo el funcionamiento correcto de las cajas y sus tiempos.

```java
public class Emulacion {

    private ColaLista<Persona> fila = new ColaLista<>(100);
    private ListaSimple cajas = new ListaSimple();
    private ListaSimple personasAtendidas = new ListaSimple();
    private int tiempoTotal = 0;

  public Emulacion() {

    }

    public void correr(int tiempoDeEmulacion) {
        int incrementoTiempo = 0;
        cajas.agregarEnCola(new Caja(0, 0, null));
        while (incrementoTiempo <= tiempoDeEmulacion) {
            if (Aleatorio.personaEnFila()) {
                fila.insertar(new Persona(0, 0));
            }
```

Se implementan los métodos que necesitaremos y se empieza a hacer la emulación del código en donde el primer método será correr aquí se irá incrementando el tiempo en el que la caja va trabajando y el tiempo de la persona en fila.

Posteriormente tenemos un if en donde se encargará del pase de personas y del tiempo que va transcurriendo en nuestra lista de tiempo que va juntando la persona y el tiempo que se le resta.

```java
if (!this.cajas.esVacia()) {
    for (int i = 0; i < cajas.getLongitud(); i++) {
        Caja tmp = ((Caja) cajas.obtenerNodo(i));
        if (!tmp.isSemaforo()) {
            tmp.setTiempoRestante(tmp.getTiempoRestante() - 1);
            if (tmp.getTiempoRestante() == 0) {
                this.personasAtendidas.agregarEnCola(new Persona(tmp.getPersona()));
                tmp.setSemaforo(true);
                tmp.setPersona(null);
            }
        }
    }
}
```

Las personas pasan a la caja y se realiza el recorrido nuevamente , la primera condición es ver si el semáforo está disponible para que otra persona pase a la caja, pero sino esta disponible el semáforo en verde, no puede pasar a la caja. En caso de que pase se tomará en cuenta el tiempo que lleva en fila, el tiempo que será atendido, estos 2 tiempos se suman y dan el resultado del tiempo total.

```java
if (!fila.estaVacia()) {
    for (int i = 0; i < cajas.getLongitud(); i++) {
        Caja tmp = ((Caja) cajas.obtenerNodo(i));
        if (tmp.isSemaforo()) {
            tmp.setPersona(fila.extraer());
            if (tmp.getPersona() == null) {
                tmp.setSemaforo(true);
            } else {
                tmp.setSemaforo(false);
                tmp.setTiempoAntender(Aleatorio.tiempoEnAtender());
                tmp.setTiempoRestante(tmp.getTiempoAntender());
                tmp.getPersona().setTiempoAtendido(tmp.getTiempoAntender());
                tmp.getPersona().setTiempoTotal(
                        tmp.getPersona().getTiempoFila() + tmp.getTiempoRestante());

            }
        }
    }
```

Si el tiempo se pasa de 15 minutos se puede abrir una nueva caja.

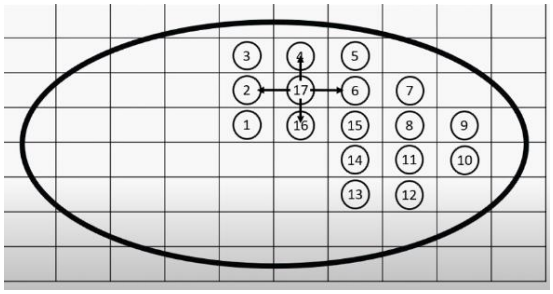
```
70              if (!personasAtendidas.esVacia()) {
71                  for (int i = 0; i < personasAtendidas.getLongitud(); i++) {
72                      Persona tmp = (Persona) personasAtendidas.eliminarEnCabeza();
73                      System.out.println(tmp.toString());
74                      if (tmp.getTiempoTotal() >= 15) {
75                          this.cajas.agregarEnCola(new Caja(0, 0, null));
76                          System.out.println("-->SE ABRE UNA CAJA NUEVA<--" + this.cajas.getLongitud());
77                      }
78                  }
79              }
```

Y por último nuestra clase supercito que es la clase principal en donde se correrá el programa.

```
3   public class Supercito {
4
5       public static void main(String[] args) {
6           Emulacion emu = new Emulacion();
7
8           emu.correr(150);
9       }
10  }
```

Recursividad pila.

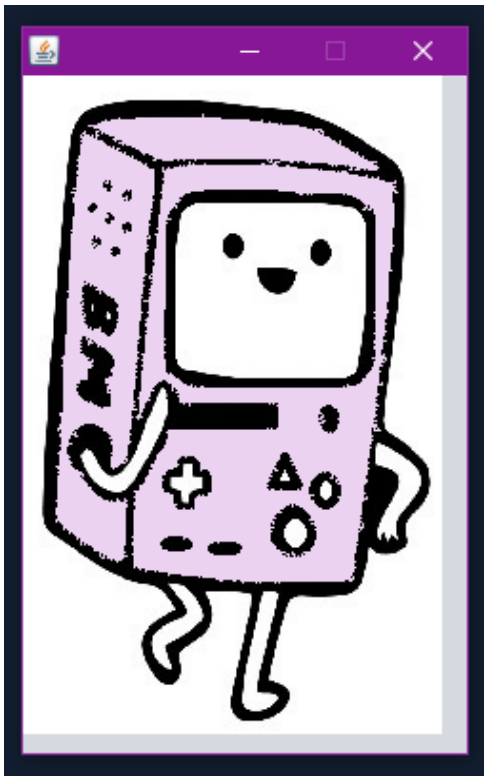
Aquí se aplicará nuestro kit de herramientas y ocuparemos la clase de pila, después se comenzará a crear la clase de inundación en donde avanzará conforme se vaya seleccionando un espacio en blanco en nuestra imagen. Como se muestra en nuestro código utilizamos swing para que se pueda mostrar en pantalla.



Se crean las clases en donde se trabajará con la clase Panel pila en donde se activará las actividades del mouse junto con la imagen , se le dará color para que al momento de darle clic se vea reflejado en pantalla, también se le dará un borde que en este caso el color es el negro.

```java
private void inundacion(int x, int y) {
    PilaInundacion<Point> pila = new PilaInundacion<>();
    pila.insertar(new Point(x, y));
    while (!pila.estaVacia()) {
        Point p = (Point) pila.extraer();
        int colorImg = img.getRGB(p.x, p.y);
        if (colorImg != -1322255 && (colorImg >= -16514044 && colorImg <= 1677721
            img.setRGB(p.x, p.y,-1322255);
            pila.insertar(new Point(p.x + 1, p.y));//llamada a la derecha
            pila.insertar(new Point(p.x - 1, p.y));//llamada a la izquierda
            pila.insertar(new Point(p.x, p.y + 1));//llamada abajo
            pila.insertar(new Point(p.x, p.y - 1));//llamada arriba
        }
        repaint();
    }
}
```

Al momento de verlo en pantalla se ve algo así:



Conclusiones

Durante este proyecto se aprendió qué es la recursividad gracias al programa de torres de Hanói y cómo es que esta misma está empleada dentro de la programación y las estructuras de datos. Del mismo modo, el equipo logró entender y analizar de una mejor manera cómo es que una programación recursiva tiene sus ventajas y desventajas frente a una programación iterativa dentro de este ámbito.

También se entendió el por qué se debería escribir programas recursivos; ya que como analizamos, estos son más cercanos a las descripciones matemáticas, generalmente son más fáciles de analizar, se logran adaptar mejor a las estructuras de datos recursivas y cómo

es que estos algoritmos recursivos nos ofrecen soluciones estructuradas, modulares y simples.

Para finalizar, y como parte del análisis. Después de saber cómo una programación recursiva nos da muchas ventajas y cómo esta nos es factible al momento de programar ya que nos deja simplificar código, acelera la ejecución de un programa, o bien, nos funciona mejor al momento de cuando una estructura de datos sea recursiva, por ejemplo, en árboles. También debemos tener en cuenta que la recursividad a veces no es factible o la mejor opción, como por ejemplo cuando los métodos usan arreglos largos, cuando el método cambia de manera impredecible de campos o simplemente cuando las iteraciones sean la mejor opción para utilizar.

En el proyecto de cajas se aprendió a trabajar con colas y observación varias características en el ámbito de implementación de esta clase a un nuevo programa, se definieron diferentes tipos de métodos de trabajar en donde se puede observar que se trabaja de manera optima y limpia. Aunque se tenían diferentes tipos de complicaciones con este programa, ya que se tenían que hacer un par de arreglos en la cola para que pudiera funcionar mejor y así el programa poder dar el resultado deseado.

Y por último tenemos el programa de inundación utilizando pila, donde se creó una nueva clase donde se implementaron todas las características y métodos de este. El relleno por inundación es una función que hemos utilizado todos a lo largo del tiempo y rara vez nos detenemos a pensar en cómo elaborar un proceso como este.

Posteriormente se le dieron las instrucciones necesarias en donde teníamos que comprender como iba a funcionar cada uno de los componentes que el usuario debería de ver sin ningún error, en este caso es el mouse y al momento que le diera clic el usuario tendría que ver como se iba coloreando la imagen sin ningún error.

Bibliografías

- Colaboradores de Wikipedia. (2020a, septiembre 4). Recursión. Recuperado 25 de enero de 2021, de https://es.wikipedia.org/wiki/Recursi%C3%B3n#Recursi%C3%B3n_en_inform%C3%A1tica
- Braybury, L. (s. f.). Diferencias entre recursión e iteración. Recuperado 25 de enero de 2021, de https://techlandia.com/diferencias-recursion-iteracion-info_254293/
- Colaboradores de Wikipedia. (2020b, octubre 27). Torres de Hanói. Recuperado 25 de enero de 2021, de https://es.wikipedia.org/wiki/Torres_de_Han%C3%B3i#:%7E:text=Las%20Torres%20de%20Han%C3%B3i%20es,postes%20fijados%20a%20un%20tablero.
- Tutorial Avanzado: Recursividad y Torres de Hanoi. (s. f.). Recuperado 25 de enero de 2021, de https://platzi.com/tutoriales/1050-programacion-basica/1498-tutorial-avanzado-recursividad-y-torres-de-hanoi/
- R. (s. f.). Torres de Hanoi. Recuperado 25 de enero de 2021, de http://programacionestructuradarrr.blogspot.com/2016/08/torres-de-hanoi.html
- Deitel, H., 2011. *Como Programar En C/C++ Y Java*. 7th ed. Ciudad de México: Pearson Educación de México, SA de CV.
- Cairo Battistutti, O. y Guardati Buemo, S., 1993. *Estructuras De Datos*. 3rd ed. Medellín: McGraw-Hill Interamericana.