Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Tarea 3

Pila de Arena Optimizada

Fecha: 08 de mayo de 2017
Autor: Daniel Araya Poblete
Email: Daniel.arayap1@gmail.com
Curso: CC3001-1
Profesor: Nelson Baloian T.

Introducción

El objetivo de este informe es presentar los resultados obtenidos a partir de una nueva solución para el problema de granos de arena de la tarea 1: Pilas de Arena. En éste se buscaba implementar un algoritmo que modele el comportamiento de una cantidad N de granos de arena, que se dejan caer sobre un mismo punto en el espacio.

En esta nueva solución, la diferencia está dada por la implementación de un TDA Pila, una estructura de datos de tipo LIFO (Last In First Out) que administre los índices de la matriz donde los granos no se encuentren en su condición de estabilidad (celdas donde el número de granos es mayor a 3). De este modo para cualquier celda que se encuentre en la pila, se le debe aplicar la condición de estabilidad y desapilar una vez se encuentre estable. Se abordará este problema con mayor detalle en el resto del informe.

Cabe mencionar que los algoritmos, las variables y los supuestos que se usaron y asumieron para la tarea 1 se mantendrán en esta tarea, es decir, la creación y utilización de un TDA es la única diferencia entre las tareas.

Como es de esperar, el tiempo de implementación del programa aplicando este recurso no será el mismo que el observado durante la primera tarea. Otro de los resultados que se entrega y discute en el presente informe son los tiempos de ejecución usando ambos programas y como varían estos tiempos mediante la aplicación de la estructura Pila previamente señalada.

En las siguientes páginas del informe se mostrará con mayor detalle las especificaciones del problema, los detalles para la implementación del TDA Pila y la aplicación de ésta en el programa. También se discutirán los resultados obtenidos con y sin la implementación de ésta en términos de eficiencia en la ejecución del código.

Análisis del Problema

El problema, como se menciona en la introducción, es idéntico al problema resuelto en la tarea 1. Se tiene una cantidad de granos N a estabilizar mediante la misma regla de estabilidad anterior: si la cantidad de granos en la celda es mayor o igual a 4, la celda se desborda sumando 1 a cada celda contigua a ésta, hasta que ya todas las celdas queden estables. Se hereda de la tarea 1, además, la misma forma de obtener el input y la clase Ventana para mostrar el resultado obtenido.

El requisito de la tarea 3 es implementar a la solución encontrada en la tarea 1, una solución alternativa que implemente un TDA Pila, con el cuál se guarden los índices de los casilleros de desborde (casilleros que deben ser desbordados), y se desapilen únicamente los que se encuentren en la pila en lugar de explorar todas las casillas presentes en la matriz.

El TDA Pila implementado (ver Anexos, archivo Pila.java) cuenta con un constructor vacío, y las operaciones básicas (ver Figura 1):

- boolean estaVacia(): indica si la Pila está vacía.
- void apilar(int i, int j): coloca los índices (i,j) de una matriz en la Pila.
- int [ ] desapilar(): extrae los índices del tope de la Pila y los entrega en un arreglo.

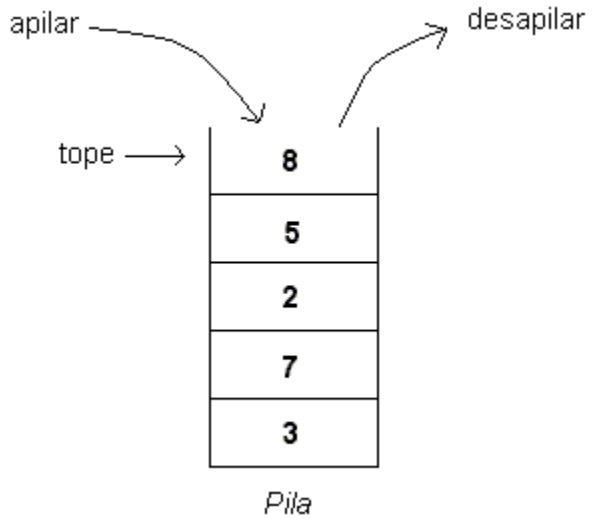

Figura 1. Esquema con las operaciones que implementa la clase Pila de la tarea 3 (tomado del apunte CC3001 Algoritmos y estructuras de datos).

Entonces, el análisis de este problema se enfocó en mantener parte del código y el algoritmo utilizado en la tarea 1, con la diferencia en la forma de explorar los índices que deben ser desbordados. En la tarea 3 los índices a desbordar se encontrarán extrayendo las

casillas en la Pila, mientras que en la tarea 1 se realizaba un recorrido completo de la matriz en busca de las celdas a desbordar. El proceso de desborde es el mismo en ambos casos y se explica brevemente en la Figura 2:

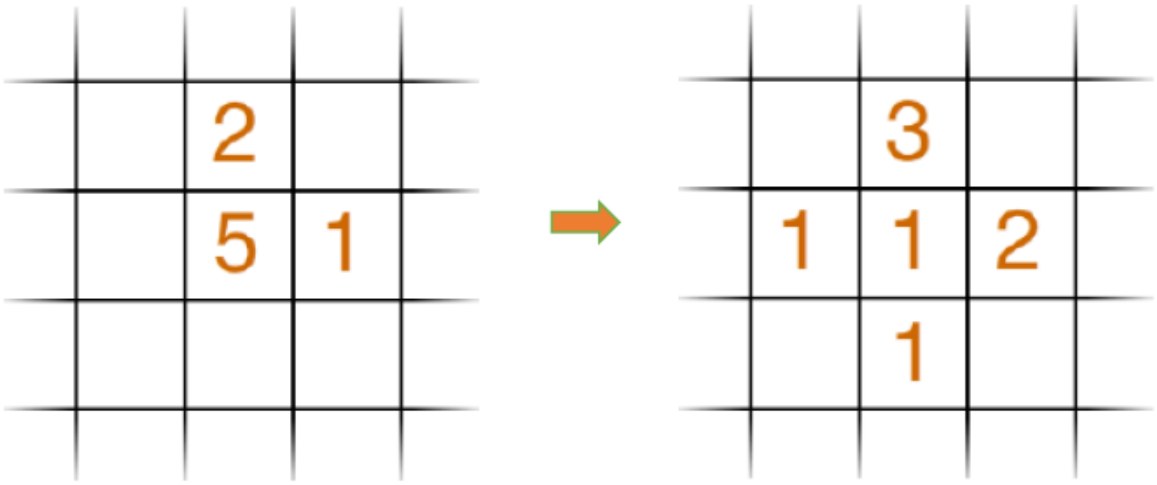

Figura 2. Puede observar que en la matriz de la izquierda el valor central es mayor a 3 granos, por lo que luego de aplicar una regla de equilibrio, a éste valor se le restan 4 granos que se añadirán en las celdas de arriba, abajo, derecha e izquierda de la celda. Queda como resultado la matriz de la derecha estabilizada (tomado de Tarea 1 CC3001).

En la segunda parte de la tarea se pide comparar los tiempos obtenidos en ambas tareas para distintos inputs y graficar las diferencias.

Solución del Problema

Inicialmente se implementó una clase Pila (con una lista enlazada) que contaba con los métodos apilar, desapilar y estaVacia. Además de éstas, se implementó un método **tope(),** vista en el apunte, que entrega los índices que se encuentran en el tope de la Pila, sin extraerlos de ésta (ver archivo Pila.java en Anexos). Esto está permitido, pues realiza las funciones desapilar y apilar en un solo método de O(1) al igual que éstos.

Para considerar el caso borde se crea una matriz cuadrada de $\sqrt{n}\times\sqrt{n}$, siendo n la cantidad inicial de granos. Para encontrar los índices a desbordar se procedió de la siguiente manera: se inicia la matriz con N granos en el centro, y los índices del centro en el tope de la Pila:

```
int lado=(int)Math.sqrt(n);
int m[][] = new int[lado][lado];//matriz de granos de arena
m[lado/2][lado/2]=n;//centro de la matriz con n granos de arena
Pila indices=new Pila();//pila donde se guardarán los índices
indices.apilar(lado/2, lado/2);//n granos de arena en la celda central
```

Figura 3. Inicialización de matriz y Pila para poder ejecutar el algoritmo.

Ahora, mientras se encuentren índices en la Pila, éstos deben ser estabilizados mediante la regla de no tener más de 3 granos en una celda:

```
Ti=System.currentTimeMillis();
while (!indices.estaVacia()){
    int[] a=indices.tope();//para obtener los índices
    int x=a[0],y=a[1];
    if (m[x][y]>=4){
        m[x][y] -= 4;
        m[x][y - 1] += 1;
        m[x - 1][y] += 1;
        m[x + 1][y] += 1;
        m[x][y + 1] += 1;
    } else indices.desapilar();
```

Figura 4. En este ciclo se desbordan las celdas que se encuentren en la Pila y de desapilan una vez que el desborde estabilizó ese casillero.

Llegando a este punto, las casillas anexas se apilarán en la Pila bajo la condición de que el valor de esa casilla **sea 4** (no mayor o igual, si se hiciera de esa manera habría muchas repeticiones ya que se apilan los índices que ya fueron descargados):

```
if(m[x+1][y]==4)indices.apilar(x+1,y);
if(m[x-1][y]==4)indices.apilar(x-1,y);
if(m[x][y-1]==4)indices.apilar(x,y-1);
if(m[x][y+1]==4)indices.apilar(x,y+1);
```

Figura 5. Se apilan una sola vez, de este modo se evita apilar muchas veces un mismo casillero de la matriz.

Finalmente se crearon dos variables Tf y Ti que miden el tiempo de ejecución desde que se inicia el programa hasta que termina, con el método *currentTimeMillis()*.

Modo de uso y resultados

Inicialmente (al igual que en la tarea 1) el programa le pedirá al usuario un número entero, cuando éste introduzca la cantidad, se muestra una ventana con la matriz mostrada en colores. Se realizaron algunas pruebas aplicando un input con distintos valores de N y evaluando el resultado entregado en la ventana "Pila de Arena 2" (Figuras 6 a 9):

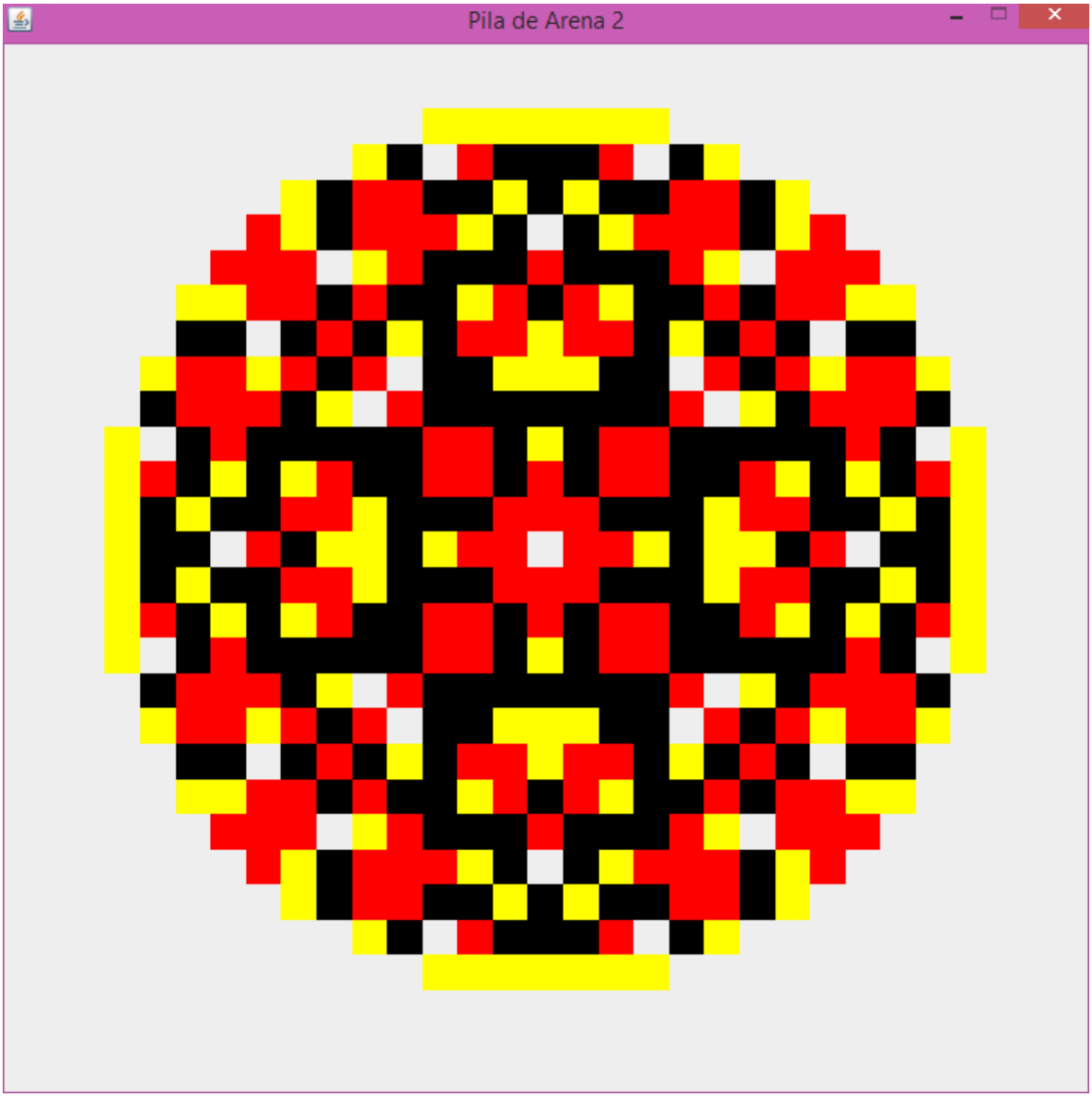

Figura 6: Mosaico con n=1000 granos de arena.

Figura 7. n=34000

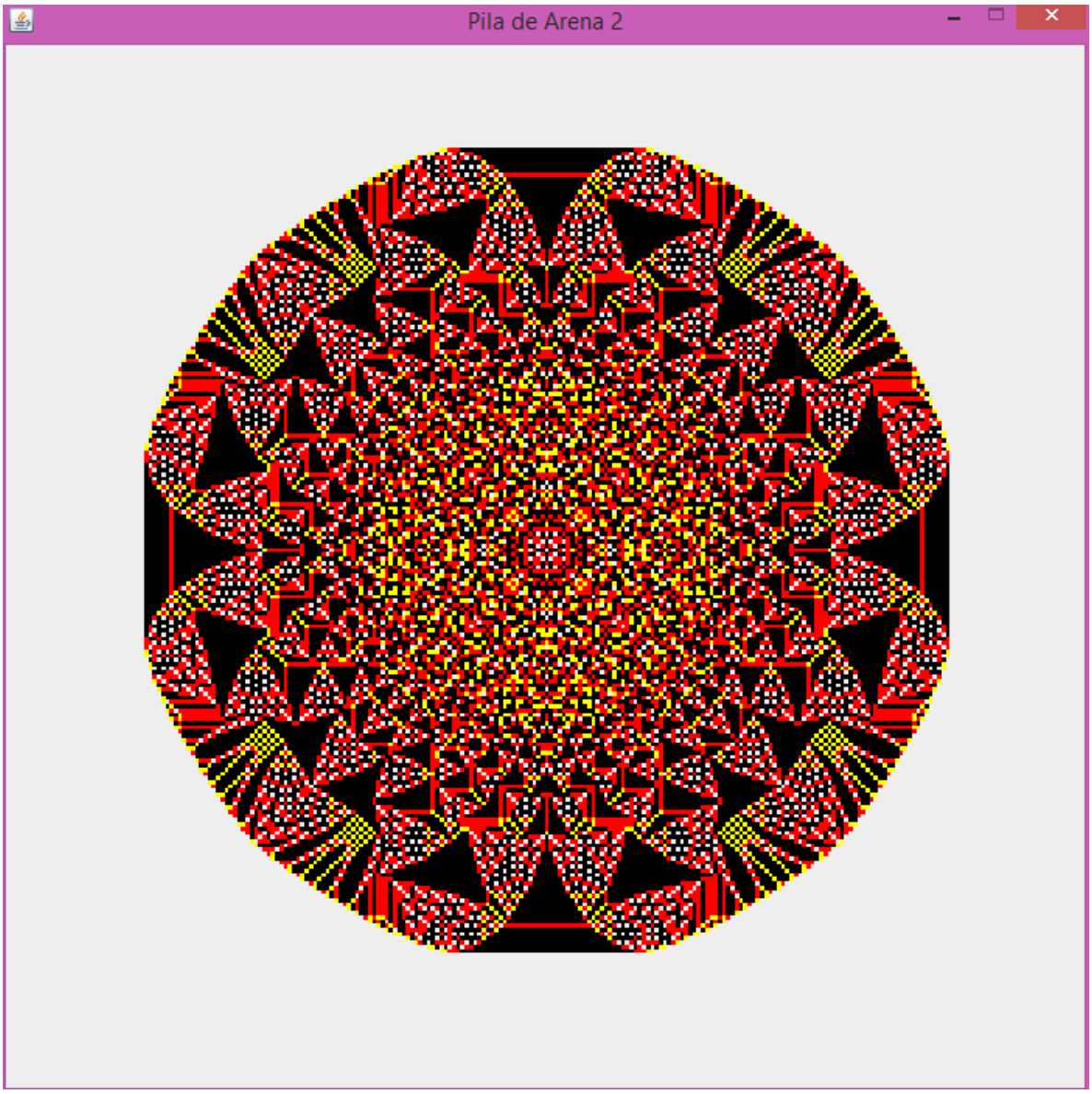

Figura 8. n=67000

Figura 9. n=100000.

Al observar los resultados entregados por el programa, se puede verificar que los resultados obtenido son idénticos a los encontrados por el programa de la primera tarea. La única diferencia significativa se da en los tiempos de ejecución de ambos programas. En la Tabla 1 se encuentran los tiempos de ejecución de ambas tareas para distintos valores de n, puede ver estos datos graficados en el Gráfico 1:

n	tarea 1 [s]	tarea 3 [s]
1000	0,032	0,031
12000	1,281	0,578
23000	3,641	1,75
34000	9,563	3,578
45000	18,531	5,891
56000	29,910	9,001
67000	45,613	12,641
78000	64,667	17,55
89000	81,679	22,811
100000	103,719	27,751

Tabla 1: Tiempos de ejecución (en segundos) al ejecutar los programas con distintos valores de n.

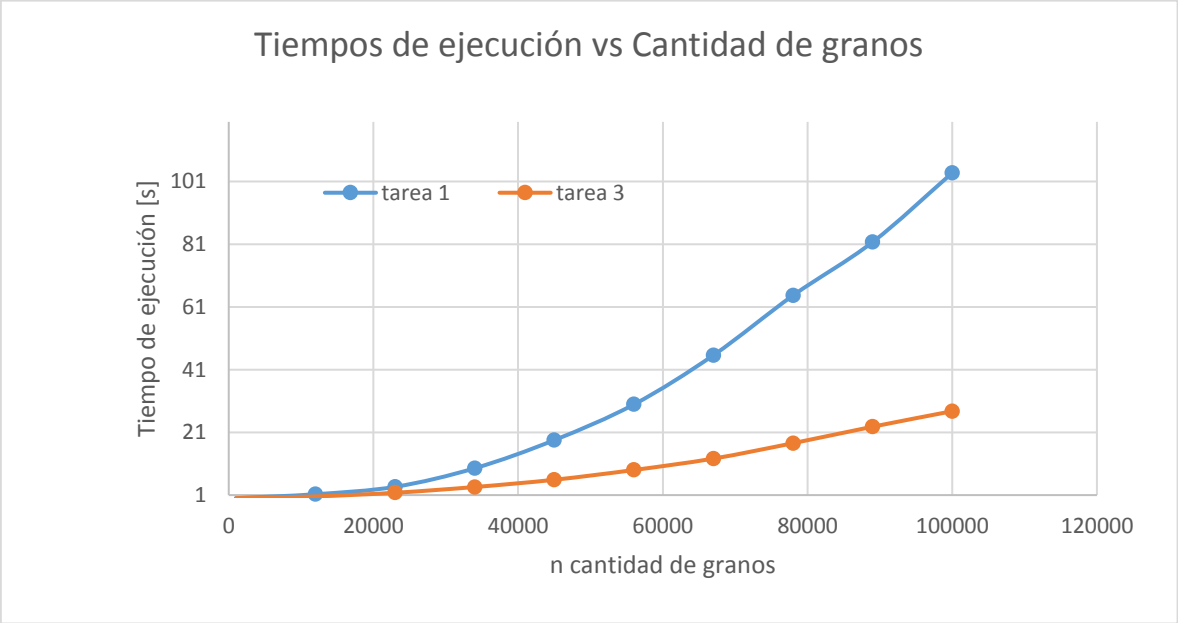

Gráfico 1. Comparación de los tiempos de ejecución entre la tarea 1 y 3. Se puede observar que el programa implementado en la tarea 3 es mucho más eficiente que el de la tarea 1.

Discusión y conclusión

Como se pudo observar en los resultados, la implementación de PilaArenaOptimizada.java resulta mucho más eficiente que PilaArena (se incluye el código fuente en la sección Anexos, y el archivo PilaArena.java en el contenido de esta tarea), por lo que se hace más claro que la implementación del TDA Pila mejora en gran parte la ejecución del programa. Esto se debe a que todos los métodos asociados a la Pila son de tiempo constante (pues trabaja solamente con la referencia al primer elemento y sus métodos no dependen del tamaño de la Pila, es decir, no depende de cuántos elementos tenga ésta).

En la solución iterativa se intentó implementar una búsqueda casillero a casillero de la matriz, lo cual hacía muy ineficiente el algoritmo puesto que realizaba demasiadas pasadas sobre casilleros vacíos innecesariamente. Como se puede ver en los resultados de la comparación entre ambas tareas (Grafico 1 y Tabla 1), los tiempos de ejecución de la tarea 1 aumentan de forma más pronunciada en función del aumento en la cantidad de granos. Es aquí donde se puede apreciar la eficiencia de usar un TDA para optimizar un algoritmo de búsqueda, ya que al ser todos sus métodos O(1) es tiempo constante en comparación a la tarea de 1.

Se puede concluir que la implementación de una estructura de datos puede solucionar de una manera más eficiente un algoritmo de búsqueda, ya que el uso de ésta disminuye significativamente el tiempo de búsqueda acotando la cantidad de desbordes a las casillas presentes en la Pila. Los TDA resultan ser entonces muy eficientes para un algoritmo de búsqueda como lo es el de la presente tarea.

Anexos

Pila.java

```
package main;
import java.util.NoSuchElementException;

class Pila{
    class Nodo {
        int[]info=new int[2];
        Nodo sgte;
        }
    private Nodo tope;
    Pila(){
        tope=null;
    }
    boolean estaVacia() {
        return tope==null;
    }
    int[] tope() {
        if (!estaVacia()) return tope.info;        // si esta vacia es un
error
        else return null;
    }
    void apilar(int i, int j){
        int[]a=new int[2];
        a[0]=i;
        a[1]=j;
        Nodo nuevo=new Nodo();
        nuevo.info=a;
        if(tope==null){
            nuevo.sgte=null;
            tope=nuevo;
        }else{
            Nodo aux=new Nodo();
            aux.info=a;
            aux.sgte=tope;
            tope=aux;
        }
    }
    int[] desapilar(){
        if (estaVacia()) throw new NoSuchElementException("Pila vacía");
        int[] aux=tope.info;
        tope=tope.sgte;
        return aux;
    }

}
```

PilaArenaOptimizada.java

```
package main;
import java.util.Scanner;
public class PilaArenaOptimizada {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Introduzca un número entero: ");
        int n = sc.nextInt();long Ti,Tf;
        int lado=(int)Math.sqrt(n);
        int m[][] = new int[lado][lado];//matriz de granos de arena
        m[lado/2][lado/2]=n;//centro de la matriz con n granos de arena
        Pila indices=new Pila();//pila donde se guardarán los índices
        indices.apilar(lado/2, lado/2);//n granos de arena en la celda
central
        Ti=System.currentTimeMillis();
        while (!indices.estaVacia()){
            int[] a=indices.tope();//para obtener los índices
            int x=a[0],y=a[1];
            if (m[x][y]>=4){
                m[x][y] -= 4;
                m[x][y - 1] += 1;
                m[x - 1][y] += 1;
                m[x + 1][y] += 1;
                m[x][y + 1] += 1;
            } else indices.desapilar();
            if(m[x+1][y]==4)indices.apilar(x+1,y);
            if(m[x-1][y]==4)indices.apilar(x-1,y);
            if(m[x][y-1]==4)indices.apilar(x,y-1);
            if(m[x][y+1]==4)indices.apilar(x,y+1);
        }
        Tf=System.currentTimeMillis();
        System.out.println(Tf-Ti);
        Ventana v = new Ventana(700, "Pila de Arena 2");
        v.mostrarMatriz(m);
    }
}
```

PilaArena.java

```
package main;
import java.util.Scanner;
public class PilaArena {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n; long Ti,Tf;
        System.out.println("Introduzca un número entero: ");
        n = sc.nextInt();
        int a = (int) Math.sqrt(n) + 1;//Se define un tamaño minimo que
debe tener la matriz
        int m[][] = new int[a][a];
        int pm = (a - 1) / 2;
        m[pm][pm] = n;//se inicializa la matriz con el dato n en la celda
central
        Ti=System.currentTimeMillis();
        for(int i=0;i<n;i++) {//ciclo que repite el recorrido n veces
            int j=0;
            while (j < a) {//ciclos de recorrido de la matriz
                int k=0;
                while (k < a) {
                    if (m[k][j] >= 4) {//condición de estabilidad
                        m[k][j] -= 4;
                        m[k][j - 1] += 1;
                        m[k - 1][j] += 1;
                        m[k + 1][j] += 1;
                        m[k][j + 1] += 1;
                    }
                 k++;
                }
             j++;
            }
        }
        Tf=System.currentTimeMillis();
        System.out.println(Tf-Ti);
        Ventana v = new Ventana(700, "Pilas de Arena");
        v.mostrarMatriz(m);
    }
}
```