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

Algoritmos y Estructuras de Datos

CC3001

Tarea 3

Pila de Arena Optimizada

Autor: Ilana Mergudich Thal
E-Mail: ilanamergudich@gmail.com

Fecha de entrega: 8 de mayo de 2017

Índice de Contenidos

1. Introducción **1**

2. Análisis del problema **2**

3. Solución del problema **4**
3.1. Clases 4
3.1.1. Pila 4
3.1.2. PilaArenaOptimizada 5
3.1.3. Ventana 8

4. Modo de uso **9**

5. Resultados **10**
5.1. Ejemplos de entradas y salidas 10
5.2. Resultados de la experimentación 11

6. Discusión **13**
6.1. Métodos del TDA Pila 13
6.2. Comparación con Tarea 1 13

7. Anexos **14**
7.1. Pila 14
7.2. PilaArenaOptimizada 15
7.2.1. desborde 15
7.2.2. main 15

1. Introducción

En esta tarea, el problema consiste en modelar el fenómeno de granos de arena apilados en el espacio y su comportamiento. Para ello, se discretizará una superficie en celdas y se representará como matriz, se definirán condiciones de inestabilidad en que los granos de arena "caen" a las celdas vecinas y se representará la matriz resultante con colores según la cantidad de granos de arena en cada celda. Para ello, se creará una clase **Pila** en base a una lista enlazada, que guarde elementos del tipo **int[2]**. Esta clase permitirá apilar y desapilar elementos de la forma LIFO (last in first out). Luego se creará la clase **PilaArenaOptimizada** donde se creará una pila en la que se guarden las posiciones de las celdas que deben ser desbordadas. Tras cada desborde, se apilarán las nuevas celdas que deban ser desbordadas y de esta forma se desbordarán las celdas de la pila hasta que esta quede vacía. Finalmente se visualizará la matriz resultante con un código de colores.

2. Análisis del problema

El problema consiste en el modelamiento del fenómeno de granos de arena apilados en el espacio. Esto significa distribuir una cierta cantidad de granos de arena en una superficie plana (potencialmente infinita). Inicialmente, se depositan n granos en el centro de la superficie, discretizada en celdas cuadradas de igual tamaño, como se muestra en la Figura 1.

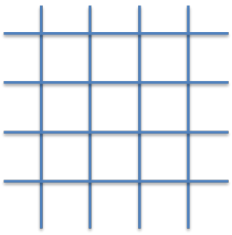

Figura 2.1: Superficie discretizada

Luego, si en alguna celda hay 4 o más granos de arena, se quitarán 4 granos de esta celda y se repartirán entre las celdas que la rodean. Este proceso debe repetirse hasta que no hayan 4 o más granos en ninguna celda. La Figura 2 representa un ejemplo de este procedimiento.

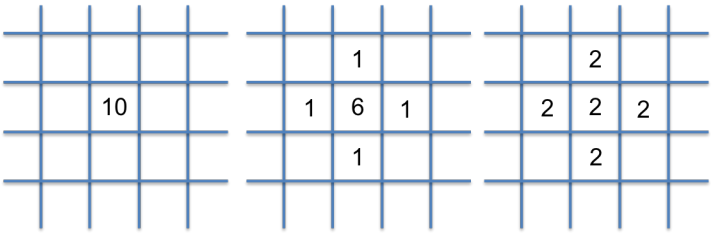

Figura 2.2: Ejemplo del procedimiento

Finalmente, se deberá visualizar la superficie resultante mediante la asignación de colores a la cantidad de granos por celda. Para realizar esta parte, se hará uso de la clase **Ventana** facilitada por el equipo docente.

Es importante mencionar que el orden en que se aplique la regla para diferentes celdas inestables (4 o más granos) no afecta el resultado final.

Para solucionar este problema se creó el método **main** en que se crea una Pila donde se guardan los índices de las celdas que deben ser desbordadas, utilizando el método **desborde**. Inicialmente se apila la celda central de la matriz, en la que se depositan los granos de arena. Luego se desborda esta posición y se revisa si es necesario apilarla nuevamente así como apilar las celdas vecinas en caso de que éstas alcancen 4 granos. Este procedimiento se repite desbordando todas las celdas que están en la Pila, hasta que ésta quede vacía.

Dado que en este modelamiento la superficie no es infinita, se produce un problema cuando celdas de los bordes son inestables, ya que éstas no tienen cuatro vecinos. Para solucionar esto,

el método **main** crea la matriz de tamaño $\sqrt{N} + 2 \times \sqrt{N} + 2$[1] dependiendo de la cantidad de granos ingresados (N). De esta forma, el tamaño de la matriz es suficiente como para que nunca sea necesario desbordar los bordes.

[1] A pesar de que en la tarea se indicaba usar una matriz de tamaño $\sqrt{N} \times \sqrt{N}$, esto fallaba para N muy pequeño, por lo que se sumó 2.

3. Solución del problema

3.1. Clases

3.1.1. Pila

Esta clase define una pila según el tamaño ("size") y último elemento de una lista enlazada. Para ello se creó una clase privada dentro de **Pila** llamada **Nodo**, donde se define un "elemento" del tipo **int[2]** (donde se guardarán las posiciones de la matriz) y un "siguiente" del tipo **Nodo**. Luego, "size" es un **int** que representa la cantidad de elementos de la lista y "ultimo" es un puntero al último **Nodo** de la lista. Además se definen diversas operaciones que se explicarán a continuación, todas ellas se rigen según el principio "LIFO" (last in first out).

3.1.1.1 Pila()
Esta operación es un constructor, se define el último como *null* y el tamaño como 0.

3.1.1.2 boolean estaVacia()
Esta operación verifica si la pila esta vacía, para ello se revisa el tamaño de la pila: si es 0, se retorna *true*, en caso contrario se retorna *false*.

3.1.1.3 void apilar(int i, int j)
Esta operación agrega el arreglo $\{i, j\}$ a la pila. Para ello, primero se crea el arreglo con estos valores. Luego existen dos casos:

1. Si la pila esta vacía, se crea un nuevo **Nodo** con donde el "elemento" es el arreglo $\{i, j\}$ y "siguiente" es *null*. Este **Nodo** será el "último".

2. Se crea un **Nodo** auxiliar, que es igual al "último" actual. Luego se modifica el último de manera que el "elemento" sea el arreglo $\{i, j\}$ y "siguiente" sea el **Nodo** auxiliar.

Sin importar el caso, se suma uno al tamaño.

3.1.1.4 int[ ] desapilar()

Esta operación quita el último elemento de la pila y retorna ese arreglo. Si la pila esta vacía, simplemente retorna *null*. En caso contrario se define un **int[ ]** x como el último elemento de la pila. Nuevamente tenemos dos casos:

1. Si la pila tenía sólo un elemento (size = 1), se define "último" como *null*.

2. Si la pila tenía más de un elemento, el nuevo "último" es el siguiente del "último", es decir, el penúltimo.

En ambos casos se resta uno al tamaño de la pila.

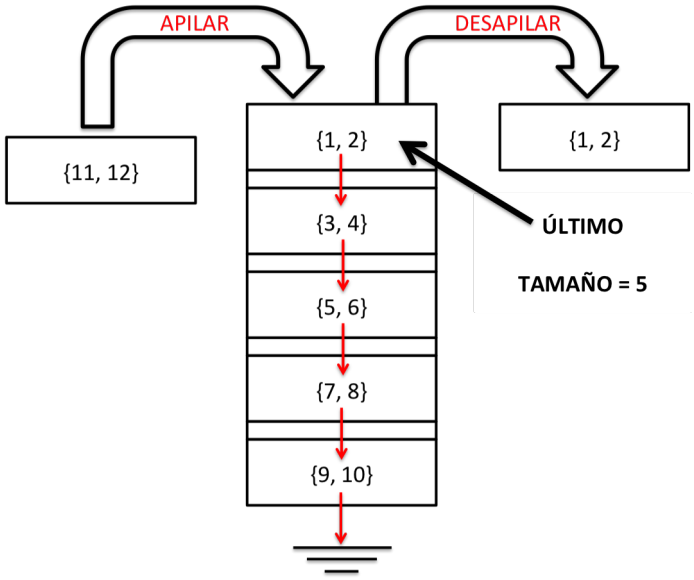

Figura 3.1: Representación de una Pila

En la Figura 3.1 se observa una representación de el **TDA Pila** definido en base a una lista enlazada. Las flechas rojas representan el "siguiente" de la clase **Nodo**, se puede notar que el elemento de más abajo apunta a *null*. "Último" representa el de más arriba de la pila, es decir el último que fue apilado. Así mismo, al desapilar se obtiene el de más arriba. El código completo de esta clase se encuentra en el Anexo 7.1.

3.1.2. PilaArenaOptimizada

En esta clase se lleva a cabo el proceso descrito anteriormente para lograr que todas las celdas queden con menos de 4 granos. Para ello, se crearon los métodos **desborde** y **main** que será explicados a continuación.

3.1.2.1 desborde

Este método recibe una matriz (int[ ][ ]) y una posición dentro de ella, representada por dos enteros i, j. Primero se restan 4 granos a la celda (posición) i, j indicada y luego se suma un grano a cada celda que la rodea.

Ejemplo de uso

Sea

$$A = \begin{pmatrix} 3 & 1 & 1 \\ 0 & 5 & 2 \\ 1 & 2 & 3 \end{pmatrix}$$

si utilizamos desborde(A, 2, 2), se retorna la siguiente matriz

$$A' = \begin{pmatrix} 3 & 2 & 1 \\ 1 & 1 & 3 \\ 1 & 3 & 3 \end{pmatrix}$$

Es posible observar que en la celda (2, 2) se restó 4 y se sumó 1 a las celdas (1,2), (3,2), (2,1) y (2,3). El código completo de este método se encuentra en el Anexo 7.2.1.

3.1.2.2 main

En primer lugar, este método le pregunta al usuario cuántos granos de arena desea en el centro de la superficie y guarda el valor N entregado. Luego, crea una matriz de tamaño $\sqrt{N}+2 \times \sqrt{N}+2$ y asigna el valor N a la celda de el centro. Dado que no se asignan valores a las demás celdas, *Java* les asigna valor 0.

Luego se crea una **Pila** (que llamaremos *casillas*) y se apila la posición del centro de la matriz. A continuación se entra en un *while* que termina sólo cuando *casillas* está vacía. En este, se desapila el último elemento y se desborda la posición de la matriz que éste indique. Entonces se verifica si tras el desborde, esa casilla sigue teniendo 4 o más granos. Si es el caso, se apila nuevamente. Por otro lado se revisa si tras el desborde las casillas vecinas quedaron con exactamente 4 granos, si es el caso, se apilan. Se define con exactamente 4 granos y no 4 o más debido a que si hay más de 4, esas casillas ya deberían estar en *casillas* debido a que ya cumplían la condición de desborde.

Finalmente se utiliza la clase **Ventana** para visualizar la matriz resultante, donde diferentes colores indican si cada casilla tiene 0, 1, 2 o 3 granos.

Ejemplo de uso

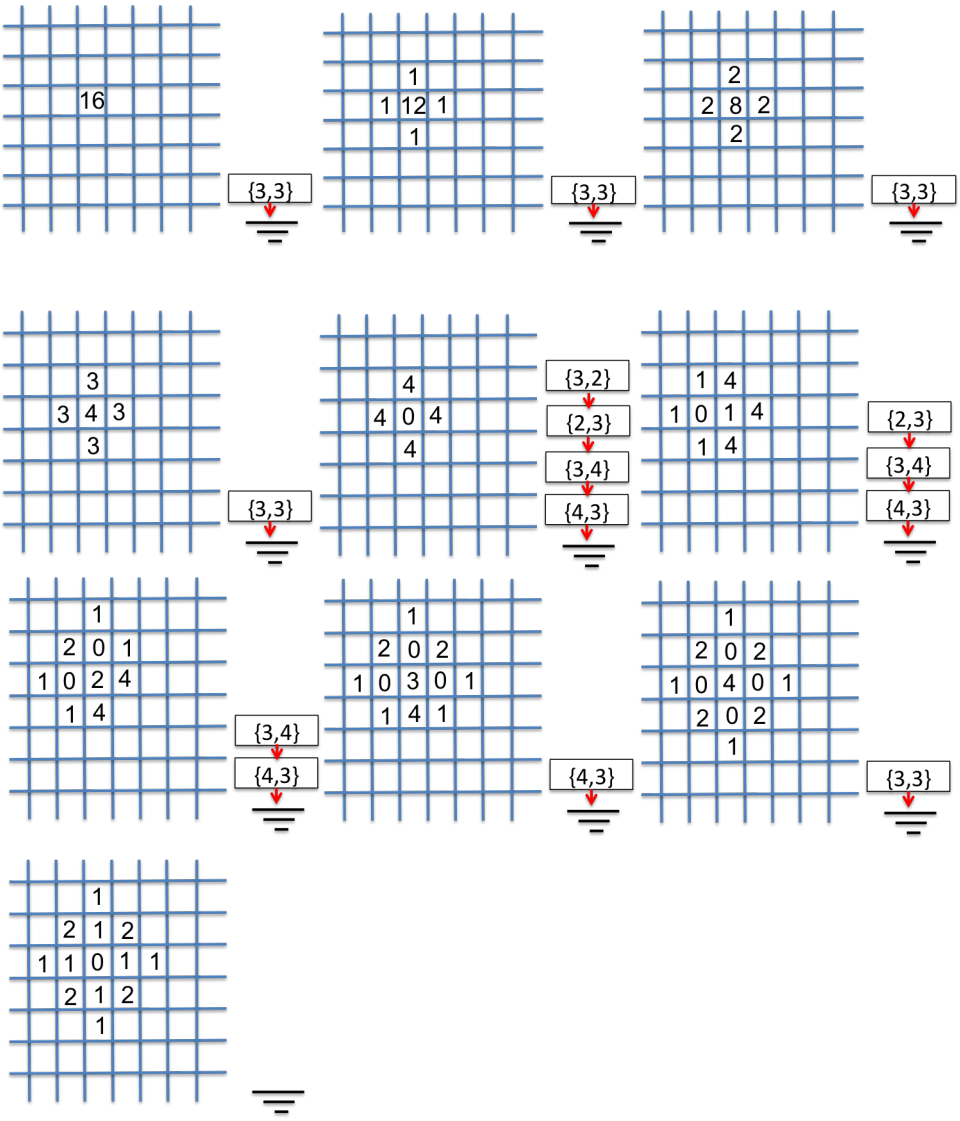

Figura 3.2: Funcionamiento de PilaArenaOptimizada

En la figura 3.2 se aprecia un ejemplo del funcionamiento de la clase **PilaArenaOptimizada** para $N = 16$. En este caso, se crea una matriz de 6×6 y se apila la posición del centro. Luego se desborda este casilla y se apila nuevamente. Esto se repite dos veces más hasta llegar a la quinta matriz que se observa. En esta, todos los vecinos llegan a 4 granos, por lo que sus posiciones son apiladas. Se van desapilado y desbordando en orden (las de más arriba primero). Al terminar de desbordarse los vecinos, la celda central vuelve a tener 4 granos, por lo que es apilada nuevamente. Tras desbordarse no quedan celdas con 4 o más granos por lo que la pila queda vacía. Se puede notar que, por ejemplo, al pasar de la séptima a la octava matriz, no se apila nuevamente la posición $\{4, 3\}$ ya que sólo se apilan los vecinos de la celda desbordada que en ese desborde alcanzaron los 4 granos. El código completo de este método se encuentra en el Anexo 7.2.2.

3.1.3. Ventana

Esta clase fue facilitada por el equipo docente. Permite crear y mostrar una ventana que muestra la matriz resultante del proceso descrito anteriormente, donde asigna un color a cada número del 0 al 3. La única intervención a esta clase, es un cambio en la asignación de colores a cada número.

Modo de uso:

- Ventana(int tamaño, String titulo): *"Constructor, crea y muestra una nueva ventana vacía cuadrada del tamaño y título especificados como parámetros."*
- mostrarMatriz(int[ ][ ] mat): *"Dibuja en una ventana la matriz indicada como cuadros de colores."*

4. Modo de uso

Para utilizar este programa, se deben guardar los archivos **Pila.java**, **PilaArenaOptimizada.java** y **Ventana.java** en una misma carpeta. A continuación, se deben compilar las tres clases y luego ejecutar **PilaArenaOptimizada.java**. Al hacer esto, se preguntará al usuario "Cuantos granos de arena desea en el centro?". Se deberá ingresar un número entero mayor o igual a 4. Entonces el programa mostrará una ventana con la matriz que representa el estado final de la superficie.

5. Resultados

5.1. Ejemplos de entradas y salidas

A continuación se observan ejemplos de entradas y salidas del programa. En cada figura se indica la cantidad de granos ingresados (N) y el tiempo en milisegundos que transcurrió entre el momento en que se ingresó el N y se mostró la figura. Cabe destacar que las figuras 5.1 y 5.2 están a una escala de 0.4 mientras que la figura 5.3 está a a una escala de 0.3.

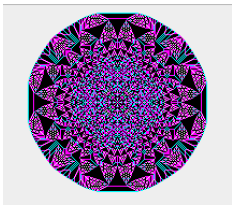

Figura 5.1:
$N = 50\ 000$
$t = 2601$ ms

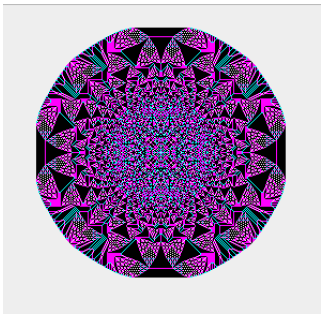

Figura 5.2:
$N = 100\ 000$
$t = 6904$ ms

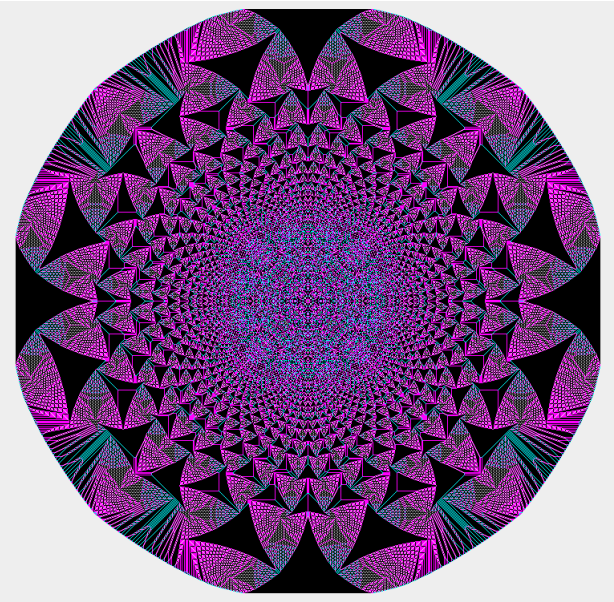

Figura 5.3:
$N = 1\ 000\ 000$
$t = 608026$ ms

5.2. Resultados de la experimentación

Para evaluar la eficiencia del **TDA Pila**, se compraron los tiempos de implementación de este programa con el realizado en la Tarea 1 para 10 valores equiespaciados de N entre 1 000 y 100 000. Para ello hubo dos consideraciones importantes. En primer lugar, la Tarea 1 fue programada con un borde sumidero en vez de que el tamaño de la matriz dependiera de N. Para poder comparar correctamente, la Tarea 1 fue modificada de tal forma que la matriz se creara según el N, al igual que esta. En segundo lugar, si se mide el tiempo total de implementación, éste depende en gran medida del tiempo que le toma al usuario ingresar la cantidad de granos. dado esto, el tiempo inicial fue definido después de que el usuario ingresa el valor de N.

A continuación se presenta una tabla y gráficos que muestran los resultados obtenidos de esta experimentación:

N	Tiempo Programa Original [ms]	Tiempo Programa Optimizado [ms]
1 000	855	874
12 000	1161	910
23 000	1915	1280
34 000	3211	1710
45 000	5039	2235
56 000	7409	2866
67 000	10300	3699
78 000	13951	4689
89 000	17918	5840
100 000	22575	7054

Tabla 5.1: Tiempos de ejecución para los diferentes programas

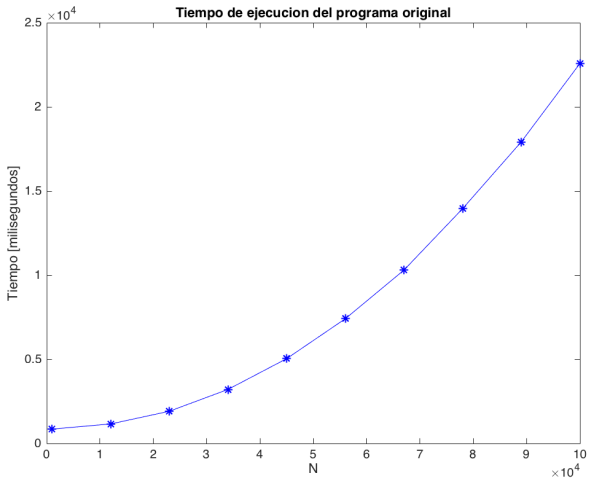

Figura 5.4: Tiempo de implementación del Programa Original

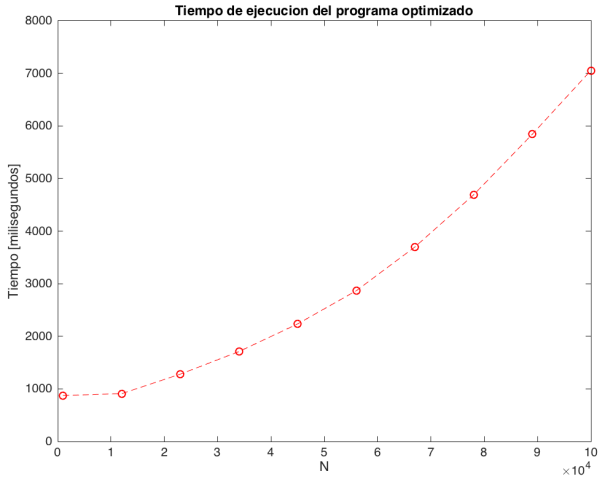

Figura 5.5: Tiempo de implementación del Programa Optimizado

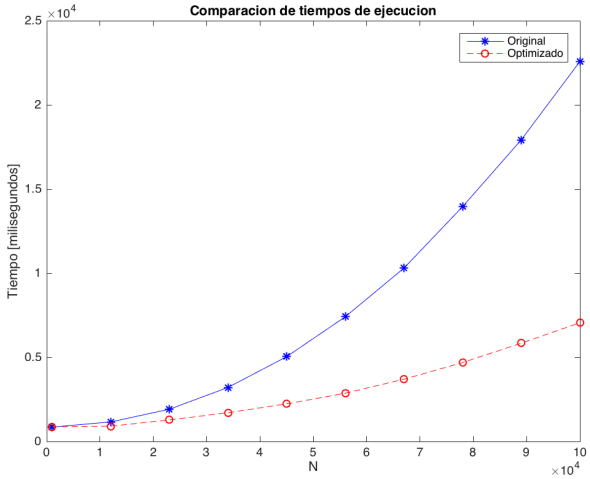

Figura 5.6: Comparación de los tiempos de implementación

6. Discusión

6.1. Métodos del TDA Pila

A partir de esta tarea podemos concluir que el **TDA Pila** es una muy buena herramienta ya que sus métodos son muy sencillos y sus costos son de tiempos constantes, es decir, de $O(1)$.

- Pila(): Este método es de $O(1)$ ya que sólo asigna 2 valores, es decir, toma un tiempo $2 = O(1)$.
- estaVacia(): Este método sólo realiza una comparación, es decir, toma un tiempo $1 = O(1)$.
- apilar(int i, int j): A pesar de que en este método existen diferente casos, simplemente se verifica si la pila esta vacía o no (ya vimos que es de $O(1)$) y se asignan valores a 2 variables, por lo que el tiempo sigue siendo constante.
- desapilar(): Este método también verifica si la pila esta vacía ($O(1)$) o si su tamaño es 1 ($O(1)$). Además de eso sólo asigna valores a 3 variables, por lo que sigue siendo de orden 1.

6.2. Comparación con Tarea 1

En el gráfico de la figura 5.6 se observa claramente que el uso del **TDA Pila** optimizó el programa de manera significativa, especialmente para valores de N grandes. Para una cantidad de granos cercana a 1000, los tiempos de implementación son casi idénticos, no obstante, a medida que aumenta este número, los tiempos se diferencian cada vez más hasta alcanzar diferencias de aproximadamente 15 segundos en el caso de $N = 100000$. Si proyectamos estos gráficos para valores de N aún mayores, las diferencias se vuelven cada vez más relevantes.

Esto se debe a que en la Tarea 1, se recorría la matriz completa cada vez que se encontraba una casilla que cumplía con la condición de desborde, por lo que para N granos de arena, se recorría la matriz completa, lo que toma un tiempo de orden $\sqrt{N} + 2 \times \sqrt{N} + 2 \sim N$ cada vez que se recorre. Digamos que para N granos se producen M desbordes, luego la matriz se recorría $M + 1$ veces, lo que tomaba un tiempo de orden $N \times M$. Por otro lado, en esta tarea, se apilan sólo las casillas que deben ser desbordadas, y además ya vimos que los métodos del **TDA Pila** son de orden 1, por lo que el tiempo total es de orden M, es decir, la Tarea 1 tomaba N veces más tiempo en implementarse. Esto calza con lo observado en los gráficos ya que al aumentar el N, mayor es la diferencia.

7. Anexos

7.1. Pila

```
1  public class Pila {
2
3      private Nodo ultimo;
4      private int size;
5
6      private class Nodo{
7          int[] elemento;
8          Nodo siguiente;
9
10     }
11
12     public Pila() {
13         ultimo = null;
14         size = 0; //inicialmente la pila está vacía
15
16     }
17
18
19     public boolean estaVacia() {
20         return size == 0;
21
22     }
23
24
25     public void apilar(int i, int j) {
26         int[] a = new int[2];
27         a[0] = i;
28         a[1] = j;
29         if(estaVacia()){
30             ultimo = new Nodo();
31             ultimo.elemento = a;
32             ultimo.siguiente = null;
33
34         }
35         else{
36             Nodo aux = ultimo;
37             ultimo = new Nodo();
38             ultimo.elemento = a;
39             ultimo.siguiente = aux;
40         }
41
42         size = size+1;
43     }
44
45     public int[] desapilar() {
46         if (!estaVacia()) {
47             int[] x = ultimo.elemento;
```

```
48              if(size == 1){
49                  ultimo = null;
50              }
51              else{
52                  ultimo = ultimo.siguiente;
53              }
54
55              size = size -1;
56              return x;
57
58          }
59          else{
60              return null;
61          }
62      }
63  }
```

7.2. PilaArenaOptimizada

7.2.1. desborde

```
1   static int[][] desborde(int[][] matriz, int i, int j){ //método que toma 4 granos de una
        celda y los reparte en las que lo rodean
2
3           matriz[i][j] -= 4; //se restan 4 granos a la celda
4           matriz[i+1][j]++; //se suma 1 grano a la celda de abajo,
5           matriz[i][j+1]++; //se suma 1 grano a la celda de la derecha,
6           matriz[i-1][j]++; //se suma 1 grano a la celda de arriba,
7           matriz[i][j-1]++; //se suma 1 grano a la celda de izquierda,
8
9           return matriz;
10      }
```

7.2.2. main

```
1   static public void main(String[] args) {
2
3           Scanner s = new Scanner(System.in);
4           System.out.print("Cuantos granos de arena desea en el centro? "); //se pregunta
                la cantidad inicial de granos de arena
5
6           int N = s.nextInt(); //se ingresa la cantidad de granos N
7
8           //long ti = System.currentTimeMillis();
9
10          double dimaux = Math.sqrt(N);
11          int dim = (int) dimaux + 2; //define la dimension de la matriz segun el N
12          int centro = dim/2; //define el centro de la matriz
13
14          int[][] matriz = new int[dim][dim]; //se crea la "superficie"
15          matriz[centro][centro] = N; //se depositan los N granos al centro
16
```

```
17        Pila casillas = new Pila(); //se crea la pila de casillas por desbordar
18        casillas.apilar(centro, centro); //se apila el centro
19
20        while(!casillas.estaVacia()){ //mientras la pila no este vacia, se desbordan las
              casillas guardadas
21            int [] indices = casillas.desapilar(); //se desapila la casilla de mas arriba
22            desborde(matriz, indices[0], indices[1]); //se desborda esa casilla
23
24            if (matriz[indices[0]][indices[1]] >= 4){ //si a pesar de ser desbordada sigue
                  teniendo mas de 4 granos
25                casillas.apilar(indices[0], indices[1]); //se apila de nuevo
26            }
27
28            int[] a = new int[2];
29            a[0] = indices[0]+1;
30            a[1] = indices[1];
31            if(matriz[a[0]][a[1]] == 4){ //se revisa si la casilla de abajo necesita ser
                  apilada
32                casillas.apilar(a[0], a[1]);
33            }
34
35            int[] b = new int[2];
36            b[0] = indices[0];
37            b[1] = indices[1]+1;
38            if(matriz[b[0]][b[1]] == 4){ //se revisa si la casilla de la derecha necesita
                  ser apilada
39                casillas.apilar(b[0], b[1]);
40            }
41
42            int[] c = new int[2];
43            c[0] = indices[0]-1;
44            c[1] = indices[1];
45            if(matriz[c[0]][c[1]] == 4){ //se revisa si la casilla de arriba necesita ser
                  apilada
46                casillas.apilar(c[0], c[1]);
47            }
48            int[] d = new int[2];
49            d[0] = indices[0];
50            d[1] = indices[1]-1;
51            if(matriz[d[0]][d[1]] == 4){//se revisa si la casilla de la izquierda necesita
                  ser apilada
52                casillas.apilar(d[0], d[1]);
53            }
54        }
55        Ventana ventana = new Ventana(dim, "Figura"); //se crea la ventana
56        ventana.mostrarMatriz(matriz); //se muestra la ventana
57
58        //long tf = System.currentTimeMillis();
59        //long t = tf - ti;
60        //System.out.println(t);
61    }
```