Capítulo 1

Jonathan Zinzan Salisbury Vega
Universitat de les Illes Balears
Palma, España
jonathan.salisbury1@estudiant.uib.cat

Joan Sansó Pericás
Universitat de les Illes Balears
Palma, España
joan.sanso4@estudiant.uib.cat

Joan Vilella Candia
Universitat de les Illes Balears
Palma, España
joan.vilella1@estudiant.uib.cat

El coste asintótico de los algoritmos ha sido uno de los principales focos de estudio (y de problemas) a lo largo de la historia de la informática. Las soluciones más sencillas suelen tener costes asintóticos no deseables dependiendo del problema a resolver. Además, encontrar soluciones óptimas suele traer verdaderos quebraderos de cabeza a los distintos programadores. En este proyecto se ha implementado una aplicación capaz de visualizar los distintos tiempos que tienen estos algoritmos. Para ello se ha utilizado el Modelo Vista Controlador. Utilizar esta arquitectura permite un mayor control de errores del programa y facilidad de reutilización como de escalado. Todo esto a coste de una mayor complejidad de código. Con esta práctica se busca poner en práctica la arquitectura MVC y ofrecer una herramienta al programador para poder discernir de una manera gráfica con la que estudiar la viabilidad de los distintos algoritmos.

Keywords—MVC, rendimiento, coste asintótico, algoritmo.

I. Introducción

Una de las máximas que ha de tener un programador a la hora de diseñar sus algoritmos, es el coste asintótico de los mismos. Muchas veces, por un simple descuido o por desconocimiento sobre la materia, se acaban implementando soluciones muy alejadas de lo que podríamos considerar como aceptable.

La herramienta que se ha desarrollado permite visualizar de una manera más fácil el coste asintótico de los distintos órdenes de complejidad. Esto permite al usuario hacerse una idea de la viabilidad de sus soluciones, no necesariamente para que sean las óptimas, pero sí para que se puedan realizar en un tiempo aceptable.

El segundo objetivo de la práctica ha sido la puesta en práctica del MVC (Modelo Vista Controlador) visto en clase. La estructura del documento será la siguiente, en primer lugar, se describe la implementación del modelo MVC y después se comentan los resultados de los distintos costes asintóticos.

II. Modelo vista controlador

A. *Contexto y entorno del estudio*

Uno de los requisitos a la hora de realizar la práctica es el uso del lenguaje de programación Java. Además, se ha dado la opción de elegir entre dos IDE (*Integrated Development Environment).*

- NetBeans
- IntelliJ

En este caso se ha escogido el IDE de NetBeans por familiaridad de uso. Además, se utiliza una herramienta de control de versiones (Git). Más específicamente su versión de escritorio *Github Desktop* por su facilidad de uso mediante interfaz gráfica.

B. *Diseño de la solución*

Como ya se ha mencionado en la introducción, para realizar esta práctica se ha utilizado el patrón de arquitectura de software MVC.

Esta técnica consiste en separar los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos.

- Modelo: En esta capa es en la que se trabaja con los datos, por lo que contendrá los mecanismos para acceder a la información y también actualizar su estado. Además, contiene toda la lógica del problema,
- Vista: Contiene el código que muestra la aplicación, es decir, que va a producir la visualización de la interfaz del usuario como de los resultados.
- Controlador: Es la parte que actúa como intermediario entre el modelo y la vista, gestionando el flujo de información entre ellos.

Las ventajas principales del MVC son: Escalabilidad, facilidad de tratamiento de errores y reutilización de componentes. Existen otras ventajas, pero esta arquitectura se aprovecha en mayor medida en aplicaciones web, el cual no es este caso.

La principal desventaja que existe del patrón MVC es la complejidad que añade a la programación. Ya que, para el mismo problema, hay que modificar el acercamiento para que quepa dentro de esta arquitectura. Lo que implica una mayor complejidad.

III. Descripción del controlador

Esta es la clase encargada de la interacción entre las clases de la vista y del modelo. Además, se encarga de gestionar los inputs generados por el usuario.

Es por esta razón que hay dos instancias constantes de dichas clases, las cuales se le pasarán por parámetro al constructor del controlador. Además, la clase tiene dos variables: Un *String* que indicará la complejidad a calcular y una *Task* que se encargará que, mediante ese *String*, se ejecute en segundo plano.

A. *Start()*

Desde la clase *main*, es el primer método que se llama una vez se han instanciado las distintas clases: Modelo, vista y controlador. Por lo que consideramos de relevancia comenzar a comentar el programa desde aquí.

Este método configura los distintos "*Listeners*[1]" para la vista y la pone visible a la espera del input del usuario. En la implementación hay definidos un total de tres *listeners* que son:

- *AnimateListener*
- *StopListener*
- *ClearListener*

Los cuales corresponden con los tres botones que hay definidos en la interfaz de usuario. Cada uno de los "Listeners" es una clase anidada, que, al no tener suficiente entidad, la declaramos dentro del controlador. Aunque a grandes rasgos cada "*Listener*" se encarga de gestionar las acciones de dichos botones, a continuación, se explican sus particularidades:

B. *AnimateListener*

Esta clase anidada implementa la interfaz "*ActionListener*". Al inicio, indica a la vista que la simulación ha comenzado y a continuación recoge mediante un *get* sobre la vista la opción que ha indicado el usuario en la interfaz (tipo de complejidad).

C. *StopListener*

Esta clase se encarga de manejar la interacción con el botón de Stop. En primer lugar, envía la señal de cancelación al *SwingWorker* (explicación más adelante). Además, indica a la vista que la

[1] Un *listener* es un método que es llamado automáticamente cuando el usuario realiza alguna acción. El método recibe un objeto *event* como parámetro que le indica al método que ha pasado exactamente.

simulación ha parado. Finalmente, hace el set correspondiente a la vista para poner la barra de progreso a 0.

D. *ClearListener*

Este *listener* maneja la interacción con el botón "*Clear*". Simplemente envía la orden a la vista para limpiar el panel.

E. *ProgressListener*

Esta clase se encarga de gestionar el avance de la simulación mediante las actualizaciones que recibe de la tarea por parte del modelo. Cuando se notifica de un cambio desde el modelo, este viene con el nombre de la propiedad, dependiendo de este se actúa de una forma distina

- *Progress:* se obtiene el valor del progreso actual y se le comunica a la vista.
- *Point:* se obtienen las coordenadas del nuevo punto y se pasan a la vista para que realice la animación.
- *State:* se comprueba si la tarea ha finalizado y lo notifica a la vista, escribe el nombre de la complejidad en el último punto.

Estos son todos los "*Listeners*" propios del controlador que se encargan de comunicar el *input* del usuario al modelo, y los avances del modelo a la vista para poder representarlos.

IV. DESCRIPCIÓN DE LA VISTA

Esta es la clase encargada de toda la visualización de la aplicación. Esta clase extiende "JFrame" que servirá para generar las distintas ventanas sobre las cuales añadiremos los distintos objetos con los que podrá interactuar el usuario. En nuestro caso, el usuario sólo podrá interactuar mediante los botones de la interfaz.

Se ha utilizado un *LaF* (*Look and Feel*) distinto al proporcionado por la librería *swing* por defecto, para modernizar la apariencia de la interfaz. Este *LaF* proviene de una librería externa que se ha incluido directamente en el código fuente del proyecto, por lo que no es necesaria ninguna modificación para compilar y ejecutar.Para la realización de la vista había dos opciones posibles: Crear y ajustar los distintos contenedores de Swing a "mano", es decir, ajustar mediante código la posición y el tamaño de los elementos o utilizar el asistente de NetBeans.

En nuestro caso se ha utilizado la herramienta de NetBeans. Ya que permite plasmar la idea que se tenía de la interfaz de una manera más directa y por familiaridad con la herramienta.

Por lo que en esta clase todo el código de la función *addComponents()* está generado por el IDE. Este método es el encargado de crear toda la disposición de la interfaz con todos sus componentes. Hecha esta salvedad, a continuación, se analizará el funcionamiento de esta clase.

Nuestra vista incorpora un total de siete tipos de componentes Swing, en los que se encuentras: JPanel, JComboBox, JButton, JLabel y JProgressBar:

- JPanel: Objetos contenedores que a su vez agrupan otros objetos. Tales como botones, etiquetas, selectores, entre otros.
- JComboBox: Objeto que permite seleccionar Strings de una lista.
- JButton: Una implementación de un botón "pulsador".
- JLabel: Área de visualización para una cadena de texto corta o una imagen, o ambos.
- JProgressBar: Un componente que muestra visualmente el progreso de alguna tarea.

V. DESCRIPCIÓN DEL MODELO

Es la clase encargada del manejo de los datos y la actualización de estos. En este caso, se ha definido una subclase llamada "*Task*" que extiende la clase *SwingWorker* [1].

Las instancias de la clase *Task* serán los que irán "simulando" las distintas complejidades e irán actualizando la barra de progreso a través del método *setProgress* de la propia clase *SwingWorker*, que luego la vista usará para mostrar correctamente la barra de progreso.

A. *SwingWorker*

Es una clase abstracta diseñada para realizar tareas largas que interactúen con la GUI en un proceso en segundo plano y de esta manera no bloquear la interfaz.

Esta clase es la recomendada por el propio lenguaje de programación Java [1] para realizar cálculos largos dentro de una interfaz dejando de esta manera el EDT (Event Dispatch Thread) para los componentes swing.

En este caso se ha utilizado la herencia para implementar la clase Task que utilice estas propiedades para simular las complejidades asintóticas se ha utilizado el sistema de eventos basados en cambios de propiedades para notificar al controlador de la aplicación.

B. *Complejidades*

El modelo nos permite simular cinco complejidades distintas:

- $\log_2(n)$
- $\sqrt{n}$
- n
- $n \cdot \log_2(n)$
- n^2

El nombre de cada complejidad se almacena en una variable estática publica, así como un conjunto de todas ellas que puede utilizarse desde la vista para mostrar las complejidades disponibles. También se almacena un array con los distintos tamaños para simular las complejidades.

El método principal es el doInBackground() del SwingWorker que se ejecuta directamente en un thread diferente al llamar al método execute() sobre la instancia. Este método llama al metodo correspondiente para simular la complejidad seleccionada, por cada N a probar, después de cada iteración notifica al controlador del nuevo punto a animar con el tiempo correspondiente.

Existe un método propio para simular cada complejidad, que recibe la posición del tamaño actual y realiza una espera activa para simularlo. Durante la simulación se almacena el tiempo transcurrido y se notifica al controlador del progreso actual.

C. *Calculo del Progreso*

Para calcular el progreso se divide el porcentaje total (*100%)* en partes iguales entre el número de *N's*. Para cada porción respectiva a una N se calcula el número de iteraciones totales a ejecutar al principio y se va actualizando con el número de iteración actual. Este cálculo quedaría definido por la siguiente formula:

$$P = S \cdot \frac{100}{N} + n \cdot \frac{100}{N}$$

Ecuación 1

Donde P es el porcentaje actual, N la longitud del vector de N's y n la posición actual de dicho vector. El valor S representa el subporcentaje de esta n en concreto y viene dado por la siguiente formula:

$$S = \frac{i}{max}$$

Ecuación 2

Donde *max* representa el número de iteraciones total a realizar en dicha complejidad con el tamaño actual, y la *i* representa la iteración actual, esta se calcula de diferentes maneras dependiendo de la complejidad.

VI. Rendimiento de algoritmos y notación Big O

En el mundo de la informática, se usa la notación Big O para medir el rendimiento espacial y temporal de los algoritmos. Esta notación es un tipo de notación asintótica, es decir, se centra en modelar correctamente el crecimiento del tiempo o espacio usados cuando el tamaño de la entrada crece (normalmente asociado a la variable *n*) sin tener en cuenta otros costes menores.

Por ejemplo, si tenemos una parte del programa que crece respecto a la entrada de forma lineal, diremos que esa parte es O(n). Si, en el mismo programa, tenemos otra parte que crece de forma cuadrática respecto a la entrada (por ejemplo, un algoritmo de ordenación "malo" de un vector con dos bucles *for*), esa parte del programa es $O(n^2)$. Por lo tanto, el programa completo pertenece al máximo de sus partes, ya que la parte que pertenece a O(n), cuando n es significativamente grande, es despreciable respecto a la parte de $O(n^2)$. Por lo tanto, el programa es $O(n^2)$.

El objetivo de todo programador es reducir el coste asintótico de su programa. Normalmente, cuando intentas reducir el coste **temporal** de tu programa, lo haces a expensas del coste **espacial**. Es decir, intercambias eficiencia en el tiempo gastando más memoria (por ejemplo, guardando resultados de operaciones para evitar así tener que recalcularlas: mejoras en el tiempo, pero ocupas más espacio), y viceversa.

Sin embargo, hay otras técnicas que te permiten reducir el coste asintótico del programa. Estas técnicas se basan en tratar de dar un enfoque distinto a las partes del programa que tienen más peso asintótico. Esto es, sustituir algoritmos que no son eficientes por algoritmos que sí lo sean. Por ejemplo, usando *Divide & Conquer*, usando algoritmos que hacen uso de memoria para guardar cálculos y así no tener que repetirlos, usar otros tipos de estructuras de datos (Hashes en lugar de listas, árboles binarios…) o usando técnicas estadísticas para disminuir la cantidad de cálculos a realizar.

VII. Técnicas usadas para mejorar los costes asimptóticos

Como hemos descrito en el apartado anterior, explicaremos un par de técnicas para mejorar el coste asintótico de un programa:

A. *Divide & Conquer*

Esta técnica de programación se basa en usar un algoritmo recursivo que divide el problema principal en N subproblemas, y luego se llama a sí mismo para esos subproblemas. Cuando se llega al caso base, el problema normalmente es de coste inmediato (O(1)).

La eficiencia de un algoritmo de *D&C* se basa en 3 puntos:

- Coste de dividir el problema en sus subproblemas.
- Coste del caso base.
- Coste de fusionar las soluciones parciales en la solución final.

También se debe tener en cuenta el número de llamadas recursivas.

Un algoritmo de D&C tiene el siguiente coste asimptótico:

$$O\left(n \cdot \log_p n\right)$$

Ecuación 3

donde:

- **n** es el tamaño del problema
- **p** es numero de subproblemas de tamaño n/p en cada paso
- El **caso base** es de **coste inmediato** O(1)

- El coste de **separar** el problema en sus subproblemas y de **juntar** las **soluciones parciales** es O(n).

Normalmente, si conseguimos transformar encontrar un algoritmo de *D&C* para nuestro programa, conseguiremos reducir el coste asintótico de este en un grado de complejidad.

B. *Uso de memoria para almacenar calculos parciales*

En algoritmos como el cálculo de números de Fibonacci, es común no tener en cuenta que, a veces, estamos calculando el mismo número muchas veces para llegar a la solución final. Por ejemplo, si queremos calcular Fib(20), tendremos que calcular Fib(19) y Fib(18). Pero si nos fijamos, al calcular Fib(19) ya tendremos calculado Fib(18), así que este último cálculo no hace falta repetirlo. Si aplicamos este principio a nuestro algoritmo, podemos agilizar muchísimo el coste del programa.

Por ejemplo, podríamos usar un *HashMap* (también se podría usar un *ArrayList*) donde en la entrada N está guardado el enésimo número de Fibonacci. Al calcular un número de Fibonacci, primero miraríamos si está ya calculado en el Hash. Si no lo está lo calculamos (de forma recursiva). Pero si lo está, simplemente cogemos el valor. Esto hará que nunca repitamos cálculos que ya hemos hecho.

Haciendo unas pruebas con Java, para calcular Fib(50), de la forma optimizada ha tardado 0,39 milisegundos, mientras que en la forma sin optimizar, casi un minuto y medio.

¿Por qué hay tanta diferencia? Pues porque el algoritmo sin optimizar es de coste $O(2^n)$ (¡sí, exponencial!) mientras que el algoritmo que guarda cada calculo es simplemente de coste lineal O(n). Esta diferencia asintótica es debido a que en el algoritmo optimizado, si ya tenemos un cálculo hecho, se dejan de hacer llamadas recursivas más profundas, ya que consultamos el valor en el Hash, de coste O(1), *podando* así el árbol de llamadas recursivas.

Esto nos muestra que, usando un poco más de memoria, podemos hacer cálculos que, sin estas optimizaciones, serian imposibles. Para calcular Fib(800), nuestro algoritmo "optimizado" tardaría aproximadamente 6 milisegundos, mientras que el sin optimizar tardaría $1{,}48 \cdot 10^{220}$ años (el universo solo tiene $13 \cdot 10^9$ años).

VIII. Manual de usuario

La primera imagen que se tiene de la interfaz una vez se ejecuta es la siguiente.

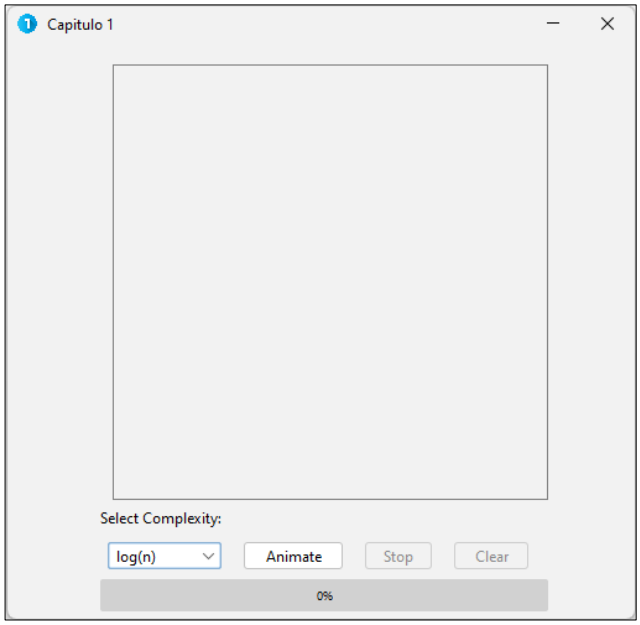

Ilustración 1

Se pueden distinguir dos partes muy diferenciadas. El panel de visualización de datos y la parte con la que puede interactuar el usuario.

Los pasos que debe de seguir el usuario a la hora de utilizar la aplicación son los siguientes.

- **Selección de complejidad a graficar**: El primer recuadro que hay en el área de interacción del usuario podemos observar un desplegable. Éste tiene siempre por defecto la complejidad $\log_2(n)$. El usuario

deberá hacer *click* sobre el recuadro y seleccionar la complejidad a graficar.

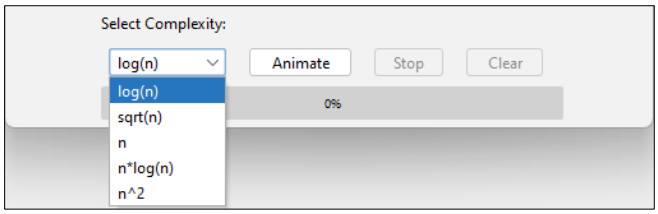

Ilustración 2

- **Animación**: El usuario deberá hacer *click* sobre el recuadro de "*Animate*". Una vez hecho esto, el programa empezará a graficar los tiempos de dicho algoritmo. Es importante remarcar que mientras se está graficando una función, el usuario no podrá poner en marcha una nueva animación.

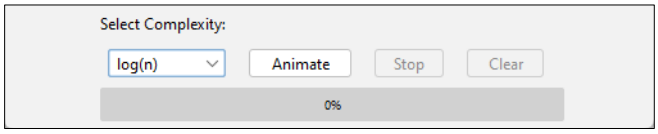

Ilustración 3

- **Nueva animación**: Una vez acabe la animación de la función, el usuario podrá volver a seleccionar una función y graficar sobre el mismo plano. Esto permite al usuario poder comparar los distintos costes asintóticos. El usuario puede incluso ejecutar varias veces la misma función para analizar si existe alguna diferencia.
- **Función de Stop**: El usuario puede, mientras se está ejecutando una animación, parar si así lo desea la ejecución. De hecho, esta opción solo se muestra disponible una vez se ejecuta una animación. El usuario puede querer parar la animación por distintos motivos, pero el más habitual será porque el tiempo de alguno de las funciones exceda el tiempo deseado por el usuario.

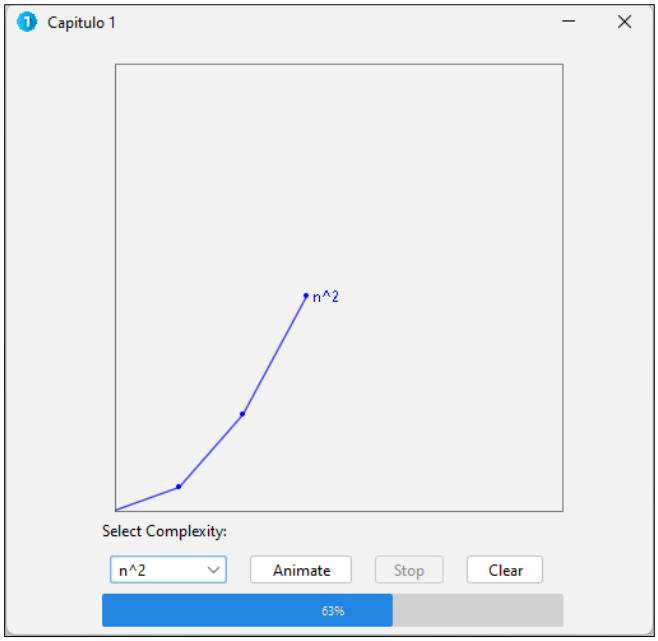

Ilustración 4

- **Función Clear**: Es muy probable que el usuario, después de hacer una serie de pruebas, quiera volver a tener el lienzo en blanco para poder visualizar los datos de una mejor manera. Esta funcionalidad está disponible mediante el botón "Clear", el cual borra todo el lienzo. Además, está programado de tal manera que solo está disponible la opción cuando hay al menos un gráfico.

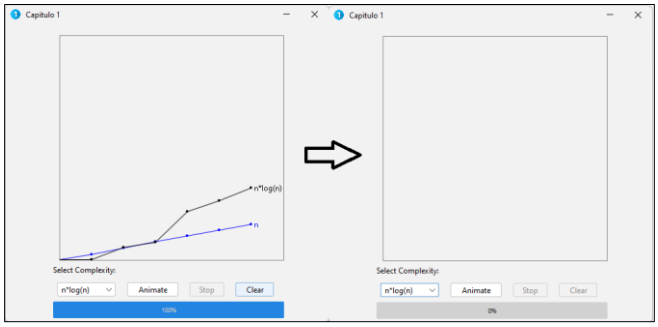

Ilustración 5

- **Cerrar el programa**: Para salir del programa es suficiente con hacer *click* en la "X" situada en la esquina superior derecha.

IX. METODOLOGÍA Y ANÁLISIS DE LOS RESULTADOS

Como se ha explicado anteriormente en el modelo hay definido un método por cada complejidad a simular.

Para simular estas complejidades se ha decidido hacer uso de la simple herramienta de programación como son los bucles, donde calculamos el número de iteraciones máximas según la complejidad y lo introducimos en uno o varios bucles. Dentro de estos bucles llamamos a un método auxiliar *sleep*() que dormita el hilo para simular el coste computacional.

A continuación, se comparan los resultados obtenidos en contraposición a las funciones matemáticas de las complejidades.

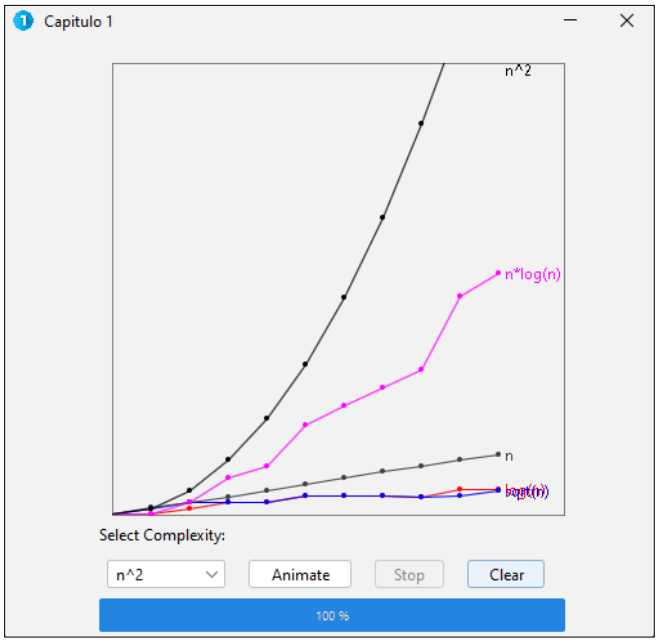

Ilustración 5

En la imagen anterior se puede observar un ejemplo de simulación de todas las complejidades. Debido a que log(n) y √n son muy parecidas en esta escala, se sobrescribe el nombre de la complejidad al final.

A continuación, se puede observar una gráfica real de las distintas complejidades.

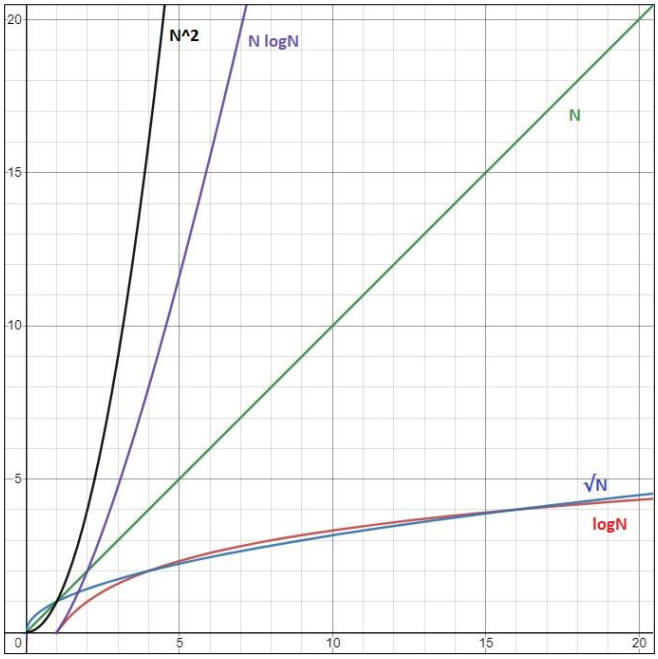

Ilustración 6

En primer lugar, cabe destacar que la representación de la aplicación ha sido escalada para que *n2* y *nlog(n)* no se salieran de proporción. Como se puede observar y se ha mencionado anteriormente *log(n)* y √*n* son muy parecidas e incluso se interseccionan dos veces, tanto en la aplicación como en la función real.

También se puede observar que la complejidad n es lineal, pero no diagonal ya que esta escalada, mientras que la complejidad cuadrática dibuja una parábola casi perfecta, pero un poco menos pronunciada que la real.

Por último, cabe destacar que las funciones log(n), √n y nlog(n) devuelven números reales, y debido al uso de la metodología explicada anteriormente, los bucles utilizan enteros para numerar las iteraciones, por este motivo se pueden observar escalones en la gráfica al realizar la conversión.

X. SUPOSICIONES Y POSIBLES MEJORAS

Dentro del modelo se han definido unas complejidades arbitrarias elegidas por su popularidad en el ámbito. Podría ser el caso que no se haya incluido alguna complejidad importante, o se haya incluido alguna no tan importante. Esto se

podría solucionar de manera sencilla modificando el Modelo gracias al patrón MVC.

También cabe destacar que dentro de dicho modelo se han definido unos tamaños a base de prueba y error que juntamente con el tiempo del sleep() crean una escala fácil de visualizar mediante la vista. Tanto los tamaños, como la duración del sleep se podrían modificar para tener una escala alternativa y esto se quería conseguir introduciendo campos nuevos en la interfaz para que el usuario pudiera modificar dichos valores, pero no se ha podido implementar debido a la falta de tiempo.

A la hora de mandar las coordenadas del nuevo punto al controlador, el modelo se encarga de escalarlos para que sea más fácil su representación. Para la coordenada del eje X se coje la posición del tamaño actual y se normaliza entre el número total de tamaños, para que de esta manera la vista pueda usar esta proporción con el tamaño del panel. Para la coordenada del eje Y se hace un simple factor de conversión para pasar de nanosegundos a una unidad de mayor escala y que sea más fácil de visualizar.

Por último, las mejoras que se hubieran hecho si se hubiera dispuesto de más tiempo, hubieran sido, permitir al usuario modificar el tamaño de la ventana, con un reescalado del panel para poder visualizar las gráficas mejores y más grandes y la introducción de los campos mencionados anteriormente para que el usuario pudiese modificar la escala.

XI. Conclusiones

Durante la realización del proyecto se han revisado distintos conceptos vistos en asignaturas anteriores, como por ejemplo la programación *Swing* de *Java*, o el uso de concurrencia y la interfaz *Runnable*.

Además, se han puesto en práctica conceptos teóricos como los costes computacionales y la complejidad asintótica.

Ya que el trabajo de cualquier ingeniero es intentar encontrar las soluciones más sencillas y eficientes a problemas, se ha intentado seguir una buena estructura a lo largo de la implementación, ya que se espera que este proyecto sirva de base practica y conceptual para los siguientes proyectos de la asignatura, y de esta manera poder menguar el trabajo a realizar posteriormente.

XII. Bibliografía

[1] Oracle, «UISwing Concurrency - Swing Worker, » [En línea]. Available: https://docs.oracle.com/javase/tutorial/uiswing/concurrency/worker.html.

[2] Wikipedia, «Wikipedia,» [En línea]. Available: https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm.

[3] «Oracle Java Documentation - SwingWorker,» [En línea]. Available: https://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html.

[4] J. M. Alarcón, «Rendimiento de Algoritmos y Big O - Campus MVP,» [En línea]. Available: https://www.campusmvp.es/recursos/post/Rendimiento-de-algoritmos-y-notacion-Big-O.aspx.

[5] P. M. H. Thomas V. Perneger, «Writing a research article: advice to beginners - Oxford Academic,» [En línea]. Available: https://academic.oup.com/intqhc/article/16/3/191/1814554.