JUEGO PARA APRENDER ESTRUCTURAS DE DATOS Y ALGORITMOS

GAME FOR LEARNING DATA STRUCTURES AND ALGORITHMS

Alejandro Jiménez Sánchez

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Grado en Ingeniería Informática

Curso 2020/2021
Febrero de 2021

Director:
Iván García-Magariño García

Índice general

1. Introducción **4**
1.1. Introduction . . . 4
1.2. Introducción a los juegos serios. Conceptos principales . . . 5
1.3. Introducción a la aplicación desarrollada . . . 7
1.4. Trabajo existente y aplicaciones similares . . . 11
1.5. Plan de trabajo . . . 13

2. Algoritmos e Interacción **15**
2.1. Introducción . . . 15
2.2. Ordenación por inserción . . . 16
2.2.1. Tutorial del juego . . . 16
2.2.2. Interacción detallada con un ejemplo . . . 18
2.2.3. Algoritmos de control y comprobación . . . 21
2.3. Ordenación por selección . . . 22
2.3.1. Tutorial del juego . . . 24
2.3.2. Interacción detallada con un ejemplo . . . 25
2.3.3. Algoritmos de control y comprobación . . . 26
2.4. Ordenación por el método de la burbuja . . . 27
2.4.1. Tutorial del juego . . . 28
2.4.2. Interacción detallada con un ejemplo . . . 29
2.4.3. Algoritmos de control y comprobación . . . 30
2.5. Ordenación por mezclas (1) . . . 31
2.5.1. Tutorial del juego . . . 33
2.5.2. Interacción detallada con un ejemplo . . . 35
2.5.3. Algoritmos de control y comprobación . . . 36
2.6. Técnica de ventana deslizante . . . 37
2.6.1. Introducción a la técnica . . . 37
2.6.2. Tutorial del juego . . . 40
2.6.3. Interacción detallada con un ejemplo . . . 42
2.6.4. Algoritmos de control y comprobación . . . 44

3. Diseño y aspectos de la aplicación **45**
3.1. Clases y procesos principales . . . 45
3.1.1. Clases . . . 45
3.1.2. Diagramas de secuencia de dos procesos principales . . . 48
3.2. Feedback proporcionado y otras funcionalidades . . . 50
3.2.1. Feedback . . . 50
3.2.2. Log con la interacción . . . 53

4. Evaluación con Usuarios **54**
4.1. Primeras pruebas con usuarios 54
4.1.1. Introducción 54
4.1.2. Pruebas realizadas 55
4.1.3. Resultados y posibles mejoras 55
4.2. Teoría sobre pruebas de usabilidad 57
4.3. Pruebas finales con usuarios 58
4.4. Resultados de las pruebas y conclusiones 59
4.4.1. Cuestionario previo 60
4.4.2. Cuestionario final 61
4.4.3. Análisis de los *logs* 62

5. Conclusiones y trabajo futuro **65**
5.1. Conclusiones 65
5.2. Trabajo futuro 65
5.3. Conclusions 66
5.4. Future work 66

Bibliografía **67**

Resumen

Las asignaturas de algoritmos y estructuras de datos son una parte fundamental en los estudios de informática. En una primera aproximación a ellas, el alumno puede tener dificultades en su comprensión. Es por ello que puede ser buena idea disminuir el nivel de abstracción, elegir un caso concreto y mostrar la aplicación del algoritmo a dicho ejemplo. Además, numerosos estudios están a favor de que sea también el estudiante el que aprende por sí mismo mediante la interacción con un juego relacionado con el concepto a estudiar. Es por ello que se implementa una aplicación en *Unity* que muestra las ideas principales de algunos de los algoritmos básicos y estructuras de datos típicas, y que no se limita a mostrar su funcionamiento, sino que permite la interacción del usuario, guardando también estadísticas sobre su rendimiento. En esta memoria se explican inicialmente los objetivos y las aplicaciones similares existentes. En los capítulos siguientes, se describe el desarrollo de la aplicación. Primero, se centra en cada juego implementado y la interacción pedida al usuario en cada caso; después, en los aspectos del diseño y desarrollo de la aplicación. La memoria finaliza con los resultados de las pruebas con usuarios.

Palabras clave

Algoritmos, aprendizaje basado en juegos, estructuras de datos, pruebas de usabilidad, Unity.

Abstract

Algorithms and data structures are core subjects in computer science courses. In a first approach, the student may find them difficult. In consequence, it could be a good idea to use a lower level of abstraction, choose a concrete example and show the flow of the algorithm applied to that case. In addition, there are a lot of articles that uphold the benefits for the students if they interact with a game about these concepts. Therefore, a *Unity* game has been developed in order to show the main concepts of these most common algorithms and data structures. It allows users to interact with it to make learning more attractive, and saves data about their performance. In this work, initially, the aims of the application are explained, and similar games are analyzed. Then, there are two chapters about the development of the application. The first one is focused on the games that are implemented and the user interaction. The second one explains the design and the development of the game. Finally, the outcomes of the usability tests are shown.

Keywords

Algorithms, game-based learning, data structures, usability testing, Unity.

Capítulo 1

Introducción

Las asignaturas de algoritmos y estructuras de datos son básicas en los cursos de ciencias de la computación. Además, son muchos los autores que defienden que incorporar videojuegos en el proceso educativo es una buena idea. Esto ha servido de motivación para desarrollar una aplicación educativa en *Unity* sobre ciencias de la computación.

En este trabajo se explica dicho juego educativo sobre algoritmos y estructuras de datos. El objetivo es que el usuario pueda simular el flujo de un algoritmo haciendo acciones similares a las que haría el ordenador sobre su memoria, aprendiendo así cómo funciona el algoritmo. Además, la aplicación guarda estadísticas sobre el rendimiento de los jugadores y tiene características típicas de los juegos como puntuaciones.

En este capítulo introductorio se exponen conceptos básicos sobre juegos serios, se analizan las aplicaciones similares y se expone el plan de trabajo tras una etapa de investigación. El segundo capítulo se centra en los algoritmos implementados en el juego y la interacción requerida al usuario. El tercero explica el diseño global así como funcionalidades adicionales. Se adjuntan también algunos diagramas UML. Por último, el cuarto capítulo explica los resultados de las pruebas de usabilidad realizadas al finalizar el desarrollo.

1.1. Introduction

Algorithms and data structures are core subjects in computer science courses. In addition, a lot of researchers uphold that incorporating videogames in education is a good idea. This has been a motivation to develop a *Unity* educational application about computer science.

In this work, this educational game about algorithms and data structures is explained. The aim is that the user can simulate the flow of an algorithm by executing actions that are similar to those executed by the computer on its memory. Thus, the student would learn how the algorithm works. In addition, the application saves statistics about the performance of the players and includes some game features like scores.

In this introductory chapter, basic concepts about serious games are explained after some research. The second chapter focuses on these algorithms implemented in the game and the interaction that is required to players. The third one is about the overall design of the application and additional functionalities. Some UML diagrams are shown in this chapter too. Finally, the outcomes of the usability tests that were performed after the development was finished are explained in the fourth chapter.

1.2. Introducción a los juegos serios. Conceptos principales

Dado que el objetivo de este trabajo es desarrollar un juego serio educativo sobre ciencias de la computación, en primer lugar se exponen conceptos sobre este campo. Esta investigación ha ayudado al diseño y desarrollo de la aplicación, proporcionando ideas ya probadas por otros autores. Esta sección introductoria se basa principalmente en la referencia [1].

Se sabe que los juegos han estado presentes en las diferentes sociedades desde hace mucho tiempo. Con la aparición de ordenadores, consolas y otros materiales informáticos se abrieron nuevas posibilidades y se comenzaron a desarrollar los llamados juegos digitales. Este campo de los videojuegos se ha desarrollado enormemente en los últimos años, y una de las claves de este crecimiento es el hecho de provocar en sus usuarios una enorme atracción. Como se definió en [2], el jugador puede llegar al llamado estado mental de flujo, donde queda totalmente concentrado e inmerso en el juego. Otro concepto relacionado es el de flujo del juego [3], caracterizado por concentración exclusiva, sentimiento de control sobre el juego, tener *feedback* inmediato de las acciones realizadas y afrontar metas claras. Por tanto, conseguir ese estado será uno de los objetivos del equipo de desarrollo de un juego digital.

Estas características de los juegos se pueden aprovechar para propósitos que abarcan no solo el entretenimiento, sino también otros campos. En [1] se comenta el ejemplo de la televisión. Pasado un tiempo tras su aparición, se empezaron a desarrollar programas con propósitos educativos, por citar un campo distinto al entretenimiento. De hecho, algunos juegos de mesa ya tenían propósitos adicionales a divertirse. Por último, es claro que los juegos deportivos tienen la ventaja adicional de mejorar la salud.

Se define juego serio como aquel juego digital que persigue entretener y conseguir al menos un objetivo más. No obstante, no hay un consenso total en esta definición, ya que algunos autores se centran en la intención del desarrollador del juego, mientras que otros se centran en la intención del usuario. En cambio, sí es común la idea de tener al menos un objetivo además del entretenimiento. Estas metas adicionales se denominan objetivos característicos. Los juegos serios pueden ser de cualquier género, ya que se puede buscar la consecución de los objetivos característicos por medio de diversas temáticas. Si un juego serio tiene éxito, el jugador será distinto después de haber jugado respecto a antes: tendrá más conocimiento, mejor salud o sus opiniones habrán cambiado. El término "juego serio" ganó popularidad en 2002 con el artículo de Sawyer y Rejetski [4] y con el juego *America's Army* [5].

Relacionado con los conceptos de flujo explicados más arriba, Sinclair [6] introduce el flujo dual para juegos serios: hay que buscar tanto el atractivo del juego (buena experiencia de usuario) como su efectividad (consecución del objetivo característico). El atractivo se consigue con un equilibrio entre la dificultad y el nivel de habilidad del usuario. Como este último es dinámico (es esperable que la habilidad mejore con la práctica del juego), habría que adaptar la dificultad dinámicamente también para que el jugador no caiga en el tedio. El otro extremo (la dificultad supera ampliamente a las habilidades) se caracteriza por frustración por parte del usuario. En cuanto a la efectividad, en este artículo [6] se presenta un juego serio relacionado con el deporte (uno de los llamados *exergames*). Por tanto, la efectividad consistiría en encontrar un equilibrio entre la capacidad física del usuario y la intensidad del ejercicio, manteniéndose siempre la exigencia del juego en un intervalo que no resulte trivial para el usuario (no generaría beneficio en la salud del mismo) pero que tampoco

exija demasiado (podrían ocurrir lesiones). Esto se extiende a muchos otros géneros de juegos.

Habitualmente se relacionan los conceptos de juegos educativos, "*edutainment*" ([7], donde se juntan *education* y *entertainment*) y aprendizaje basado en juegos (*game-based learning*, [7]) con el concepto de juegos serios, ya que se establece la identificación del objetivo característico con el aprendizaje. Esto no es cierto, ya que el objetivo característico puede ser muy diverso (ponerse en forma, desarrollar habilidades sociales o de resolución de problemas, promover ciertos valores). Por tanto, los juegos educativos y resto de conceptos relacionados se identifican más bien como un subconjunto estricto de los juegos serios.

Pese a que la industria de los juegos serios ha crecido recientemente, está muy lejos de llegar al campo de los videojuegos de entretenimiento (nomenclatura usada para aquellos cuyo objetivo es solo entretener). Una razón de esto es que el público objetivo suele ser mucho más reducido y específico que en el caso de videojuegos de entretenimiento. Otro inconveniente es que los presupuestos para el desarrollo de la aplicación suelen ser menores. Además, es más complejo su desarrollo, ya que habitualmente hay que incluir en el proceso a expertos del dominio del objetivo característico. Esta multidisciplinaridad puede complicar el desarrollo de la aplicación, al convivir personas con diferentes formas de resolver problemas y de pensar, o incluso las disciplinas pueden tener terminología distinta. Por último, como es un campo menos amplio, los desarrolladores de juegos serios tienen menos referencias en las que basarse.

Como ya se ha comentado, es buena idea (y no solo en juegos serios, sino en cualquier tipo de juego) adaptarse a la habilidad del usuario. Además, el hecho de que cada juego serio tenga unos usuarios objetivo muy específicos, hace aún más necesario si cabe la adaptación de la dificultad a las habilidades.

Dado que una de las principales ramas de los juegos serios son los juegos educativos y en esta memoria se explica el diseño y desarrollo de una aplicación cuyo objetivo característico es el aprendizaje, es interesante centrarse también en algunos aspectos educativos. En el artículo mencionado anteriormente [7], M. Prensky asegura que la motivación que despiertan los juegos en los niños se debe al aprendizaje que proporcionan (y esta motivación es deseable al enfrentarse a un aprendizaje). Los videojuegos enseñan, en un nivel subyacente, a recopilar información y tomar decisiones rápidamente, desarrollar estrategias para superar problemas o a cooperar con otros jugadores.

En cuanto a las diferentes formas de aprender, el modelo de referencia teórico 8LEM (*Eight Learning Events Model*) [8] describe el aprendizaje a partir de ocho eventos de aprendizaje básicos, siendo un evento de aprendizaje la descripción de la actividad de un alumno y un maestro en un contexto educativo. Los ocho eventos básicos son los siguientes:

- Imitación: Observar y, posteriormente, imitar. El profesor sirve como modelo, pero este modelo no tiene por qué ser suyo.
- Recepción: Recibir información emitida por el profesor intencionadamente para enseñar.
- Ejercicio: En campos en los que hay que automatizar habilidades, se aprende practicando. El tutor debe guiar y proporcionar *feedback* adecuado.
- Exploración: Aprender mediante propia investigación con cierta libertad, buscando entre datos. El profesor proporcionaría fuentes de información adecuadas.

- Experimentación: Prueba y error, es decir, manipular el entorno y observar consecuencias. El alumno probaría así todas las combinaciones posibles o aquellas que considere interesantes.
- Creación: Crear nuevo contenido, donde el término "nuevo" se refiere al alumno, no necesariamente contenido que no esté ya creado por ninguna otra persona.
- Autorreflexión: Reflexionar sobre los propios procesos cognitivos.
- Debate: Aprender mediante interacciones sociales, como debates en los que se defiende una posición ante un conflicto, y están promovidos y moderados por un tutor. La interacción puede ser asíncrona (como en un foro) o síncrona.

Como ya se ha dicho, el desarrollo de juegos serios es multidisciplinar. Por tanto, si se quiere desarrollar un *software* educativo (por ejemplo), conviene conocer alguna de estas teorías para poder aplicar los principios al desarrollo. También relativo a este campo, el artículo [9] y otros resultados análogos proponen más claves para un juego educativo de éxito: proporcionar *feedback* inmediato, tener gran nivel de interacción, desafío y competitividad; y conseguir que el usuario tenga motivación intrínseca para jugar. Como se indica en [1], un juego serio también puede ser utilizado a partir de motivación extrínseca (usar la aplicación como herramienta para conseguir cierta meta). Por ejemplo, un estudiante que quiere superar una prueba: utiliza la aplicación para estudiar y aprobar el examen, y después ya no tiene la necesidad de usarla.

En el diseño de videojuegos y juegos serios intervienen muchos otros factores. Además de los obvios (como podría ser definir el público objetivo) hay otros como pensar en el perfil de los posibles jugadores (Bartle [10] publicó un modelo para juegos MUDs basado en *killers*, *achievers*, *socializers* y *explorers*) y la adaptación del juego a cada tipo. También hay que considerar aspectos como la necesidad de supervisión por parte de un instructor, el entorno donde se debe jugar, la posibilidad de volver a jugar (algunos juegos están diseñados para jugar una sola vez; otros, para jugar varias veces) o el tiempo necesario para jugar (el tiempo necesario para cada partida es un aspecto importante).

Las ideas y artículos sobre juegos serios son en la actualidad muy numerosos. La aplicación que se explica en esta memoria ha seguido algunos de los conceptos expuestos en esta sección introductoria, con mecanismos muy simples pero que (se pretende) mejoran la experiencia de usuario respecto a las primeras versiones que se implementaron, donde no se tuvieron en cuenta estos conceptos.

1.3. Introducción a la aplicación desarrollada

Hay diferentes artículos sobre el proceso de estudio de cursos relacionados con las ciencias de la computación. Por ejemplo, en [11] se presentan los resultados de una encuesta en la que participaron más de 500 estudiantes y profesores, con el objetivo de encontrar aquellos puntos que más cuestan a los estudiantes de los primeros cursos de informática. Este artículo llega a la conclusión de que muchos alumnos desarrollan conceptos superficiales sobre la programación (dominan la sintaxis de cierto lenguaje de programación y dominan instrucciones aisladas) pero fallan al integrar esto en programas completos que resuelvan un problema. Además, los resultados de la encuesta aseguraban que los conceptos que más difíciles resultaban eran dividir un programa en subprogramas, punteros, recursión y tipos abstractos de

datos (TADs). En cuanto a métodos de aprendizaje, el preferido es basarse en programas de ejemplo, y destacan los métodos prácticos sobre los teóricos, aunque la teoría sea también importante.

En todo caso, entender las estructuras de datos y los algoritmos más básicos es imprescindible para cualquier estudiante de informática. En primer lugar, conocerlos permite tomar la mejor decisión de diseño frente a un problema en cuanto a eficiencia y simplicidad. Pero, más allá de esto, entender su funcionamiento en profundidad permite poder implementar modificaciones de los mismos. Esto sirve para adaptarlos a problemas concretos en los que las estructuras o algoritmos necesarios difieren en algún punto de los que se estudian como base, pero la idea subyacente es similar y puede ser usada para resolver dicho problema.

Ir formando esta capacidad para pensar en qué usar para resolver el problema, así como hacer que el aprendizaje sea gradual y comenzar por códigos sencillos, es lo que puede explicar que se estudien algoritmos que no se usan al tener ya implementaciones mejores (aunque hay una discusión sobre si la sencillez de código justifica su enseñanza frente a ineficiencia e inutilización del mismo en el artículo [12]). En muchos centros se mantiene la enseñanza de los algoritmos de ordenación de complejidad cuadrática $\mathcal{O}(n^2)$. Ante una primera aproximación, es interesante comprender estos algoritmos para después poder pasar al estudio de los algoritmos óptimos en cuanto a complejidad $\mathcal{O}(n \log n)$, pero menos intuitivos.

En cualquier caso, no hay duda de la importancia de las asignaturas relacionadas con estos conceptos. Para muchos estudiantes, la mejor forma de entender el funcionamiento de un algoritmo es ver la aplicación del mismo a un caso concreto, lo que hace mucho más intuitivo el código asociado y permite, incluso, que no sea necesario memorizarlo en absoluto. Esto se relaciona con el artículo [11], donde no destaca especialmente el método de aprendizaje de visualizaciones interactivas, pero tiene una buena aceptación (3.3 sobre 5, en línea con otros métodos a excepción del ya mencionado uso de programas de ejemplo). Surgen así interesantes aplicaciones que muestran el flujo de ciertos algoritmos o la visualización de algunas estructuras de datos: en [13] se presenta un visualizador de estructuras de datos a nivel conceptual (por ejemplo, los árboles se presentan de la forma esquemática habitual en lugar de mostrar el contenido de la memoria del ordenador). El usuario puede realizar acciones sobre estas estructuras de datos y apreciar cómo van cambiando de forma conceptual. En definitiva, surgen aplicaciones que aportan un *feedback* mucho más visual que herramientas típicas de depuración de código (*debugging*).

Todo esto ha servido de motivación para implementar un juego (con más carga educativa que lúdica) que explique algunos de los algoritmos básicos y estructuras de datos principales de las ciencias de la computación. La intención es simular la aplicación del algoritmo sobre un caso concreto y aleatorio de entrada, siendo el usuario el que hace acciones similares a las que haría un ordenador en ese caso concreto. Por ejemplo, para ordenación por inserción podríamos pensar que la fase en la que se hace hueco al elemento a insertar consiste en que el valor de una posición se desplaza a la posición siguiente, cuando realmente se obtiene el valor de la posición i y se copia en la posición $i + 1$. Esto, de forma interactiva, se puede modelar haciendo click en dicha posición i y arrastrando a la posición $i + 1$, lo que copiaría el valor. Estas metáforas claramente no intervienen en la comprensión del algoritmo (sería simplemente una pequeña abstracción), y podemos decir que el usuario está haciendo acciones similares a las que haría el ordenador sobre su memoria. Con la descripción dada, el juego se englobaría en el género de juegos de lógica o puzzles.

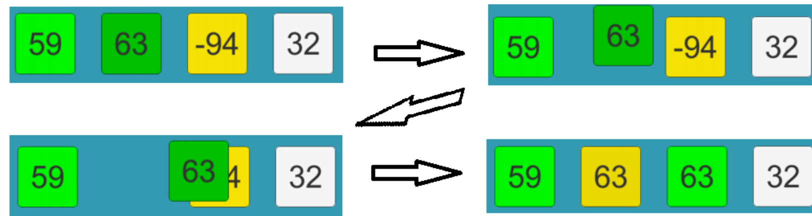


Figura 1.1: Interacción usada para representar la instrucción de alto nivel `lista[i+1] = lista[i]`

Así, mediante la imitación del ordenador por parte del usuario a casos concretos, el estudiante retendría las ideas de los algoritmos; igualmente, afianzaría los conceptos asociados a las estructuras de datos con las que se trabaje. Se incluyen además conceptos típicos de juegos como estadísticas del rendimiento y puntuaciones asociadas a las diferentes actividades.

Se estarían usando, por tanto, algunos de los principios de aprendizaje del modelo anteriormente comentado 8LEM [8]. Por ejemplo, el principio de ejercicio (automatizar habilidades), que encaja perfectamente con la idea de aprender un algoritmo. Aunque se ha pretendido que no sea así, también podría aparecer el principio de experimentación de prueba y error. En las pruebas con usuarios de las fases iniciales de desarrollo se apreció que los primeros minutos probaron diferentes interacciones hasta que encontraron el patrón pedido. Ante esto, surgió la idea de añadir un tutorial explicando la interacción solicitada al entrar en cada juego. Por otro lado, esta aplicación iría en la línea de incluir visualizaciones interactivas o conceptos de gamificación en el aprendizaje, concretamente para los primeros cursos de ciencias de la computación, aunque la población objetivo puede ser mucho más amplia.

La aplicación que se desarrolla tiene varios requisitos funcionales principales, donde el primero de ellos destaca claramente sobre el resto:

- El usuario podrá elegir la pantalla del juego asociado al algoritmo o estructura de datos que quiera practicar y realizar dicha actividad. Esta interacción consistirá en acciones guiadas (o no, según la habilidad del usuario), y tras cada una de ellas el sistema dará siempre *feedback* sobre si la acción es correcta o no, siguiendo así los principios de [3] o [9].
- El usuario podrá consultar estadísticas sobre su rendimiento, como partidas jugadas, acciones requeridas y tiempo empleado para realizar las acciones.
- El usuario, en cualquier momento de su experiencia en la aplicación, podrá interrumpir la actividad que esté realizando, pasando a los menús en cualquier caso o reiniciando la partida en caso de estar en una pantalla asociada a un juego.

En cuanto a requisitos no funcionales o de calidad, es una aplicación sencilla que no debería exigir grandes recursos. No es necesaria conexión a Internet para usarla. Se ha probado y está disponible tanto en Windows como en Android.

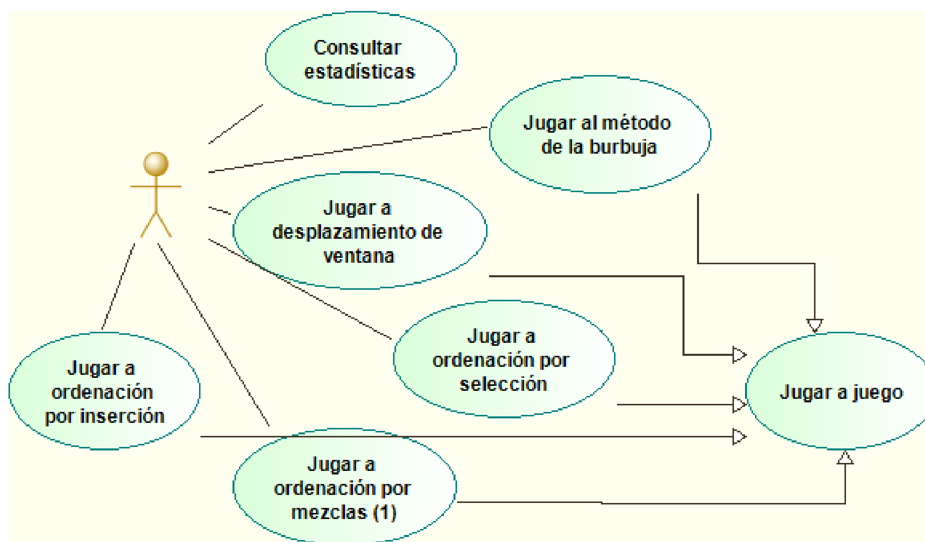


Figura 1.2: Diagrama de casos de uso de la aplicación.

Como se comentará en la siguiente sección, hay varias aplicaciones ya existentes en las que predomina el carácter lúdico, y otras en las que predomina el carácter teórico (mediante explicaciones teóricas o simulaciones en las que el usuario observa pero no puede interaccionar). El objetivo del trabajo es conseguir una aplicación totalmente interactiva, donde el componente educativo destaca sobre el lúdico, pero donde también apreciamos conceptos típicos de juegos. En cuanto a la interacción, se usan patrones habituales (como por ejemplo *drag-and-drop*) pero se intenta no abstraer demasiado las ideas subyacentes para que el juego sea bastante fiel a la realidad (como se discutió en el párrafo previo a la figura 1.1).

La aplicación explicada en esta memoria implementa finalmente juegos asociados a ordenación por inserción, ordenación por selección, método de la burbuja, ordenación por mezclas (una parte de su funcionamiento, la asociada a la mezcla de dos listas ordenadas) y técnica de desplazamiento de ventana para reducir complejidad.

Otro objetivo del trabajo es añadir adaptabilidad a la aplicación según la habilidad que desempeñen los jugadores en sus últimos movimientos, siguiendo las ideas de [6]. Se ha implementado un mecanismo muy sencillo para proporcionar pistas en casos necesarios, y para adaptar la dificultad. Esto se explicará en detalle en el capítulo 3.

Un último objetivo es realizar pruebas con usuarios para comprobar la usabilidad (atractivo) y la utilidad (efectividad) de la aplicación. Para ayudar en este propósito, se implementa un sistema de *logging* con toda la interacción del usuario: el *log* de cada prueba se procesará para poder obtener datos cuantitativos. Esta idea se obtiene de los artículos [14] y [15]. En este último se presentan varios métodos de análisis de toda esta información recopilada, aunque seguramente el método más útil en el caso de la aplicación expuesta en esta memoria venga dado por las llamadas curvas de aprendizaje [16]. Por supuesto, no hay que olvidarse de los datos cualitativos de las pruebas, que se obtendrán por medio de grabaciones. Se explican más detalles de estos métodos de análisis y de las evaluaciones realizadas finalmente en el capítulo 4, junto a las conclusiones.

En cuanto a la población objetivo a la que estaría dirigida preferentemente la aplicación, tenemos a usuarios que conocen los fundamentos principales de la programación, y se en-

cuentran aprendiendo diferentes algoritmos y estructuras de datos básicas; otra opción serían los usuarios que ya han estudiado y conocen todos estos algoritmos y estructuras de datos, y quieren recordar o afianzar cómo funcionaba cada uno de ellos. En las pruebas también se ha incluido a algunos usuarios sin conocimientos de informática, para ver la evolución de sus interacciones durante su experiencia en el juego.

La aplicación ha sido desarrollada en Unity (en concreto, usando la versión 2019.2.11f1). Como se comenta en [17], Unity fue creado por tres amigos (David Helgason, Joachim Ante y Nicholas Francis) en Dinamarca, lanzando el producto en 2005. Dicho sistema surgió al escribir Nicholas Francis una pregunta solicitando ayuda para el desarrollo de un sistema de sombreado que quería implementar. Fue Joachim Ante quien respondió a dicha cuestión, y ambos colaboraron en desarrollar un sistema así. Posteriormente, David Helgason se unió al proyecto. Desde entonces, dicho entorno de desarrollo y motor de videojuegos ha crecido hasta convertirse en uno de los más importantes en la actualidad.

El lenguaje de programación usado, dentro de los compatibles con Unity, ha sido C#. Como se explica en la referencia [18], este lenguaje totalmente orientado a objetos fue desarrollado por Microsoft para soportar características del *framework* .NET. Siendo un lenguaje bastante moderno, se engloba en la familia de C, C++ y Java.

1.4. Trabajo existente y aplicaciones similares

Centrándonos en juegos serios relativos a aprendizaje, en concreto de temática de ciencias de la computación, es interesante el artículo [19], en el que se hace un estudio alrededor de un juego (*Program your robot*) que motiva el pensamiento computacional. Esto es diferente del concepto de programación, y consiste en usar métodos, lenguajes y sistemas de las ciencias de la computación para resolver diferentes problemas. Esta definición se inspira en la que dio Wing en [20]. Las categorías que se suelen aceptar dentro del pensamiento computacional son lógica condicional, construcción de algoritmos, *debugging*, simulación y computación distribuida. Muchos autores defienden la inclusión del pensamiento computacional en los programas educativos en todos los niveles y todas las disciplinas. J. Qualls y L. Sherrell hacen en [21] una recopilación de distintos artículos relacionados.

Volviendo al artículo [19] se mencionan también diversos ejemplos de juegos serios con el objetivo de aprender conceptos relacionados con programación o ciencias de la computación. Por ejemplo, *Robocode* (2001), uno de los juegos serios pioneros sobre programación y que está dedicado al aprendizaje de Java. Consiste en programar un tanque para, posteriormente, batallar con otros tanques programados. Además, *Robocode* es software libre. Otros ejemplos pueden ser *Colobot*, *The Catacombs* [22], *Saving Sera* [22] o *EleMental* [23]. Este último trata sobre recorrido en profundidad (*Depth-First Search*, DFS) en árboles binarios.

También en este artículo [19] se hace una distinción interesante: herramientas de programación más interactivas que los *IDEs* habituales, como podría ser *Scratch*, no se consideran entornos de *game-based learning*. Una diferencia, por ejemplo, es que en el juego presentado (*Program your robot*) hay una puntuación asociada a la elegancia de la solución (correcta ejecución y reusabilidad del código). Esta retroalimentación típica en los juegos podría animar al usuario a pensar cómo se podría mejorar su código para obtener más puntos, con el aprendizaje que esto conlleva.

Retomando el caso de *Program your robot*, en este juego se intentan fomentar los conceptos del pensamiento computacional, como construcción de algoritmos, *debugging* o simulación. Además, se introducen los principios básicos de programación, como instrucciones y funciones. Es destacable que a medida que avanza el juego, se hace necesario emplear buenas prácticas de programación (usar funciones en vez de repetir código, por ejemplo). El juego consiste en un robot que debe escapar de unas plataformas mediante un algoritmo solución proporcionado por el usuario. Cuanto mayor elegancia tenga el algoritmo proporcionado (buenas prácticas de programación comentadas anteriormente, como uso de funciones) mayor será la puntuación. Los resultados que se obtuvieron en la fase de evaluación fueron muy positivos, ya que los estudiantes tuvieron la sensación de que la aplicación era sencilla, intuitiva y cumplía el propósito de enseñar conceptos de ciencias de la computación.

Si nos acercamos aún más a la temática de la aplicación objetivo (es decir, algoritmos o estructuras de datos) sigue habiendo trabajo ya realizado. Por ejemplo, en [14] se presenta un juego para entender lo relativo a las pilas, *The Stack Game*. Se estudian aplicaciones de esta estructura de datos como podría ser convertir notación infija en postfija para operaciones. En este artículo también se hace hincapié en que además de una introducción teórica y visualizaciones dinámicas del funcionamiento de la estructura, es favorable que el estudiante interaccione directamente mediante este tipo de aplicaciones.

Otros artículos sobre juegos para aprender estructuras de datos y algoritmos son [24], donde se presenta un entorno interactivo web llamado DSLEP (*Data Structure Learning Platform)*, que permite estudiar conceptos de arrays, pilas (mediante el famoso juego de las torres de Hanoi), colas, listas doblemente enlazadas y árboles con sus recorridos, e incluye elementos de juegos como puntos, niveles y clasificaciones; también [25], donde se presenta *Play and Learn DS*, una plataforma que permite visualizaciones interactivas sobre conceptos de las estructuras típicas ya mencionadas en otros casos, aunque en este caso destaca también la presencia de conceptos sobre árboles binarios de búsqueda; y, por último, el artículo [26] que presenta *Wu's Castle*. Este juego se engloba en el proyecto *Game2Learn*, que consiste en que alumnos de cursos avanzados de ciencias de la computación crean juegos educativos que se usarán en los cursos iniciales de dicha disciplina. En este juego destaca la enseñanza de bucles y bucles anidados. Los resultados fueron positivos, ya que los usuarios aseguraron ver interesante la aplicación al ser más interactiva y visual (en concordancia con lo que ya se ha indicado en esta memoria analizando otros estudios).

En cuanto a algoritmos como tema central del juego, destacan las aplicaciones que tratan sobre los de ordenación. Por ejemplo, la que se puede descargar desde `https://play.google.com/store/apps/details?id=com.meet1234&rdid=com.meet1234`, llamada *SORTit*, que es bastante cercana a lo que se pretende en este trabajo. Sin embargo, la abstracción usada en algunas interacciones es de mayor nivel, y por tanto menos próxima a cómo actúa realmente el algoritmo. Por ejemplo, la ordenación por inserción permite poner el elemento a insertar en su posición correcta con un solo *click*, perdiendo la parte del algoritmo consistente en copiar de derecha a izquierda (visualmente) a la posición siguiente, lo que a la hora de implementar el algoritmo podría dar problemas aun habiendo conseguido dominar completamente el juego. Además, tiene el inconveniente de solo tener implementadas ordenaciones por inserción, selección y método de la burbuja. Por último, otro recurso interesante es *Algoritmos: Explicados y Animados*, que se puede descargar mediante `https://play.google.com/store/apps/details?id=wiki.algorithm.algorithms`. El contenido que abarca es bastante mayor, pero tiene el inconveniente de no ser interactiva,

sino que explica los algoritmos mediante simulaciones de casos concretos y mediante teoría.

1.5. Plan de trabajo

Se dedica una primera etapa al aprendizaje del motor de videojuegos *Unity*, plataforma en la que se desarrolla la aplicación, así como a analizar aplicaciones similares y diferentes artículos para obtener ideas a incorporar en el juego. Posteriormente, se comienzan a implementar diferentes pantallas a modo de *Scenes* (usando terminología de *Unity*), cada una correspondiente a un menú o un juego (asociado al aprendizaje de cierto algoritmo). Cada una de estas pantallas se desarrolla en forma de prototipo exploratorio. Es decir, tras una primera versión muy sencilla, se van añadiendo funcionalidades y mejorando la interacción y usabilidad poco a poco, pero no se descarta el trabajo realizado en el prototipo. Los diferentes juegos principales se fueron añadiendo uno tras otro, esto es, se desarrolla una pantalla hasta su versión final y posteriormente se comienza de cero con la nueva pantalla. Paralelamente al desarrollo de estos juegos, también se añaden otras funcionalidades como guardar estadísticas sobre el rendimiento del jugador o guardar un *log* con la interacción del usuario de cara al análisis de la efectividad de la aplicación.

Tras esto, en una versión intermedia del desarrollo, se realizan las primeras pruebas con dos usuarios para descubrir potenciales errores. Como es sabido, el coste de arreglar un error se incrementa cuanto más adelante se descubra en la etapa de desarrollo. Estas pruebas han servido para corregir algunos problemas en la parte que ya estaba implementada, así como recibir valioso *feedback* de cara a la segunda etapa de desarrollo, en la que se incorporan más algoritmos y funcionalidades a la aplicación.

Finalmente, se realizan las evaluaciones con usuarios, que tienen un doble objetivo: en primer lugar, comprobar si la interfaz es intuitiva y el usuario comete pocos errores asociados a la misma (sin tener en cuenta los errores asociados a la falta de conocimientos en el contenido que se esté estudiando); en segundo lugar, comprobar si el usuario comprende mejor el concepto que se estudia en una cierta pantalla después de jugar respecto a antes de empezar la experiencia de juego (análogamente, aquí no hay que tener en cuenta los errores asociados a una mala interfaz). Para todo esto será de mucha utilidad el *log* guardado. Dado que no estamos hablando de una aplicación de propósito general sino de un juego educativo, se complementan las ideas básicas de pruebas de usabilidad con otros artículos específicos para este caso, como [27], [16] y [15].

Se adjunta a continuación una imagen del diagrama Gantt con la planificación temporal que se siguió.

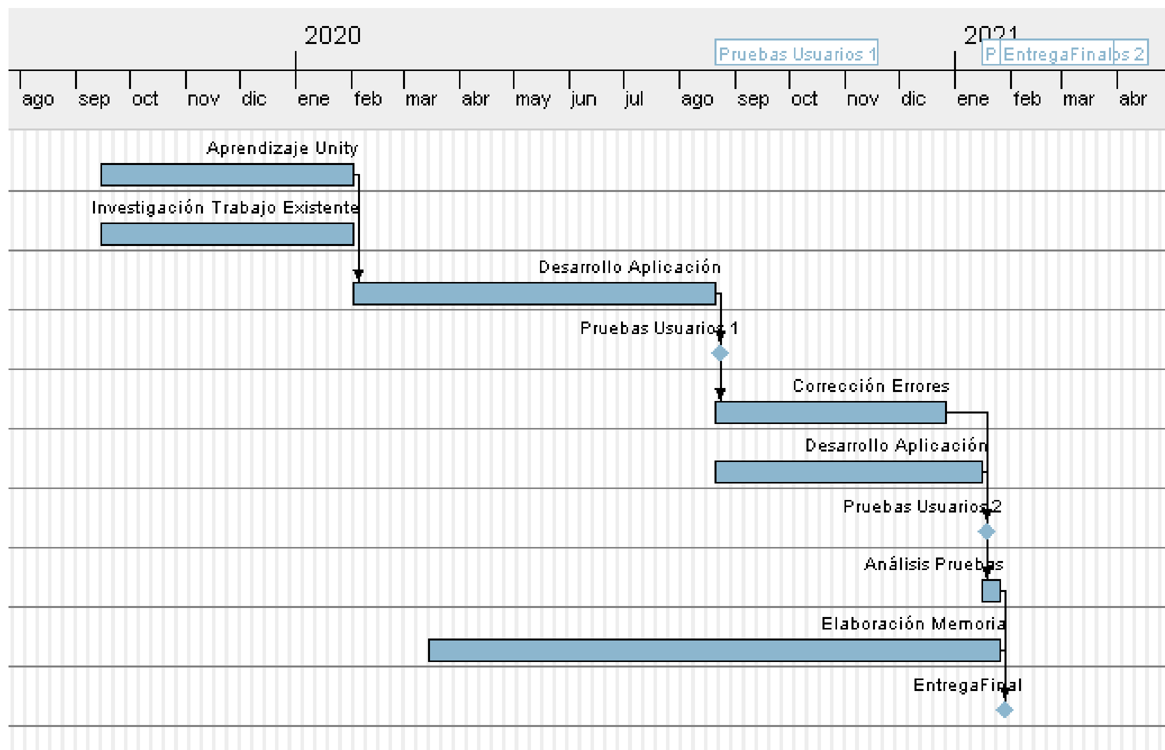


Figura 1.3: Diagrama Gantt con la planificación temporal seguida en el proyecto. Se han marcado las pruebas con usuarios como hitos, así como la entrega final. El diagrama está realizado con la herramienta de software libre *GanttProject*.

Capítulo 2

Algoritmos e Interacción

2.1. Introducción

En este capítulo se explicará la interacción pedida en cada uno de los juegos implementados en la aplicación, recordemos: ordenación por inserción, ordenación por el método de la burbuja, ordenación por selección, ordenación por mezclas (parte de la mezcla de dos listas ordenadas) y técnica de la ventana deslizante. Para cada uno de ellos, se muestran las ideas que transmite el tutorial asociado en el juego y se explica la interacción requerida, poniendo también un ejemplo. El diseño y estructura global de la aplicación, junto con diagramas UML, se explica en el siguiente capítulo; se deja este para lo específico de los algoritmos implementados. Además de lo mencionado, se explica para cada juego cómo se ha realizado el control del flujo; es decir, el control del avance de la simulación del algoritmo por parte del usuario, así como el control de la consistencia del modelo interno en todo momento respecto a la vista, esto es, respecto al estado de la estructura de datos mostrada en la pantalla.

Se ha intentado que todas estas imágenes que se adjuntan cubran lo mejor posible todas las opciones y diferentes interacciones de la aplicación. En particular, se intenta mostrar en esta memoria tanto los diferentes textos sin pistas activas como aquellos en los que se han activado las pistas (esta técnica de adaptación de la dificultad se explica con detalle en el capítulo 3).

Para ver también la detección tanto de interacciones erróneas como de interacciones correctas o las diferentes puntuaciones resultantes en función de los aciertos y fallos durante el juego y del tiempo empleado, se adjunta aquí un vídeo conjunto de toda la experiencia en la aplicación sobre la parte de los tutoriales y de los algoritmos. A lo largo del vídeo se incluyen acciones incorrectas para mostrar la activación de las pistas. El vídeo está disponible en `https://youtu.be/KvPyXDBppLQ`

Antes de pasar a analizar cada algoritmo, una observación que valdrá para todos los juegos es que las posiciones de los *arrays* se inicializan con valores aleatorios entre -99 y 99. En el caso de ordenaciones, el usuario tendrá que ordenar estos valores de menor a mayor simulando el algoritmo que esté practicando; en el caso de la técnica de la ventana deslizante, se opera con sumas y restas de los valores, ya que se ilustra la técnica mediante el problema de obtener la suma máxima de k elementos consecutivos del array.

2.2. Ordenación por inserción

En ordenación por inserción, uno de los primeros algoritmos estudiados, se distinguen tres fases en el juego, así como las tres etapas asociadas en el algoritmo que comprueba que el usuario interacciona correctamente. Este algoritmo pertenece a ordenación natural: se realizan menos operaciones cuanto más ordenado esté inicialmente el array. La complejidad es $\mathcal{O}(n^2)$.

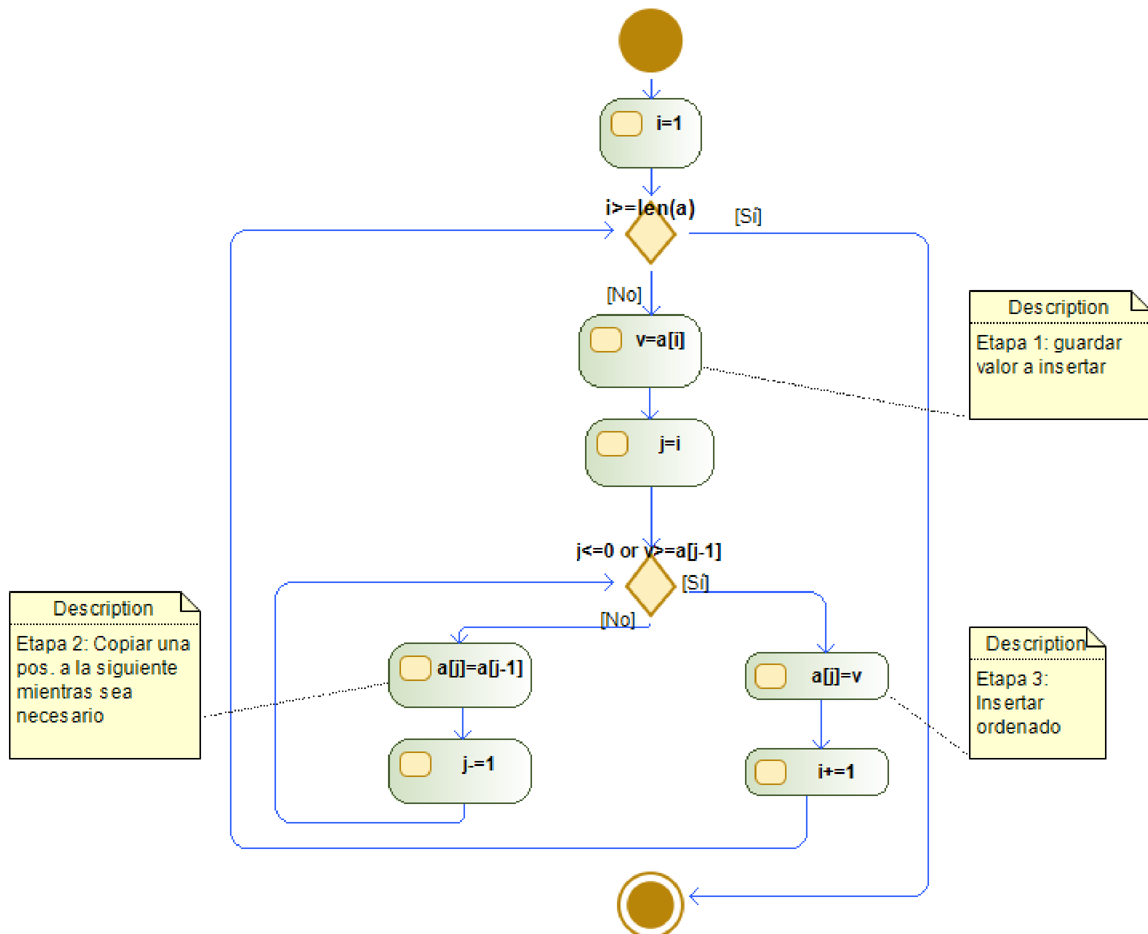


Figura 2.1: Diagrama de flujo correspondiente a ordenación por inserción, con anotaciones sobre la interacción requerida.

2.2.1. Tutorial del juego

Se intentan enseñar las bases del algoritmo y de la interacción pedida con las siguientes etapas del tutorial:

- Antes de entrar en las fases del algoritmo, se introduce ordenación por inserción. Se distinguen dos partes en el *array*, una primera que siempre está ordenada (inicialmente con el primer elemento, y está indicada en el juego usando color verde) y una segunda parte sin estar necesariamente ordenada (indicada en el juego en color blanco). Se indica que la intención es, en todo momento, insertar dentro de la parte ordenada el primer elemento de la segunda parte, hasta que el array quede completamente ordenado.

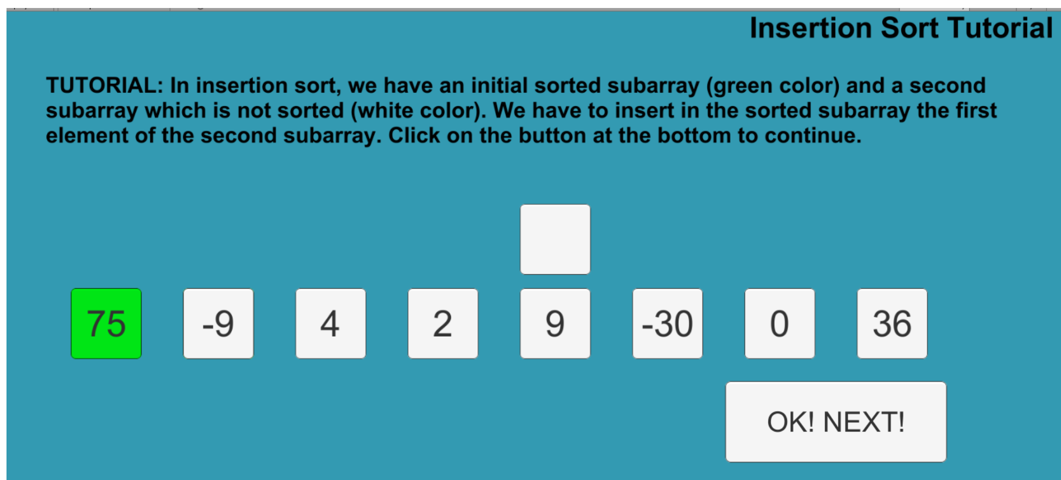


Figura 2.2: Ideas básicas iniciales en el tutorial de ordenación por inserción.

En la siguiente etapa del tutorial, comienzan las animaciones que explican las tres fases que se repiten en el algoritmo y los tres tipos de interacción usados en el juego.

1. **Elegir la siguiente posición a insertar y guardarla en una variable auxiliar.** En la etapa 1 se pide al usuario buscar la siguiente posición cuyo valor se insertará en la parte ordenada del *array*, y arrastrar dicha posición a la variable auxiliar (representada por una posición sin datos y colocada en el juego fuera del *array*). Se explica que este paso es para evitar perder el valor a insertar, ya que va a ser sobrescrito.

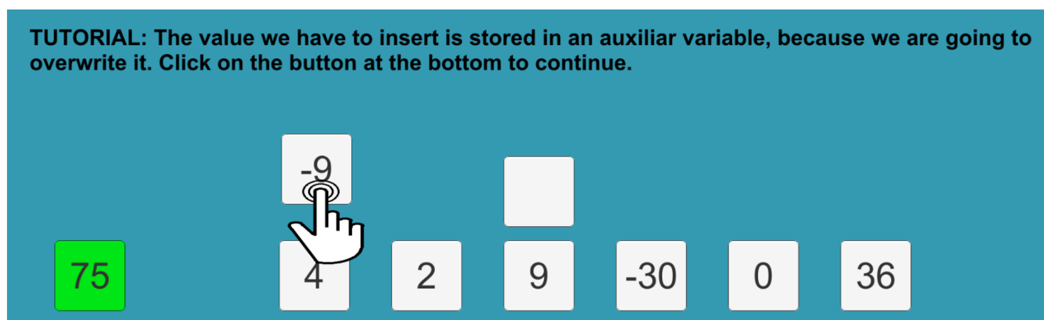


Figura 2.3: Paso 1: guardar el valor de la posición a insertar dentro de la variable auxiliar. La animación muestra el proceso de arrastrar la posición con el valor -9.

2. **"Hacer hueco" en la parte ordenada del *array* al elemento a insertar.** En este segundo paso se pide arrastrar, mientras sea necesario, la posición $j-1$ a la posición j para copiar el valor del *array* en una cierta posición a la siguiente. Este proceso, acorde con el algoritmo, se realiza comenzando por la posición i (siendo i la posición del elemento a insertar) y yendo hacia el comienzo del *array* (visualmente, hacia la izquierda). Se va indicando la posición actual en color amarillo para hacer más visual el proceso.

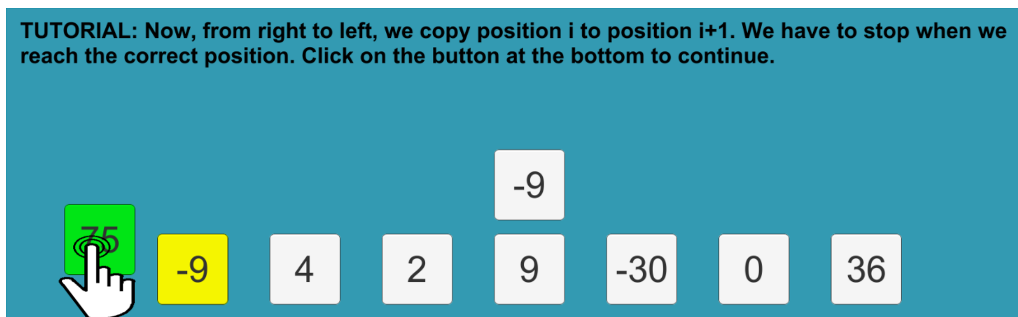


Figura 2.4: Animación que arrastra la posición 0 a la 1 para replicar el valor 75. Es necesario este desplazamiento porque $-9 < 75$. Se pierde el valor -9 del *array*, pero se mantiene visible en todo momento en la variable auxiliar.

3. **Insertar el elemento en la parte ordenada.** Cuando en el segundo paso se llegue a un elemento que ya es menor que el elemento a insertar o se alcance el inicio del *array*, la segunda fase termina (nótese que puede ocurrir que la etapa 2 sea vacía). En ese momento, podremos insertar ordenadamente el valor correspondiente a la etapa 1 de forma que la primera parte del array (ya ordenada) crece una unidad. Posteriormente, el proceso vuelve a la etapa 1. La interacción pedida es arrastrar la variable auxiliar a la posición donde se debe insertar el valor (es decir, donde se ha realizado hueco).

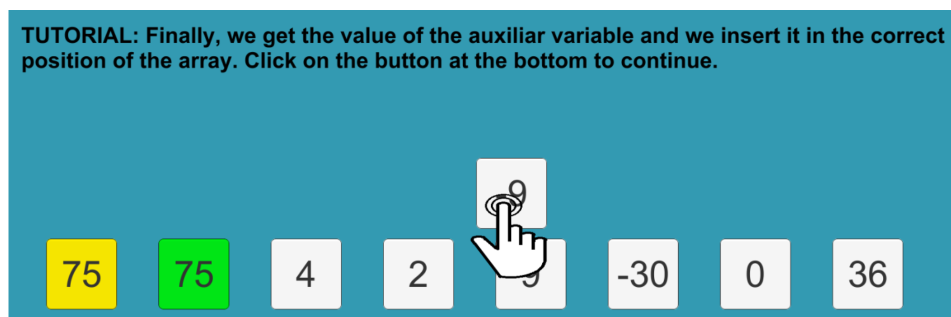


Figura 2.5: Animación arrastrando el valor de la variable auxiliar hacia la posición 0 de la parte ordenada (la correcta en este caso), copiando el valor origen en la posición destino y logrando por tanto que la parte ordenada del *array* crezca en una unidad.

2.2.2. Interacción detallada con un ejemplo

El tutorial muestra las ideas básicas del algoritmo y de la interacción pedida, pero veamos la interacción de forma más detallada, ya con un ejemplo en el que se reta al usuario a simular el algoritmo a partir de un *input* aleatorio. Se intentan abarcar todos los posibles *feedbacks* de todas las etapas, por lo que en las imágenes siguientes se ha provocado la activación y desactivación de las pistas para mostrar los diferentes textos.

Figura 2.6: Paso 1: Llevar la siguiente posición a insertar a la variable auxiliar.

Figura 2.7: Paso 2: Desplazar copiando para hacer hueco al 61 en la parte ordenada (texto sin pistas).

Figura 2.8: Paso 2: Desplazar copiando para hacer hueco al 61 en la parte ordenada (texto con pistas).

Figura 2.9: Paso 3: Arrastrar la variable auxiliar a la posición correcta para copiar el valor, y lograr un *subarray* ordenado de 2 posiciones. El proceso volvería a empezar, teniendo que insertar el valor -50 en la parte ordenada.

Un último comentario es que si la fase 2 es vacía (no hacen falta desplazamientos), entonces no se pide ninguna interacción en esta fase. Las partes 1 y 3 permanecerían iguales, aunque en este caso sean acciones innecesarias al estar guardando un valor en una variable auxiliar e inmediatamente recuperando el valor. Se ha implementado así por homogeneidad y para fomentar la idea de actuar de forma metódica.

2.2.3. Algoritmos de control y comprobación

El modelo se corresponde con la clase `InsertionSortManager`, que se comunica principalmente con `ArrayPosInsSort` y que tiene los siguientes atributos y métodos.

- **Atributos**.
    - `values`: *Array* con los valores a ordenar, que se va modificando acorde a las acciones del usuario.
    - `stepNumber`: Número de etapa en que se encuentra el proceso.
    - `nextPositionStep1`: Entero que indica la siguiente posición a insertar en la parte ordenada.
    - `numberToInsert`: Valor a insertar en la parte ordenada.
    - `movesStep2`: Número de acciones requeridas en el paso 2. Se considera acción de la segunda etapa a cada uno de los desplazamientos de una posición a la siguiente para copiar el valor.
    - `completedStep2`: Acciones ya completadas en el paso 2.
    - `posStep3`: Posición del *array* donde insertar el valor en la etapa 3.
- **Métodos**.
    - Métodos de inicialización, *get* y *set*.
    - `restart()`: Reinicia los valores para empezar un nuevo juego. La etapa pasa a ser la 1, la siguiente posición a insertar pasa a ser la segunda (posición 1) e indica también que no se ha completado ningún paso en la etapa 2.
    - `step1Completed()`: Debe ser llamado cuando el usuario completa con éxito la etapa 1. Recalcula y actualiza sus atributos, cambiando `stepNumber` al valor 2, guardando el valor que hay que insertar en el atributo `numberToInsert`, calculando la posición donde hay que insertar ese valor (mediante el método privado `findPosToInsert()` que se explica más adelante) y calculando por último el número de desplazamientos necesarios, usando la diferencia entre la posición por la que se encuentra el proceso de ordenación y la posición en la que hay que insertar el valor, es decir, `nextPositionStep1 - findPosToInsert(nextPositionStep1)`. Se usa una variable auxiliar para evitar los cálculos redundantes que provocaría llamar a la función `findPosToInsert()` dos veces.
    - `step2Position()`: Indica la siguiente posición con la que se debería interaccionar en la etapa 2.
    - `subStep2Completed()`: Debe ser llamado tras cada movimiento correcto del usuario en la etapa 2. Actualiza sus atributos, como el *array* `values`. Incrementa el atributo del número de desplazamientos ya completados en la etapa 2 y, de forma análoga, decrementa el número de desplazamientos restantes.

- `step2Completed()`: Debe ser llamado tras completar toda la etapa 2. Actualiza sus atributos indicando que la etapa es ahora la tercera, e inicializando el número de desplazamientos completados en la etapa 2 a cero, de cara a la siguiente iteración.
- `step3Completed()`: Llamado tras completar la etapa 3. Recalcula los atributos, actualizando el *array* mediante `values[posStep3] = numberToInsert;` y devolviendo el estado a la etapa 1.
- `finished()`: Devuelve un valor *bool* que indica si el proceso de ordenación ha terminado. Basta comparar si se está en la etapa 1 pero la siguiente posición a insertar coincide ya con la longitud del array (este tamaño viene dado como constante en la clase `Utilities`).
- `findPosToInsert()`: Método privado que ayuda a recalcular los atributos tras finalizar la etapa 1. Se realiza una búsqueda desde una cierta posición origen del *array* hacia el comienzo del mismo, hasta que se alcanza el inicio de la estructura o hasta que se encuentra una posición tal que su valor es menor o igual que el valor de la posición origen. Es decir, este método calcula la posición objetivo a la que tiene que llegar el siguiente elemento a insertar.

```
private static int findPosToInsert(int begin) {
    int pos = begin;
    while (pos > 0 && values[begin] < values[pos-1]) {
        --pos;
    }
    return pos;
}
```

Figura 2.10: Código del método privado `findPosToInsert()`

.

2.3. Ordenación por selección

Otro de los algoritmos de complejidad cuadrática es la ordenación por selección. También se distingue entre una parte inicial ordenada (inicialmente vacía e indicada en el juego en verde) y el resto del *array* (indicado en el juego en blanco); el algoritmo consiste en buscar el mínimo de esta segunda parte e insertarlo en la primera parte, intercambiando los valores de las dos posiciones que intervienen. Una propiedad de este algoritmo es que siempre realiza el mismo número de operaciones (con la salvedad de posibles intercambios "triviales" (una posición con sí misma) que se podrían evitar).

La interacción que se pide al usuario en este juego es, probablemente, la más sencilla de todas, como se verá a continuación. Como ya se ha dicho, esta implementación de ordenación por selección realiza siempre el mismo número de operaciones. Esto se representa claramente en el *feedback* que se da al completar el juego, ya que siempre se necesitan el mismo número de acciones. Sin embargo, esto no debe confundir al usuario: la interacción en este juego permite que el usuario acceda al elemento mínimo de la segunda parte de un vistazo, sin consumir acciones. Estas operaciones en una computadora sí hay que realizarlas.

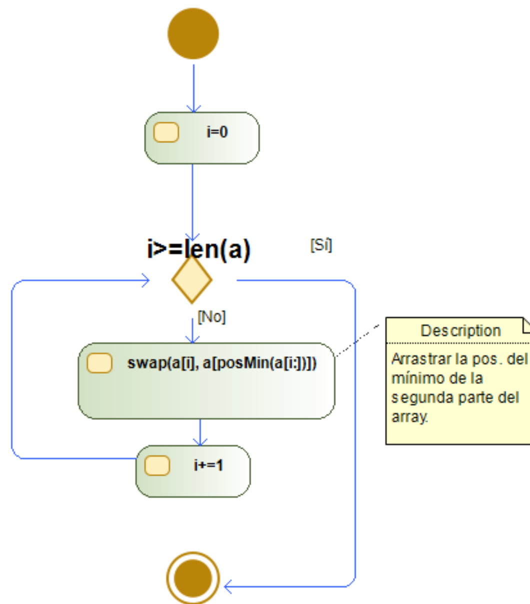


Figura 2.11: Diagrama de flujo de ordenación por selección, con anotaciones sobre la interacción del juego.

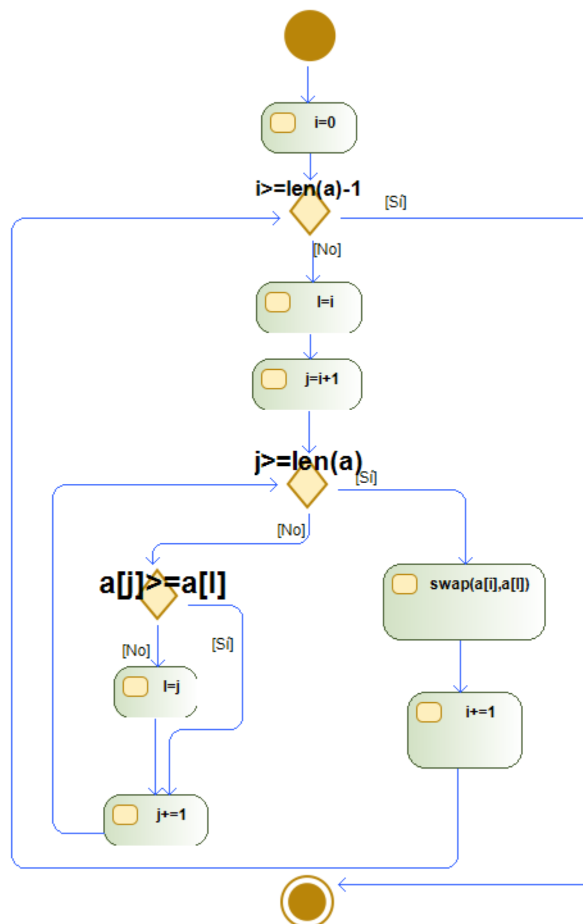


Figura 2.12: Diagrama de flujo de más bajo nivel de ordenación por selección, detallando la instrucción del diagrama anterior `swap(a[i], a[posMin(a[i:])])`.

2.3.1. Tutorial del juego

- En primer lugar, se presenta una introducción a esta ordenación:

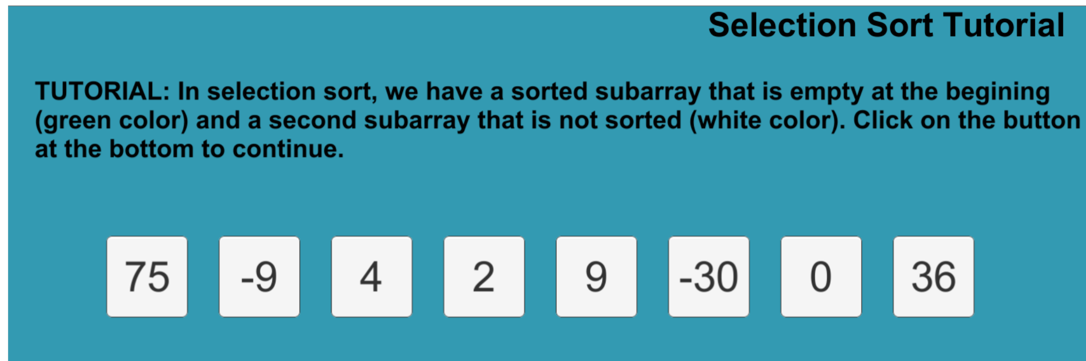


Figura 2.13: Introducción del tutorial de ordenación por selección.

Una vez comenzado el juego, se presentan dos opciones de interacción:

1. Si el mínimo de la segunda parte no queda ya en la primera posición de esta parte no necesariamente ordenada, se busca dicho valor y se arrastra a la siguiente posición a ordenar:

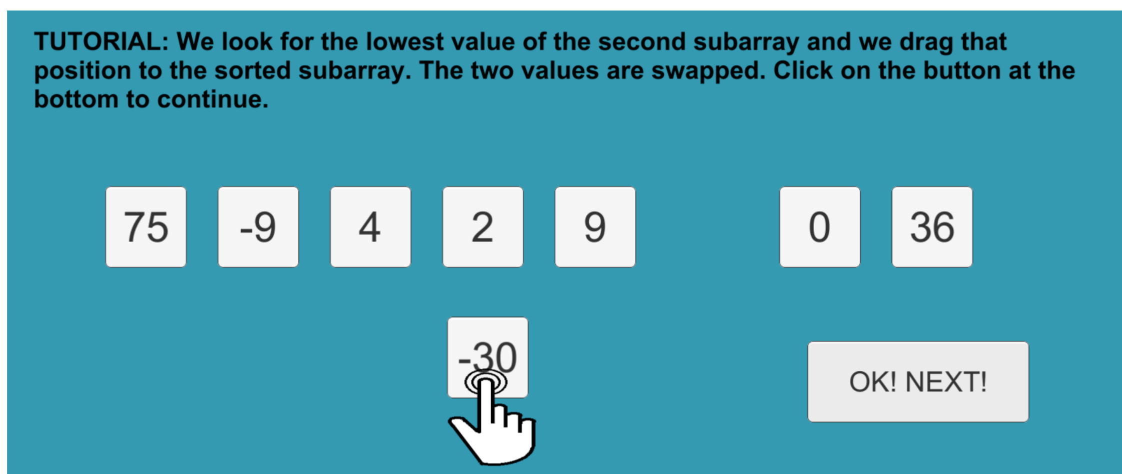


Figura 2.14: El mínimo de la segunda parte (al inicio, la segunda parte coincide con todo el *array*) es arrastrado a la siguiente posición a ordenar.

2. Si el mínimo de la segunda parte en alguna etapa coincide con su posición destino (es decir, queda ya ordenado), la interacción que se requiere es simplemente hacer *click* en esa posición:

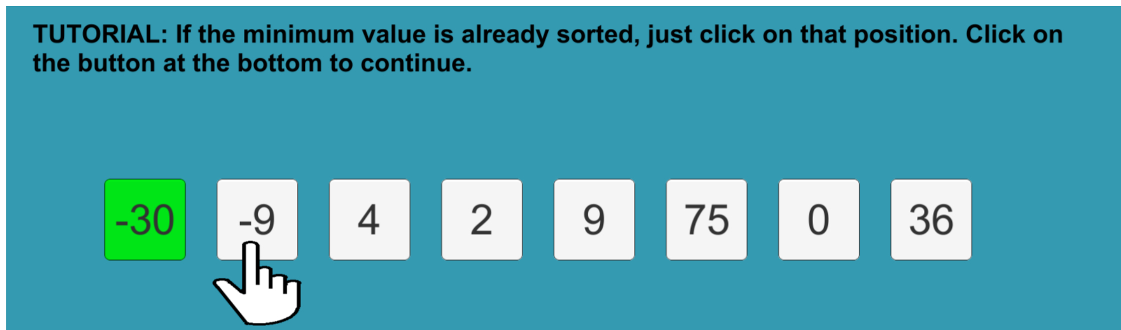


Figura 2.15: Aquí, el mínimo de la segunda parte coincide con su posición destino. Basta con hacer *click* en el valor -9

2.3.2. Interacción detallada con un ejemplo

El comienzo de la resolución de un *input* aleatorio usando estas dos interacciones que introduce el tutorial sería el siguiente:

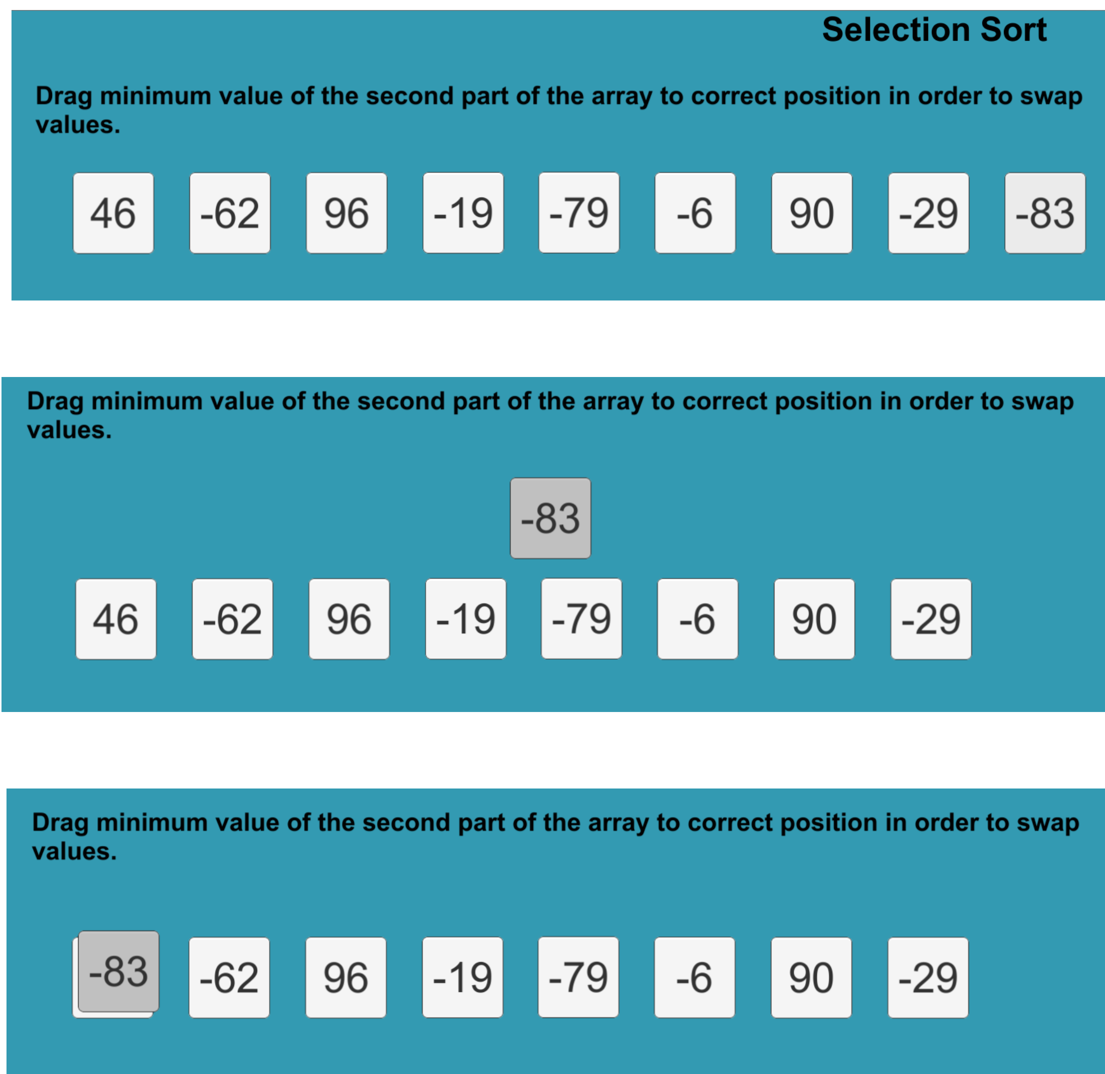


Figura 2.16: El mínimo de la segunda parte (al inicio, todo el *array*) es la posición 8, de valor -83. Hay usar la primera interacción, y arrastrar dicha posición hacia la 0 para intercambiar valores con el 46. Se muestra el texto sin pistas.

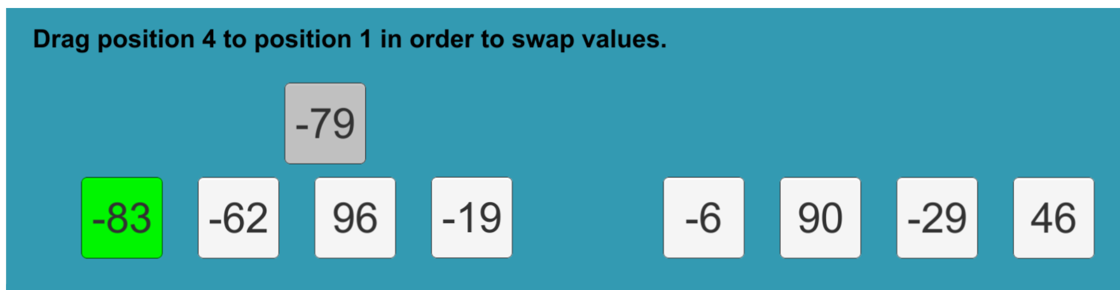


Figura 2.17: El valor 46 ahora está en la posición 8. El mínimo de la segunda parte pasa a ser el -79, por lo que de nuevo se arrastra, ahora a la segunda posición (posición 1). En esta imagen se muestra el texto con pistas, donde (si el usuario se queda bloqueado) se indica que la interacción esperada es mover la posición 4 hacia la posición 1.

Siguiendo este sencillo mecanismo, la lista queda ordenada siempre en un número de movimientos del usuario igual a la longitud de la lista, aunque hay que recordar la observación sobre la búsqueda del mínimo: se ha decidido no implementar detalles sobre esta búsqueda para centrar la atención en el algoritmo de ordenación. Se aligera así un proceso que seguramente resultaría tedioso para el usuario. Por tanto, este juego se ha implementado con un nivel de abstracción algo superior que el anterior (ordenación por inserción).

2.3.3. Algoritmos de control y comprobación

El modelo es la clase `SelectionSortManager`, que se comunica con `ArrayPosSelSort` y que tiene los siguientes atributos y métodos.

- **Atributos**.
  - `values`: *Array* con los valores a ordenar. Se va actualizando acorde a las acciones del usuario.
  - `pos1swap`: Primera posición que interviene en cada intercambio de dos elementos.
  - `pos2swap`: Segunda posición que interviene en cada intercambio de dos elementos.
- **Métodos**.
  - Métodos de inicialización, *get* y *set*.
  - `iniPos2Swap()`: Inicializa el atributo `pos2Swap`, lanzando una búsqueda del mínimo en todo el *array*. Para ello, se ayuda de `findPosMinimumFrom()`, método privado que se comentará posteriormente. En concreto, como se quiere considerar todo el *array*, la llamada es `pos2swap = findPosMinimumFrom(0);`
  - `stepCompleted()`: Llamado tras cada acción correcta del usuarios, recalcula los atributos. En concreto, se intercambian `values[pos1swap]` y `values[pos2swap]`, se incrementa `pos1swap` para preparar la siguiente acción y, si el proceso no ha terminado, se calcula también `pos2swap` para la siguiente acción, mediante una llamada `findPosMinimumFrom(pos1swap);`
  - `finished()`: Indica si el proceso de ordenación ha terminado. Basta comparar `pos1swap` con la última posición del *array*, es decir, `Utilities.LENGTH - 1`.
  - `restart()`: Reinicia el modelo, inicializando los atributos. La primera posición que interviene en el intercambio pasa a ser la 0. Para calcular la restante, se ejecuta una búsqueda del mínimo en todo el *array*.

- `findPosMinimumFrom()`: Método privado usado en los cálculos de los atributos tras cada interacción. Lanza una búsqueda del mínimo en cierto segmento del *array*. En concreto, desde la posición dada en el parámetro hasta el final de la estructura.

2.4. Ordenación por el método de la burbuja

La ordenación por el método de la burbuja (más conocida por su nombre en inglés, *bubble sort*) es otro de los algoritmos de complejidad cuadrática $\mathcal{O}(n^2)$. En el artículo [12] se discute sobre si es adecuado enseñar estos algoritmos de complejidad cuadrática, y en especial *bubble sort*, ya que es el que peor se comporta entre los tres típicos (burbuja, inserción y selección). Sin embargo, lo cierto es que en muchos centros sí se enseñan, y esta actitud se puede defender al intentar hacer que el aprendizaje sea gradual.

Esta aplicación incluye el estudio de *bubble sort*, ya que (como también se comenta en [12]) este algoritmo es de los más famosos entre los básicos de las ciencias de la computación.

Hay diferentes implementaciones de *bubble sort*. La que este juego considera opera del final del *array* hacia el comienzo, comparando la posición relativa de cada par de elementos que se encuentra. Si el par está ordenado, se sigue avanzando hasta el comienzo del *array*. Si el par está desordenado, los elementos se intercambian (se suele decir que "la burbuja sube"). Cuando la burbuja alcanza la primera parte del *array* (aquella posición tal que estamos seguros que hasta ahí tenemos un *array* ordenado), se comienza una nueva comparación de pares de elementos desde el final del *array* hacia el comienzo.

Por último, hay que mencionar que la implementación que este juego considera de *bubble sort* sí tiene en cuenta la optimización de detenerse cuando en un recorrido entero del bucle interno no se realizan intercambios (lo que significa que el *array* está ya ordenado). Si esto sucede, el juego acaba automáticamente; es decir, esto no tiene que tenerlo en cuenta el usuario.

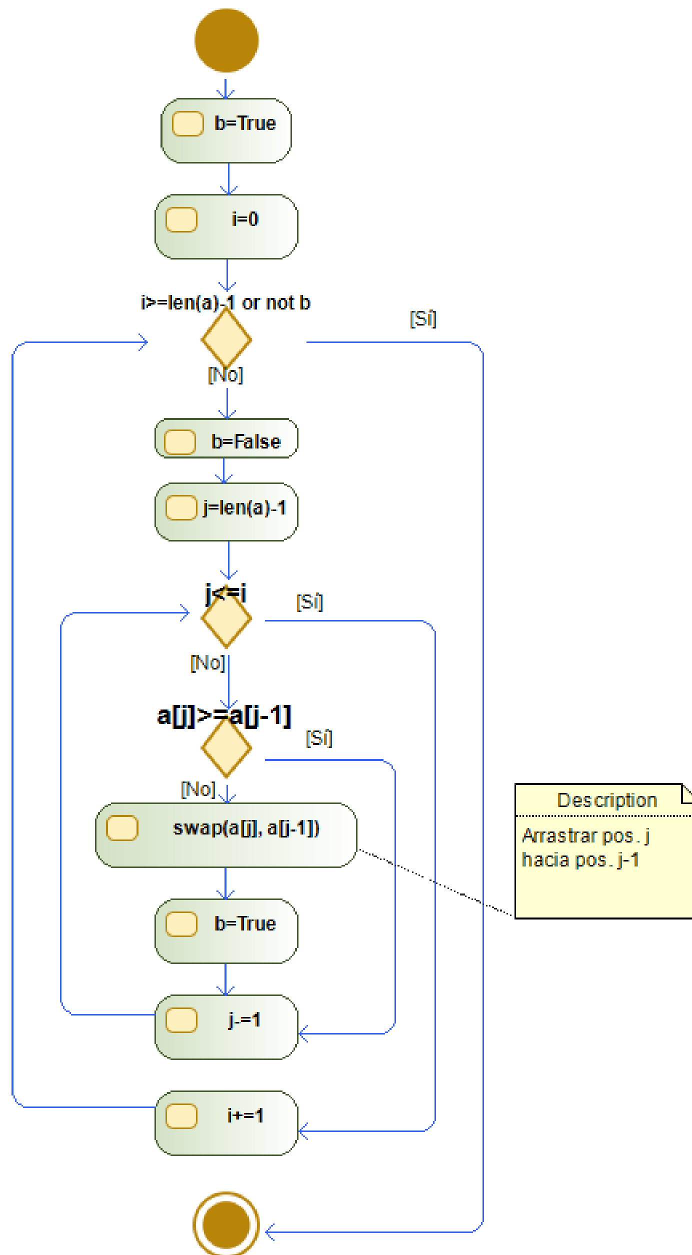


Figura 2.18: Diagrama de flujo del método de la burbuja.

2.4.1. Tutorial del juego

Bubble Sort Tutorial

TUTORIAL: This bubble sort implementation works from right to left of the array, comparing two positions. If these two positions are not sorted, they are swapped. Click on the button at the bottom to continue.

75 -9 4 2 9 -30 0 36

Figura 2.19: Inicialmente, se explica cómo funciona la implementación de *bubble sort* tratada.

Pasando a la siguiente animación del tutorial, se muestran dos flechas en el extremo derecho que van avanzando hacia el comienzo del *array*, intercambiando los dos elementos que intervienen en la comparación si están desordenados entre sí.

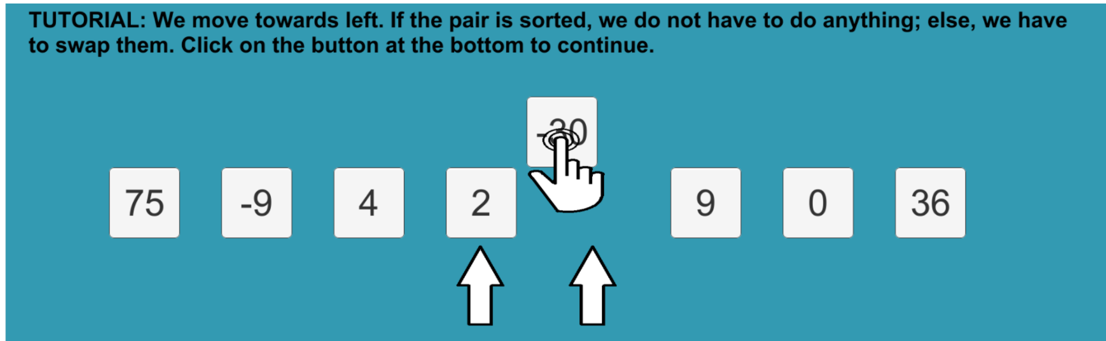
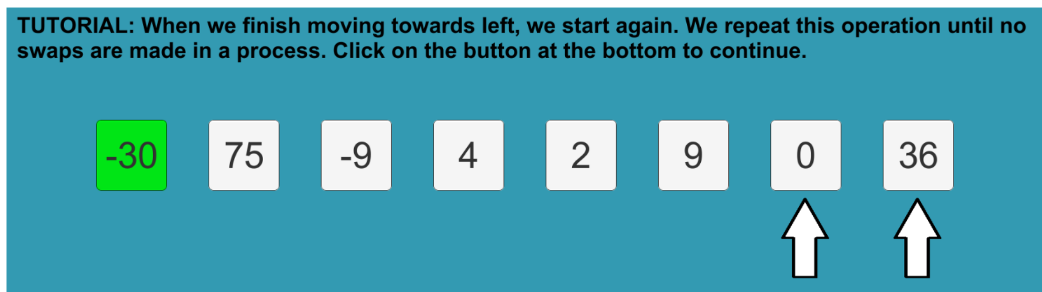


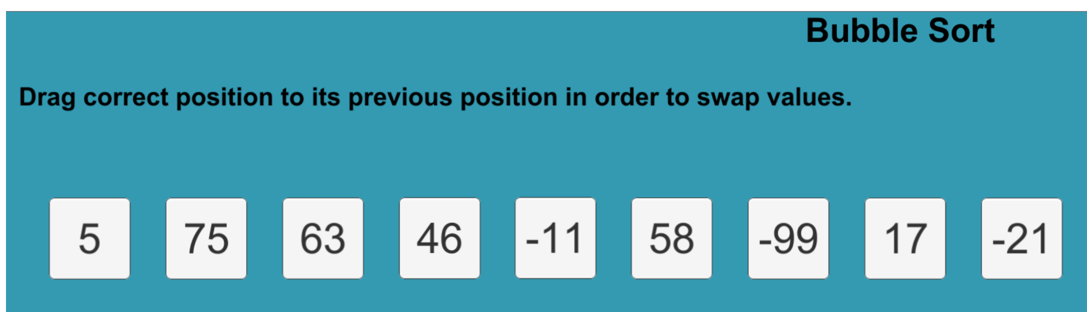
Figura 2.20: En este caso, se representa la comparación entre 0 y 36 (están ordenados y simplemente se desplazan las flechas), posteriormente entre -30 y 0 (igual que antes, están ordenados), entre 9 y -30 (no están ordenados, por lo que se intercambian arrastrando el -30 a su posición anterior); finalmente, entre 2 y -30 (vuelven a no estar ordenados, por lo que se intercambian llevando el -30 a su posición anterior, que es justamente el paso que se representa en la imagen).

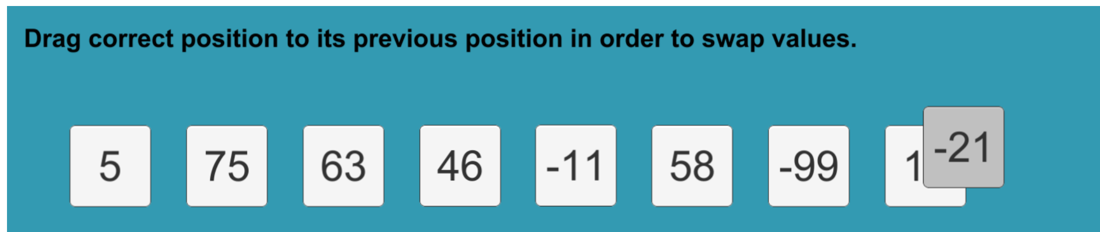
Cuando la burbuja llega a su posición destino, la animación muestra ambas flechas volviendo al extremo derecho (final del *array*) para indicar que el proceso vuelve a empezar. Se contará con una posición adicional ordenada, que se marca en color verde en el juego.



2.4.2. Interacción detallada con un ejemplo

De nuevo, veamos un ejemplo de acciones del usuario en el juego en sí, para ver los diferentes textos que se muestran (con ayudas y sin ellas) así como la interacción requerida con más detalle:





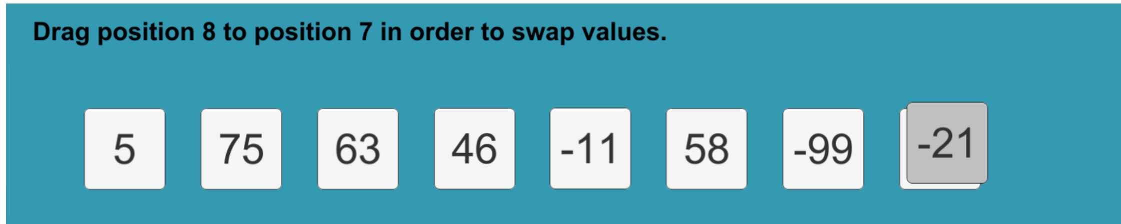
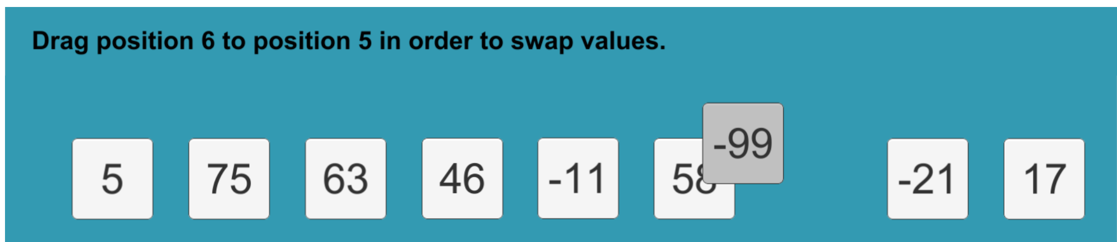


Figura 2.21: $17 > -21$, luego este par está desordenado. Hay que arrastrar la posición de valor -21 a la posición anterior para intercambiar valores. En la última de estas imágenes se muestra el texto cuando se activan las pistas, que indica con detalle qué movimiento hay que realizar.



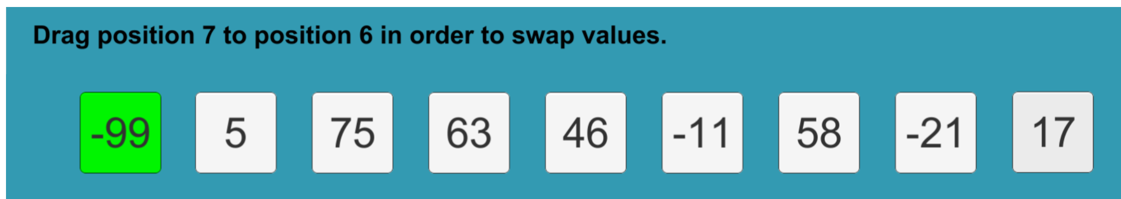


Figura 2.22: Se sigue avanzando hacia el comienzo del *array*, buscando el siguiente par que esté desordenado. El valor -99 intercambia valores con el 58, y posteriormente se va intercambiando con todos los siguientes, ya que es el valor que debe llegar al comienzo al ser el menor de los restantes. Como se muestra en la última imagen, las posiciones que están con seguridad ordenadas al comienzo del *array* se representan en verde; el texto muestra el siguiente intercambio (tras considerar de nuevo en la parte final del array).

2.4.3. Algoritmos de control y comprobación

El modelo se corresponde con la clase `BubbleSortManager`, que se comunica con el correspondiente *script ArrayPos*, es decir, `ArrayPosBubbleSort`. Los atributos y métodos de `BubbleSortManager` son los siguientes.

- **Atributos**.
  - `values`: Array con los elementos, se va actualizando al avanzar el proceso.

- `posSorted`: Posición hasta la cual se tiene seguridad que el comienzo del *array* está ordenado. Es decir, el rango [0, `posSorted`) está ordenado.
- `posBubble`: Posición del elemento que hay que intercambiar con su posición anterior, es decir, con el que hay que interactuar en cada momento.

- **Métodos**.
  - Métodos de inicialización, *get* y *set*.
  - `initialize()`: Inicializa el atributo `posBubble` con ayuda del método privado que se explicará posteriormente `calculatePosBubble()`, pasando como parámetro el valor `Utilities.LENGTH-1` para que la búsqueda se realice en toda la estructura.
  - `subStepCompleted()`: Actualiza los atributos tras cada interacción correcta, intercambiando `values[posBubble]` y `values[posBubble-1]` y recalculando la siguiente posición con la que interaccionar mediante `calculatePosBubble()` con el parámetro adecuado.
  - `stepCompleted()`: Actualiza los atributos cada vez que el usuario completa un recorrido desde el final del *array* hasta el comienzo intercambiando posiciones. Tendremos una posición adicional ordenada, y para la siguiente posición con la que hay que interaccionar se lanza una búsqueda en todo el *array* de nuevo mediante `calculatePosBubble()`.
  - `finished()`: Indica si el proceso de ordenación ha terminado. Se implementa comprobando que la búsqueda del siguiente elemento con el que interaccionar llega al inicio de la estructura, es decir, el *array* ya está ordenado.
  - `restart()`: Reinicia el modelo, inicializando los atributos. Se actualiza a 0 la posición hasta la cual tenemos un *subarray* ordenado con seguridad.
  - `calculatePosBubble()`: Método privado auxiliar que busca desde la posición indicada en el parámetro hacia el inicio del *array* el primer par de elementos consecutivos (a, b) tal que $a > b$. Nótese que este método auxiliar se usa en muchos de los anteriormente explicados.

```
private static int calculatePosBubble(int from)
{
    int sol = from;
    while (sol > 0 && values[sol - 1] <= values[sol])
    {
        --sol;
    }
    return sol;
}
```

Figura 2.23: Código del método privado auxiliar que permite implementar muchos de los métodos restantes de esta clase.

2.5. Ordenación por mezclas (1)

La ordenación por mezclas (en inglés, *merge sort*) es ya un algoritmo óptimo en cuanto a complejidad en tiempo ($\mathcal{O}(n \log n)$). Este algoritmo, perteneciente al grupo de los algoritmos *"Divide y vencerás"*, fue inventado por John von Neumann en 1945. Se basa en dos ideas:

recursión y mezcla de dos listas ordenadas. En la escena *MergeSort1* del juego se pretende que el usuario trabaje sobre la mezcla de dos *arrays* ordenados.

En esta escena, al principio, se presentan dos *arrays* de cuatro elementos ya ordenados, separados entre sí. Debajo, se coloca la lista (inicialmente en blanco) sobre la que se construirá el *array* ordenado resultado de la mezcla de las dos listas superiores.

La interacción consiste en pensar el índice que se lleva en cada lista en un momento determinado (aunque esto se indica claramente usando colores). Ahora, se compararan los elementos de cada lista en sus respectivos índices (es decir, `lista0[indice0]` y `lista1[indice1]`). Tras elegir el menor de los dos, dicha posición se arrastra a la siguiente posición a completar del array inferior.

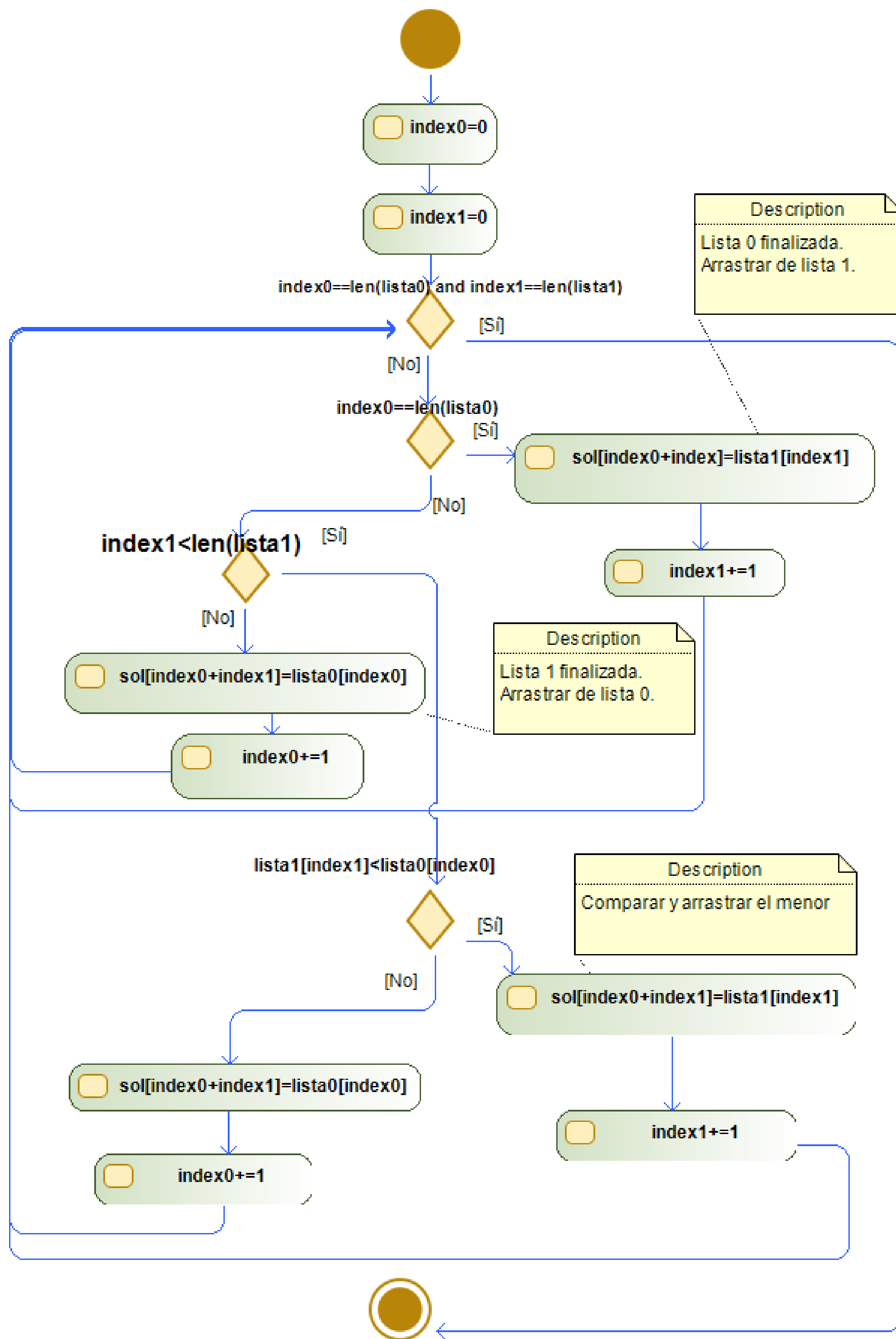


Figura 2.24: Diagrama de flujo de la mezcla de dos listas ordenadas.

2.5.1. Tutorial del juego

El tutorial de este juego explica mediante animaciones lo que se ha indicado en la introducción anterior. Inicialmente se explica el propósito. Después, comienzan las animaciones que muestran el proceso.

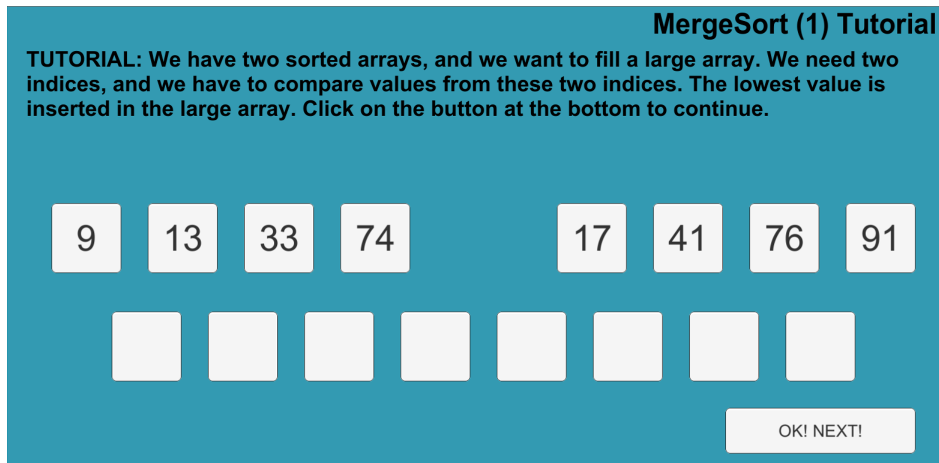


Figura 2.25: En primer lugar, se introduce la mezcla de dos listas ordenadas.

1. Ambos índices comienzan en cero, y se comparan por tanto las posiciones iniciales de ambos *arrays*. El valor inferior es el que se inserta en el *array* resultado, mediante una interacción *drag-and-drop* como habitualmente.

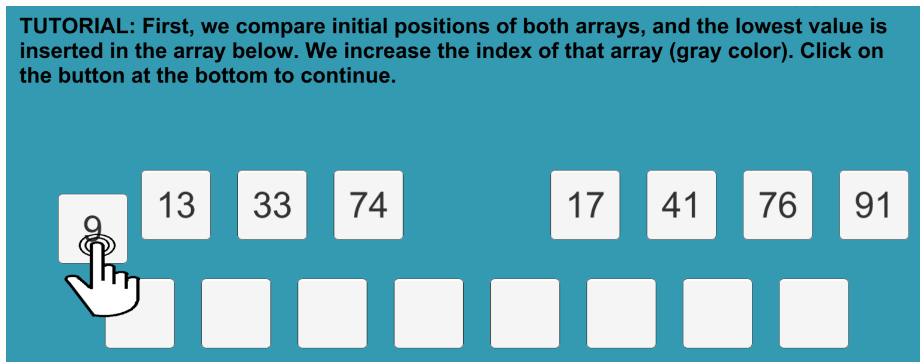


Figura 2.26: `lista0[indice0] = lista0[0]` $= 9 < 17 =$ `lista1[0] = lista1[indice1]`, por lo que se arrastra la primera posición del *array* 0 a la estructura resultado. Se incrementa el índice de la primera lista (`indice0`), indicándolo en el juego con posiciones en gris (ver siguiente punto).

2. Se comparan de nuevo los valores apuntados por los índices correspondientes (uno de los dos índices tiene que haber cambiado). En el tutorial, la animación muestra el desplazamiento del segundo elemento, de nuevo tomando como origen la primera lista.

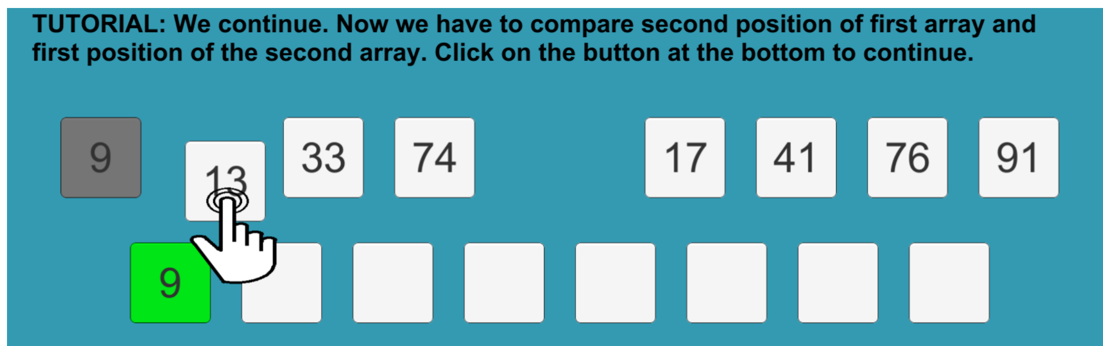
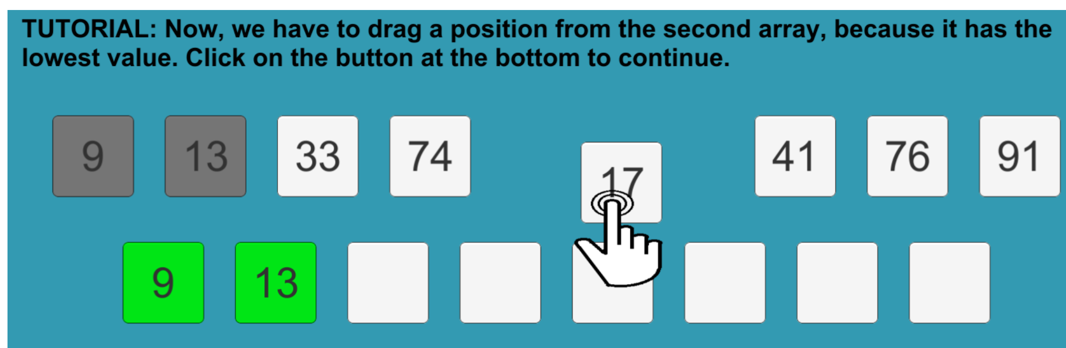
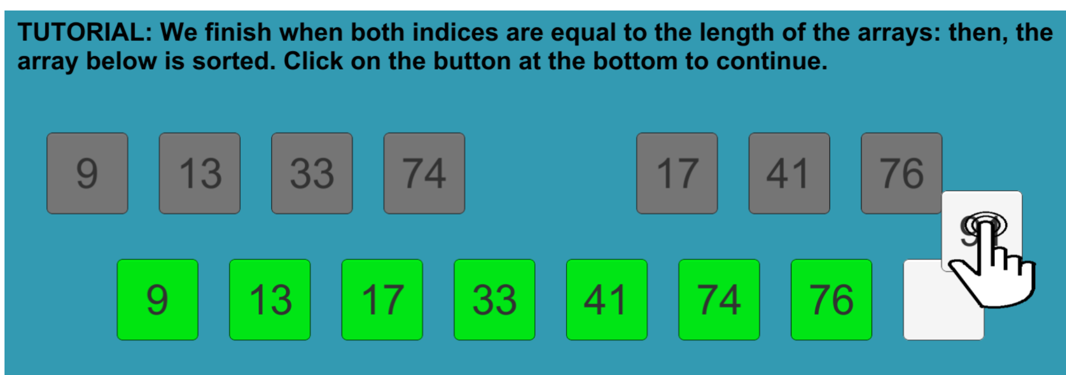


Figura 2.27: El tutorial pasa a mostrar que `indice0 == 1` utilizando una posición en gris en la primera lista. Como 13 $<$ 17, la animación muestra que se debe arrastrar el valor 13 al *array* resultado.

3. Para ilustrar un ejemplo en el que se toma como origen la segunda lista, el tutorial continúa un paso más.



4. Finalmente, aunque es muy intuitivo cuándo va a acabar el juego, el tutorial termina con la animación del movimiento final, más bien para poder explicar en el texto asociado que el proceso acaba cuando los índices de ambas listas origen son iguales a su longitud:



2.5.2. Interacción detallada con un ejemplo

En la subsección anterior ya se han explicado con suficiente detalle las interacciones pedidas, usando el ejemplo del tutorial (que es un ejemplo fijo (es decir, no aleatorio) y seleccionado con cuidado para ser ilustrativo). Sin embargo, para ver el texto que se muestra al usuario durante el juego, tanto en su versión sin pistas como en su versión con pistas si el usuario se bloquea en un paso, veamos las dos primeras interacciones de otro ejemplo, con *input* aleatorio:

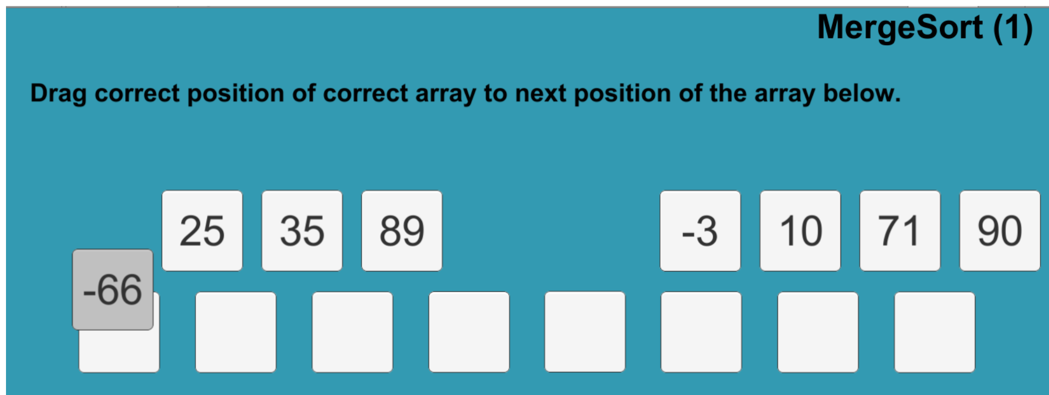


Figura 2.28: Primera interacción, con el texto sin pistas: se indica que hay que arrastrar cierta posición de cierto *array* a las posiciones inferiores.

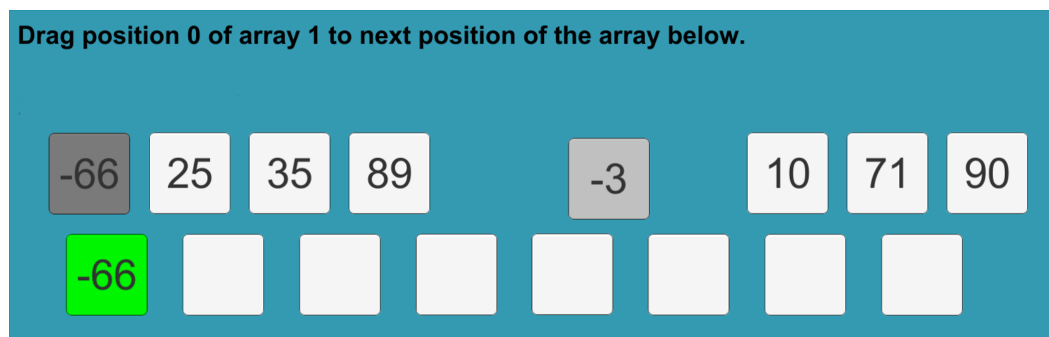


Figura 2.29: Segunda interacción, provocando que se activen las pistas al fallar varias veces: se indica la posición y la lista con la que se debe interaccionar.

2.5.3. Algoritmos de control y comprobación

La inicialización de los valores en este caso es ligeramente distinta al resto: como los dos *arrays* a mezclar tienen que estar ya ordenados, cada posición no se puede inicializar de forma individual, ya que depende de otras.

La clase correspondiente al modelo de esta pantalla es `MergeSort1Manager`, que interacciona principalmente con `ArrayPosMergeSort1`. Se ha creado en el modelo la estructura `interactionMergeSort1` para encapsular la información de la siguiente interacción que se le pedirá al usuario: número de lista donde está la siguiente posición a insertar, así como el índice de la posición. Los atributos y métodos de `MergeSort1Manager` son los siguientes.

- **Atributos**.
  - `values0`, `values1`: Listas con los valores a mezclar.
  - `index0`, `index1`: Índices de las respectivas listas.
- **Métodos**.
  - Métodos de inicialización y *get*.
  - `stepCompleted()`: Llamado tras una interacción correcta, incrementa el índice adecuado.

- `getNextInteraction()`: Devuelve un objeto `interactionMergeSort1` que encapsula la siguiente interacción que debe hacer el usuario. Se comprueba si alguna de las dos listas ha sido agotada. Si ninguna de las dos está agotada, se busca aquella tal que su índice apunta al valor menor. Se adjunta imagen de este método.

```
public static interactionMergeSort1 getNextInteraction()
{
    if (index0 >= Utilities.LENGTH_MSORT1)
        return new interactionMergeSort1(1, index1);
    if (index1 >= Utilities.LENGTH_MSORT1)
        return new interactionMergeSort1(0, index0);
    return values0[index0] <= values1[index1] ?
        new interactionMergeSort1(0, index0) :
        new interactionMergeSort1(1, index1);
}
```

- `getNextIndex()`: Devuelve la siguiente posición a rellenar del *array* resultado (`index0 + index1`).
- `finished()`: Indica si el proceso ha terminado, comprobando si el índice de ambas listas es igual a su longitud.
- `restart()`: Reinicia el juego, inicializando de nuevo los atributos. Los índices vuelven a 0, y las listas se crean con nuevos valores aleatorios.

2.6. Técnica de ventana deslizante

2.6.1. Introducción a la técnica

Informalmente, la ventana deslizante se podría explicar como una técnica en la que una subestructura se desplaza por una estructura mayor. Por ejemplo, un *subarray* que se desplaza por un *array* (es decir, en cada caso se tienen en cuenta solamente ciertas posiciones consecutivas del *array* mayor); también podría ser una pequeña matriz que se desplaza por una matriz mayor. En definitiva, la clave de esta técnica es que en cada movimiento la subestructura solamente cambia algunas de sus características, y no todas, por lo que no hace falta recalcular la operación asociada a la subestructura en cada paso, sino hacer un cálculo inicial y posteriormente ajustar el cálculo solamente teniendo en cuenta los cambios que se han realizado.

Expliquemos esto con más detalle, usando el ejemplo que se ha implementado en el juego: dado un *array* de longitud n cuyos valores son números enteros, queremos encontrar la suma máxima de k elementos consecutivos (para $k \leq n$). Una posible implementación que devuelve el resultado correcto consiste en coger los k primeros elementos (posiciones 0 a $k-1$, ambas incluidas), sumarlos e inicializar el máximo a este valor. Posteriormente, se toman los elementos correspondientes a las posiciones 1 hasta la k, se suman y se compara el resultado con el máximo inicializado anteriormente, tomando como máximo provisional el mayor valor de entre estos dos. El proceso continúa, calculando en cada caso la suma de k elementos consecutivos del *array* subyacente y comparando el resultado con el máximo provisional. Al final, obtenemos el resultado correcto mediante este algoritmo de complejidad $\mathcal{O}(n \cdot k)$. Pero esta complejidad se puede reducir, ya que se están realizando muchos cálculos redundantes: al pasar de sumar las posiciones i hasta $k+i-1$ (supongamos un ejemplo y un valor de

i tal que $k+i$ siga en rango) a sumar las posiciones $i+1$ hasta $k+i$ (ambas incluidas), solamente están cambiando en la suma los términos correspondientes a las posiciones i (cuyo valor desaparece en la suma) y $k+i$ (cuyo valor no estaba en la suma, y ahora aparece). Por tanto, la forma adecuada de implementar el proceso sería hacer el cálculo inicial, pero después basta con restar a la suma ya calculada el valor de la posición que deja de ser considerada, y sumar a la suma ya calculada el valor de la posición que entra a ser considerada. Podemos ver esto como una ventana (subestructura, *subarray*) de longitud k, que abarca inicialmente las posiciones i a $k+i-1$, y que se desplaza una unidad "hacia adelante": ahora abarca las posiciones $i+1$ a $k+i$. Bastaría entonces restar el extremo izquierdo de la ventana original, y sumar el valor de la posición del extremo derecho de la misma, que ha pasado a considerarse. La complejidad en este caso pasa a $\mathcal{O}(n)$.

El caso de una matriz (*array* bidimensional) es similar: al mover una submatriz una unidad (una fila más, una fila menos, una columna más o una columna menos), en general (salvo casos extremos, como considerar una submatriz fila y moverla por filas) hay elementos que siguen estando dentro de dicha submatriz, mientras que otros sí que cambian. Bastaría hacer el ajuste considerando los elementos que deja de abarcar la subestructura y aquellos que pasan a estar abarcados por ella.

Esta técnica también se puede adaptar a problemas que no presentan una relación tan obvia con la misma: encontrar anagramas de cierta palabra dentro de un texto. Si buscamos anagramas de una palabra de longitud k en un texto de longitud n, se desplaza por el texto (que es un *array* de caracteres) un *subarray* de longitud k. En este caso, la operación asociada a la ventana es la frecuencia con que aparecen las letras: inicialmente se calcula la frecuencia de aparición de los k primeros caracteres, pero al desplazar la ventana basta con restar una aparición al caracter que sale de la subestructura, y sumar una aparición al caracter que entra. Si las frecuencias de aparición de la ventana coinciden con las frecuencias de aparición de la palabra de la que buscamos anagramas, hemos encontrado un anagrama, es decir, una permutación de los mismos caracteres.

```
 2 def ventana(a, k):
 3     i=0
 4     suma=0
 5     # Etapa 1: Suma inicial.
 6     # Arrastrar al boton suma.
 7     while i < k:
 8         suma += a[i]
 9         i += 1
10     maxi = suma
11     # Etapa 2: Desplazar ventana.
12     while i < len(a):
13         # Etapa 2.1: Resta.
14         # Arrastrar al boton resta.
15         suma -= a[i-k]
16         # Etapa 2.2: Suma.
17         # Arrastrar al boton suma.
18         suma += a[i]
19         i += 1
20         if suma > maxi:
21             # Etapa 2.3: Nuevo maximo.
22             # Pulsar boton nuevo maximo.
23             maxi = suma
24     return maxi
25
```

Figura 2.30: Código *Python* para el problema de la suma máxima de k elementos consecutivos de un *array*. Se adjuntan comentarios con las etapas que se distinguen en la interacción.

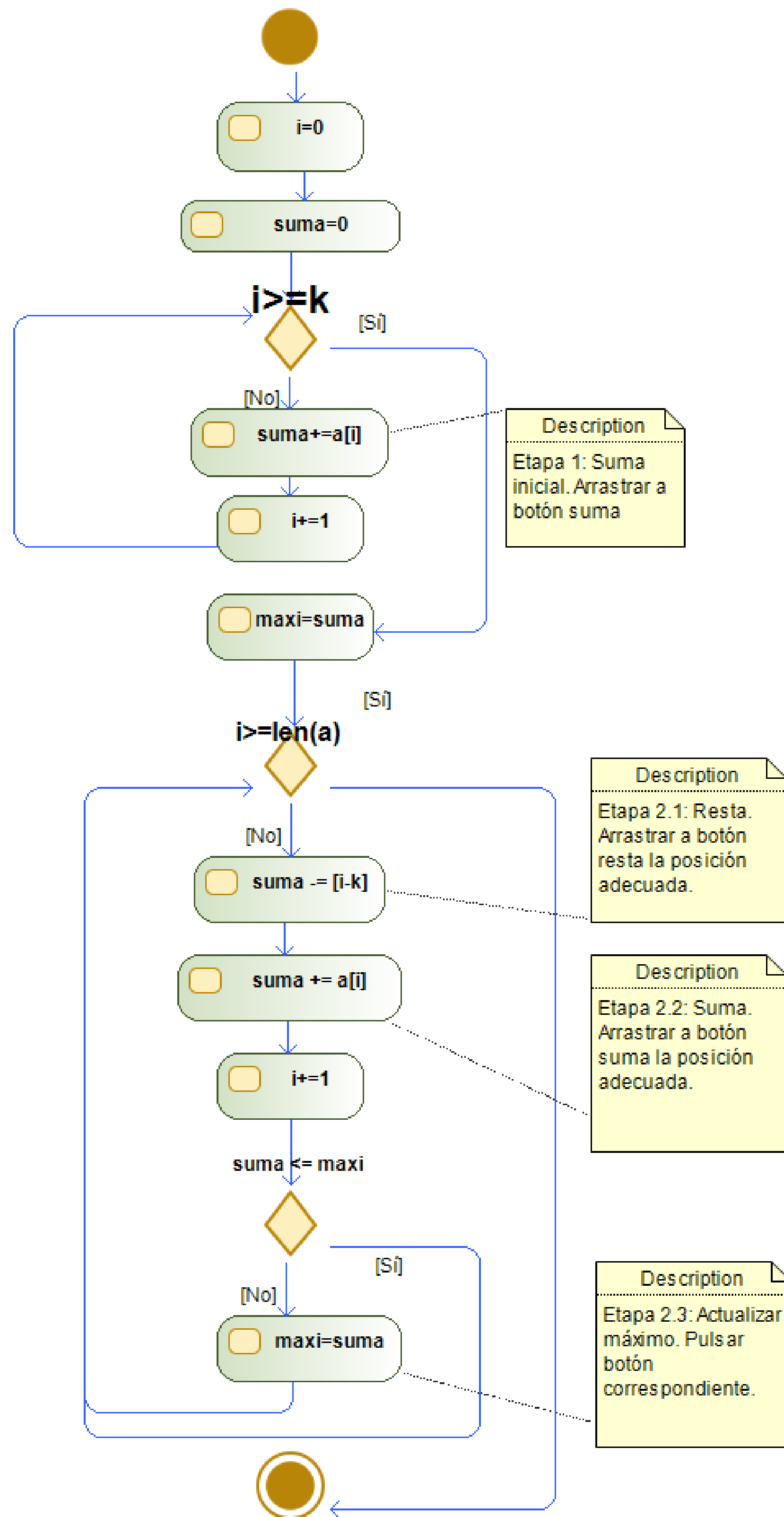


Figura 2.31: Diagrama de flujo del problema de la suma máxima de k elementos consecutivos. Las anotaciones indican las diferentes etapas consideradas y la interacción requerida.

2.6.2. Tutorial del juego

- En primer lugar, se expone el problema que ya se ha comentado, así como los elementos con los que se va a interaccionar:

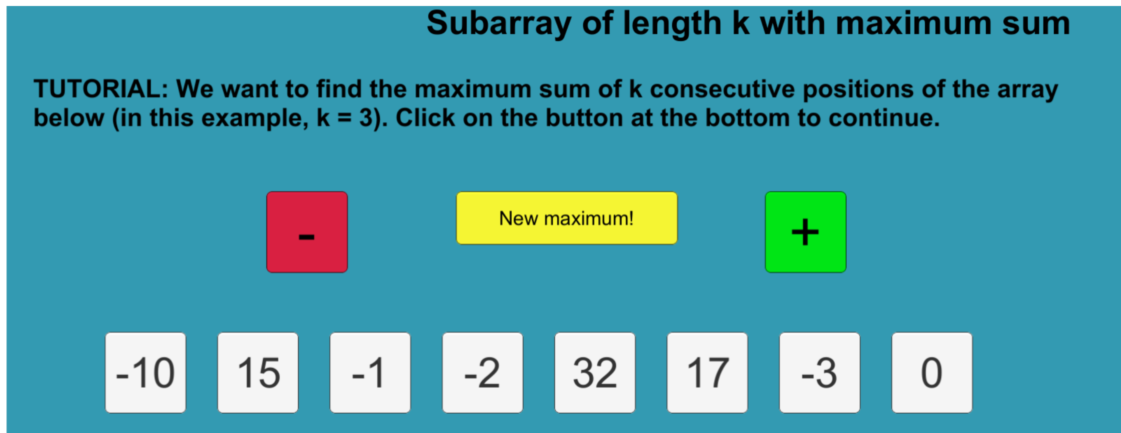


Figura 2.32: Introducción del tutorial del juego.

Al pulsar el botón para pasar al siguiente paso del tutorial, se pasa al paso 1 del algoritmo (se distinguen dos pasos, donde el segundo tiene tres subetapas posibles):

1. Hacer la suma inicial: el tutorial muestra la animación de sumar los primeros k elementos, arrastrando las posiciones al botón verde de "+".

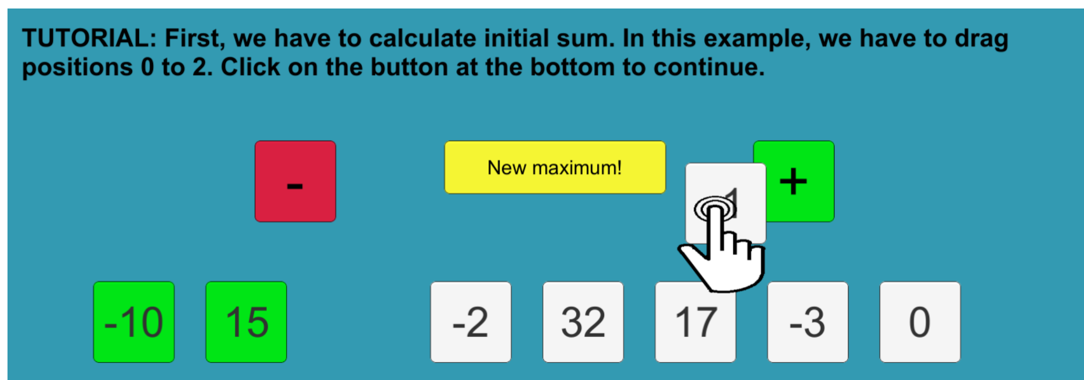


Figura 2.33: Hacer la suma inicial: se muestra la animación de la suma de las tres primeras posiciones. En la imagen, ya se han sumado las dos primeras posiciones (color verde) y se está sumando la tercera posición.

2. Desplazar la ventana y comparar las sumas con el máximo provisional. Esta etapa tiene tres subetapas, con tres interacciones asociadas:

    a) Desplazar la ventana (1): Se muestra la animación que arrastra el extremo izquierdo de la ventana al botón de la resta, para dejar de tener en cuenta ese elemento en la suma total.

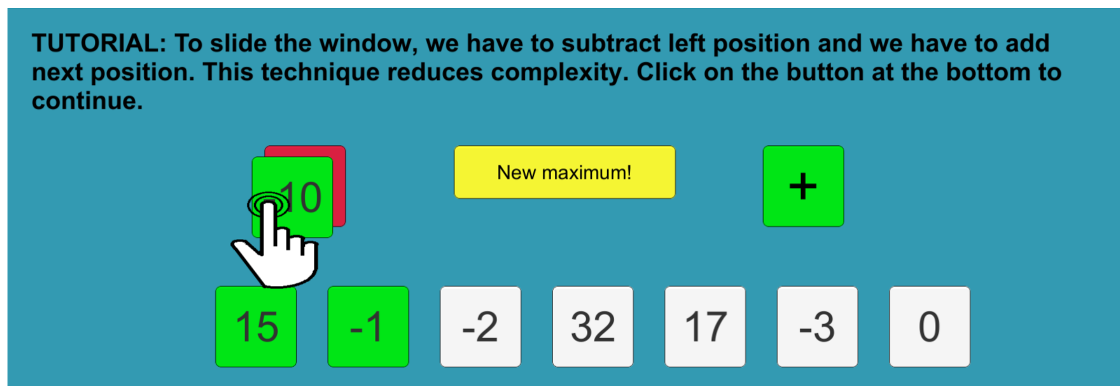


Figura 2.34: Restar a la suma actual la posición 0 para empezar a desplazar la ventana.

b) Desplazar la ventana (2): Se muestra la animación que arrastra la siguiente posición a considerar en la ventana al botón de la suma, para añadir ese elemento a la suma actual.

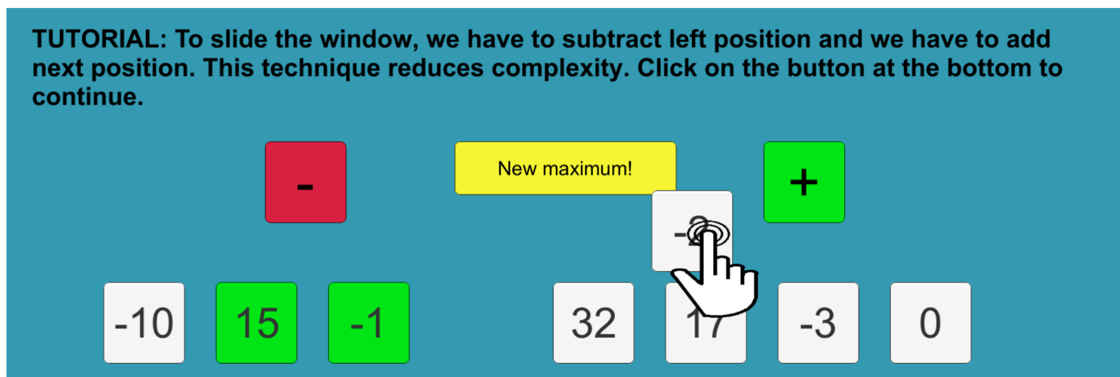


Figura 2.35: Sumar el siguiente elemento para acabar de desplazar la ventana

c) Nuevo máximo (etapa quizás vacía, es decir, sin interacción y por tanto omitida): Si la suma de los elementos de la ventana considerada es mayor que el máximo actual, la animación muestra que hay que hacer *click* en el botón de nuevo máximo.

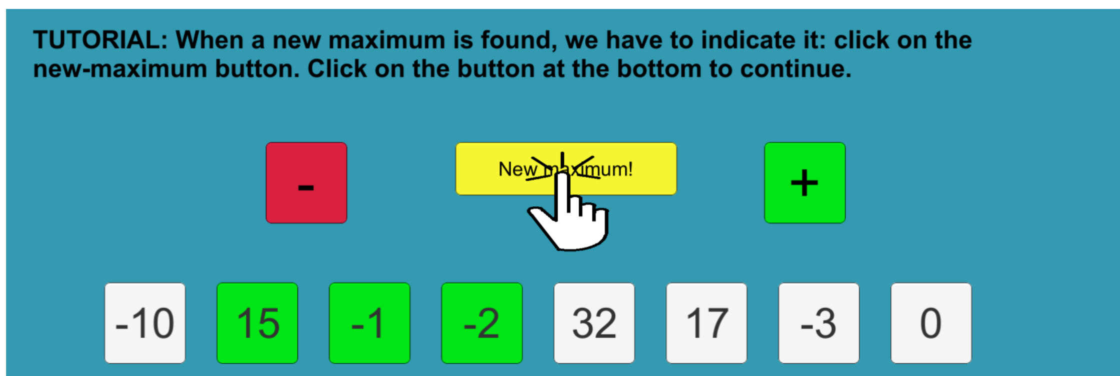


Figura 2.36: La animación muestra un nuevo máximo encontrado, que hay que indicar en el botón correspondiente.

2.6.3. Interacción detallada con un ejemplo

Pese a que ya se ha visto con mucho detalle en la subsección anterior cómo funciona la interacción de este juego, veamos algunos pasos aplicados a un ejemplo que ya tiene que resolver el usuario tras el tutorial, es decir, el juego en sí. Esto nos servirá para ver los diferentes textos que aparecen en la escena, con pistas o sin ellas:

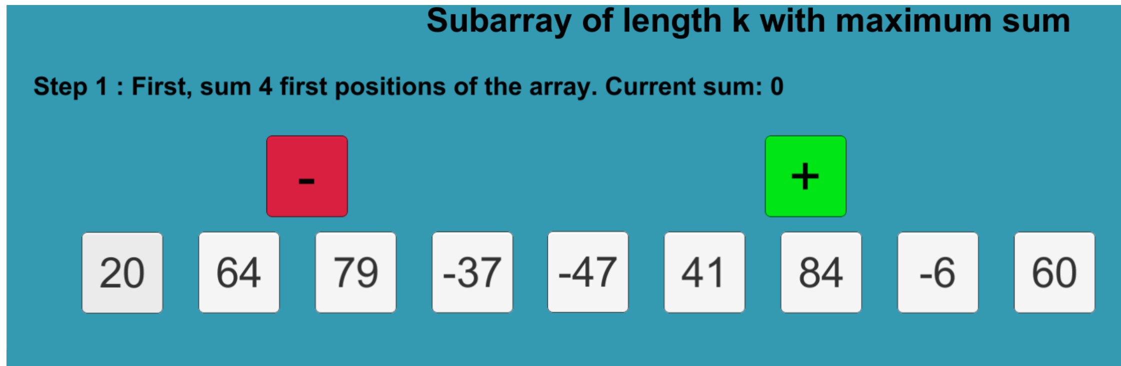
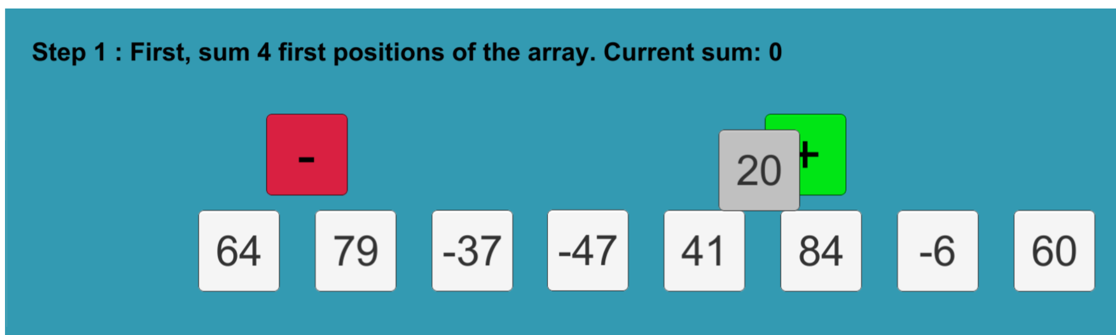


Figura 2.37: En este caso, $k = 4$ (en el juego, k se calcula aleatoriamente dentro de unos valores adecuados). Hay que empezar sumando los cuatro primeros valores del array.



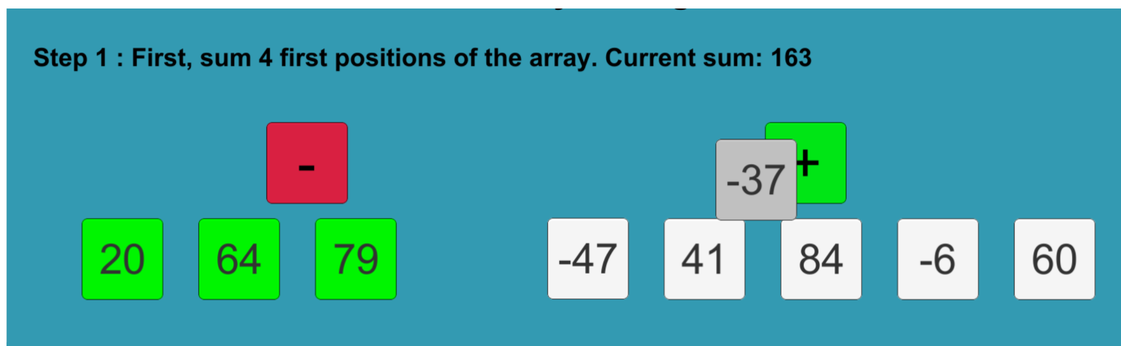
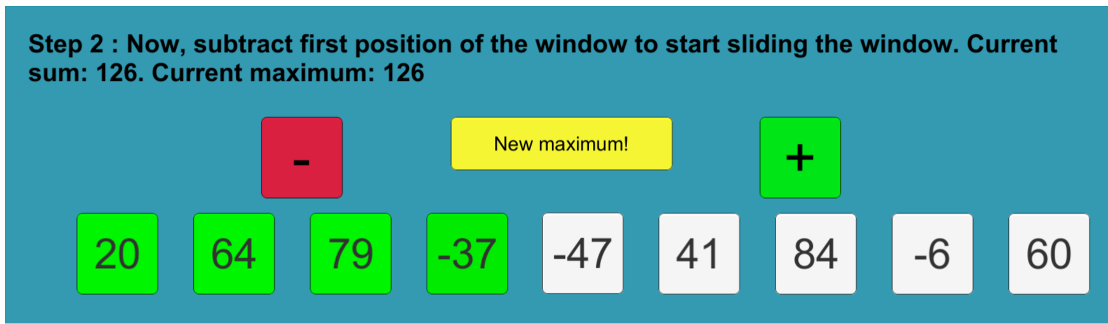


Figura 2.38: Al ir sumando posiciones, se actualiza el campo de suma actual con la cifra concreta. Además, las posiciones de la ventana se van coloreando de verde.



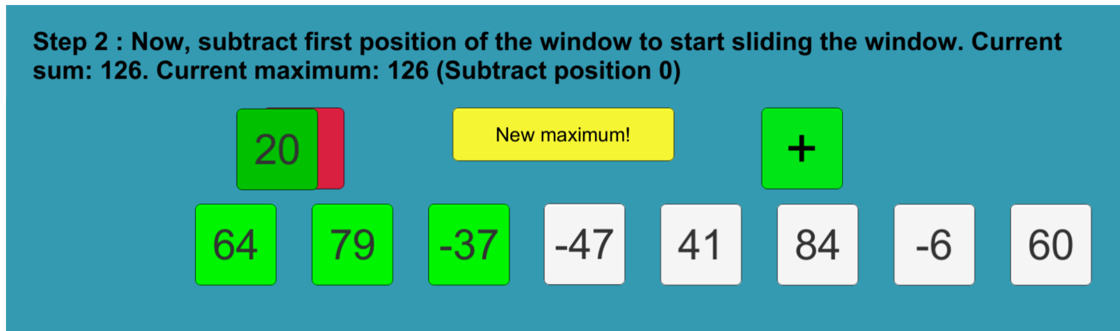
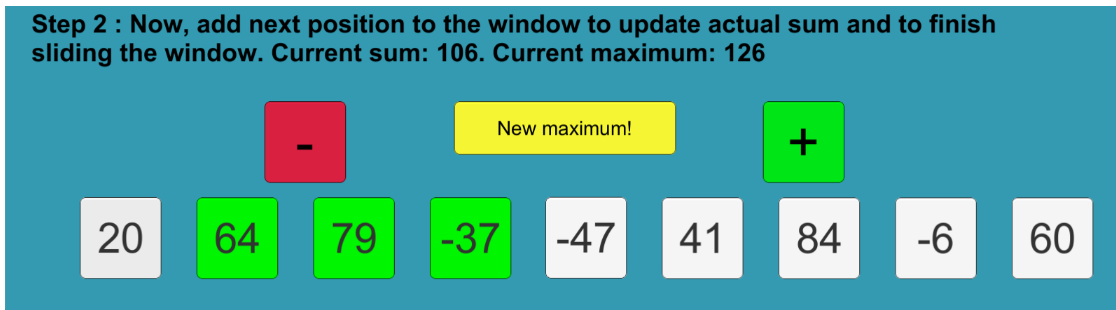


Figura 2.39: Una vez tenemos la ventana inicial, se inicializa el máximo y procedemos a desplazar la ventana. Primero, hay que restar el extremo izquierdo (en la imagen superior se muestra el texto sin pistas, mientras que en la inferior se muestran las pistas y se indica con qué posición hay que interaccionar)

.



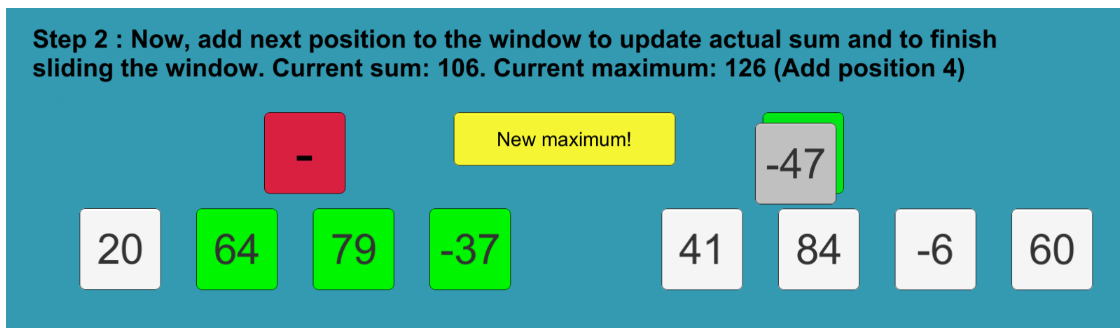


Figura 2.40: Para terminar de desplazar la ventana hay que sumar la siguiente posición. De nuevo, la imagen inferior añade el texto con pistas, donde se indica también la posición con la que hay que interaccionar.

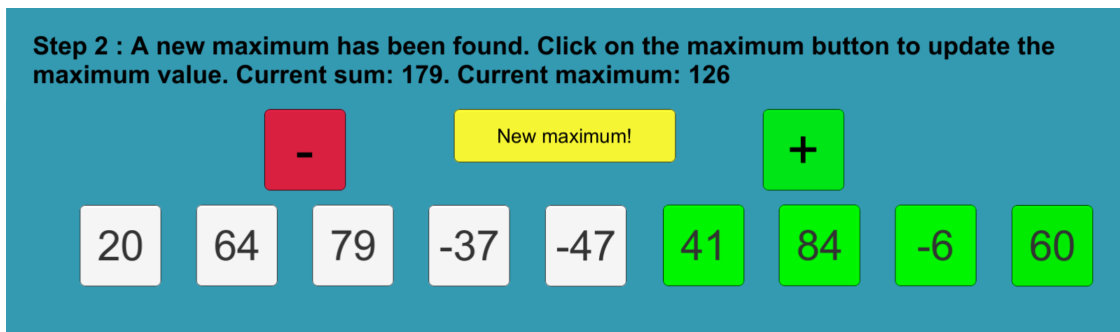


Figura 2.41: Cuando en algún momento la ventana tenga una suma mayor que el máximo provisional, hay que hacer *click* sobre el botón del máximo. En este caso, la ventana de suma máxima se encuentra al final, y al indicar que se ha encontrado un nuevo máximo, el juego terminará.

2.6.4. Algoritmos de control y comprobación

El modelo corresponde a la clase `SlidingWindowManager`, que interacciona principalmente con `ArrayPosSlidWindow` y en menor medida con `SlidWindowNewMax`. Sus atributos y métodos son los siguientes.

- **Atributos**.
  - `values`: *Array* con los valores.
  - `k`: Longitud del *subarray* del que encontrar la suma máxima. Se inicializa aleatoriamente.
  - `stepNumber`: Indica en cuál de las dos etapas se encuentra el proceso.
  - `subStepNumberStep2`: Número de subetapa dentro de la etapa 2.
  - `newMaximum`: *bool* que controla si un nuevo máximo provisional ha sido encontrado.
  - `posEndWindow`: Entero con la posición del final de la ventana (se sabe que el inicio está k posiciones por detrás).
  - `actualSum`: Suma de la ventana en cada momento.
  - `provMaximum`: Valor de la suma máxima provisional.
- **Métodos**.
  - Métodos de inicialización y métodos *get*.
  - `finished()`: Indica si la partida ha sido completada. Se comprueba que `posEndWindow` coincida con la última posición del *array* y que no se esté a la espera de indicar que se ha encontrado un nuevo máximo.
  - `restart()`: Reinicia los atributos para comenzar un nuevo juego.
  - `correctInteraction()`: Es llamado tras cada interacción correcta, y actualiza el modelo recalculando los atributos adecuadamente tras distinguir la etapa en la que se encuentra el proceso.

Capítulo 3

Diseño y aspectos de la aplicación

En este capítulo se explican las clases que integran el proyecto y que no han sido tratadas aún (las clases de los *Managers* de los juegos se detallaron en el capítulo anterior). A continuación, se muestra algún proceso ilustrativo mediante diagramas de secuencia. Además, hay una segunda sección con la explicación de las funcionalidades adicionales a los juegos.

3.1. Clases y procesos principales

3.1.1. Clases

- `Player`. Clase serializable para permitir el guardado y carga de objetos de este tipo. Sus atributos son la información de cada uno de los juegos implementados, es decir, `insSortInfo`, de tipo `GameInfoInsSort`; `selSortInfo`, de tipo `GameInfoSelectionSort`; `bubbleSortInfo`, de tipo `GameInfoBubbleSort`; `mergeSort1Info`, de tipo `GameInfoMergeSort1`; y, por último, `slidingWindowInfo`, de tipo `GameInfoSlidingWindow`. El método `updateGameData()`, que recibe un enumerado `Game game` e información sobre la partida, actualiza el atributo correspondiente al juego `game`.

- `GameInfo`. También es serializable. Guarda la información de un juego. En concreto, guarda puntuación máxima, partidas acabadas, total de acciones necesarias para terminar esas partidas y el tiempo dedicado a ese juego. Estos atributos son actualizados cada vez que se acaba una partida por medio del método `updateGameData()`. Su método `ToString()` devuelve en texto estos datos y es la base para mostrar las estadísticas de todos los juegos. `GameInfo` es la clase padre de `GameInfoInsSort`, `GameInfoBubbleSort`, `GameInfoSelectionSort`, `GameInfoMergeSort1` y `GameInfoSlidingWindow`. Estas clases sobrescriben el método `ToString()`, indicando de qué juego se trata y llamando después a `ToString()` de la clase padre.

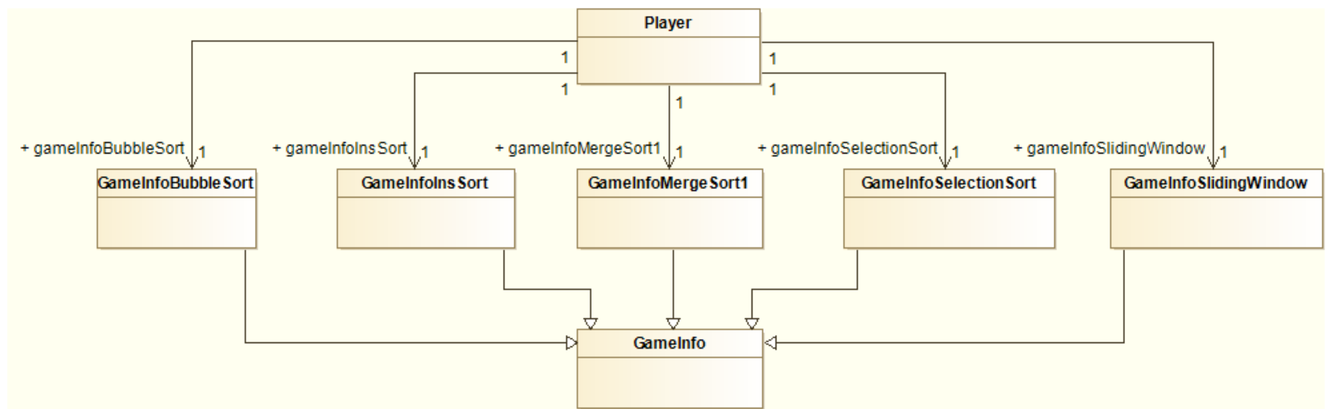


Figura 3.1: Relación entre las clases `Player`, `GameInfo` y las clases que heredan de `GameInfo`.

- **Tutoriales**. Las clases `InsertionSortTutorial`, `BubbleSortTutorial`, `SelectionSortTutorial`, `MergeSort1Tutorial` y `SlidingWindowTutorial` son *scripts* que gestionan los tutoriales de sus juegos mediante una instancia de `animator`. Se controla el número de etapa del tutorial y el texto mostrado en cada etapa. Cuando se pulsa el botón de continuar el tutorial, basta incrementar el número de etapa e indicar a `animator` que pase a la siguiente animación mediante el método `SetTrigger()`.

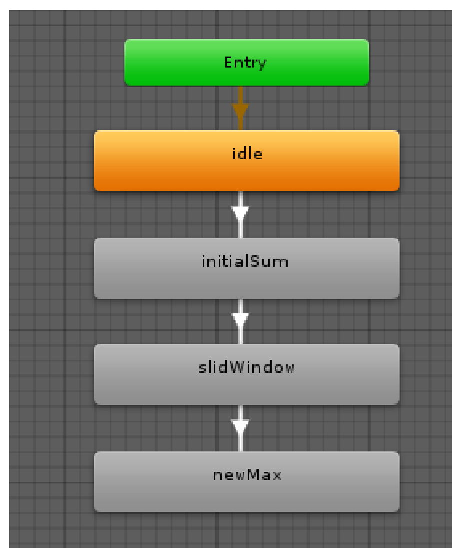


Figura 3.2: Estructura lineal en la animación de los tutoriales. En este caso, el de desplazamiento de ventana. Al finalizar la última animación, se carga una nueva escena.

- `ArrayPos` ***(Juego)***. Las clases `ArrayPosInsSort`, `ArrayPosBubbleSort`, `ArrayPosSelSort`, `ArrayPosMergeSort1` y `ArrayPosSlidingWindow` siguen también una estructura común. Son el controlador entre el modelo y la vista. Estos *scripts* están asociados a cada uno de los botones que conforman los *arrays* de los juegos, y por tanto deben implementar las interfaces `IPointerDownHandler, IPointerUpHandler` e `IDragHandler` para permitir el patrón *drag-and-drop*. Cada botón conoce qué posición ocupa en el array por medio del atributo `posThis`. Los métodos principales son `checkInteraction()`, `setInformationText()` y `updateInstructions()`. El primero comprueba si la interacción realizada es correcta; el segundo muestra el texto con el *feedback* indicando el

acierto o el error del usuario; por último, el tercero actualiza las instrucciones según la etapa del juego y según si se han activado las pistas.

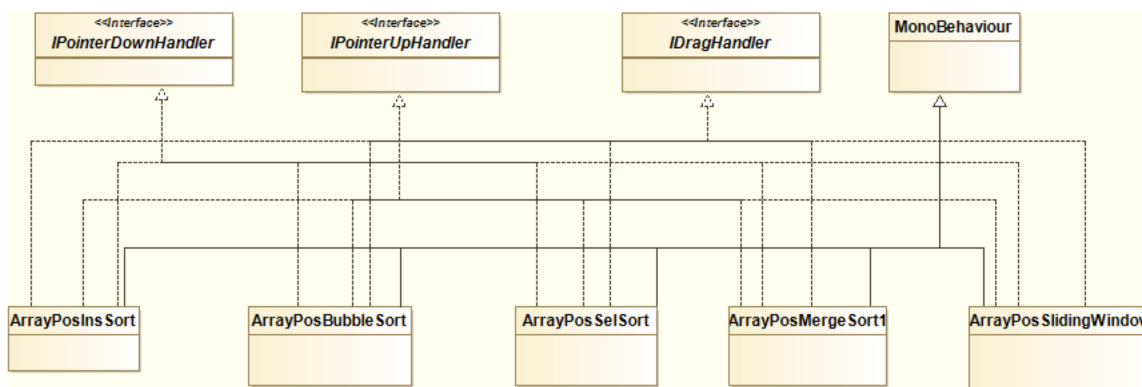


Figura 3.3: Todas las clases `ArrayPos` implementan las interfaces que permiten hacer *drag-and-drop*. Además, heredan de `MonoBehaviour`, es decir, son scripts.

- `Game`. Tipo enumerado con un posible valor para cada juego y menú, con un valor adicional para detectar errores.

- `ArrayController` y `MergeSort1Controller`. Gestionan todos los botones que componen el array de forma global. También realizan mediante código las animaciones del comienzo de cada juego. `MergeSort1Controller` es necesario porque la distribución que se presenta en esta pantalla es distinta al resto. Su método más importante es `locate()`, que recoloca las posiciones a su lugar adecuado en la escena tras cada acción del usuario.

- `GameOptions`. Se encarga de los botones de reinicio y volver hacia atrás de todos los juegos. Conoce la acción que debe realizar según el parámetro `game` que recibe en los métodos `restartSelected()` y `backSelected()`.

- `SceneController`. Gestiona el cambio de escena y el cierre de la aplicación al hacer *click* en el botón de salir.

- `OtherOptionsController`. Controla la animación del *pop-up* del panel de otras opciones, desplegándolo u ocultándolo según el botón pulsado.

- `SlidWindowNewMax`. Es similar a los *scripts* que tienen los botones que conforman el *array*, es decir, `ArrayPos`*(Juego)*, aunque este caso es algo excepcional. La razón es que en la actividad de desplazamiento de ventana hay un elemento adicional que sí admite interacción. Así, el botón de nuevo máximo necesita tener un *script* con código que se ejecute al ser pulsado para comprobar si esa interacción es correcta o no. El código se corresponde con su método `newMaxButtonClicked()`.

- `ShowStatistics`. Realiza el volcado del texto de los datos de rendimiento al área de texto presente en la pantalla de estadísticas, para poder mostrarlas. Se ayuda del método `getStatisticsText()` de `DataManager`, que a su vez tiene una instancia de `Player` con esa información requerida.

- `DataManager`. Clase que gestiona los datos de la aplicación. Se implementa usando el patrón de diseño *Singleton* para tener una única instancia en todo el juego (además,

es seguro frente a concurrencia). Sus atributos son la propia instancia (como en cualquier *Singleton*), el *lock* de protección, una variable *bool* `loaded` que indica si ya se han cargado los datos (para cargarlos una sola vez aunque se reciban más peticiones), varias rutas para la gestión de archivos, atributos para gestión de puntuaciones y de estadísticas, atributos para controlar el sistema de *log* y un entero con el estado de las pistas (solamente se muestran las pistas en el estado 2). Además tiene una instancia de `Player`, encapsulando la información de un usuario. En la carga de datos (método `loadData()`) se usa la clase `BinaryFormatter`, y es por esto por lo que se ha explicado unos párrafos antes que ciertas clases son serializables. `DataManager` también se encarga de gestionar las estadísticas de un juego, como la puntuación. Cuando un juego acaba, el método `gameFinished()` indica a la instancia de `Player` que actualice sus estadísticas añadiendo las de la partida que acaba de terminar. También tiene métodos destinados al sistema de *log*, aunque estos se explicarán más adelante. Por último, es el encargado de guardar los datos para que la próxima vez que se entre a la aplicación las estadísticas sigan disponibles. Para esto, de nuevo se usa `BinaryFormatter` y se guarda la instancia de `Player` mediante el método `Serialize()`.

- `LoadData`. Al abrir la aplicación, ejecuta la carga de datos, indicando también al `DataManager` la ruta `Application.persistentDataPath`.

- `Utilities`. Clase con constantes y con algunos métodos auxiliares, como comprobar si dos vectores representan posiciones cercanas (con cierto margen `EPS_NEAR`) o transformar una variable de tipo `TimeSpan` a segundos.

3.1.2. Diagramas de secuencia de dos procesos principales

En primer lugar se muestra un diagrama de secuencia ilustrativo para el caso de uso de consultar estadísticas. Esto permite entender mejor y complementar lo expuesto en la sección anterior relativo a las estadísticas. Por último, se representa mediante otro diagrama de secuencia la anidación de las llamadas en el proceso de comprobar si una interacción del usuario es correcta. Se toma como ejemplo el juego de ordenación por selección y se muestra uno de los posibles recorridos (no se consideran en el diagrama las distintas opciones de etapas en el juego o interacción correcta o errónea debido a cuestiones de espacio).

Consultar estadísticas. Uno de los casos de uso de la aplicación es consultar las estadísticas del usuario. La cadena de llamadas que se genera da lugar al siguiente diagrama de secuencia.

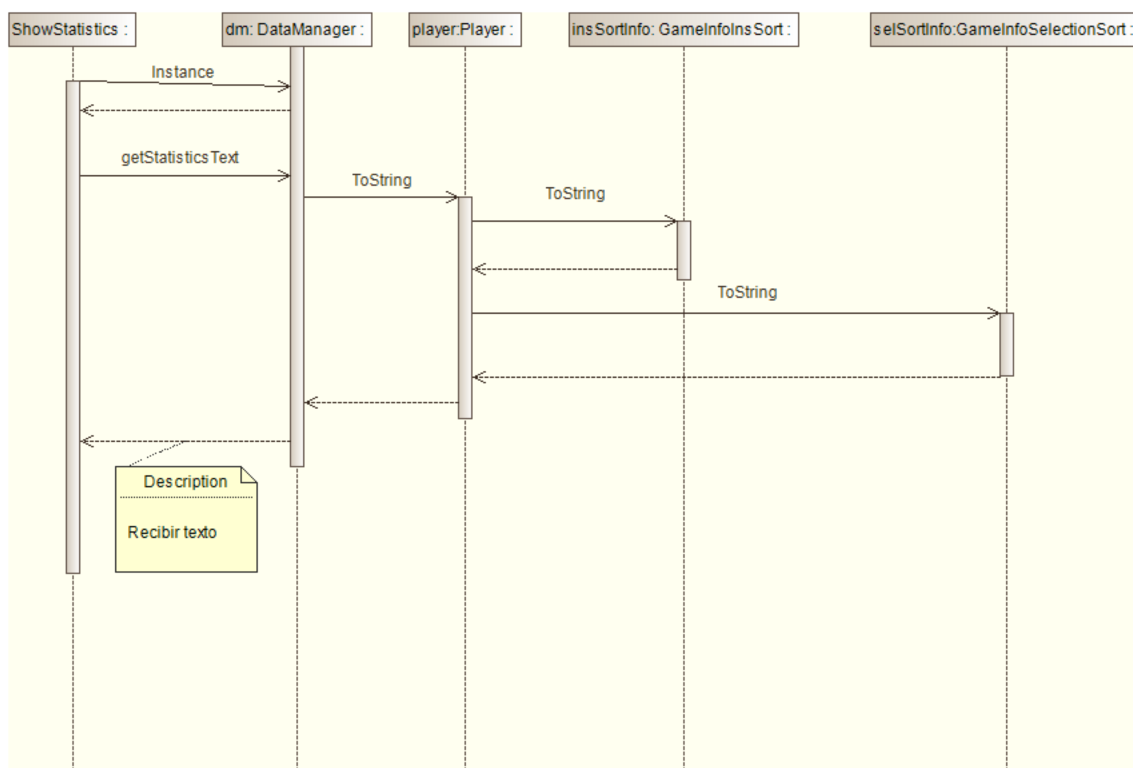


Figura 3.4: Diagrama de secuencia para el caso de uso de consultar estadísticas. Se consideran solo implementados dos juegos por cuestiones de espacio, pero las llamadas desde la instancia de `Player` seguirían al resto de instancias `GameInfo`**(Juego)**.

Comprobar interacción. Otra de las funciones esenciales que debe realizar la aplicación es verificar que las acciones que realiza el usuario son correctas. Se adjunta un diagrama de secuencia correspondiente a una de estas comprobaciones.

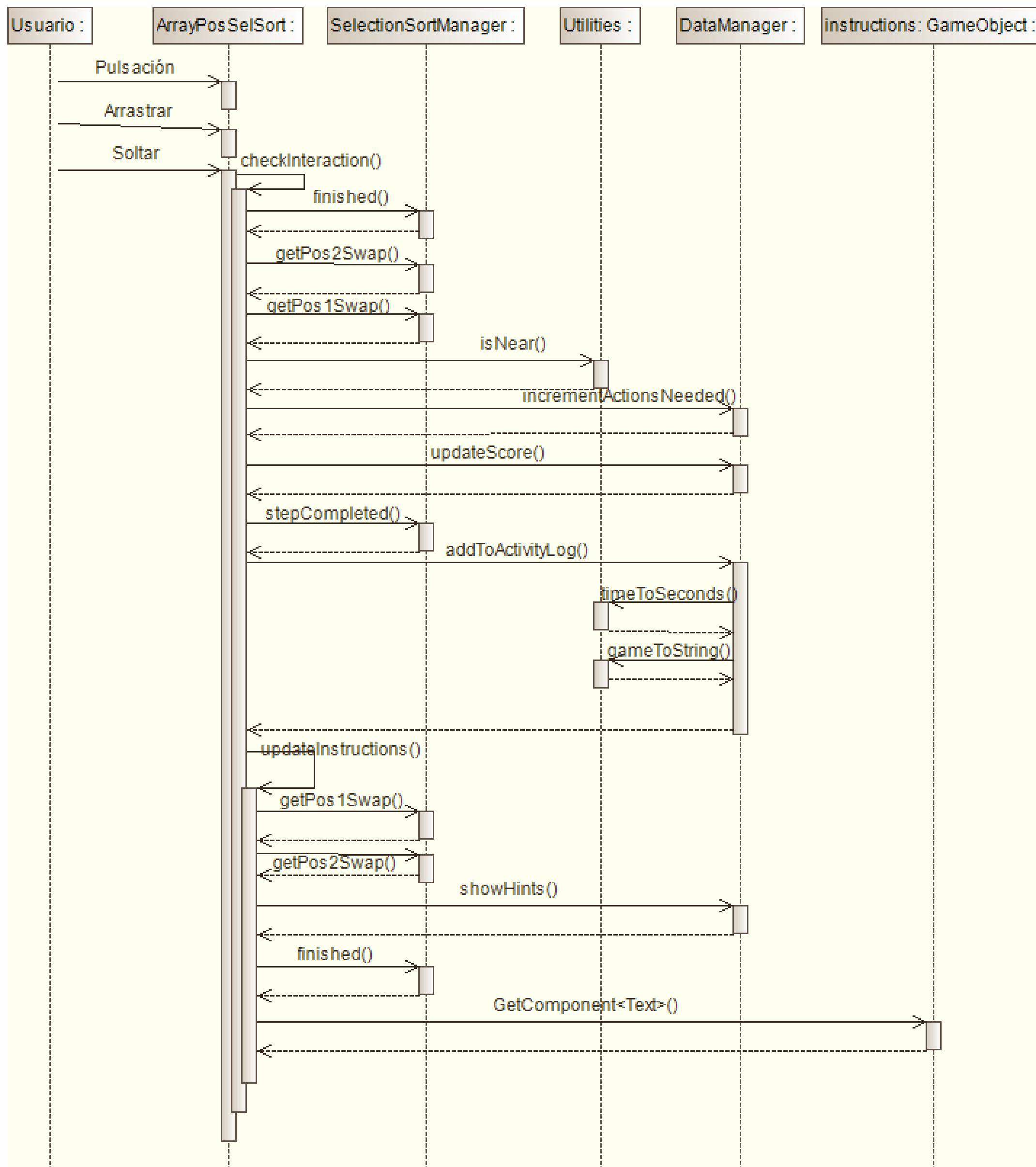


Figura 3.5: Diagrama de secuencia de la comprobación de una interacción del usuario en el juego de ordenación por selección. Se considera solo un recorrido de las distintas opciones que se presentan y se muestra la anidación de las llamadas en el caso en que la siguiente posición no está ya ordenada, el usuario interacciona correctamente, el juego no finaliza tras esa interacción y el *array* con el *log* no se llena al escribir una posición adicional.

3.2. Feedback proporcionado y otras funcionalidades

3.2.1. Feedback

Como se ha discutido analizando artículos en el capítulo 1 de esta memoria, muchos autores coinciden en la necesidad de proporcionar *feedback* adecuado, así como adaptar la dificultad del juego a la habilidad del usuario. Por ejemplo, estos temas se discuten en [1], [6] o [3]. El juego que se presenta en esta memoria tiene tres tipos de *feedback* de diferente naturaleza:

- **Aciertos/Errores**: Tras cada acción, el sistema informa de si la interacción es correcta

o errónea mediante texto en la pantalla del juego. Internamente, se actualiza la puntuación que se muestra al finalizar dicho juego y se actualiza el modelo de adaptación de la dificultad dinámicamente. Estos dos conceptos que se acaban de mencionar constituyen el resto de formas de *feedback* de la aplicación, y se tratarán a continuación.

- **Puntuaciones y estadísticas**: Para introducir más componente lúdico y típico de juegos, se incorporan puntuaciones y estadísticas. Al acabar cada partida se muestra el tiempo que se ha tardado en completarla y una puntuación asociada que depende de dicho tiempo y del número de aciertos y de errores. Finalmente, también se muestra el número de acciones necesarias para completar el juego (este último dato es más bien solo informativo). En cuanto a la puntuación, se busca un equilibrio entre configuraciones iniciales. Si es sencilla, se requerirán menos acciones y por tanto habrá menos oportunidades de sumar puntos, pero estas configuraciones sencillas también deberían completarse en menos tiempo, penalizando menos la puntuación. Configuraciones tales que son necesarias muchas acciones para acabar llevarían a una mayor cantidad de aciertos (también de fallos, seguramente) pero también penalizaría más la puntuación por el tiempo tardado.

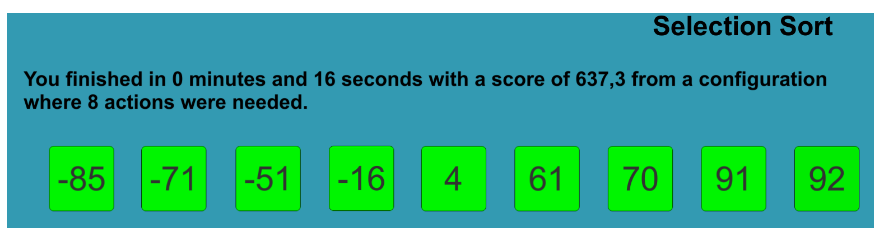


Figura 3.6: Imagen del *feedback* proporcionado al terminar un juego, en este caso ordenación por selección.

Además del sistema de puntuaciones, se guardan estadísticas interesantes para el usuario desde el punto de vista teórico. Por ejemplo, para cada pantalla se cuentan el número de partidas finalizadas, el número de acciones requeridas totales en dichas partidas o el tiempo total empleado. Combinando estos datos, podemos mostrar al usuario estadísticas como el número medio de acciones requeridas en una partida. También podemos conseguir datos sobre su rendimiento, como el tiempo medio empleado para cada acción. Se ha decidido mostrar en la pantalla de "Estadísticas", para cada juego implementado, la mayor puntuación conseguida, el número de partidas terminadas, la media de acciones por juego, el tiempo total empleado en dicha pantalla y la media de tiempo empleado en cada acción. Si un juego no tiene partidas terminadas, se indica que no hay datos.

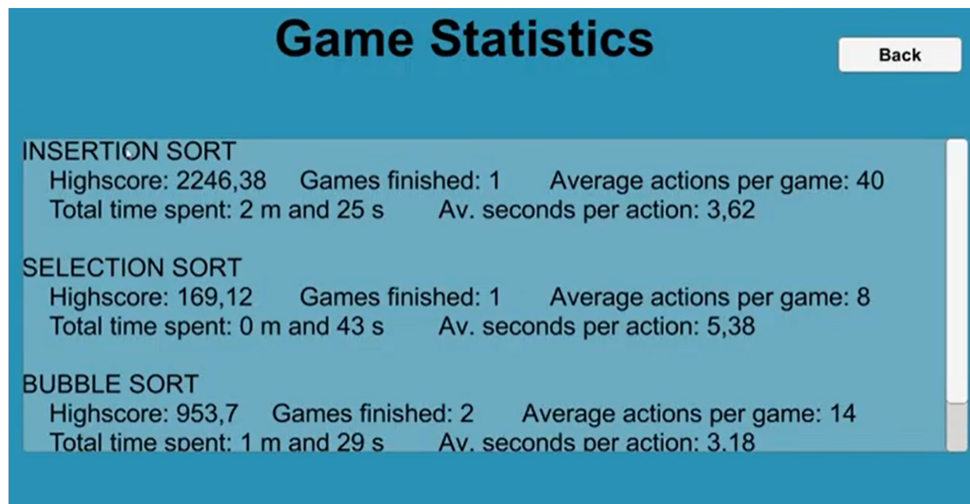


Figura 3.7: Captura de pantalla de las estadísticas mostradas.

- **Adaptación dinámica de la dificultad**: Seguramente el método más eficaz para este tipo de aplicaciones. En este caso, se basa en proporcionar mayor o menor información en el texto que se muestra en cada etapa de cada juego. Se puede modelar como un diagrama de tres estados, pero donde solamente se han implementado dos salidas (mostrar pistas o no). El diagrama de estados es el siguiente.

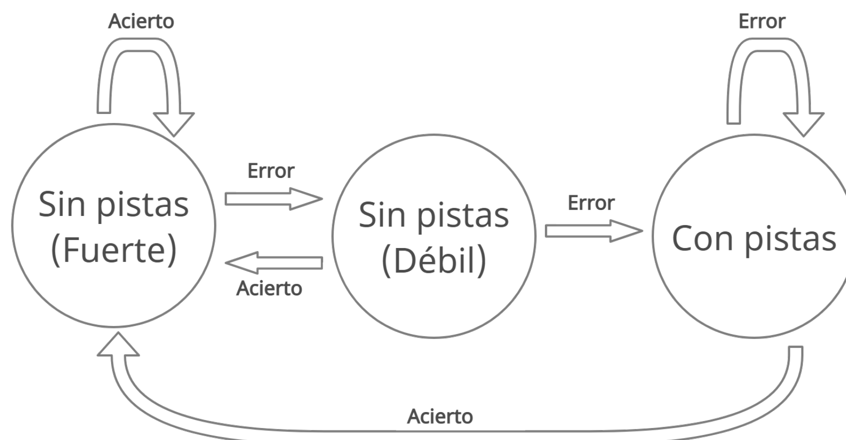


Figura 3.8: Diagrama de estados correspondiente al sistema de adaptación dinámica de la dificultad.

En dicho diagrama apreciamos:

- Estado "Sin pistas (fuerte)": Es el estado inicial. Primero se reta al usuario a resolver el problema sin pistas. Mientras se acierte, el sistema permanece en este estado; ante un error, el sistema cambia al siguiente estado que se va a explicar a continuación, donde seguirá sin mostrar las pistas aún.
- Estado "Sin pistas (débil)": Se sigue sin mostrar ayuda extra, pero el usuario ya ha cometido un fallo. Ante un acierto, se vuelve al estado anteriormente explicado; sin embargo, si falla, ya sabemos que ha tenido dos fallos seguidos. Para evitar la frustración del usuario y que se quede bloqueado en algún paso, el estado cambia

a "Con pistas" y el texto se actualiza dando más información, como la posición con la que hay que interaccionar o el movimiento que hay que realizar.

- Estado "Con pistas": Las últimas dos interacciones (al menos) han sido fallidas. Las pistas se activan y se muestra más información intentando que el usuario encuentre la interacción correcta. Esto, a su vez, le puede dar más pistas para entender la interacción pedida en general y con ello el algoritmo en estudio. Si se acierta, se desactivan las ayudas completamente (esto es, se vuelve al estado "Sin pistas (fuerte)"); mientras se falle, el sistema se mantiene en dicho estado con pistas.

Para implementar esto, la clase `DataManager` guarda un atributo entero `hintsState`, donde 0 se corresponde con "Sin pistas (fuerte)", 1 se corresponde con "Sin pistas (débil)" y 2 se corresponde con el estado "Con pistas". En el constructor y ante nuevos comienzos de juegos, como ya se ha comentado, se inicializa a 0; por otro lado, ante cada interacción, se actualiza el valor de la siguiente forma. Si se ha acertado, se pasa al estado 0; si no, el estado se incrementa en una unidad, con la excepción de estar ya en el estado 2 (pistas activas) en cuyo caso el atributo no cambia. Por último, el método de `DataManager` `showHints()` indica si se deben mostrar las pistas o no: basta comparar el atributo `hintsState` con el valor 2. En los métodos `updateInstructions()` que se comentaron anteriormente, el *string* asociado a las instrucciones adquiere un valor u otro según lo que devuelve este método `showHints()`.

3.2.2. Log con la interacción

Por último, se almacena un archivo de texto con toda la interacción del usuario. Esto en versiones posteriores del juego podría tener utilidad para conseguir estadísticas más elaboradas. Sin embargo, el objetivo del *log* en este trabajo es servir de ayuda en las pruebas finales con usuarios.

Se recoge la secuencia de acciones realizadas en cierta pantalla (momento en que sucede medido en segundos desde que el usuario entra en la aplicación, se indica si la interacción es correcta y también se registran otras acciones como el final o el reinicio del juego). Posteriormente, los archivos se analizarán para sacar conclusiones de los usuarios que participan en las pruebas.

En cuanto a la implementación de dicho *log*, se guarda como atributo de `DataManager` un array de tipo *string* de `LOG_BUFFER_SIZE` posiciones. Para no estar actualizando el archivo de texto tras cada acción, la información de la interacción se va insertando en dicho *array*; cuando este se llena, se traslada al archivo de texto, y el índice de escritura del *array* pasa de nuevo al valor 0 (no hace falta borrarlo, se va sobrescribiendo). Una pequeña consideración que hay que tener en cuenta al utilizar este mecanismo es que al cerrar la aplicación, se encuentre donde se encuentre el índice de escritura, hay que volcar la información del array al archivo de texto. Para permitir esto está el método `forceLogFileUpdate()` de la clase `DataManager`. Para cargar y guardar la información se usa la clase `File` de `System.IO`.

Capítulo 4

Evaluación con Usuarios

En este capítulo se explican las pruebas de usabilidad llevadas a cabo para la aplicación. Una primera sección desarrolla los primeros ensayos que se relizaron, con una versión de la aplicación aún no definitiva. Ante la necesidad de métodos más fiables y ante los malos resultados de las pruebas, se dedicó una fase intermedia (mientras se seguía desarrollando la aplicación y mientras se corregían algunos errores que salieron a la luz en dichas pruebas) para investigar artículos sobre métodos para realizar las pruebas de usabilidad, así como el análisis de las mismas. La teoría que se encontró más aplicable al caso de esta aplicación se expone resumida en la segunda sección. La tercera explica las pruebas finales con usuarios. Finalmente, el capítulo termina con los resultados y las conclusiones sobre todo este proceso realizado relativo a pruebas de usabilidad.

4.1. Primeras pruebas con usuarios

4.1.1. Introducción

Una vez se consiguió una versión estable con ciertos algoritmos implementados, se pudo probar dicha aplicación con dos usuarios para detectar posibles errores en fases tempranas del desarrollo (faltaban por añadir funcionalidades y algunos juegos más frente a la versión final). Como se mencionó al final del capítulo 1, el coste de arreglar un error se incrementa considerablemente según aumenta el desarrollo de la aplicación y, efectivamente, estas pruebas sirvieron para corregir lo que se estaba haciendo incorrectamente y seguir desarrollando de forma adecuada.

Las pruebas se tuvieron que realizar en remoto, lo que tiene algunos inconvenientes (aunque también ventajas, como que el usuario está en su entorno). Por tanto, fue una evaluación no moderada pero sí guiada (*scripted test*), ya que se le indicó al usuario lo que tenía que hacer. En concreto, primero se les proporcionó un guion con los pasos a realizar y herramientas a usar. Tras esto, los usuarios debían rellenar un cuestionario previo a la prueba. Posteriormente, se podían fijar en la lista de tareas a realizar en la aplicación (grabando simultáneamente la pantalla tras haber comprobado que el archivo de *log* se guardaba correctamente en su dispositivo). Finalmente, tenían que rellenar otro cuestionario con las impresiones que les generó la aplicación. Como se realizaron dos pruebas, las respuestas cuantitativas al cuestionario apenas tienen relevancia (la muestra es muy pequeña). Sin embargo, la fase de interacción con la aplicación así como las respuestas cualitativas y sugerencias de mejora sí resultaron muy útiles.

El artículo [27] explica que tanto los tests de usabilidad como el *game-based learning* están ganando importancia. Por tanto, también se debe desarrollar el diseño de dichos juegos y sus pruebas. Mientras que en el software habitual el hecho de no cometer errores es bueno, un videojuego debe desafiar constantemente al usuario, y se vuelve aburrido si no se comete ningún error. En definitiva, aunque está claro que las pruebas de usabilidad habituales hay que modificarlas para videojuegos y, en concreto, para juegos serios educativos, en esta primera fase de las pruebas no se introdujo ninguna distinción. Esta fue otra razón por la que se decidió hacer un segundo conjunto de pruebas de usabilidad.

4.1.2. Pruebas realizadas

Los dos usuarios que participaron en las pruebas tenían ya conocimientos de programación. En la versión que probaron, era el usuario el que elegía si quería tener pistas o no (esto se cambió más adelante al sistema de tres estados ya mencionado, tras conocer la defensa que realizaban diversos artículos científicos del *feedback* personalizado y de la adaptación dinámica y automática de la dificultad). En esta versión antigua que se usó en las primeras pruebas, uno de los usuarios realizó la prueba sin ayuda de las pistas, mientras que el otro podía consultar las pistas en todo momento. Las pruebas estaban pensadas para realizarse en un tiempo de entre 15 y 20 minutos. Las tareas del guion se presentaron en forma de tareas directas (solamente se informa de lo que tiene que hacer, en contraposición a las tareas escenario que añaden un contexto). Los usuarios realizaron la evaluación con sus dispositivos móviles tras proporcionarles el archivo *.apk* de la aplicación.

Tras hacer el cuestionario inicial, poner en marcha las herramientas de grabación y comprobar que el *log* se guarda correctamente, el usuario podía comenzar con las tareas, que consistían en jugar 4 partidas de cada uno de los siguientes juegos: ordenación por inserción, método de la burbuja, ordenación por selección y ordenación por mezclas (1). Además, el usuario debía comprobar sus estadísticas en el juego. Tras finalizar, tenía que completar la encuesta final dando sus impresiones y enviar los resultados.

4.1.3. Resultados y posibles mejoras

Aunque no era un número de usuarios significativo para resultativos cuantitativos, sí se obtuvieron sugerencias de mejora, así como posibles cambios en la aplicación tras analizar las grabaciones:

- Un error observado en estas pruebas fue la precisión en el patrón *drag-and-drop*. El usuario hacía muchos movimientos que eran correctos pero el sistema los detectaba como incorrectos. Este problema se considera como crítico. Había aparecido en fases iniciales del desarrollo, pero tras ajustar el valor de la variable asociada a la precisión, quedó solventado. Más adelante, se descubrió un error asociado en el código. Tras solucionarlo, el problema despareció en las segundas pruebas.

- En ordenación por inserción, en la primera implementación realizada, había una fase en que desaparecía de la pantalla el elemento a insertar, ya que en la fase de hacer hueco a dicho elemento era solapado al hacer la copia. Esto despistó a los usuarios. En la versión final se incluye la variable auxiliar ya mencionada para tener disponible en todo momento cualquier elemento. Además, esto es fiel a cómo actúa realmente el algoritmo.

- Los usuarios coinciden en que era necesaria una primera parte de teoría (quizás incluso con pseudocódigo) para evitar el proceso inicial de prueba y error. Por esto, para que la aplicación sea autocontenida, se implementa un tutorial guiado que se despliega cuando el usuario entra a la pantalla de un juego.

- Un usuario reporta que en algunos casos el sistema de interacción podría ser diferente a *drag-and-drop*. Por ejemplo, al intercambiar elementos se realiza arrastrando un elemento concreto al otro. Esto hacía pensar al usuario que un elemento tiene mayor jerarquía que el otro, cuando esto no es así.

- También se encontró algún aspecto que resultó especialmente agradable a los usuarios: el uso de colores en los botones para dar ciertos *feedbacks* fue bienvenido, por lo que se generalizó el uso de esta característica en el desarrollo posterior.

- Aunque esto no lo reportó ningún usuario, en el análisis de las grabaciones se apreciaba que en ocasiones el usuario comienza una acción, pero observa que es una acción incorrecta antes de finalizarla, por lo que devuelve el botón a su posición inicial. Sin embargo, el juego lo detecta como interacción errónea y penaliza la puntuación. Quizás esto se podría distinguir de alguna forma, de manera que si el usuario vuelve al estado inicial, no se considere un error. Se distinguirían así estas acciones de las correctas que se completan totalmente y de las incorrectas que se completan totalmente.

El tiempo que emplearon los usuarios en completar las tareas fue de 20 y de 26 minutos (poco más de lo esperado). El problema de la precisión contaminó totalmente el archivo de *log*, ya que en la grabación se apreciaba que el usuario iba entendiendo los conceptos, pero seguía fallando mucho debido a la precisión: movimientos correctos eran clasificados como incorrectos. Esta es otra razón para volver a realizar pruebas con una versión mejorada. La siguiente imagen, correspondiente a los errores acumulados en el tiempo de los usuarios, muestra que las pendientes de las curvas no bajan, como era previsible. Además, el procesamiento del *log* se debe hacer agrupado según el juego. Si no, al cambiar a uno nuevo, volverán a aparecer errores, dificultando mucho la obtención de conclusiones.

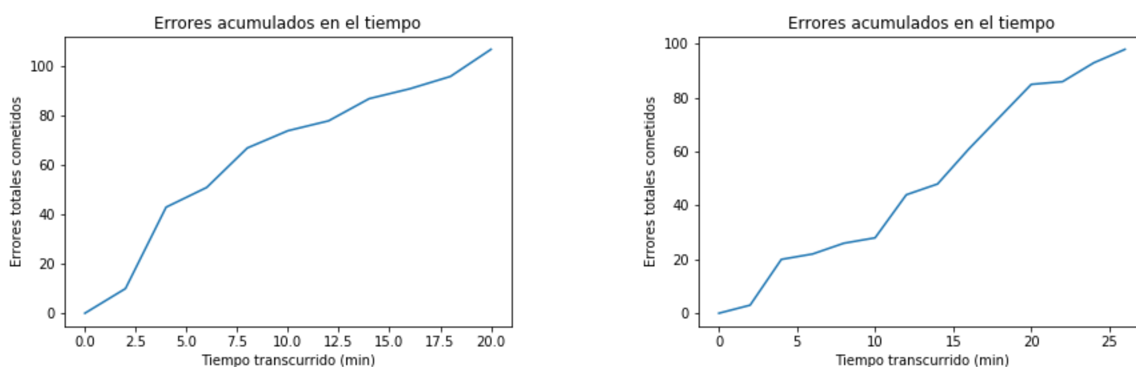


Figura 4.1: Gráficas de errores acumulados a lo largo del tiempo tras analizar los *logs* de los usuarios de las pruebas iniciales. Se aprecia que la pendiente apenas desciende a lo largo del tiempo. La gráfica es similar a la de [27], con la diferencia de anotar errores en vez de eventos, como se hace en el modelo de este artículo. Posteriormente se encuentra el artículo [16], que expone la teoría de las curvas de aprendizaje. Curiosamente, si la métrica usada es el número de errores y se toma de forma acumulada, se podría decir que estas gráficas corresponden a versiones de curvas de aprendizaje.

4.2. Teoría sobre pruebas de usabilidad

En el artículo [15], donde se analizan los datos de la interacción de usuarios con un juego educativo para enseñar física muy básica, se comentan varios métodos. En primer lugar, si la mecánica del juego es cercana al objetivo, se indica que se pueden usar métricas muy directas y fáciles de implementar como el número de niveles superados. En el caso de la aplicación de esta memoria la mecánica del juego es exactamente la misma que el objetivo, es decir, se representa totalmente igual lo que se pretende aprender. Por tanto, se podría usar una métrica así. Sin embargo, es una medida muy sencilla y además las pruebas están pensadas para realizar un número determinado de partidas y observar la evolución más que para medir cuántos niveles es capaz de superar el usuario o cuántas partidas puede jugar en cierto tiempo. Por tanto, esta opción se descartó.

Siguiendo en [15], en segundo lugar se comenta otra técnica como es BKT (*Bayesian Knowledge Tracing*), que intenta inferir los conocimientos del estudiante a través de su rendimiento utilizando redes bayesianas. Sin embargo, lo que se ha decidido implementar para las pruebas finales con usuarios son las curvas de aprendizaje (*learning curves*). Esta técnica también se comenta en [15], pero se explica y se desarrolla en profundidad en [16].

En lo que respecta a este artículo sobre curvas de aprendizaje, el eje y se corresponde con la medida del rendimiento (hay que decidir una métrica, como número de intentos para resolver un paso o la tasa de error) y el eje x se corresponde con el tiempo (aunque puede ser, por ejemplo, en forma de número natural incremental según cada oportunidad que se presenta para demostrar el concepto a aprender). Estas últimas palabras, "el concepto a aprender", son importantes, ya que lo ideal es tener una curva de aprendizaje para cada concepto a aprender. Si realmente se está aprendiendo el concepto, la curva debería seguir la ley de potencia de la práctica (*power law of practise* [16]). Esta ley viene dada por $Y = Ax^{-B}$, donde A y B dependen de la curva, representando A la tasa de error (o la correspondiente métrica asociada) para $x = 1$ (por ejemplo, si x mide cada oportunidad para demostrar el concepto aprendido, A mediría la tasa de error en el primer intento, sin practicar antes). Intuitivamente, B mide la velocidad de aprendizaje, ya que se relaciona con la pendiente y la forma de la curva.

Como ya se ha dicho, lo ideal sería tener una curva de aprendizaje para cada concepto a aprender, pero también es deseable tener muchos datos. Para solventar este problema, una posible solución es agregar los datos para todos los usuarios según cierto concepto a aprender.

Unas últimas observaciones sobre esto son que estamos suponiendo que el límite de los errores del usuario en un juego educativo es 0 cuando el tiempo de práctica tienda a infinito (es lógico suponer esto ya que aunque el feedback sea muy inadecuado, solamente por prueba y error se acabarían encontrando patrones); además, el feedback puede influir en cómo se aplana la curva: si la curva comienza con mucha pendiente descendente (es decir, pendiente negativa pero de gran valor absoluto) pero a partir de cierto momento t la pendiente negativa se suaviza pero convergiendo a un valor estrictamente positivo de la tasa de error en vez de a cero, nos estará indicando que el *feedback* no es efectivo a partir de ese momento t. Por último, también en [16] se hace hincapié en que comparar dos curvas diferentes es difícil y no se puede hacer directamente por la naturaleza de las funciones que representan.

4.3. Pruebas finales con usuarios

En las pruebas con usuarios que se realizaron una vez el desarrollo de la aplicación estaba finalizado, se contó con 20 ejercicios. El objetivo era sacar datos más fiables, comprobar si los problemas encontrados en las primeras pruebas estaban solucionados y mejorar algunos aspectos del análisis.

Aunque se explica en detalle a continuación, a grandes rasgos el proceso consistió en compartir a cada usuario el archivo *.apk* junto con unas instrucciones, un cuestionario previo, un guion de tareas a realizar y un cuestionario final, ya que las pruebas se tuvieron que realizar en remoto. Los resultados obtenidos de los cuestionarios, *logs* con la interacción y observaciones extra se adjuntan en la siguiente sección para finalizar el capítulo.

En la estructura de las pruebas, en primer lugar, se indica a los usuarios que el ejercicio está pensado para durar media hora, y se les invita a leer y pensar cada paso en el juego. Se les informa del sistema de pistas (adaptación dinámica de la dificultad al usuario) para indicarles que si se quedan bloqueados el texto se actualizará dando más detalles de lo que se pretende. Como también se ha probado con usuarios fuera del público objetivo principal, se recuerda que en informática las posiciones comienzan en 0 para la mayoría de lenguajes de programación. Por último, se les indica que sus pruebas serán incorporadas a la memoria de forma anónima, que los *logs* no recopilan ninguna información más allá de aciertos y errores dentro del juego, y que las estadísticas obtenidas de los *logs* se incorporarán a la memoria de forma conjunta.

Se indica a continuación que las tareas que deben realizar son, en este orden, responder el cuestionario inicial, comenzar la grabación de pantalla (si es posible), realizar el guion de tareas una vez estén dentro de la aplicación y enviar el archivo *algLog.txt* que se genera en su dispositivo móvil y que contiene el *log* que se analizará finalmente.

En los cuestionarios se hacen afirmaciones con respuestas según una escala Likert (donde 1 es totalmente en desacuerdo y 5 es totalmente de acuerdo), aunque también hay alguna cuestión donde se pide respuesta escrita. Pasando al cuestionario inicial, se preguntan los siguientes puntos (los usuarios sin conocimientos informáticos no responden las primeras cinco cuestiones).

- En mi estudio de la informática he estudiado el funcionamiento de algoritmos mediante el código asociado.
- En mi estudio de la informática he estudiado el funcionamiento de algoritmos mediante vídeos.
- En mi estudio de la informática he estudiado el funcionamiento de algoritmos recreando por mí mismo el flujo del algoritmo en un papel.
- En mi estudio de la informática he usado videojuegos o aplicaciones interactivas para entender mejor los algoritmos.
- Pienso que un juego interactivo en el que se imita el comportamiento de los algoritmos puede ayudar a entenderlos mejor.
- En mis estudios en general, he usado videojuegos educativos o aplicaciones interactivas para aprender.

- Creo que incorporar videojuegos educativos y aplicaciones interactivas en el aprendizaje es una buena idea.
- Describe con tus palabras (o con pseudocódigo si lo prefieres) cómo resolverías el siguiente problema: Tenemos una lista de un millón de números. Queremos encontrar la suma máxima de 100 elementos consecutivos, es decir, cuánto podemos sumar en total sumando los valores de 100 posiciones consecutivas de la lista.
- Comentarios adicionales (Opcional).

Cabe destacar que la penúltima cuestión anterior se pregunta para observar posibles diferencias entre la aproximación que tomarían antes de realizar la actividad de la ventana deslizante sobre la suma máxima de k elementos consecutivos frente a después de jugar al mismo.

Posteriormente, el usuario puede abrir la aplicación y realizar el siguiente guion de tareas:

- Jugar cuatro partidas del algoritmo de ordenación por selección.
- Jugar cuatro partidas del algoritmo de ordenación por el método de la burbuja.
- Jugar cuatro partidas del algoritmo de ordenación por mezclas (1).
- Jugar cuatro partidas del algoritmo de ordenación por inserción.
- Jugar cuatro partidas de la técnica de desplazamiento de ventana.

Una vez el usuario ha completado las tareas, finaliza la prueba enviando el archivo *algLog.txt* con el *log* y realizando el cuestionario final, donde de nuevo la mayoría son afirmaciones pensadas para una escala Likert:

- Me ha resultado fácil llegar a las pantallas donde comienzan los juegos pedidos.
- Una vez en un juego determinado, me ha resultado fácil entender la interacción pedida.
- Me han ayudado los tutoriales de los juegos.
- ¿La interacción de qué juego te ha resultado más intuitiva? ¿Por qué?
- ¿La interacción de qué juego te ha resultado más difícil? ¿Por qué?
- Tras jugar estas 20 partidas en total, tengo la sensación de haber entendido bien el funcionamiento de cada juego (o haber recordado completamente los algoritmos).
- Me parece una aplicación útil a la hora de enseñar conceptos.
- Comentarios adicionales (opcional).

Además, se pregunta de nuevo el problema de la suma máxima de k elementos consecutivos de un *array*, para observar si el usuario mejora el método que dio en el cuestionario previo.

4.4. Resultados de las pruebas y conclusiones

Tras analizar los resultados de la encuesta previa, de la encuesta final y de todos los archivos *algLog.txt* que la aplicación generó a los usuarios participantes, se expone a continuación un resumen de los resultados y se comentan los hechos más destacados.

4.4.1. Cuestionario previo

Haciendo la media de las respuestas para cada pregunta de escala Likert se obtienen los siguientes resultados. Hay que recordar que si el usuario no tenía estudios de informática pasaba directamente a responder a las preguntas 6 y 7.

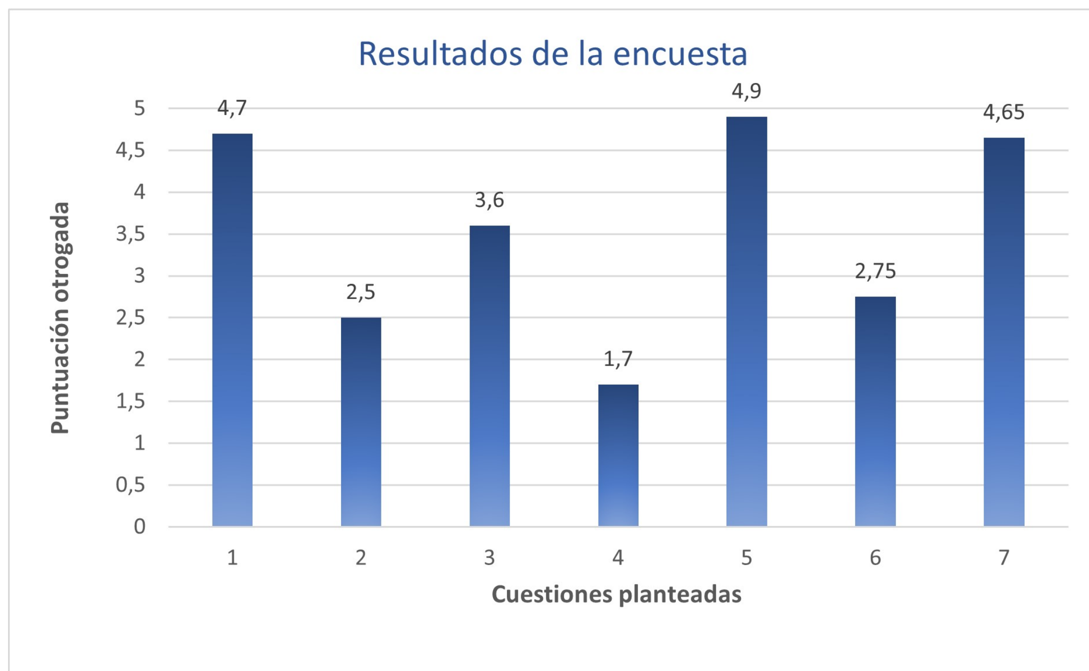


Figura 4.2: Media de las respuestas para cada pregunta en el cuestionario previo.

A modo de leyenda de la figura anterior, se recuerda cuál era cada pregunta a continuación.

1. En mi estudio de la informática he estudiado el funcionamiento de algoritmos mediante el código asociado.

2. En mi estudio de la informática he estudiado el funcionamiento de algoritmos mediante vídeos.

3. En mi estudio de la informática he estudiado el funcionamiento de algoritmos recreando por mí mismo el flujo del algoritmo en un papel.

4. En mi estudio de la informática he usado videojuegos o aplicaciones interactivas para entender mejor los algoritmos.

5. Pienso que un juego interactivo en el que se imita el comportamiento de los algoritmos puede ayudar a entenderlos mejor.

6. En mis estudios en general, he usado videojuegos educativos o aplicaciones interactivas para aprender.

7. Creo que incorporar videojuegos educativos y aplicaciones interactivas en el aprendizaje es una buena idea.

Los resultados que muestra el gráfico de la figura 4.2 indican que los métodos más habituales para estudiar algoritmia son, en este orden, estudiar el código asociado, recrear el flujo del algoritmo en papel, usar vídeos y, finalmente, usar videojuegos o aplicaciones interactivas. Sin

embargo, con un valor de 4.9 sobre 5, se afirma que un juego interactivo en el que se simula el flujo de un algoritmo puede ser útil. Además, se está de acuerdo (con una puntuación de 4.65 sobre 5) en que incorporar videojuegos educativos en el aprendizaje puede ser una buena idea.

4.4.2. Cuestionario final

Los resultados de la encuesta final, con la puntuación media de las respuestas a las preguntas de escala Likert, se adjuntan a continuación.

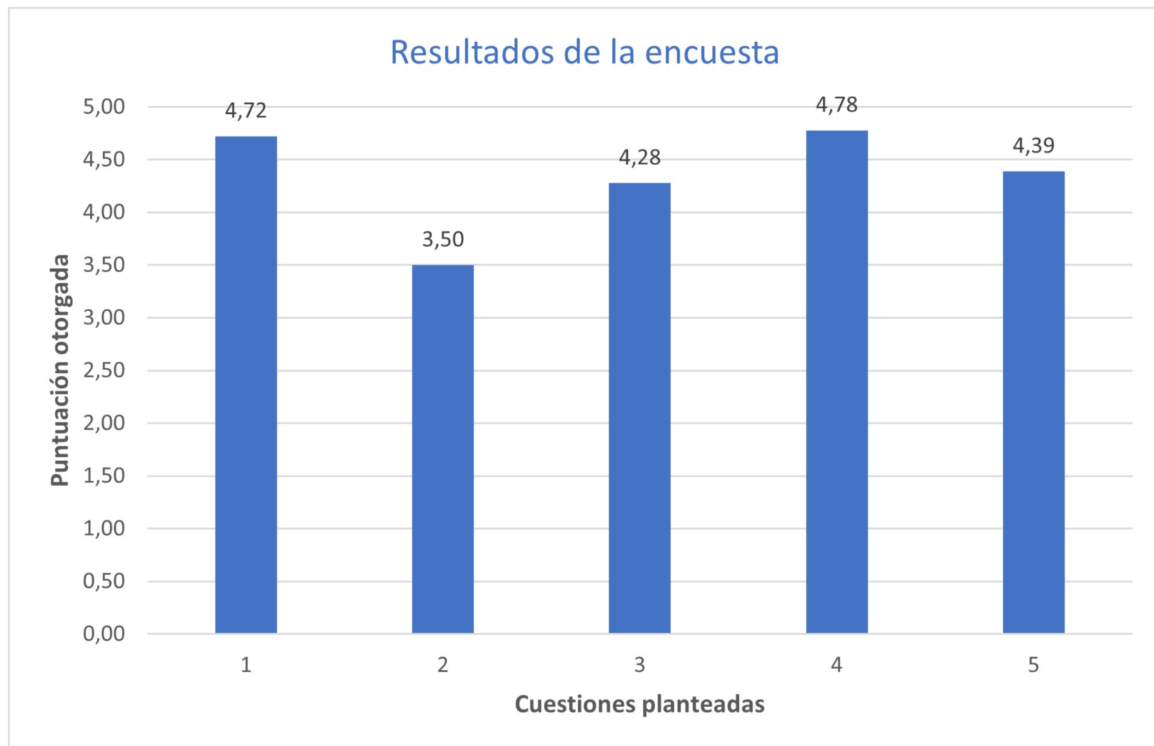


Figura 4.3: Respuestas numéricas al cuestionario final.

El número de cuestión de la gráfica 4.3 se corresponde con las siguientes afirmaciones:

1. Me ha resultado fácil llegar a las pantallas donde comienzan los juegos pedidos.
2. Una vez en un juego determinado, me ha resultado fácil entender la interacción pedida.
3. Me han ayudado los tutoriales de los juegos.
4. Tras jugar estas 20 partidas en total, tengo la sensación de haber entendido bien el funcionamiento de cada juego.
5. Me parece una aplicación útil a la hora de enseñar conceptos.

En general los resultados son positivos, por lo que se puede suponer que los usuarios quedaron satisfechos con la aplicación. El valor más bajo (3.5 sobre 5) es el de entender la interacción propuesta; el valor más alto (4.78 sobre 5) se corresponde con la afirmación de haber entendido bien el funcionamiento del juego y del algoritmo finalmente en cada caso. Esto último, al fin y al cabo, era el objetivo de la aplicación.

También se preguntó por el juego con la interacción más sencilla, así como el juego con la interacción más complicada. Los resultados son los siguientes.

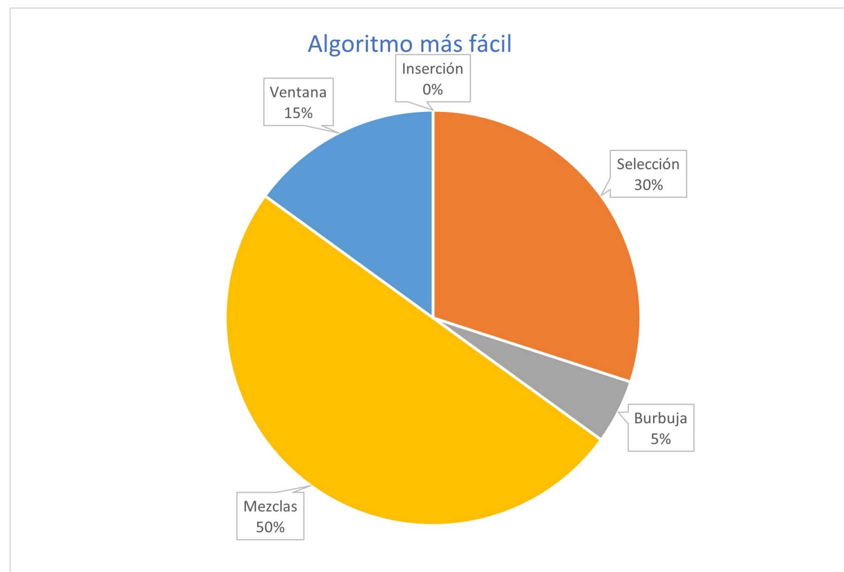


Figura 4.4: Juego que ha parecido más sencillo a los usuarios.

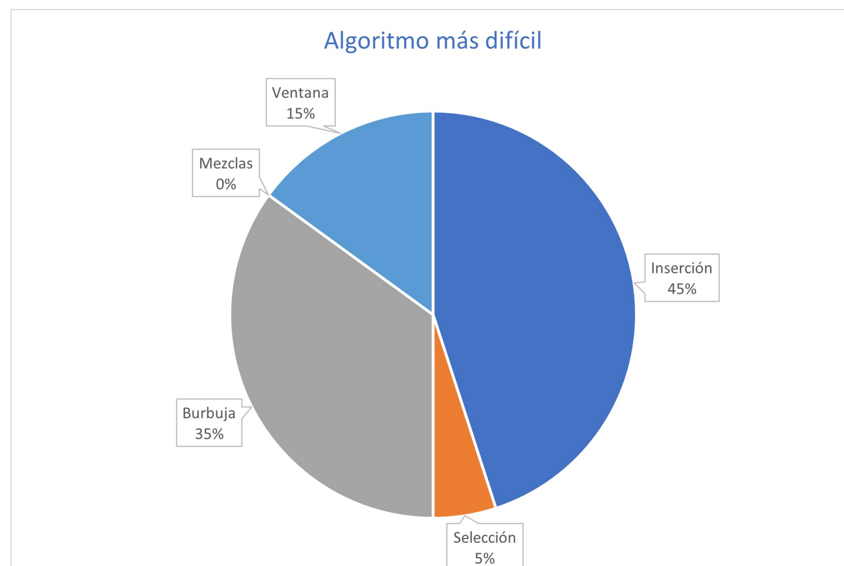


Figura 4.5: Juego que ha parecido más complicado a los usuarios.

Los algoritmos más fáciles han resultado ser la mezcla de dos listas ordenadas y ordenación por selección; el más difícil, ordenación por inserción. Vemos que el algoritmo que se ha implementado a más bajo nivel ha resultado ser el más difícil.

4.4.3. Análisis de los *logs*

El *log* generado ha resultado ser de mucha utilidad para obtener estadísticas globales y curvas de aprendizaje. Se ha implementado un script en *Python* que procesa todos los archivos recibidos, considerando independientemente cada juego pero, para cada uno de estos juegos, agregando los resultados de todos los usuarios.

El funcionamiento de este *script* de análisis consiste en guardar en una variable de tipo diccionario el momento de comienzo de un nuevo juego. Se registran los errores y aciertos en ese juego en intervalos temporales a partir de ese instante inicial.

Es muy ilustrativa la evolución de la tasa de errores a lo largo del tiempo en el caso de ordenación por inserción, que además es el juego que más acciones registra y el que los usuarios han clasificado como menos intuitivo. La gráfica se adjunta a continuación, donde se ha incorporado la recta que aproxima los puntos, usando el módulo `scipy.optimize`. Se aprecia una clara pendiente negativa, señal de que el usuario está aprendiendo el concepto a lo largo del tiempo. Debajo de esta imagen se adjunta otra gráfica con los errores acumulados. Se aprecia una disminución de la pendiente al avanzar en el tiempo.

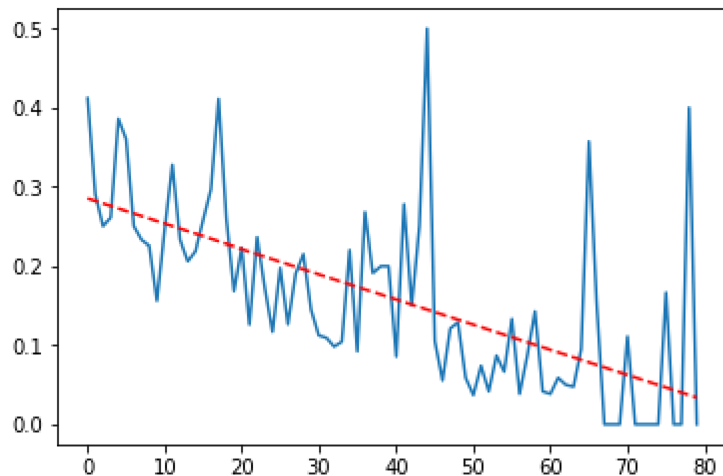


Figura 4.6: Tasa de errores a lo largo del tiempo para ordenación por inserción, en intervalos de 10 segundos (estamos por tanto ante una curva de aprendizaje). El eje x representa el número de intervalo considerado, y el eje y indica el tanto por uno de errores. La recta de ajuste muestra pendiente negativa.

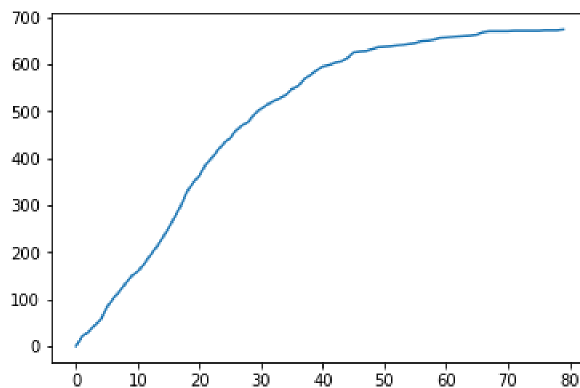


Figura 4.7: Errores acumulados a lo largo de intervalos de tiempo, para el juego de ordenación por inserción. El eje x representa el número de intervalo de tiempo (igual que en la figura anterior). El eje y representa el total de errores.

Por último, se adjunta una tabla con estadísticas globales de todos los *logs* sobre los diferentes juegos.

	Acciones	**Aciertos**	**Errores**	**Tasa de aciertos**
Inserción	3713	3017	696	81.26 %
Selección	740	617	123	83.38 %
Burbuja	1643	1299	344	79.06 %
Mezclas	729	624	105	85.60 %
Ventana	1465	1238	227	84.51 %

Cuadro 4.1: Estadísticas sobre las acciones realizadas.

La tasa de aciertos final es buena en todos los casos, en torno al 80 %. Como era esperable, los juegos que han parecido más complicados son los que tienen menor tasa de aciertos. La diferencia no es grande, pero ante la gran cantidad de acciones registradas sí es representativa.

Capítulo 5

Conclusiones y trabajo futuro

5.1. Conclusiones

El objetivo de este trabajo es crear una aplicación educativa que simula el funcionamiento de algoritmos o estructuras de datos y que permita la interacción del usuario. Las pruebas realizadas al final del desarrollo demuestran que esto ha sido conseguido, ya que tanto el cuestionario final sobre las impresiones con la aplicación como el análisis de los *logs* arrojan resultados positivos. La interacción puede resultar difícil al principio, pero el usuario acaba entendiendo las acciones que realiza el algoritmo y, tras alguna partida, puede replicarlo sin problema. Las pruebas también han demostrado que las funcionalidades adicionales implementadas agregan valor a la aplicación, como el sistema de adaptación dinámica de la dificultad o el sistema de puntuaciones.

Por otro lado, en este trabajo se han utilizado conceptos estudiados a lo largo del grado en ingeniería informática. Como es obvio, las asignaturas de algoritmia tienen un gran peso, aunque se hayan implementado algoritmos básicos. Por otro lado, el paradigma de programación orientada a objetos se ha utilizado para desarrollar un código reutilizable, por lo que asignaturas como Tecnología de la programación o Ingeniería del software también han ayudado al desarrollo de esta aplicación. Para las pruebas con usuarios ha sido útil la asignatura Desarrollo de sistemas interactivos, proporcionando un método para este tipo de evaluaciones. Por último, este trabajo también ha servido para aprender nuevos conceptos que no habían sido tratados a lo largo del grado. Por ejemplo, el motor de videojuegos *Unity*, usando el lenguaje de programación C#, que ha resultado novedoso igualmente.

5.2. Trabajo futuro

La aplicación implementada se podría mejorar de diversas formas. En primer lugar, la manera más inmediata es incluir nuevos juegos asociados a otros algoritmos. Por otro lado, se podría utilizar el sistema de puntuaciones para crear una base de datos que fomente la competitividad en el juego, con el aprendizaje asociado. El sistema de adaptación dinámica de dificultad ha resultado efectivo, pero no deja de ser muy sencillo, por lo que la aplicación podría personalizarse más aún.

Otra posible mejora es ampliar la interacción *drag-and-drop*, con otras formas de realizar las acciones en el juego. Por ejemplo, si se implementan árboles, una interacción interesante podría ser inclinar el teléfono hacia el lado correcto para representar que el árbol rota para quedar equilibrado.

Conclusions and future work

5.3. Conclusions

The aim of this work is to develop an educational application that simulates the flow of algorithms and teaches the basics of data structures, allowing users to interact with it. The usability tests performed at the end of the development show that these goals have been achieved. Both the survey about the impressions of the users and the log file analysis show positive outcomes. Initially, users may find interaction difficult, but they understand how algorithms work after playing a few games. Furthermore, usability tests have shown that additional functionalities in the game like the score system or dynamic adaptability improve the application.

In addition, concepts studied throughout the degree in computer science have been used. Obviously, subjects about algorithms and data structures are very important in this work, although algorithms implemented in the game are basic. Object-oriented programming paradigm has been used in order to develop reusable code, so subjects like Computer Programming Technology or Software Engineering have played an important role in this work too. Interactive systems development subject has been useful for usability testing, because it provides a method to make these tests. Finally, I have learned new concepts too, like *Unity* game engine or C# programming language.

5.4. Future work

The application that has been developed could be improved in several ways. First, new games about algorithms and data structures could be included in the application. Secondly, the score system may be used to create a database that fosters competitiveness. The dynamic difficulty adjustment system has been effective, although it is very simple and the application itself could better adjust to the needs of each user.

Another possible improvement of the game is to extend *drag-and-drop* interaction pattern with other ways of executing actions. For example, if trees were implemented, it would be interesting to allow the user to tilt the device in order to rotate and balance them.

Bibliografía

[1] R. Dörner, S. Göbel, W. Effelsberg, and J. Wiemeyer, *Serious Games.* Springer, 2016.

[2] M. Csikszentmihalyi, *Flow: The psychology of optimal experience.* Harper & Row New York, 1990.

[3] P. Sweetser and P. Wyeth, "Gameflow: A model for evaluating player enjoyment in games," *ACM Computers in Entertainment*, vol. 3, 07 2005.

[4] B. Sawyer and D. Rejeski, "Serious games: Improving public policy through game-based learning and simulation," 01 2002. Woodrow Wilson International Center for Scholars, Washington, DC.

[5] P. Games and M. Knight, "America's army," 2002. Crown Publishing Group, New York.

[6] J. Sinclair, "Feedback control for exergames," 2011. https://ro.ecu.edu.au/theses/380/ (Accessed 13 December 2020).

[7] M. Prensky, "Digital game-based learning," *ACM Computers in Entertainment*, vol. 1, 10 2003.

[8] D. Leclercq, "The 8 Learning Events Model and its principles," 2005. http://www.labset.net/media/prod/8LEM.pdf (Accessed 13 December 2020).

[9] M. D. Kickmeier-Rust and D. Albert, "Micro-adaptivity: Protecting immersion in didactically adaptive digital educational games," *Journal of Computer Assisted Learning*, vol. 26, no. 2, pp. 95–105, 2010.

[10] R. Bartle, "Hearts, clubs, diamonds, spades: Players who suit muds," *Journal of MUD research*, vol. 1, no. 1, 1996.

[11] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen, "A study of the difficulties of novice programmers," *Acm sigcse bulletin*, vol. 37, no. 3, pp. 14–18, 2005.

[12] O. Astrachan, "Bubble sort: an archaeological algorithmic analysis," *ACM Sigcse Bulletin*, vol. 35, no. 1, pp. 1–5, 2003.

[13] R. S. Baker, M. Boilen, M. T. Goodrich, R. Tamassia, and B. A. Stibel, "Testers and visualizers for teaching data structures," *ACM SIGCSE Bulletin*, vol. 31, no. 1, pp. 261–265, 1999.

[14] D. Dicheva and A. Hodge, "Active learning through game play in a data structures course," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pp. 834–839, 2018.

[15] E. Harpstead, B. A. Myers, and V. Aleven, "In search of learning: Facilitating data analysis in educational games," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, (New York, NY, USA), p. 79–88, Association for Computing Machinery, 2013.

[16] B. Martin, K. R. Koedinger, A. Mitrovic, and S. Mathan, "On using learning curves to evaluate its," in *Proceedings of the 2005 Conference on Artificial Intelligence in Education: Supporting Learning through Intelligent and Socially Informed Technology*, (NLD), p. 419–426, IOS Press, 2005.

[17] J. K. Haas, "A history of the unity game engine," 2014. Worcester Polytechnic Institute, https://digitalcommons.wpi.edu/iqp-all/3207/ (Accessed 13 December 2020).

[18] E. Balagurusamy, *Programming in C#: A Primer.* McGraw-Hill Education, forth ed., 2010.

[19] C. Kazimoglu, M. Kiernan, L. Bacon, and L. Mackinnon, "A serious game for developing computational thinking and learning introductory computer programming," *Procedia - Social and Behavioral Sciences*, vol. 47, pp. 1991 – 1999, 2012. Cyprus International Conference on Educational Research (CY-ICER-2012)North Cyprus, US08-10 February, 2012 http://www.sciencedirect.com/science/article/pii/S1877042812026742 (Accessed 13 December 2020).

[20] J. M. Wing, "Computational thinking," *Communications of the ACM*, vol. 49, no. 2, pp. 33–35, 2006.

[21] J. A. Qualls and L. B. Sherrell, "Why computational thinking should be integrated into the curriculum," *Journal of Computing Sciences in Colleges*, vol. 25, no. 5, pp. 66–71, 2010.

[22] T. Barnes, H. Richter Lipford, E. Powell, A. Chaffin, and A. Godwin, "Game2learn: building cs1 learning games for retention," vol. 39, pp. 121–125, 01 2007.

[23] A. Chaffin, K. Doran, D. Hicks, and T. Barnes, "Experimental evaluation of teaching recursion in a video game," in *Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*, pp. 79–86, 2009.

[24] E. B. Costa, A. M. Toda, M. A. A. Mesquita, and J. D. Brancher, "Dslep (data structure learning platform to aid in higher education it courses)," *International Journal of Computer and Systems Engineering*, vol. 8, no. 4, pp. 1143 – 1148, 2014.

[25] N. Kaur and G. Geetha, "Play and learn ds: Interactive and gameful learning of data structure," *International Journal of Technology Enhanced Learning*, vol. 7, pp. 44–56, 09 2015.

[26] M. Eagle and T. Barnes, "Experimental evaluation of an educational game for improved learning in introductory computing," *SIGCSE'09 - Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, vol. 41, pp. 321–325, 03 2009.

[27] P. Moreno-Ger, J. Torrente, Y. G. Hsieh, and W. T. Lester, "Usability testing for serious games: Making informed design decisions with user data," *Advances in Human-Computer Interaction*, vol. 2012, 2012. Hindawi, https://www.hindawi.com/journals/ahci/2012/369637/ (Accessed 13 December 2020).