Algoritmos y estructuras de datos

Genetic

El algoritmo genético es usado en el sistema como uno de los dos algoritmos encargados de generar y solucionar los códigos secretos recibidos como parámetro con el objetivo de garantizar la victoria de la "máquina".
Este algoritmo radica en la imitación de los genes y su evolución condicionada por el azar a lo largo del tiempo, aun así, nuestra implementación influencia en este algoritmo para que se favorezca el azar en dirección a la solución del código secreto mediante el *fitness*.

En primer lugar, se genera una población/conjunto de códigos inicializados con valores al azar, pero todos de acuerdo a la longitud y los números de colores diferentes configurados en la partida en cuestión. Esta población deberá pasar por un cierto número de evoluciones que desencadenará en una generación por evolución, hasta que se resuelva el código secreto y/o se alcance el número de generaciones máximas configuradas o maxSteps.
La clave de este algoritmo reside en cómo se realiza esta evolución y que conjunto de algoritmos internos se usan para influenciarla. En nuestro caso, la evolución va a conformarse en el orden en que se mencionan, de los siguiente algoritmos internos: selección, recombinación, mutación, permutación y por último, inversión.

Todo empieza con la recombinación, precedida de dos selecciones de un número de "torneos" configurados inicialmente, que son códigos escogidos al azar de entre la población a evolucionar y que terminaran con la "selección" del mejor código (fitness más alto) tanto en la primera selección como en la segunda. Así, es como obtendremos los dos códigos a recombinar. La recombinación no es más que la creación de un nuevo código a partir de los dos seleccionados, favoreciendo la aportación de colores (genes) de uno ellos, el que se acerque más al código secreto o como se establece en el sistema, al que sea más fitness. En este caso, usamos el producto del umbral de recombinación, configurado antes de la ejecución, por un factor fitness, que como se ha mencionado anteriormente, decanta la balanza hacia la mayor aportación de un código u otro, dando cabida al azar de la genética.

Debemos incidir en el término fitness, ya que es el parámetro que determina qué código es mejor y cual es peor. El parámetro fitness es calculado y guardado internamente en cada código como atributo, y este cálculo se basa en contar el número de aciertos totales (acierto en color y posición), y aciertos parciales (acierto en color) que contiene el código respecto al código secreto, en nuestro caso, definiendo su máximo igual a la longitud de los códigos, y dando entonces, el peso de 1 unidad a los aciertos totales y de 0.2 a los aciertos parciales, premiando los aciertos totales frente a los parciales, ya que estos últimos a medida que se agranda el abanico de colores posibles pierden relevancia.

Ésta recombinación se realiza tantas veces como códigos tenga la población, característica configurada previamente a la ejecución del algoritmo.

Seguidamente, se lleva a cabo la mutación de todos y cada uno de los códigos de la población afectada por recombinación, dando, de nuevo y como será habitual en todos los algoritmos internos, al azar mediante la tasa de mutación. Cabe recalcar que la mutación se realiza a nivel de cada color que contiene el código, por tanto, un máximo de “longitud del código” veces por código.

A continuación, tras la recombinación y mutación, procedemos a la permutación, donde de igual manera que en los procesos de evolución anteriores, influimos en todos y cada uno de los códigos condicionados por el producto del umbral de permutación (configurada previamente a la ejecución) por el tanto por uno de la diferencia del fitness del código respecto al máximo fitness possible entre el máximo fitness ((*maxFitness - fitnessCodigo) / maxFitness*). Ésta permutación consiste en “permutar” y/o intercambiar la posición de dos colores de un código completamente al azar.

El antepenúltimo proceso de evolución que sufre la población a evolucionar es la llamada, inversión, donde de la misma forma que en la permutación, se escogen dos posiciones aleatorias de un código y se invierte el orden de esa sección de colores del código. Como ya es habitual, este proceso afectará a todos los códigos condicionados por el producto del umbral de inversión (configurado previamente a la ejecución) por el mismo tanto por uno definido en el proceso de permutación.

Ésta población evolucionada es valorada y se almacena su mejor código como parte de una lista que se devolverá en la función “solve” encargada de realizar el algoritmo explicado, dando así lugar a una lista de los mejores códigos, uno por generación, hasta llegar a un código que corresponda con la solución, o sea se, igual al código secreto introducido como parámetro inicialmente, o se alcance el número máximo de generaciones permitidos y/o maxSteps.

Como bien se puede observar en este algoritmo adaptado al juego en cuestión, el azar es influenciado por el parámetro fitness, haciendo así de un algoritmo existente en la naturaleza basado en el azar, otro basado en el azar però mermado por el fitness y por tanto, se podría considerar al conjunto de procesos como procesos fitness del algoritmo genético.

Five Guess

El algoritmo Five Guess es el segundo algoritmo encargado de generar y solucionar los códigos secretos recibidos como parámetro con el objetivo de garantizar la victoria de la “máquina”.

En primer lugar el algoritmo inicializa un conjunto de códigos posibles, el número de códigos posibles será igual al número de colores elevado a la longitud del código, la inicialización de estos códigos se hace mediante una función recursiva.

La máquina escogerá como primer código una pareja de números tal y como nos explica Donald Knuth, ya que si no cogemos una pareja de números y cogemos números aleatorios el algoritmo no será capaz de resolver el código en 5 turnos.

En caso de que el primer código escogido sea la solución el algoritmo termina, en caso contrario entra en un bucle que terminará en caso de haber encontrado la solución o haber superado los turnos máximos de la configuración de la partida.

Dentro de este bucle la máquina primero de todo obtiene la corrección del último código enviado y elimina del conjunto de códigos posibles todos aquellos que quedan descartados con dicha corrección, la forma de descartar un código, es comparar la última corrección obtenida con la que generaría dicho código en caso de ser correcto, en caso de que sea la misma el código sigue siendo un codigo candidato, en caso contrario es descartado.
En segundo lugar selecciona el mejor código de los restantes para ser enviado, esta selección se hace mediante el algoritmo “MiniMax” que lo que busca es que código en el peor de los casos será capaz de eliminar un mayor número de códigos posibles, en caso de empate escoge el primero que encuentra.
Finalmente el código escogido es enviado y añadido al conjunto de códigos de la máquina, tras esto comprueba si el código enviado es la solución e incrementa en 1 el número de turno actual. En caso de no haber adivinado el código y seguir sin superar el número de turnos máximos volverá a empezar el bucle.

Una vez terminado el bucle devuelve el conjunto de códigos de la máquina.

Principales estructuras de datos del sistema

- List<Integer> codigo:
    - También usado en ED como guess, resultado o codigoSecreto. Es mostrado ya en las estructuras de datos del algoritmo genético, ésta estructura de datos es una de las dominantes del sistema. És una lista de enteros que representan los colores (valor en entero), tanto de los códigos correspondientes a las adivinanzas como de los correspondientes a las correcciones y/o resultados respectivos a estas adivinanzas. Podemos encontrar usos en: IPartida, Partida, Tablero, CalculoFitness, Evolucion.
- List<List<Integer>> tablero:
    - También usado en ED como tableroResultados, códigos o, se guardan los tableros, que no dejan de ser una matriz (filas x columnas) donde en cada posición tenemos un Integer que indica el color de la pelota. Podemos encontrarlo en Tablero, MaquinaGenetic.
- Vector<Color> colores:
    - Es un vector de objetos de tipo Color, clase que contiene el valor en Integer del color, oscilando entre 0 y el número máximo de colores diferentes - 1, que representa los colores del código, ítem representado por la clase Código.
- Codigo[] codigos:
    - Es una array de objetos de tipo Código que representa el atributo fundamental y principal de la clase contenedora o clase conjunto de códigos, PoblacionCodigos, que reúne todos los códigos de la población en cuestión. Podemos encontrarlo en PoblacionCodigos.
    - También es del mismo tipo el array que indica un tema y el array con los temas disponibles
- List<List<Integer>> codigosPosibles:
    - Es un arrayList donde se almacenan los posibles códigos para el algoritmo FiveGuess, cada Integer representa un color. Podemos encontrarlo en MaquinaFiveGuess.
- List<List<Object>> partidas:
    - Tenemos un arrayList que contiene una lista de objetos. Aquí encontramos información sobre las partidas tanto para el ranking global como para los récords. Podemos encontrarlo en las clases mencionadas anteriormente además de en CtrlPersistencia entre otros..
- ArrayList<Integer> partidasGuardadas:
    - En este arrayList pasamos los identificadores de las partidas guardadas. Esta estructura la podemos encontrar en algunos lugares del código como en VistaPartidasGuardadas, ya que, se pasan muchas veces,

Otras estructuras de datos relevantes del sistema:

- List<Object> configuracion:
    - Lista de objetos de tipo genérico utilizada para almacenar los atributos de configuración de la clase Configuración fuera de ésta, para así evitar un acoplamiento incoherente e innecesario de la clase Configuración y aislar el uso de objetos tipo Configuración con la excepción de la classe Partida, donde sí es coherente, útil y necesario la transferencia de información mediante este tipo de objetos.
- HashMap<String,Perfil> Usuarios
    - Mapa para referenciar los usernames de los usuarios con el Perfil, teniendo así, los atributos de este último. Encontramos esta ED en ControladorDominio.
- ArrayList<Partida>
    - Lista de Partidas conteniendo los atributos de esta para inicializarlas. Podemos encontrarlo en la clase de Perfil.
- List<Object> listaObjetos
    - Lista de objetos tipo genérico utilizada para almacenar los atributos de .La encontramos en la clase DriverPerfil.
- Integer[] codigoRevelador
    - Es un array de objetos tipo Integer que representa el atributo fundamental en cuanto a pistas se refiere. Este codigoRevelador contiene posiciones con la solución, y, a medida que pidas pistas, dará una respuesta correcta del código secreto teniendo en cuenta las que ya ha dado y el número de pistas que tienes. Podemos encontrarlo en PartidaCodebreaker.
- List<List<Double>> tablaDificultadToSteps
    - Teniendo esta especie de matriz, una lista de listas double, simulamos un tablero donde cargamos un archivo de dificultad. Podemos encontrarlo en PartidaCodemaker.
- Partida[] partidas
    - Array de objetos tipo Partida que representa el atributo fundamental y principal de la clase RankingGlobal, que reúne el top 10 de todas las partidas jugadas en función de la puntuación obtenida. Podemos encontrarlo en RankingGlobal.
- Pair<Integer,String>[][] records
    - Matriz de pares donde almacenamos el número de turnos empleados y el nombre de usuario del jugador que, con cierta longitud de código y cierto número de colores, ha conseguido resolver el puzzle en el menor número de turnos. Podemos encontrarlo en RecordsMoviments.
- Map<Color,String>colorMap:
    - En este mapa se almacenan algunos colores principales, y su respectiva traducción a un string, es decir, tenemos un color en primera posición, y su nombre en la segunda. Podemos encontrar esta estructura en VistaPartidaCodeBreaker o VistaPartidaCodeMaker.

- String[] column:
    - Este array de strings lo utilizamos en clases como VistaRankingGlobal y VistaRecords para almacenar datos que se vayan a introducir en la tabla cuando se muestren las vistas.
- String[][] data:
    - Matriz de strings donde se almacenan los datos de los jugadores (rango, nombre de usuario, puntuación) para rellenar la tabla de la VistaRankingGlobal y VistaRecords.
- List<Color[]> temas
    - En este arrayList de un vector de colores, podemos encontrar todos los temas. Un tema es un conjunto de colores, que se usa para la configuración del juego. En esta lista, guardamos todos los temas para usarlos en clases como VistaTemas.

Algoritmos y costes

A continuación están todas las clases y sus métodos que generan costes listados y ordenados. Nótese que los métodos y funciones que no están incluidos en la siguiente ordenación, suponen un coste constante. Esta suposición está hecha en base a que todas las operaciones y llamadas que podemos realizar en java (desde sumas a getters y setters y operaciones de conteo ofrecidas por java), las suponemos que tienen un coste constante en todo momento. Variables como numeroTurnos, longitudCodigo y numeroCodigos, las documentamos como si pudieran valer n, pero en nuestro caso, estas variables están acotadas, por lo tanto provocaría siempre, y en todos los casos y métodos en los que los bucles hagan este número de iteraciones, costes constantes.

- ControladoresDominio
    - ControladorDominio
        - Método para crear una partida (creaPartida())
            - Este método tiene una llamada a GuardaPerfiles() de la misma clase, que tiene un coste de n, lo que provoca, que esta función, también tenga un coste lineal.
              Coste función: O(n)
        - Método para ejecutar un turno (ejecutarTurno())
            - Este método ejecuta un turno de la partida. Tiene una llamada a GuardaPerfiles() de esta misma clase, que tiene un coste de n. Por tanto, el coste de este método, también es lineal.
              Coste función: O(n)
        - Método para registrar a un usuario (registrar())
            - Tiene una llamada a GuardaPerfiles(), con coste n, por lo que provoca un coste, también, de n o lineal en esta función.
              Coste función: O(n)
        - Método para consultar el ranking global (obtenRanking())
            - Este método recorre el array RankingPartidas guardando información en caso de que la posición accedida tenga algo contenido. Sabiendo que la longitud del array puede ser 10 como máximo, tenemos un vector que da 10 vueltas, por tanto, tiene un coste de 10 o constante.
              Coste función: O(1)
        - Método para obtener las partidas por acabar(obtenPartidasPerfilPorAcabar())
            - Este método llama a obtenIdsPartidasSinAcabar() de perfil, que tiene un coste de n, por tanto, esta función también tiene coste lineal.
              Coste función: O(n)
        - Método para obtener las partidas acabadas(obtenPartidasPerfilAcabadas())

        - Este método llama a obtenIdsPartidasAcabadas() de perfil, que tiene un coste de n, por tanto, esta función también tiene coste lineal.
          Coste función: O(n)
- Método para cargar una partida (cargarPartida())
    - Este método recorre las partidas sin acabar usando un bucle for, y cuando encuentra la que tiene un id igual que el pasado por parámetro, la devuelve. El bucle llama a la función ObtenPartida de Perfil y hace n iteraciones. Como la función tiene coste n, el coste de este método es nxn = n^2 o cuadrático.
      Coste función: O(n^2)
- Método para guardar los perfiles ·GuardaPerfiles())
    - Esta función contiene un for donde añade todos los perfiles de usuario en una lista. Esto provoca que el bucle haga tantas iteraciones como usuarios haya, además, llama al método escribePerfiles del ControladorPersistencia, que también tiene un bucle for para recorrer los perfiles. Esto se hace uno a continuación del otro, por tanto, este método tiene coste n+n = 2n, que significa coste n o lineal.
      Coste función: O(n)
- Método para obtener una lista de usuarios (ObtenListaUsuarios())
    - Este método contiene 2 bucles for simples, uno a continuación del otro, por tanto, en caso peor, cada bucle harà n iteraciones, por tanto, coste 2n, que se traduce a coste n o lineal, en general.
      Coste función: O(n)
- Método para obtener los temas disponibles (obtenTemas())
    - Esta función contiene un bucle recorriendo todos los temas disponibles, pero al solamente tener 3 temas disponibles, tenemos un coste 1 o constante.
      Coste función: O(1)
- Método para eliminar el perfil (eliminaPerfil())
    - Este método llama a GuardaPerfiles, que como hemos visto tiene un coste n o lineal y no hace mucho más. Por tanto el coste de esta función es n.
      Coste función: O(n)
- Método para eliminar todos los perfiles (eliminaTodosPerfiles())
    - Funciona de la misma manera que el método anterior.
      Coste función: O(n)

        - Método para eliminar la partida de un perfil (eliminaPartidaPerfil()) y método para eliminar una partida sin guardar (eliminarPartidaSinGuardar())
            - Funcionan ambos igual que los 2 anteriores.
              Coste función: O(n)
- Dominio
    - Partida
        - Método para generar el resultado de la adivinanza (generaResultado())
            - Este método hace 2 recorridos al código secreto. El primero para obtener los aciertos totales y el segundo para obtener los aciertos parciales. Dado que ambos recorridos hacen tantas iteraciones como longitudCodigo, y suponiendo que pueda ser n, tenemos un coste de n+n, por tanto, 2n, o lo que es lo mismo en costes, coste n o lineal.
              Coste función: O(n)
        - Método abtracto que obtiene una pista (obtenPista())
            - En PartidaCodebreaker tenemos un bucle while que itera hasta encontrar una posición que no esté revelada. Dado que calculamos la posición a mirar con un random, en peor caso, el coste de esta función es infinito, pero, dada la imposibilidad de esto, supondremos un coste de n o lineal al tener todas las posiciones la misma probabilidad de ser escogida. En PartidaCodeMaker se llama a generaResultado, que como hemos visto tiene coste n, por tanto, el coste también sería n.
              Coste función: O(n)
        - Método para informar la adivinanza al tablero
            - Esta función hace una comprobación de los colores introducidos usando un bucle for que hace tantas iteraciones como posiciones tenga adivinanza. Ya que puede tener n posiciones, el bucle haría n iteraciones, por tanto, tendría un coste de n o lineal.
              Coste función: O(n)
        - Método para informar el resultado al tablero
            - Este método es similar al anterior. Comprueba los valores de una lista resultado recorriendola haciendo uso de un bucle for. Este bucle obviamente, hará tantas iteraciones como valores contenga resultado, y por tanto, si tiene n valores, hará n iteraciones provocando un coste de n o lineal.
              Coste función: O(n)
    - PartidaCodebreaker

        - Método para generar un código secreto aleatorio (generarCodigoSecreto())
            - Este método se encarga de generar un código secreto aleatorio para la partida. Tiene un solo bucle que hará tantas iteraciones como longitudCodigo, por tanto n, es decir, coste n o lineal.
              Coste función: O(n)
        - Método para obtener pistas (obtenPista())
            - Este método tiene como finalidad dar pistas al usuario cuando éste lo requiera. Tenemos un bucle while que itera hasta encontrar una posición que no esté revelada. Dado que calculamos la posición a mirar con un random, en peor caso, el coste de esta función es infinito, pero, dada la imposibilidad de esto, supondremos un coste de n o lineal al tener todas las posiciones la misma probabilidad de ser escogida.
              Coste función: O(n)
    - PartidaCodemaker
        - Método para ejecutar el turno (ejecutarTurno())
            - Este método ejecuta un turno para la máquina intentando resolver el código propuesto. Hace 2 llamadas a solve() de Maquina, que es lo único que puede llegar a ser costoso dado que són los algoritmos más costosos del programa. Sabiendo que solve() tiene un coste de n^3 en maquinaGenetic, y, un coste de n^m, en maquinaFiveGuess, provoca que esta función, tenga el coste mayor, por tanto, coste n^m.
              Coste función: $O(n^m)$
            - Método para obtener pistas (obtenPista())
                - Este método tiene como finalidad dar pistas al usuario cuando éste lo requiera. Hace una llamada a generaResultado, en peor caso, que tiene un coste n o lineal, como ya hemos visto.
                  Coste función: O(n)
    - Tablero
        - Constructor de tablero e inicializador (Tablero)
            - Es solo un constructor que va añadiendo listas en función de la longitud del código. Son las columnas de la matriz, por tanto, el número de turnos configurado, n. El coste al solo recorrer esto es n o lineal.
              Coste función: O(n)
    - CalculoFitness
        - Método para informar el código secreto (informaCodigoSecretoGenetic())

        - Simplemente informa el código. Únicamente depende del tamaño del código secreto, por tanto, coste n o lineal.
          Coste función: O(n)
    - Método para el cálculo de la fitness para un código (fitnessCodigo())
        - Hace un recorrido para puntuar los aciertos totales y parciales del resultado. Dado que el número de iteraciones que hace el único bucle depende de la longitud del código y la del código secreto, tenemos que el coste es n o lineal.
          Coste función: O(n)
    - Método para obtener el resultado mediante el código introducido (obtenResultado())
        - Este método se encarga de obtener el resultado, es decir, la corrección del código que se ha introducido. Tiene 2 bucles que dan tantas iteraciones como tamaño tenga el codigoSecreto. Por tanto, tenemos un coste de n+n, que es lo mismo que 2n lo que significa, en cuanto a costes, coste n o lineal.
          Coste función: O(n)
- Codigo
    - Constructor de Codigo (Codigo())
        - Es solo un constructor que va añadiendo colores a un vector de longitudCodigo posiciones. Por tanto, esto provoca que el bucle pueda hacer n iteraciones, entonces, tendríamos coste n o lineal.
          Coste función: O(n)
    - Método para dar una lista de enteros (toListInteger())
        - Este método rellena una lista con los valores de Color definidos en enteros. El bucle único da tantas vueltas como longitudCodigo, por tanto, provoca un coste de n o lineal.
          Coste función: O(n)
    - Método que llama a fitnessCodigo en caso de que fitness sea 0 (fitnessCodigo())
        - Este método llama a fitnessCodigo() de CalculoFitness en caso de que el valor de fitness sea 0. Como hemos visto anteriormente, la función llamada tiene un coste de n o lineal, por lo tanto, también está.
          Coste función: O(n)
- Evolucion
    - Método para la evolución de una población (evolucionaPoblacionCodigos)

- Este método va almacenando el mejor código teniendo en cuenta recombinaciones. Tiene un único bucle que da tantas vueltas como numeroCodigos, pero, dentro de este hace dos llamadas a seleccionaCodigo() y a fitnessCodigo y una a recombinaCodigos(), mutaCodigo(), permutaCodigo() e invierteCodigo(), que tienen todos coste n, por tanto, deducimos que tiene un coste nx(2x(n)+n+n+2x(n)+n+n), en peor caso, que es lo mismo que nx(8n), es decir, cuadràtico o coste n^2.
  Coste función: $O(n^2)$

- Método para la selección de un código en función del fitness(seleccionaCodigo())
  - Como se acaba de decir, el método hace selección de código en función del fitness. Volvemos a tener un único bucle que da tantas vueltas como numTorneos, por tanto, coste n o lineal.
    Coste función: O(n)
- Método para la recombinación de códigos (recombinaCodigos())
  - Este método genera un código a partir de combinar otros 2 teniendo en cuenta el fitness y un umbral de recombinación. Hace 2 llamadas a fitnessCodigo. Además, la función solo tiene un bucle, que hace tantas iteraciones como longitudCodigo, lo que nos da coste 2x(n)+n=3n o lo que es lo mismo, coste n o lineal.
    Coste función: O(n)
- Método para la mutación de un código en función del fitness (mutaCodigo())
  - El método consiste en obtener un código nuevo, mutado, a partir de otro existente teniendo en cuenta, ahora, la tasaMutacion y la diferencia de fitness en tanto por uno. Tiene 2 bucles, que iteran sobre longitudCodigo por tanto, coste n+n, es decir, 2n, que en costes se traduce a coste n o lineal.
    Coste función: O(n)
- Método para la permutación de 2 posiciones random del código introducido (permutaCodigo())
  - Dado un código, este método se encarga de permutar 2 de las posiciones de este, generando así, un nuevo código con los mismos colores que el anterior. Hay un solo bucle que recorre en función de longitudCodigo. Por tanto, al igual que en el caso anterior, tenemos un coste n o lineal.
    Coste función: O(n)

        - Método para la inversión de 2 posiciones random del código introducido(invierteCodigo())
            - Esta función se comporta similar a la anterior, haciendo, como ya se ha dicho, una inversión entre 2 posiciones del código. Nos encontramos con 2 bucles que no dependen el uno del otro, y el segundo solo se ejecuta dependiendo de la condición del if. Por tanto, en peor caso, tenemos que los 2 se ejecutan, con coste n, lo que significa 2n, que traducido a costes, significa coste n o lineal.
              Coste función:O (n)
    - PoblacionCodigos
        - Constructor de PoblacionCodigos (PoblacionCodigos())
            - Es un simple constructor para inicializar la población. Se basa en un bucle que itera tantas veces como numeroCodigos, por tanto, podemos llegar a un coste n o lineal.
              Coste función:O (n)
        - Método para comprobar el código más apto (mejorCodigo())
            - En este método comprobamos entre todos los códigos almacenados cuál es el mejor, el que mejor se ajusta, el fittest. Solo tenemos un bucle que comprueba 1 a 1 todos los códigos y que llama a la función fitnessCodigo() ya mencionada, entonces tenemos tantas iteraciones como numeroCodigos, por la llamada de la función, provoca un coste nxn= n^2, es decir coste n^2 o cuadràtico.
              Coste función:O (n^2)
    - MaquinaFiveGuess
        - Método que inicializa los posibles códigos
            - Este método llama a la función inicializaPosiblesCodigosRecursivos() de la propia clase con un coste de n^m, entonces, esta función también tiene ese coste.
              Coste función:O (n^m)
        - Método que inicializa los posibles códigos (inicializaPosiblesCodigosRecursivos())
            - Esta función añade a codigosPosibles todos los códigos que puedan ser la solución del puzzle. Esta función contiene un único bucle for que hace tantas iteraciones como número de colores tenga como parámetro. Dentro de este propio bucle se llama a sí misma. Si el bucle numeroColores vale n, y la longitud del código es m, tenemos un coste de n^m.
              Coste función: O(n^m)
        - Método para resolver el código secreto introducido (solve())

        - Este método comienza llamando a la función anterior (inicializaCodigosPosibles()), seguidamente llama a generarCodigoInicial(), y procede a ejecutar un bucle while de pasosMaximos iteraciones (n en peor caso), en el que dentro se llama a obtenResultado y a actualizaCodigosPosibles de esta misma clase. Por tanto, en conjunto, el coste de esta función sería (n^m)+n+nx(n^3+n^2), por lo que tendríamos un coste en peor caso de n^m.
          Coste función: O(n^m)
    - Método que genera el primer código (generarCodigoInicial())
        - Este método genera el primer código introducido por la máquina. Contiene 2 bucles for que se ejecutan uno después del otro. Cada bucle se ejecuta longitudCodigo número de veces/2. Por tanto, tenemos un coste de n/2 + n/2 = n, por tanto un coste n o lineal.
          Coste función: O(n)
    - Método que actualiza los posibles códigos (actualizaCodigosPosibles())
        - Esta función contiene un bucle for que itera tantas veces como codigosPosibles haya, actualizando los códigos que había y modificándose por otros que más se ajusten. Dentro de este bucle for se hace una llamada a obtenResultado(), una función de la misma clase con un coste de n^2. Por tanto,el coste de esta función se resumiría en nx(n^2)=n^3.
          Coste función: O(n^3)
    - Método que devuelve la corrección del código introducido (obtenResultado())
        - La función comienza con un bucle for itera tantas veces como longitudCodigo, y le sigue otro bucle for, que contiene otro bucle for en su interior. Por tanto, si longitudCodigo tiene un valor n, el coste de esta función sería n+(nxn) = n^2+n. En esencia, esta función tiene un coste de n^2 o cuadrático.
          Coste función: O(n^2)
    - Método que devuelve todos los resultados posibles (obtenResultadosPosibles())
        - En esta función encontramos un bucle for dentro de un bucle for. Ambos iteran tantas veces como longitudCodigo, por tanto, si tiene un valor de n, el coste sería de nxn=n^2, por tanto coste n^2 o cuadrático.
          Coste función: O(n^2)
    - Método que escoge la mejor opción (miniMax())

- Esta función escoge y devuelve la mejor opción para ser el siguiente código enviado. Contiene 3 bucles for, uno principal, uno dentro del principal, y el último dentro del secundario. El primer bucle itera sobre todos los codigosPosibles, el segundo según los resultadosPosibles y el tercero vuelve sobre los codigosPosibles. Todos estos valores, en peor caso, son n. En el segundo bucle, el número máximo de iteraciones se calcula usando la función obtenResultadosPosibles(), que como hemos visto, tiene coste n^2. Todo esto provoca un coste de nx(n^2+nx(n))=2n^3, por tanto, coste n^3.
  Coste función: O(n^3)

- MaquinaGenetic
  - Método para resolver el código secreto introducido (solve())
    - Este método hace una llamada a la constructora de PoblacionCodigos() que, como hemos visto tiene coste n, y va haciendo iteraciones para conseguir encontrar el mejor código hasta encontrar el correcto o se tope con límite de generaciones i/o maxSteps. Dentro de este while hace una llamada a evolucionaPoblacionCodigos(), que tiene coste n^2 y otra a toListInteger(), que tiene coste n. Tenemos que en peor caso, haga tantas iteraciones como se le permita, es decir, n, por tanto, el coste nos quedaría nx(n^2+n), lo que provoca un coste de n^3. **ATENCIÓN:** como ya se ha indicado varias veces, estamos suponiendo que variables como longitudCodigo, numeroTurnos y numeroCodigos puedan ser n. En el caso pràctico, estas variables están limitadas a valores, lo que provocaría en todo caso costes constantes.
      Coste función: O(1)

- Perfil
  - Método para obtener una lista con los ids de las partidas con/sin acabar (obtenIdsPartidasSinAcabar() y obtenIdsPartidasAcabadas())
    - Estos 2 métodos hacen, en esencia, lo mismo, recorren todas las partidas, y dependiendo de si están acabadas o no, las almacenan en una lista. Sabiendo que el número total de partidas es n, y suponiendo que el número de partidas empezadas sea m, entonces, el número de partidas acabadas será n-m, que, en todo caso, m<n. Dado que los costes sean m y n-m, provoca un coste lineal, y por tanto, en general, el coste de ambas funciones es n.
      Coste función: O(n)

        - Método para obtener el id de la última partida (obtenIdUltimaPartida())
            - Este método contiene un bucle for que hace tantas iteraciones como partidas haya y obtiene su id para comprobar si es el último. Si hay n partidas guardadas, esta función provoca un coste n o lineal.
              Coste función: O(n)
        - Método para eliminar una partida (eliminarPartida())
            - Este método recorre todas las partidas comprobando su id, si el id coincide con el pasado como parámetro, la elimina. Si hay n partidas, y, en peor caso, resulta que la partida a borrar es la última, hará n iteraciones buscándola. Esto provocaría un coste de n o lineal.
              Coste función: O(n)
        - Método para obtener una partida (ObtenPartida())
            - Dado un identificador de una partida, busca entre las partidas una con este identificador para devolverla. Igual que en el caso anterior, provoca un coste de n o lineal.
              Coste función: O(n)
    - Ranking Global
        - Método para añadir una Partida (anadirPartida())
            - Esta función añade una partida al ranking global. Dado que en este ranking solo tenemos 10 partidas guardadas, todas los bucles harán, como máximo, 10 iteraciones, por lo que podemos afirmar, que, al haber 3 bucles, con longitud fija, el coste sería 10+10+10 (en peor caso), lo que se obtiene un número entero, por tanto tenemos coste 30, lo que significa tener un coste constante.
              Coste función: O(1)
- ControladoresPresentacion
    - ControladorPresentacio
        - Método para ejecutar un turno (ejecutarTurno())
            - Esta función llama a ejecutarTurno de controladorDomino, que tiene un coste de n o lineal, provocando el mismo coste en esta función.
              Coste función: O(n)
        - Método para obtener las partidas guardadas (obtenPartidasGuardas())
            - Este método llama a obtenPartidasPerfilPorAcabar() de controladorDominio, que tiene un coste de n, y también lo tiene, entonces, esta función.
              Coste función: O(n)
        - Método para cargar una partida (cargarPartida())

        - Esta función llama al método cargarPartida de ControladorDominio, que tiene un coste de n^2, lo que provoca un coste de n^2 o cuadrático en esta función.
          Coste función: O(n^2)
- Método que carga la vista de la partida (cargarVistaPartida())
    - Este método llama a cargar() en vistaPartidaCodeBreaker o vistaPartidaCodeMaker. En el primer caso, tiene un coste de n^2, y en el segundo, un coste de n^3. Por tanto, tenemos un coste de n^2+n^3, lo que provoca un coste en peor caso de n^3.
      Coste función: O(n^3)
- Método para obtener los temas disponibles (obtenTemas())
    - Este método obtiene todos los temas disponibles llamando a obtenTemas() del controlador de dominio, que tiene un coste constante, ya que el bucle siempre hace 3 iteraciones, por tanto el coste de esta función también es constante o 1.
      Coste función: O(1)
- Método para obtener el código inicial (obtenCodigoInicial())
    - Este método hace una llamada a creaPartida() del controlador de dominio que tiene un coste de n o lineal lo que provoca el mismo coste en esta función.
      Coste función: O(n)
- Método para iniciar como CodeBreaker (inicializarPartidaCodeBreaker())
    - Este método hace una llamada a creaPartida() del controlador de dominio que tiene un coste de n o lineal lo que provoca el mismo coste en esta función.
      Coste función: O(n)
- Método para informar del código (informaCodigo())
    - Este método hace 2 llamadas a ejecutarTurno() del controladorDominio. Cada llamada tiene un coste de n, lo que provoca que esta función tenga un coste de n+n=2n, o lo que es lo mismo, un coste de n o lineal.
      Coste función: O(n)
- Método que ejecuta un turno vacío (procesar())
    - Este método hace una llamada a ejecutarTurno() del controladorDomino, que tiene un coste de n o lineal, provocando el mismo en esta función.
      Coste función: O(n)
- Método para guardar partidas (guardarPartidas())
    - Este método hace una llamada a GuardaPerfiles(), que tiene un coste de n o lineal, provocando el mismo en la función.

Coste función: O(n)

- Método para eliminar una partida (eliminarPartida())
  - Este método hace una llamada a eliminarPartidasSinGuardar del controlador de dominio que tiene un coste de n o lineal, provocando el mismo en esta función.
    Coste función: O(n)
- Método para eliminar una partida (eliminarPartida())
  - Este método hace una llamada a eliminarPartidaPerfil del controlador de dominio que tiene un coste de n o lineal, provocando el mismo en esta función.
    Coste función: O(n)

- ControladorVistaSesion
  - Método para empezar la partida (empezarPartida())
    - Este método hace una llamada a inicializarPartidaCodeBreaker() o a obtenCodigoInicial() del controlador de presentación. Ambas tienen un coste de n o lineal, lo que provoca el mismo coste en esta función.
      Coste función: O(n)
  - Método para obtener las partidas guardadas (obtenPartidasGuardadas())
    - Este método llama a obtenPartidasGuardas() del controlador de presentación. Esta función tiene un coste de n, ya que el método al que llama también tiene ese coste.
      Coste función: O(n)
  - Método para cargar una partida (cargarPartida())
    - Este método carga una partida llamando a la función cargarPartida del controlador de presentación, que tiene un coste de n^2. Esta función, también tiene un coste de n^2 o cuadrático, entonces.
      Coste función: $O(n^2)$
  - Método para mostrar la vista de los temas (mostrarVistaTemas())
    - Este método hace una llamada a obtenTemas() del controlador de presentación, que como hemos visto, tiene un coste constante, lo que provoca un coste constante o 1, también en la función.
      Coste función: O(1)
  - Método para actualizar el tema (actualizaTema())
    - Este método hace una llamada a actualizar() de la vista de configuración. Esta llamada tiene un coste de n, lo que provoca un coste de n o lineal en esta función.

Coste función: O(n)

- Método para eliminar una partida (eliminarPartida())
  - Este método llama a eliminarPartida() del controladorPresentacion, con un coste de n, lo que provoca un coste también de n o lineal en la función.
    Coste función: O(n)

- Persistencia
  - CtrlPersistencia
    - Método para escribir los perfiles (escribePerfiles())
      - Este método guarda los perfiles en un fichero con el nombre “Perfiles.txt”. Contiene un bucle for que recorre todos los perfiles. Si contamos que tenemos n perfiles, provocaría que el bucle hiciera n iteraciones y por tanto un coste de n o lineal.
        Coste función: O(n)
- Vistas
  - BotonPelota
    - Método para acciones de los botones (actionPerformed())
      - Este método hace una llamada a pintarPelotaConColorSeleccionado() de VistaIntroducirCodigo con un coste de n, lo que provoca un coste de n o lineal también en esta función.
        Coste función: O(n)
  - CargarPartidaListener
    - Método para acciones de los botones (actionPerformed())
      - Este método llama a cargarPartida de VistaPartidasGuardadas con un coste de n^2, lo que provoca el mismo coste en esta función.
        Coste función: $O(n^2)$
  - EliminarPartidaListener
    - Método para acciones de los botones (actionPerformed())
      - Este método llama a eliminarPartida de VistaPartidasGuardadas con un coste de n, lo que provoca el mismo coste en esta función.
        Coste función: O(n)
  - VistaConfiguracionPartida
    - Método para acciones de los botones (actionPerformed())
      - Este método llama a empezarPartida de ControladorVistaSesion con un coste de n, lo que provoca el mismo coste en esta función.
        Coste función: O(n)
  - VistaConfiguracionPerfil
    - Método para iniciar los componentes (iniciarComponentes())

- Este método contiene un bucle for que itera tantas veces como numeroColores. En caso de que numeroColores pudiera ser n, el bucle haría n iteraciones y provocaría un coste de n o lineal.
  Coste función: O(n)
- Método para acciones de los botones (actionPerformed())
  - Este método llama a mostrarVistaTemas() de ControladorVistaSesion con un coste constante, lo que provoca el mismo coste en esta función.
    Coste función: O(1)
- Método para actualizar el tema (actualizar())
  - Este método hace un bucle for con tantas iteraciones como componentes del panel de tema. En el peor caso, hace n iteraciones, lo que provoca un coste de n o lineal.
    Coste función: O(n)

- VistaGuardarPartida
  - Método para acciones de los botones (actionPerformed())
    - Este método llama a guardarPartidas(), eliminarPartida() y guardarPartidas() de ControladorPresentacion, con un coste de n+n+n = 3n, lo que provoca un coste de n o lineal.
      Coste función: O(n)
- VistaIntroducirCodigo
  - Método para iniciar los componentes (iniciarComponentes())
    - Este método contiene 2 bucles seguidos independientes que iteran sobre longitudCodigo el primero y sobre numeroColores el segundo. Si estos valores pudieran ser n, el coste de esta función sería n+n=2n, o lo que es lo mismo, coste n o lineal.
      Coste función: O(n)
  - Método para obtener el código (obtenCodigo())
    - Este método contiene un bucle for que itera sobre los componentes del panel del código propuesto. Si este valor fuera n, tendríamos que el bucle haría n iteraciones y en cada iteración, una llamada a convertirColorANumero(), que tiene coste n. Tendría entonces, la función, coste nxn=n^2 o cuadrático.
      Coste función: $O(n^2)$
  - Método que pasa de color a número (convertirColorANumero())
    - Este método contiene un bucle que itera sobre el numeroColores, que, si vale n, tendríamos que la función tiene coste n o lineal.
      Coste función: O(n)

        - Método para pintar una pelota (pintarPelotaConColorSeleccionado())
            - Este método consiste en un bucle for iterando por cada componente del panel del código propuesto pintando la pelota del color que se haya seleccionado. Si tenemos n componentes en el panel, el bucle hará n iteraciones y provocará un coste de n en la función.
              Coste función: O(n)
    - VistaPartidaCodeBreaker
        - Constructor de VistaPartidaCodeBreaker (VistaPartidaCodeBreaker())
            - Es el constructor de la clase. Llama a la función iniciarComponentes() que tiene un coste de n^2.
              Coste función: O(n^2)
        - Método para iniciar los componentes de la vista (iniciarComponentes())
            - La función empieza con 2 bucles for uno dentro del otro que ejecutan numeroTurnos y longitudCodigo iteraciones. Le siguen otros 2 bucles que funcionan de la misma manera, un bucle único que itera sobre numeroColores. Por último hay otro bucle único que itera sobre numeroTurnos. En total provoca un coste de nxn+nxn+n+n= 2n^2+2n , lo que provoca un coste n^2 o cuadrático.
              Coste función:O (n^2)
        - Método para actualizar la pelota (actualizaPelota())
            - Este método contiene un bucle for que hace tantas iteraciones como componentes tenga el panel de juego. Si tiene n, entonces hace n iteraciones y provoca un coste n o lineal.
              Coste función:O (n)
        - Método que desbloquea una fila (desbloqueaFila())
            - Este método contiene 2 bucles independientes que iteran sobre los componentes del panel de juego y sobre los del panel de correcciones. Sabiendo que el número de componentes puede ser n, el coste de esta función vale n+n=2n, es decir, coste n o lineal.
              Coste función:O (n)
        - Método para llenar una fila (llena())
            - Este método contiene 1 bucle que itera sobre los componentes del panel de juego. Sabiendo que el número de componentes puede ser n, el coste de esta función vale n o lineal.
              Coste función: O(n)

        - Método para pintar los resultados (pintaResultados())
            - Este método contiene 2 bucles independientes que iteran sobre las filas y sobre los componentes del panel de correcciones. Sabiendo que el número puede ser n, el coste de esta función vale n+n=2n, es decir, coste n o lineal.
              Coste función: O(n)
        - Método para obtener el código (getCodigo())
            - Este método contiene 2 bucles. El primero hace tantas iteraciones como componentes del panel de juego, y el segundo, incluido en el primero, tantas como numeroColores. Si estos valores fueran n, tendríamos un coste de nxn=n^2, o cuadrático.
              Coste función: O(n^2)
        - Método para ejecutar un turno (ejecutarTurno())
            - En esta función se hace una llamada a ejecutarTurno() del controladorPresentacion y a desbloqueaFila() de la misma clase en peor caso, los cuales tienen un coste de n, por lo que provoca un coste de n+n=2n, por tanto, coste n o lineal.
              Coste función: O(n)
        - Método para cargar una partida (cargar())
            - Este método contiene un bucle dentro de otro. El primero itera tantas veces como filasHechas, y el segundo como componentes del panel de juego. Si suponemos que ambos pueden ser n, esta función tendría un coste de n^2 o cuadrático.
              Coste función: O(n^2)
    - VistaPartidaCodeMaker
        - Método para iniciar los componentes de la vista (iniciarComponentes())
            - La función empieza con 2 bucles for uno dentro del otro que ejecutan numeroTurnos y longitudCodigo iteraciones. Le siguen otros 2 bucles, uno dentro de otro que iteran sobre numeroTurnos y longitudCodigo. Independientemente a estos hay otros 2 bucles, uno dentro del otro que funcionan de la misma manera. Y por último, tenemos un único bucle que itera sobre numeroTurnos. Teniendo en cuenta que estos valores puedan ser n, tendríamos un coste de (nxn)+(nxn)+n=2n^2+n, lo que significa un coste de n^2 o cuadrático.
              Coste función: O(n^2)
        - Método para hacer visible la vista (hacerVisible())

        - Este método tiene 3 bucles independientes que iteran sobre longitudCodigo, los componentes del panel de juego y los del panel de correcciones. Suponiendo que estos valores puedan ser n, tenemos un coste de n+n+n=3n, lo que provoca un coste de n o lineal.
          Coste función: O(n)
- Método para acciones de los botones (actionPerformed())
    - Este método llama a procesar 2 veces y, en peor caso, a ejecutarTurno(). Dado que estas funciones tienen todas un coste de n, el coste de nuestra función es 2xn+n=3n, lo que significa un coste de n o lineal.
      Coste función: O(n)
- Método para escribir la respuesta (escribeRespuesta())
    - Este método contiene un único bucle for que itera sobre los componentes del panel de juego, que pueden ser hasta n. Esto provoca n iteraciones, y por tanto, un coste de la función de n o lineal
      Coste función: O(n)
- Método para obtener la corrección (obtenCorreccion())
    - Este método consiste en un bucle for que hace tantas iteraciones como componentes haya en el panel de correcciones. En caso de n componentes, hace n iteraciones, y provoca un coste de n o lineal.
      Coste función: O(n)
- Método para mostrar una pista (mostrarPista())
    - Este método consiste en un bucle for que hace tantas iteraciones como componentes haya en el panel de correcciones. En caso de n componentes, hace n iteraciones, y provoca un coste de n o lineal.
      Coste función: O(n)
- Método para pintar los resultados (pintaResultados())
    - Este método contiene 2 bucles independientes que iteran sobre las filas y sobre los componentes del panel de correcciones. Sabiendo que el número puede ser n, el coste de esta función vale n+n=2n, es decir, coste n o lineal.
      Coste función: O(n)
- Método para ejecutar un turno (ejecutarTurno())
    - Este método consiste en un bucle for que hace tantas iteraciones como componentes haya en el panel de correcciones y hace una llamada a escribeRespuesta() de esta misma clase, con un coste de n. Así que en total, tenemos un coste de n+n=2n, lo que implica un coste de n o lineal.

Coste función: O(n)

- Método para cargar una partida (cargar())
  - Este método contiene un bucle principal. Dentro de este 2 bucles independientes, y el segundo contiene otro bucle dentro. En peor caso, atravesamos 3 bucles, uno que itera sobre filasHechas, el siguiente sobre saveTurno y el último sobre componentes del panel de corrección. En caso de que todos estos valores sean n, tenemos un coste de nx(n+(nxn)) = n^3+n^2, lo que es, en esencia, un coste de n^3.
    Coste función: O(n^3)

- VistaPartidasGuardadas
  - Método para iniciar los componentes de la vista (iniciarComponentes())
    - Esta función tiene un único bucle que itera tantas veces como numeroPartidas. Suponiendo que este valor pudiera ser n, el bucle haría n iteraciones y provocaría un coste de n o lineal.
      Coste función: O(n)
  - Método para reiniciar los componentes (reiniciarComponentes())
    - Hace una llamada a iniciarComponentes() para reiniciar la vista. Tiene el mismo coste que la función anterior.
      Coste función: O(n)
  - Método para cargar una partida (cargarPartida())
    - Este método hace una llamada a cargarPartida() de controladorVistaSesion con un coste de n^2, lo que provoca un coste de n^2 o cuadrático en esta función.
      Coste función: O(n^2)
  - Método para eliminar una partida (eliminarPartida())
    - Este método hace una llamada a eliminarPartidaPerfil de VistaSesion, una llamada a reiniciarComponentes(), y otra a iniciar() del ControladorVistaSesion. Estas 3 funciones tienen coste n, por tanto, en total, tenemos un coste de n+n+n=3n, lo que provoca un coste n o lineal.
      Coste función: O(n)
- VistaRankingGlobal
  - Método para convertir un vector de Pair a una matriz de String (convertirParejaAMatrizDeString())
    - Este método recorre el vector de pair por lo que provoca un coste n.
      Coste función: O(n)
- VistaRecords

        - Método para convertir un vector de Pair a una matriz de String (convertirParejaAMatrizDeString())
            - Este método contiene 2 bucles for uno dentro del otro que iteran 4 y 3 veces. Esto nos conlleva a un coste de 4x3=12, por tanto, tenemos un coste 1 o constante.
              Coste función: O(1)
    - VistaTemas
        - Método para iniciar los componentes de la vista (iniciarComponentes())
            - La función hace 3 bucles for numeroColores iteraciones. En total provoca un coste de n+n+n= 3n , lo que provoca un coste n.
              Coste función: O(n)
        - Método para acciones de los botones (actionPerformed())
            - Este método llama a 4 veces a actualizaTema() de controladorVistaSesion, en condicionales discriminatorios. Por lo tanto, solo se ejecutará 1 vez en el peor caso. Sabiendo que el coste de esta llamada es n, el coste de esta función, también es n o lineal.
              Coste función: O(n)