# 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.

## 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

## 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 y en DriverPerfil.
- ArrayList<Partida>
    - Lista de Partidas conteniendo los atributos de esta para inicializarlas en la clase ControladorDominio. Podemos encontrarlo en ControladorDominio, 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 PartidaCodemader.
- ArrayList<Integer> listaPartidas
    - Esta lista de Integers se usa para tener y almacenar los ids de las partidas. Estas pueden estar finalizadas. Podemos encontrarlo en Perfil.
- 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.

### 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 esta 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.

- ControladorDominio
    - Método para consultar el ranking global (consultarRanking())
        - 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.

- DriverControladorDominio
  - main()
    - Dado que en el main tenemos un while, que a la vez llama a otros metodos, y la condición de la finalización de este depende del usuario, tenemos que el coste de este puede ser infinito en el peor caso. Quede claro que no es un algoritmo por sí solo, pero es posible que se haya de tener en cuenta el posible coste.
  - Método para listar el historial de partidas sin acabar (TestMostrarPartidasEmpezadas()) y método para listar el historial de partidas acabadas (TestMostrarPartidasAcabadas())
    - Estos 2 métodos hacen, en esencia, lo mismo, recorren las partidas por acabar o acabadas, dependiendo del caso, y las 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.
  - Método para consultar el ranking global (TestConsultarRankingGlobal())
    - Este método llama al método consultarRanking(), que como ya hemos visto, tiene coste n, y recorre el ranking global, es decir, las 10 (número escogido por decisión nuestra) mejores partidas, e imprime cierta información. Tenemos que la llamada a este método consultarRanking() provoca un coste de n o lineal, que es el que define el coste de nuestra función al haber un bucle que siempre hará como máximo 10 iteraciones.
  - Método para consultar todos los récords (TestConsultarRecordsMoviments())
    - El método se encarga de obtener información de todos los récords hasta el momento. Por tanto, hemos de recorrer una matriz, por cada número de colores desde 0 a n y por cada tamaño de código de (al menos) 2 a n (en caso de que no limitemos el número de colores y el tamaño de código máximo). Esto provoca, que, al tener que recorrer filas y columnas, tenemos un coste de nxn o n^2, coste cuadràtico o exponencial.
  - Método para jugar partida como CodeBreaker (jugarPartidaCB())
    - El método simula una partida del jugador como jugando como CodeBreaker. En este se hacen tantas iteraciones como turnos hayamos definido (n en caso de no limitarlos). Comprobamos pistas (solo si tenemos, pero, asumimos que si, en el peor caso),

por tanto coste n si n es el tamaño de la lista de enteros listaPista. Tenemos coste n también en el bucle que recorre según longitud de código (que si no está limitada, el coste es n). Por tanto, en este método tendríamos un coste de: nx(n+n), por tanto, nx(2n), o lo que es lo mismo en costes, n^2, coste cuadràtico o exponencial. Esta deducción está hecha a partir de la suposición de no estar limitados, pero en nuestro caso, como tenemos numeroTurnos y longitudCodigo limitados, el coste sería constante.

- Método para jugar partida como CodeMaker (jugarPartidaCM())
  - El método simula una partida del jugador como jugando como CodeMaker. Tenemos coste n en el bucle que recorre según longitud de código (que si no esta limitada, el coste es n). A continuación, se hacen tantas iteraciones como turnos hayamos definido (n en caso de no limitarlos). Dentro de este bucle tenemos otro que recorre según la longitudCodigo (que, como ya hemos dicho, puede ser n). Por tanto, en este método tendríamos un coste de: n+(nxn), por tanto, n+(n^2), o lo que es lo mismo en costes, n^2, coste cuadràtico o exponencial. Esta deducción esta hecha a partir de la suposición de no estar limitados, pero en nuestro caso, como tenemos numeroTurnos y longitudCodigo limitados, el coste seria constante.
- Método para imprimir el estado de la partida (imprimirEstadoPartida())
  - Este sencillo método imprime los elementos de las listas de lado a lado. Por tanto, hace un solo recorrido dependiendo del tamaño del código introducido (que en caso de no limitarse podría ser n). Por tanto, en este método tendríamos un coste n o lineal.

- DriverPerfil
  - main()
    - Pasa lo mismo que el main del DriverControladorDominio.
- 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.
- 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.
    - Método para obtener pistas (obtenPista())
        - Este método tiene como finalidad dar pistas al usuario cuando éste lo requiera. En el peor de los casos, el usuario tiene tantas pistas como longitud tiene el código, por tanto, si gastara estas, provocaría n iteraciones en el bucle, lo que significa coste n o lineal.
- 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.
- 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.
    - 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.
    - 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.
- 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.
    - 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.
- 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 una a recombinaCodigos(), mutaCodigo(), permutaCodigo() e invierteCodigo(), que tienen todos coste n, por tanto, deducimos que tiene un coste nx(2x(n)+n+n+n+n), en peor caso, que es lo mismo que nx(6n), es decir, cuadràtico o coste 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.
    - 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. La función solo tiene un bucle, que hace tantas iteraciones como longitudCodigo, lo que nos da coste n o lineal.
    - 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.
    - Método para la permutacion 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 de n o lineal.
    - 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.
- 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.
    - 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, entonces tenemos tantas iteraciones como numeroCodigos, por tanto, provoca un coste n o lineal.
- 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.
- Perfil
    - Método para obtener una lista con los ids de las partidas con/sin acabar (getIdsPartidasSinAcabar() y getIdsPartidasAcabadas())
        - Estos 2 métodos hacen, en esencia, lo mismo, recorren todas las partida,  y dependiendo de si estan acabadas o no, las almacena 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.
    - Método para saber el número de logros superados
        - Esta función recorre por tu Perfil, todos los logros que has obtenido y hace un conteo. Solo hace un bucle que itera sobre todos los logros existentes, y comprueba si el jugador lo ha logrado. Por tanto, si tenemos n logros, hace n iteraciones, lo que provoca un coste n o lineal.
- Ranking Global

- Método para añadir una Partida
  - 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 seria 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.