Projecte KenKen

Segona Entrega

Identificador d'equip: 1

Participants: Sergi Alonso (sergi.alonso.morillas),

Nicolas Belloch (nicolas.belloch),

Carla Edo (carla.edo.garzon),

Enric Segarra (enric.segarra)

Versió: 2.0

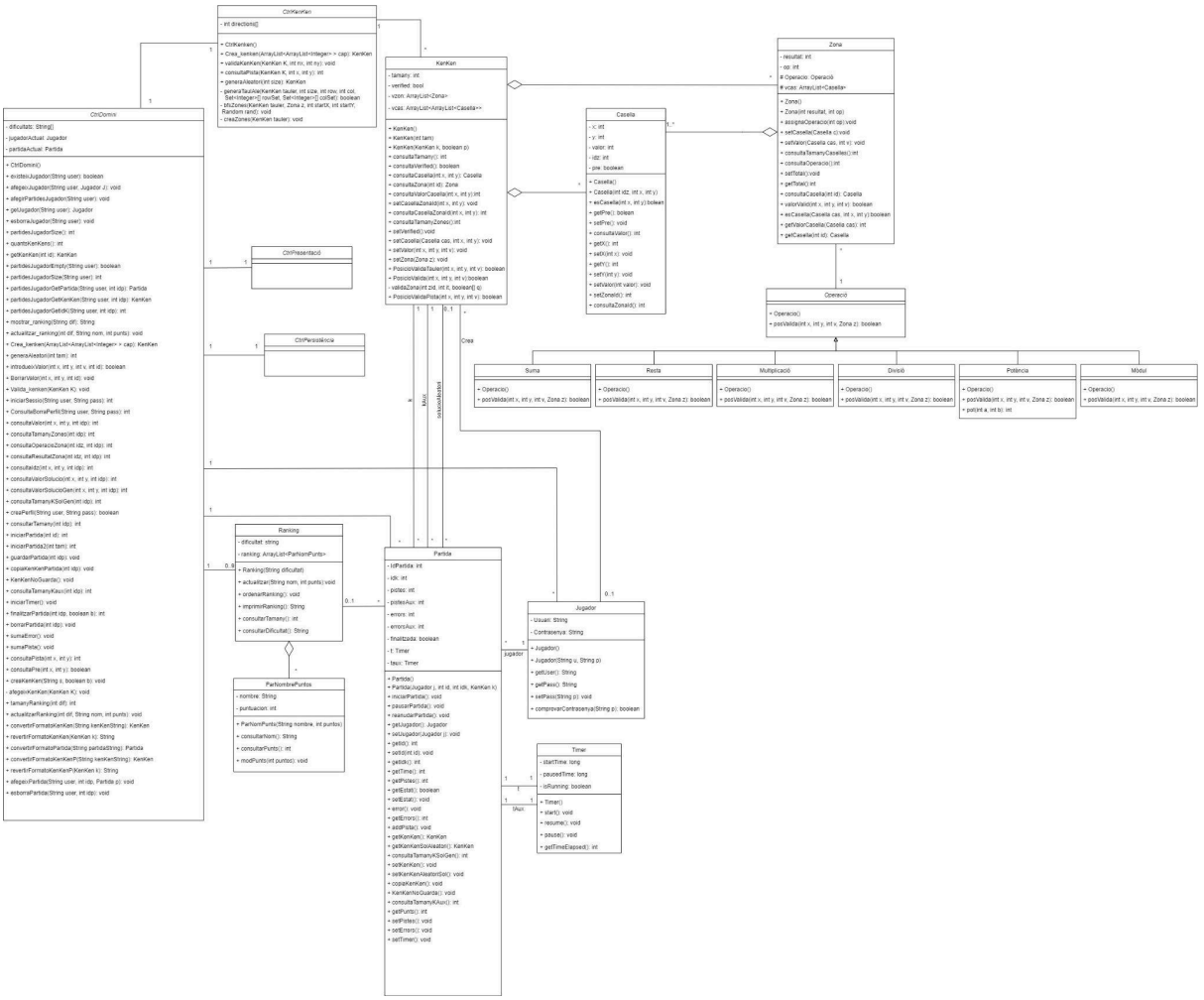
Assignatura: Projecte de Programació (PROP)

Professor Responsable: Carles Arnal

Índex

Diagrames UML

Capa de Domini



Link per a visualitzar-lo millor:

https://drive.google.com/file/d/1DSw4NOdmF3-gZYfNfoJFzHWQWJ51Y6Ab/view?usp=drive_link

Restriccions Textuals

RT1: Claus Primàries: (Ranking, dificultat), (Partida, idPartida), (Jugador,Usuari), (ParNombrePuntos,(nombre,puntuacion))

RT 2: Un jugador només pot sortir una vegada per ranking.

RT3: Una zona amb operació del tipus resta, divisió, potència o mòdul només pot tindre dues caselles.

RT4: Una zona amb operació nul·la (op == 0) només pot tindre una casella.

RT5: No pot haver-hi més d'una partida amb el mateix jugador i instància de KenKen.

RT6: Un KenKen no pot tindre més zones que caselles.

RT7: Les caselles que pertanyen a zona han de pertànyer al mateix KenKen al qual pertany la zona.

RT8: El resultat d'una zona sempre serà vàlid per a la seva operació (Exemple: suma de dues caselles amb resultat 1 no pot passar).

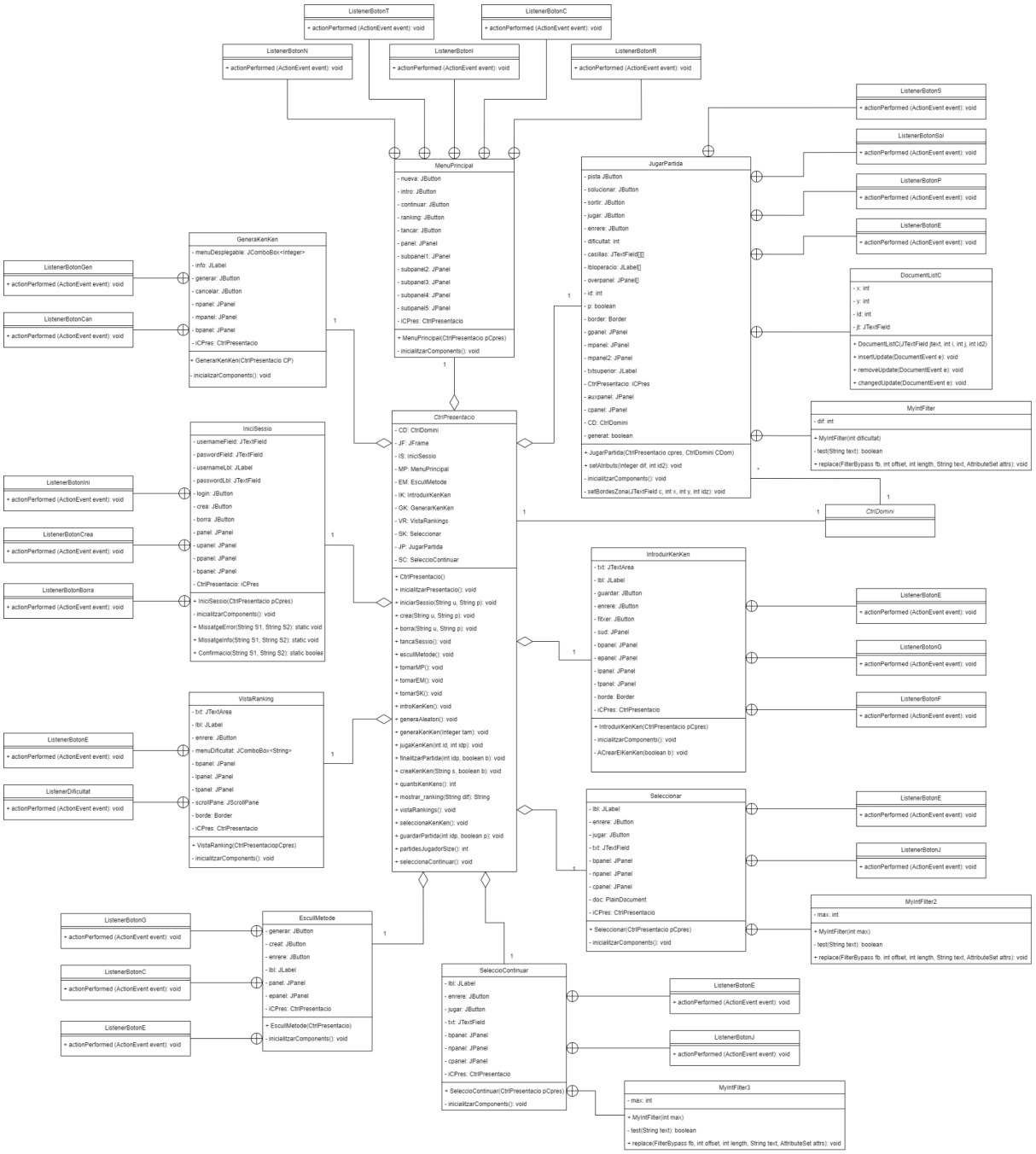
RT9: No poden haver-hi dues o més caselles amb la mateixa posició (x,y) per al mateix KenKen ni per a la mateixa Zona.

RT10: Dues Zones no poden tenir la mateixa casella d'un KenKen.

RT11: Una partida només pertany a un Ranking si està finalitzada (finalitzada = true).

RT12: Una partida només pot pertànyer al Ranking de dificultat igual a la del seu KenKen.

Capa de Presentació



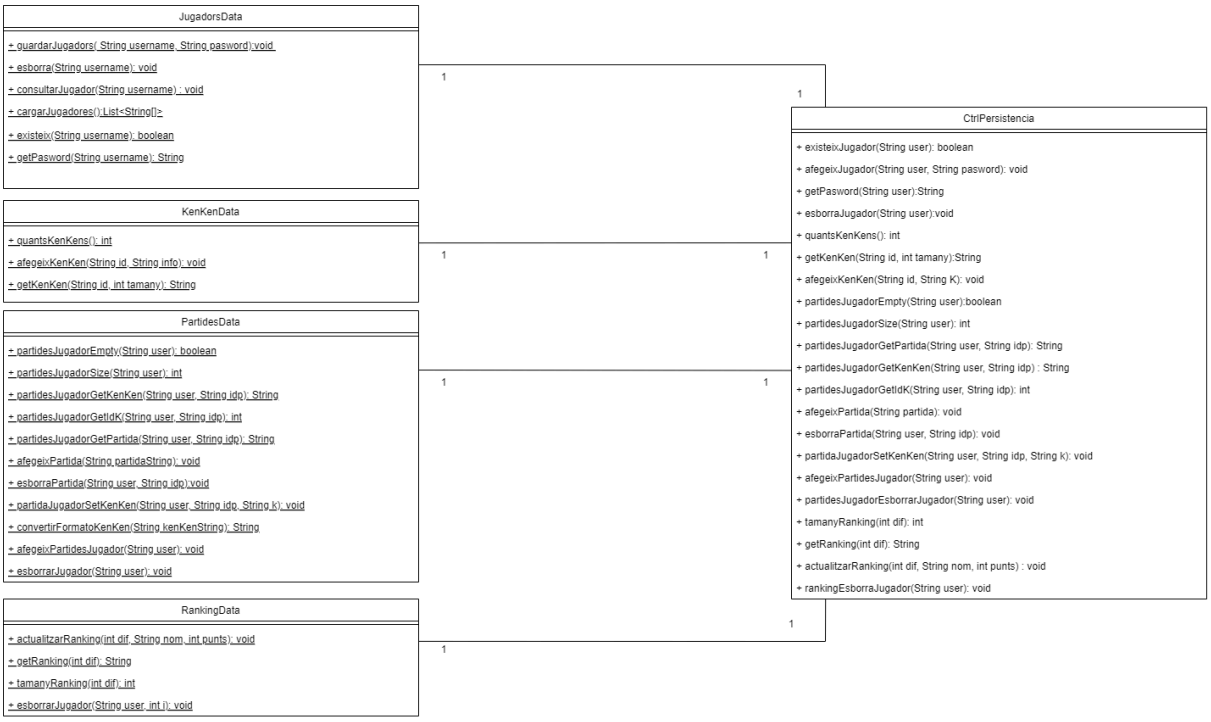
Link per a visualitzar-lo millor:

https://drive.google.com/file/d/1vc3Ge-PdP-qfdCg6knNlVqALdI0qXBOq/view?usp=drive_link

Restriccions Textuals

RT1: El controlador de domini (CD) de CtrlPresentacio i de JugarPartida són el mateix objecte.

Capa de Persistència



Link per a visualitzar-lo millor:

https://drive.google.com/file/d/1wHgBgMExHL0CT9kfSTDJVaNaCIMU4iy/view?usp=drive_link

Restriccions Textuals

Aquest diagrama no té restriccions textuals.

Descripció d'atributs i mètodes

Capa de Domini

Casella

Atributs:

- **int x:** indica la fila de la Casella al KenKen
- **int y:** indica la columna de la Casella al KenKen
- **int valor:** indica el número que conté la Casella
- **int idz:** indica l'identificador de la Zona a la qual pertany la Casella
- **boolean pre:** indica si la Casella del KenKen és un valor fix inicialitzat

Mètodes:

- **Casella():** Creadora d'una casella buida, x, y, idz s'inicialitzen a -1 per detectar que la casella no és vàlida. Valor és 0 per defecte per qualsevol casella buida, i pre és fals per defecte. El seu cost és Θ(1).
- **Casella(int idz, int x, int y):** Creadora de casella vàlida. S'assigna un idz per identificar la Zona de la casella i s'assignen els x i y passats per paràmetre. El valor de la casella és 0 per defecte i pre és fals també. El seu cost és Θ(1).
- **boolean esCasella(int x, int y):** Retorna true si la x i y de la casella coincideixen amb els passats per paràmetre. El seu cost és Θ(1).

Zona

Atributs:

- **int resultat:** Representa el resultat de la combinació de valors de la Zona un cop passats per l'operació d'aquesta.
- **int op:** Número que identifica quina operació té la casella, on 0 no té operació, 1 és Suma, 2 és resta, 3 és multiplicació, 4 és divisió, 5 és potència i finalment 6 és mòdul.
- **ArrayList<Casella> vcas:** Arraylist que conté els objectes de Casella que conté la Zona
- **Operacio operacio:** Objecte de classe Operació assignat a la Zona. Aquest s'ocupa d'operar amb els valors de les caselles de vcas.

Mètodes:

- **Zona():** Creadora de Zona buida, s'inicialitza l'array de caselles i es posa op a -1 per determinar que la casella encara no té operació. (El valor 0 per op significa que la casella no té operació, és a dir qualsevol número és vàlid per emplenar-la). El seu cost és Θ(1).
- **Zona(int resultat, int op):** Creadora de Zona que inicialitza el resultat i l'operació d'aquesta. Si op > 0 s'assigna un objecte Operació, contràriament, l'operació serà 0. El seu cost és Θ(1).
- **void assignaOperacio(int op):** Funció que s'encarrega de crear i assignar un objecte Operació a la Zona. El seu cost és Θ(1).
- **boolean valorValid(int x, int y, int v):** Funció que s'encarrega de comprovar si el valor v és vàlid per a la casella amb posició (x,y). Depèn de cada operació, així que el cost és variable.
- **boolean esCasella(Casella cas, int x, int y):** Crida a l'operació esCasella (int x, int y) de l'objecte Casella passat per paràmetre, per comprovar que les coordenades coincideixen. El seu cost és Θ(1).

Operació

Mètodes:

- **abstract void Operacio():** Creadora de la classe. El seu cost és Θ(1).
- **abstract boolean posValida(int x, int y, int v, Zona z):** Operació abstracta a ser implementada.

Suma

Mètodes:

- **void Suma():** Creadora de classe. El seu cost és Θ(1).
- **boolean posValida(int x, int y, int v, Zona z):** Comprova que la suma dels valors del vector de caselles de la Zona z sigui igual al resultat de z. Té cost Θ(n), sent n el nombre de caselles de la zona, ja que hi ha un bucle que accedeix a totes les caselles de la zona.

Resta

Mètodes:

- **void Resta():** Creadora de classe. El seu cost és Θ(1).
- **boolean posValida(int x, int y, int v, Zona z):** Comprova que la resta dels valors del vector de caselles de la Zona z sigui igual al resultat de z. Té cost Θ(1), ja que totes les operacions que realitza són constants.

Multiplicació

Mètodes:

- **void Multiplicació():** Creadora de classe. El seu cost és Θ(1).
- **boolean posValida(int x, int y, int v, Zona z):** Comprova que la multiplicació dels valors del vector de caselles de la Zona z sigui igual al resultat de z. Té cost Θ(n), sent n el nombre de caselles de la zona, ja que hi ha un bucle que accedeix a totes les caselles de la zona.

Divisió

Mètodes:

- **void Divisió():** Creadora de classe. El seu cost és Θ(1).
- **boolean posValida(int x, int y, int v, Zona z):** Comprova que la divisió dels valors del vector de caselles de la Zona z sigui igual al resultat de z. Té cost Θ(1), ja que totes les operacions que realitza són constants.

Mòdul

Mètodes:

- **void Mòdul():** Creadora de classe. El seu cost és Θ(1).
- **boolean posValida(int x, int y, int v, Zona z):** Comprova que el mòdul dels valors del vector de caselles de la Zona z sigui igual al resultat de z. Té cost Θ(1), ja que totes les operacions que realitza són constants.

Potencia

Mètodes:

- **void Potencia():** Creadora de classe. El seu cost és Θ(1).
- **boolean posValida(int x, int y, int v, Zona z):** Comprova que la potència dels valors del vector de caselles de la Zona z sigui igual al resultat de z.
- **int pot(int a, int b):** Calcula la potència dels dos valors. Té cost Θ(1), ja que totes les operacions que realitza són constants.

KenKen

Atributs:

- **int tamany:** Indica la mida del KenKen.
- **ArrayList<ArrayList<Casella>> vcas:** ArrayList d'ArrayLists (matriu de caselles) que guarda la informació de les caselles del KenKen.
- **ArrayList<Zona> vzon:** ArrayList que guarda la informació de les Zones del KenKen.
- **boolean verified:** Indica si el KenKen ha estat verificat.

Mètodes:

- **KenKen():** Crea un KenKen de tamany -1, inicialitza les arraylists buides i posa verified a false. El seu cost és Θ(1).
- **KenKen(int tam):** Crea un KenKen de la mida indicada pel paràmetre, inicialitza també les arraylists amb caselles i posa verified a false. Té cost Θ(tam^2), ja que crea el tauler de caselles amb dos bucles que recorren una matriu de mida tam*tam. Si considerem el context de l'aplicació, la mida màxima seria 9 i el cost màxim seria Θ(81), es podria considerar constant.
- **KenKen(KenKen k, boolean p):** Serveix per crear una còpia del KenKen passat per paràmetre. El booleà ens indica si hem de copiar tots els valors o només els valors predefinits del KenKen. Té cost Θ(tam^2), mateix raonament que la creadora anterior.
- **boolean posicioValidaTauler(int x, int y, int v):** Comprova si el valor de la posició indicada no es repeteix en files ni columnes. Té cost *O*(tam), ja que hi ha un bucle que pot recórrer tot tam, però pot retornar abans. Si considerem el context de l'aplicació seria cost constant (tam és com a màxim 9).

- **boolean posicioValida(int x, int y, int v):** Comprova si el valor de la posició indicada és vàlid pel tauler i per les zones. Considerarem que PosicioValidaTauler té cost constant pel context general, llavors el cost variarà segons la subclasse d'operació que faci ValorValid, sent el cost màxim *O*(n), on n és el nombre de caselles de la zona de la casella amb posició (x,y).
- **void ValidaZona(int zid, int it, boolean[] q)**: Comprova si la zona té una possible solució. El cost és de *O*(n^(2+m)), com s'explica a la descripció de l'algorisme corresponent.
- **boolean PosicioValidaPista(int x, int y, int v)**: Comprova si el valor v per a la casella amb posició (x,y) és vàlid actualment per al kenken i si amb aquest valor la zona tindrà una solució a futur. El cost és de *O*(n^(2+m)), per la crida a ValidaZona.

Jugador

Atributs:

- **String user:** Identifica a cada objecte d'usuari.
- **String password:** Comprova la identitat de l'usuari.

Mètodes:

- **Jugador():** Creadora buida de Jugador. El seu cost és Θ(1).
- **Jugador(String u, String p):** Creadora per paràmetres de jugador, l'inicialitza amb user = u i password = p. El seu cost és Θ(1). .
- **boolean comprovaContrasenya(String p):** Valida si la contrasenya entrada és la de l'usuari. El seu cost és Θ(1). .

Partida

Atributs:

- **KenKen k:** Objecte de KenKen que pertany a la Partida.
- **KenKen kAux:** Objecte de KenKen auxiliar on, en una partida ja guardada, es còpia el kenken k a kAux al reprendre-la, ja que si finalment no es guarda es còpia kAux a k.
- **KenKen solucioAleatori:** Objecte de KenKen que guarda la solució del KenKen k. Només s'utilitza per kenkens generats aleatòriament.
- **Jugador jugador:** Objecte de Jugador que juga la Partida.
- **int idPartida:** Identificador que distingeix la Partida.
- **int idk:** Id del KenKen que pertany a la Partida.
- **int pistes:** Nombre de pistes demanades fins al moment.
- **int pistesaux:** Nombre de pistes auxiliars, que es copien al reprendre una partida. Mateix funcionament que kAux.
- **int errors:** Nombre d'errors comesos.
- **int errorsaux:** Nombre de pistes auxiliars, que es copien al reprendre una partida. Mateix funcionament que kAux.
- **boolean finalitzada:** Indicador de si la partida està completada.
- **Timer t:** Objecte de Timer per comptabilitzar el temps de la partida.
- **Timer taux:** Objecte de Timer auxiliar, que es copia al reprendre una partida. Mateix funcionament que kAux.

Mètodes:

- **Partida():** Creadora de Partida sense variables inicialitzades. El seu cost és Θ(1).
- **Partida(Jugador j, int id, int idk, KenKen k):** Creadora de Partida on s'assignen totes les seves variables a partir de les passades per paràmetre. El seu cost és Θ(1).
- **void iniciarPartida():** Inicia el Timer de la Partida. El seu cost és Θ(1).
- **void pausarPartida():** Pausa el Timer de la Partida. El seu cost és Θ(1).
- **void reanudarPartida():** Resumeix el Timer de la Partida. El seu cost és Θ(1).
- **void error():** Suma un error a la variable error. El seu cost és Θ(1).
- **void addPista():** Suma una pista a la variable pista. El seu cost és Θ(1).

- **int getPunts():** Retorna el càlcul dels punts de la partida. El seu cost és Θ(1).
- **void copiaKenKen():** Copia el KenKen k a kAux, els errors a errorsaux, les pistes a pistesaux i el Timer t al Timer taux. Aquesta funció es crida cada cop que es reprèn una partida i té cost Θ(1) (realment té cost Θ(tam^2), on tam és la mida dels kenkens, però ho considerem constant).
- **void KenKenNoGuarda():** Copia el KenKen kAux a k, els errors a errorsaux, les pistes a pistesaux i el Timer t al Timer taux. Aquesta funció només es crida al no guardar una partida que ja s'havia guardat un mínim d'un cop. Té cost Θ(1) (mateix raonament que a copiaKenKen()).

Ranking

Atributs:

- **String dificultat:** Indica el tamany de KenKen al qual es refereix el Ranking.
- **ArrayList<ParNomPunts> ranking:** Guarda els noms dels usuaris i els seus punts.

Mètodes:

- **Ranking(String dificultat):** Creadora de Ranking que inicialitza els seus parametres. El seu cost és Θ(1).
- **void actualitzar(String nom, int punts):** Entra el nom d'un usuari i els seus punts. L'afegeix al Ranking si no hi era, actualitza els seus punts si hi era, i reordena el Ranking. Té cost Θ(n*log(n)), ja que és el cost de la crida a collections.sort() dintre de la crida del mètode ordenarRanking(). Tot i haver-hi un bucle, el cost del bucle (*O*(n)) és inferior al del sort.
- **void ordernarRanking():** Sorter que ordena el Ranking per punts. Té cost Θ(n*log(n)), per la crida a collections.sort().
- **void imprimirRanking():** Fa un print dels jugadors i punts del Ranking. Té cost Θ(n), sent n els elements del ranking, ja que hi ha un bucle que visita tots els elements.

Timer

Atributs:

- **long startTime:** Guarda el valor del temps del sistema quan s'inicia el Timer.
- **long pausedTime:** Guarda el valor del temps del sistema quan s'atura el Timer.
- **boolean running:** Indica si el Timer es troba en funcionament.

Mètodes:

- **Timer():** Creadora del Timer. S'inicialitza running com a fals. El seu cost és Θ(1).
- **void start():** Inicialitza el Timer si running == false. El seu cost és Θ(1) (la funció del sistema System.currentTimeMillis() la considerarem constant).
- **void pause():** Pausa el Timer si running == true. El seu cost és Θ(1).
- **void resume():** Continua el Timer a partir del moment pausat si running == false. El seu cost és Θ(1).
- **int getTimeElapsed():** Retorna el temps transcurrit des de l'inici del Timer. El seu cost és Θ(1).

CtrlKenKen

Atributs:

- **int[][] directions:** Array de parells de valors per indicar les direccions a fer servir per als algorismes de la classe.

Mètodes:

- **CtrlKenKen():** Creadora del Controlador. El seu cost és Θ(1).
- **Crea_kenken(ArrayList<ArrayList<Integer> > cap):** Aquesta funció s'ocupa de crear un KenKen a partir de arraylist d'integers al que se li passa com a paràmetre (sent aquest l'input per fitxer o manual de l'usuari). El cost de la funció és Θ(n^2), com s'explica a la descripció de l'algorisme corresponent.
- **validaKenKen(KenKen K, int nx, int ny):** Funció de backtracking que comprova si el KenKen passat per paràmetre es pot resoldre. Té un cost de *O*(n^5), el qual s'explica a la descripció de l'algorisme corresponent.

- **consultaPista(int x, int y, int idp):** Retorna un possible valor per a la casella amb posició (x,y), tal que sigui vàlid actualment i a futur per a la zona. En cas de no haver-hi retorna un -1. El seu cost és de O(n^(3+m)), tal com s'explica a la descripció de l'algorisme corresponent.
- **generaAleatori(int size):** Aquesta funció genera un KenKen de forma aleatòria a partir del tamany indicat per paràmetre. Pel seu funcionament intern de backtracking explicat a la part d'algorismes del document, té un cost de Θ(n^(n^2)).
- **generaTaulAle(KenKen tauler, int size, int row, int col, Set<Integer>[] rowSet, Set<Integer>[] colSet):** Funció d'ús per a generaAleatori(int size). Aquesta funció s'encarrega d'emplenar el tauler amb valors aleatoris de forma correcta per la normativa d'un KenKen. Té un algorisme de backtracking que fa que el cost de la funció sigui de Θ(n^(n^2)).
- **bfsZones(KenKen tauler, Zona z, int startX, int startY, Random rand):** Algorisme encarregat de generar les zones del KenKen. Com indica el seu nom fa servir un mètode BFS per assignar zones recorrent el tauler de forma iterativa. Al fer servir aquesta metodologia el cost de la funció és deΘ(n^2) pel tauler de tamany n.
- **creaZones(KenKen tauler):** Aquesta funció prepara una sèrie d'arrays per tal que bfsZones pugui funcionar correctament. Al cridar a la funció de bfsZones multiples vegades, té un cost de Θ(n^(n^2)).

CtrlDomini

Atributs:

- **CtrlPersistencia CP:** Controlador de Persistència.
- **CtrlKenKen CK:** Controlador dels KenKens.
- **String[] dificultats:** Array dels diferents tamanys disponibles (3-9).
- **Jugador jugadorActual:** Objecte de Jugador per referenciar al jugador que té iniciada la sessió.
- **Partida partidaActual:** Objecte de Partida per referenciar a la partida que s'està duent a terme.

Mètodes:

- **CtrlDomini():** Creadora del Controlador. Crea les instàncies de CtrlKenKen i CtrlPersistència. El seu cost és Θ(1).
- **boolean existeixJugador(String user):** Comprova l'existència d'un jugador amb username user.
- **void afegeixJugador(string user, Jugador J):** Afegeix el jugador J a persistència.
- **void afegeixPartidesJugador(String user):** Afegeix l'usuari de nom user a la persistència de Partides de Jugador.
- **Jugador getJugador(String user):** Retorna el jugador amb nom user des de la capa de persistència.
- **void esborraJugador(String user):** Esborra de persistència el jugador de nom user.
- **int partidesJugadorSize():** Retorna el nombre de partides que té el jugador actual.
- **int quantsKenKens():** Retorna el nombre de KenKens que hi ha a guardats a persistència.
- **KenKen getKenKen(int id):** Retorna un objecte KenKen a partir d'un KenKen guardat a persistència identificat per id.
- **boolean partidesJugadorEmpty(String user):** Retorna si el jugador d'username user té cap partida guardada.
- **int partidesJugadorSize(String user):** Retorna el nombre de partides guardades que té el jugador d'username user.
- **Partida partidesJugadorGetPartida(String user, int idp):** Retorna un objecte partida de persistència a partir de la partida guardada identificada per idp del jugador identificat per user.
- **KenKen partidesJugadorGetKenKen(String user, int idp):** Retorna un objecte de KenKen de persistència de la partida guardada identificada per idp, del jugador identificat per user.
- **int partidesJugadorGetIdK(String user, int idp):** Retorna el id del KenKen de la partida guardada identificada per idp, del jugador identificat per user.
- **String mostrar_ranking(String dif):** Retorna la informació del Ranking actual de la dificultat indicada per dif.

- **void actualitzar_ranking(int dif, String nom, int punts):** Actualitza el Ranking de dificultat dif amb el user del jugador nom, i els punts passats per paràmetre.
- **KenKen Crea_kenken(ArrayList<ArrayList<Integer> > cap):** Retorna un objecte de KenKen creat a partir de l'ArrayList d'enters, passat com a paràmetre.
- **int generaAleatori(int tam):** Genera un KenKen de forma aleatoria amb el tamany indicat per paràmetre. Retorna el tamany actual de la llista de KenKens.
- **boolean introdueixValor(int x, int y, int v, int id):** Retorna si la posició x y, amb valor v, és vàlida per al KenKen de la partida identificada per id, de les partides del jugador actual.
- **void BorrarValor(int x, int y, int id):** Esborra el valor de la posició indicada per x, y, del KenKen identificat per id, de les partides del jugador actual.
- **void Valida_kenken(KenKen K):** Valida l'objecte de KenKen passat com a paràmetre. Si no és vàlid saltarà una excepció.
- **int iniciarSessio(String user, String pass):** Inicia sessió pels paràmetres entrats. Si les credencials són vàlides, el jugador actual passa a ser el creat pels paràmetres (que serà una còpia d'un existent).
- **int ConsultaBorraPerfil(String user, String pass)**: Si el perfil de jugador indicat pels paràmetres existeix, s'esborra.
- **int consultaValor(int x, int y, int idp):** Retorna el valor de la casella indicada per x i y, del KenKen de la partida identificada per idp de les partides del jugador actual.
- **int consultaTamanyZones(int idp):** Retorna quantes zones té el KenKen de la partida indicada per idp del jugador actual.
- **int consultaOperacioZona(int idz, int idp):** Retorna l'identificador de l'operació de la zona identificada per idz, del KenKen de la partida identificada per idp, del jugador actual.
- **int consultaResultatZona(int idz, int idp):** Retorna el resultat de la zona identificada per idz, del KenKen de la partida identificada per idp, del jugador actual.
- **int consultaIdz(int x, int y, int idp):** Retorna l'identificador de la zona de la casella x, y, del KenKen de la partida identificada per idp, del jugador actual.

- **int consultaValorSolucio(int x, int y, int idp):** Retorna el valor de la casella x, y, de la solució del KenKen de la partida identificada per idp, del jugador actual.
- **public int consultaValorSolucioGen(int x, int y, int idp):** Retorna el valor de la casella x, y, de la solució del KenKen generat aleatòriament de la partida actual.
- **int consultaTamanyKSolGen():** Retorna el tamany de la solució del KenKen generat aleatòriament de la partida actual.
- **boolean creaPerfil(String user, String pass)**: Si no existeix un usuari amb les credencials indicades per paràmetre, es crea i guarda.
- **int consultarTamany(int idp):** Retorna el tamany del KenKen de la partida identificada per idp, del jugador actual.
- **int iniciarPartida(int id):** Inicia una partida pel jugador actual, amb el KenKen identificat per id. Retorna el idp.
- **int iniciarPartida2(int tam):** Inicia una partida pel jugador actual, amb el KenKen generat aleatoriament de tamany tam. Retorna el idp.
- **void guardarPartida(int idp):** Pausa la partida identificada per idp passat per paràmetre i la guarda en el seu estat actual en la capa de persistència com a partida del jugador actual.
- **void copiaKenKenPartida(int idp):** Funció que reprèn la partida (guardant les pistes, els errors, el timer i el kenken als atributs auxiliar de la partida).
- **void KenKenNoGuarda(int idp):** Es reinicia la partida actual tal com estava abans de reprendre-la.
- **int consultaTamanyKaux(int idp):** Retorna el tamany del KenKen auxiliar de la partida.
- **void iniciarTimer():** Inicia el timer de la partida actual.
- **int finalitzarPartida(boolean b):** Estableix com a finalitzada la partida actual. Si b és true s'actualitza el ranking de la mida del KenKen de la partida.
- **void borrarPartida(int idp):** Esborra la partida identificada per idp del jugador actual.
- **void sumaError():** Afegeix un error al contador de la partida actual del jugador actual.
- **void sumaPista():** Afegeix una pista al contador de la partida actual del jugador actual.

- **consultaPista(int x, int y):** Retorna un possible valor per a la casella amb posició (x,y), tal que sigui vàlid actualment i a futur per a la zona. En cas de no haver-hi retorna un -1. El seu cost és de O(n^(3+m)), per la crida a consultaPista del controlador de kenkens.
- **consultaPre(int x, int y):** Consulta si la casella de coordenades x, y, del KenKen de la partida actual és predeterminada, és a dir té valors inicialitzats.
- **void creaKenKen(String s, boolean b):** Crea un objecte KenKen a partir del string s passat per paràmetre. El paràmetre b indica si el KenKen es vol guardar en versió buida.
- **void afegeixKenKen(KenKen K):** Guarda el KenKen passat per paràmetre a persistència.
- **int tamanyRanking(int dif):** Retorna el tamany actual del Ranking de dificultat dif.
- **void actualitzarRanking(int dif, String nom, int punts):** Actualitza el Ranking de dificultat dif, amb un usuari i punts passats per paràmetre.
- **KenKen convertirFormatoKenKen(String kenKenString):** Retorna el KenKen obtingut a partir de convertir el string passat per paràmetre.
- **String revertirFormatoKenKen(KenKen k):** Retorna un string obtingut a partir de convertir el KenKen passat per paràmetre.
- **Partida convertirFormatoPartida(String partidaString):** Retorna la partida obtinguda a partir de convertir el string passat per paràmetre.
- **KenKen convertirFormatoKenKenP(String kenKenString):** Retorna el KenKen obtingut a partir de convertir el string passat per paràmetre (amb un format diferent a revertirFormatoKenKen(KenKen k)).
- **String revertirFormatoKenKenP(KenKen k):** Retorna un string obtingut a partir de convertir el KenKen passat per paràmetre (amb un format diferent a convertirFormatoKenKen(String kenKenString)).
- **void afegeixPartida(String user, int idp, Partida p):** Guarda a persistència la partida passada per paràmetre per l'usuari user amb identificador de partida idp.
- **void esborraPartida(String user, int idp):** Esborra la partida identificada per idp de l'usuari user de persistència.

Capa de Presentació

Per a la capa de presentació hem utilitzat GUI (*graphic user interfaces*), creades amb java *AWT* i *Swing*. Pel que fa a l'organització, hem utilitzat un únic controlador que conté el marc principal (*JFrame*) i que es relaciona amb totes les vistes, les quals actuen com a panells (hereten de *JPanel*) que s'introdueixen al *JFrame*. Aquestes vistes contenen els components i panells necessaris, que junt amb diferents *Layouts* ens permeten distribuir els components a la pantalla correctament.

CtrlPresentacio

L'únic controlador de presentació de l'aplicació consta amb el marc principal que conté tota la interfície gràfica i amb un objecte de cada vista, de les quals col·loca la corresponent en cada context al marc. A més disposa de tots els mètodes necessaris per a accedir a la capa de domini (i des d'allà a la capa de persistència, si és necessari).

IniciSessio

És la primera vista de l'aplicació, on l'usuari es pot autenticar.
Utilitza dos camps de text (*JTextField*) per introduir l'usuari i el nom i dues etiquetes (totes les etiquetes que hem utilitzat són *JLabel*) per informar a on introduir-los respectivament.
Conté tres botons (tots els botons que hem utilitzat son *JButton*) associats a listeners que permeten iniciar sessió i passar a la següent vista i crear o esborrar un perfil en base a les credencials introduïdes als camps de text descrits anteriorment.

MenuPrincipal

Representa la vista que fa la funció de menú principal, des d'aquí podem accedir a les principals funcionalitats de l'aplicació.
Cinc botons associats a listeners dirigeixen a l'usuari a altres vistes on podrà iniciar una nova partida (classe EscullMetode), introduir, validar i guardar un KenKen (classe IntroduirKenKen), reprendre una partida guardada anteriorment (classe SeleccioContinuar), consultar els rankings (classe VistaRankings) i tanca la sessió actual (retorna a IniciSessio).

VistaRanking

Des d'aquesta vista podrem consultar el respectiu ranking associat a la dificultat dels KenKens.

La vista consisteix en una llista desplegable (les llistes desplegables que utilitzem són *JComboBox*) que permet escollir la dificultat associada al ranking a consultar (de 3x3 a 9x9) i la mostra via un camp de text gran (*JTextArea*) amb l'ajuda d'un *listener* associat a la llista desplegable. El camp de text és només per a sortida de text (no permet l'entrada d'aquest per part de l'usuari).

Una etiqueta informa de la funció de la llista desplegable i per acabar també conté un botó que retorna al menú principal.

EscullMetode

Aquesta vista és prèvia a començar a jugar una partida. Des d'aquí escollirem si volem jugar un KenKen generat aleatòriament o, per altra banda, si volem jugar un que hagi guardat el propi o un altre usuari anteriorment.

Per escollir entre aquestes dues opcions disposem de dos botons associats a listeners que permeten dirigir a l'usuari a les següents vistes. En cas d'escollir generar un kenken aleatori es passaria a la vista GeneraKenKen i en cas de decidir jugar-ne un ja creat es passaria a la vista representada per la classe Seleccionar.

També disposem d'una etiqueta que ens informa del que estem escollint i un botó que ens permet retornar a la vista anterior (MenuPrincipal).

GeneraKenKen

Després de seleccionar generar un kenken aleatori a EscullMetode, aquesta vista s'encarrega de proporcionar a l'usuari una llista desplegable que permet escollir la dificultat desitjada per al KenKen a generar.

A més de la llista desplegable, la vista disposa de dos botons, un per a començar a jugar (i passar a la vista JugarPartida, un cop s'ha seleccionat la dificultat) i un per a cancel·lar la generació i retornar a EscullMetode.

IntroduirKenKen

És la classe des de la qual podem introduir KenKens perquè els validi i guardi.
Podem introduir el KenKen de dues formes, introduint-lo nosaltres manualment en un camp de text gran (*JTextArea*) o mitjançant un botó que ens permetrà importar-lo des d'un fitxer de text i el copiarà al camp de text. Un cop introduït podem validar-lo a través d'un botó i se'ns preguntarà si volem guardar-lo.
Addicionalment, tenim un botó que ens permet tornar a la vista anterior (MenuPrincipal) i una etiqueta que ens indica la funcionalitat del camp de text.

JugarPartida

Aquesta vista és la que ens permetrà jugar la partida, creant el kenken amb dificultat*dificultat camps de text (*JTextField*), els quals disposen d'un *listener* propi, per a poder interactuar amb el kenken, i d'un filter propi per a poder limitar els valors a introduir (entre 0 i dificultat). Per a jugar la partida, aquesta vista és l'única amb accés directe al controlador de domini, tal com estipula l'exemple d'aquesta excepció a les transparències.
A més disposa d'una previsualització del kenken (no interactiu) amb una etiqueta que pregunta si es vol jugar aquest kenken i dos botons, un per a retornar a la vista anterior i un per a començar a jugar.
Al començar a jugar, desapareixen l'etiqueta i els dos botons, s'habilita la interacció amb el kenken i apareixen tres botons nous, un per a demanar la solució del kenken, un altre per demanar una pista i un per a sortir. En sortir es pregunta al jugador si es vol guardar la partida, en cas negatiu se li demana una confirmació.

Seleccionar

Aquesta classe en permet escollir el KenKen que jugarem a continuació. Per fer-ho disposem d'un camp de text (*JTextField*) en el que tenim hem d'introduir l'identificador numèric del KenKen guardat que volem jugar. Una etiqueta ens informa del que hem de fer. Finalment, tenim dos botons, un per començar el joc amb el KenKen escollit i un altre per tornar a la vista anterior (EscullMetode).

SeleccioContinuar

La classe que accedim per continuar una partida. A aquesta classe escollirem la partida que volem reprendre introduint el seu identificador a un camp de text (*JTextField*). Existeixen dos botons que ens permetran tornar a la vista anterior (MenuPrincipal) i reprendre la partida identificada pel valor introduït respectivament. També conté una etiqueta que informa de la utilitat del camp de text.

Capa de Persistència

CtrlPersistencia

La classe CtrlPersistencia és la que comunica la capa de Domini amb la de Persistència. S'encarrega de rebre i passar la informació necessària de la capa de Domini a la de Persistència a través d'estructures de dades simples com Strings i int.

JugadorsData

La classe JugadorsData s'encarrega de guardar, modificar i esborrar la informació dels jugadors que es troba en l'arxiu jugadors.json. També s'encarrega d'adaptar aquesta informació a un format d'estructura de dades simple per tal que sigui fàcil de transferir de Persistència a Domini.

KenKenData

La classe KenKenData s'ocupa de guardar a la carpeta kenkens els kenkens, que es generin o s'introdueixin amb el valor de la solució proporcionada un cop es valida el kenken, en format arxiu .txt cadascun amb nom igual "k"+idk sent idk el número que identifica el kenken.

PartidesData

La classe PartidesData s'encarrega de guardar a l'arxiu partides.json el nom dels usuaris amb els identificadors de les seves partides i la informació d'aquestes. Per fer-ho transforma la informació del format de l'arxiu partides.json a Strings o int, segons sigui necessari, i viceversa.

RankingData

La classe RankingData és l'encarregada d'afegir, modificar i esborrar els jugadors i els seus punts al fitxer rankingX.json sent X la mida del kenken que ha jugat el jugador.

Estructures de dades i Algorismes

Estructures de dades

Ranking

Per representar el Ranking hem utilitzat un *ArrayList* de ParNomPunts (classe que simula un *pair* de C++), ja que és l'estructura més senzilla per les nostres necessitats perquè creiem que el cost que pot representar no és significatiu. El cost màxim que ens podria penalitzar seria de *O*(n) en el moment en el qual actualitzem el rànquing i el jugador encara no apareix en ell.

KenKen

Les dels KenKens consisteixen en un *ArrayList* de zones i un *ArrayList* d'*ArrayLists* de caselles intentant representar una matriu d'aquestes. La justificació de l'ús d'aquestes estructures varia en cada cas, per a les zones s'ha optat per aquesta opció, ja que el nombre que emmagatzemarà sempre serà reduït, en canvi, la representació d'una matriu per a les caselles ens facilita la implementació exponencialment, ja que ens possibilita l'accés a la casella que desitgem només amb la seva posició (x,y).

Zona

Cada zona utilitza un ArrayList on emmagatzema les caselles que pertanyen a aquesta, les quals pertanyen al KenKen al que pertany la zona (referència compartida). El nombre de caselles sempre és petit i, per tant, no ens preocupa que el cost pugui escalar gaire.

Persistència

A la persistència hem guardat els jugadors, les partides i els rankings en arxius json. Hem optat per aquest tipus d'arxius, ja que permeten emmagatzemar dades de manera senzilla i llegible en una estructura lleugera. A més permeten generar objectes dins del fitxer el que fa que sigui més fàcil accedir o modificar un objecte concret com pot ser un usuari, una partida, un kenken dins d'una partida, o un usuari dins d'un ranking. Concretament, hi ha els següents arxius:

- jugadors.json: on es guarden per parelles els jugadors amb les seves contrasenyes.
- partides.json: on es guarden els usuaris pel seu nom com a objecte a partir dels quals podem accedir a les partides que es guarden dins de cada usuari com a objecte cada una. L'estructura JSON és ideal per a aquest tipus de dades jeràrquiques, on cada usuari pot tenir múltiples partides, i cada partida conté diversos atributs (KenKen associat, pistes, errors, estat, etc.)
- ranking3.json, ranking4.json, rankin5.json, ranking6.json, ranking7.json, ranking8.json, ranking9.json: cadascun representa el ranking de la mida de kenken que s'indica en el seu nom, conté els jugadors com a clau dels seus punts. La separació per tamany permet un accés més directe i eficient als rànquings específics de cada nivell, evitant la necessitat de filtrar grans quantitats de dades.

Pels kenkens hem utilitzat diferents fitxers .txt, un per cada kenken creat. Això ha estat perquè, al tenir cada KenKen en el seu propi fitxer .txt es facilita l'accés i la modificació d'un KenKen específic sense afectar els altres. A més l'estructura per guardar els kenkens és un string i no requereix dividir el contingut en diferents objectes, ja que no s'ha d'accedir a aquests. En aquest cas, els .txt ofereixen una solució senzilla i més adequada, mantenint la llegibilitat.

Algorismes

Generació Aleatoria de KenKen

Aquest algorisme es compon de dos algorismes principals:

1. Generació de valors del tauler.
2. Generació de zones del tauler.

Fent una breu explicació general, aquest algorisme crea primer un tauler KenKen de tamany NxN, sent N el valor entrat per l'usuari, i primer l'emplena de valors aleatoris fent servir l'agorisme 1, posteriorment es passa aquest tauler emplenat a l'algorisme 2, el qual s'ocupa de generar una serie de zones de forma aleatoria però seguint les restriccions que té la normativa de zones d'un KenKen. A aquestes zones se les assigna una operació depenent del seu tamany, i un resultat corresponent al contingut de les caselles de la Zona i la operació assignada.

Anàlisi del cost

El cost complet de l'algorisme és de *O*(n^5), que ve donat per la funció de backtracking de generació de valors del tauler NxN.

1. Generació de valors del tauler

Aquest algorisme es compon de dues funcions: "generaAleatori" i "generaTaulAle"

La funció "generaAleatori" és la funció encarregada de crear l'objecte de tauler de KenKen i inicialitzar-lo, i posteriorment crida a la funció "generaTaulerAle".
Aquesta primera funció d'inicialització té un cost de Θ(n^2) degut al tauler de NxN.

La funció "generaTaulerAle()" és una funció de backtracking recursiva, que va emplenant les caselles del KenKen amb valors aleatoris que iteren entre 1 i N, sent N el tamany del KenKen. Aquest algorisme té en compte que per una fila i columna un valor no es pot repetir. Per aquesta raó necessitem el backtracking, ja que al emplenar el tauler de forma aleatoria, ens podem trobar en la situació en la que per la casella actual, el o els valors que encara tenim disponibles (els que no s'han fet servir a la fila de la casella) per emplenar-la no són vàlids per la posició, és a dir, en

aquella columna ja hi ha un de cada disponible per la fila, llavors haurem de tornar una casella enrere i provar amb un altre valor. El cost de fer el backtracking és de *O*(n^5).

2. Generació de zones del tauler

La generació de zones es composa també de dues funcions: “creaZones” i “bfsZones”.

La funció “creaZones” prepara un ArrayList de les caselles que estan per assignar, i fa un shuffle per iterar les caselles de forma aleatoria. Aquesta té un cost de Θ(n^2) ja que ha d’agafar les cel·les no assignades del tauler NxN. Quan es fa el shuffle per aleatoritzar el procediment també és té un cost de Θ(n^2)

Per a cada casella a iterar, es crida a la funció “bfsZones” que com indica el seu nom, és un algorisme basat en un BFS que genera zones explorant caselles no assignades, en direccions aleatòries respecte la casella actual en la que itera.
Mentre itera la zona a crear, apunta el valor de les caselles en un vector, el qual serveix un cop ha creat la zona, per calcular el resultat d’aquesta a partir de la operació que se li assigni, segons el seu tamany (degut a les característiques de les nostres operacions, les zones de tamany 2 es dediquen exclusivament a potències, mòduls, restes i divisions, qualsevol altre tamany es fa servir per sumes i multiplicacions) . La creació de zona acaba ja sigui perquè ha arribat al màxim de caselles per zona indicades com a paràmetre fixe, o perquè ha sortit amb antel·lació de forma aleatoria (per tal de generar zones de diferents tamanys, sino sempre tindriem zones de tamany màxim menys les que ha no hi capiguessin). La complexitat de l’algorisme BFS és de Θ(n^2) ja que recorre com a màxim totes les caselles.

Creació d'un KenKen

La vista IntroduirKenKen rep tota l'entrada com un string, l'envia al controlador de presentació i aquest al del domini. Ara, el domini s'encarrega de convertir el string en un ArrayList d'ArrayLists d'Integers (una matriu d'enters), separant el string primer per salts de fila (.split(/n)) i després cada fila per espais (.split(" ")). Aquesta funció s'encarrega, amb l'ús de l'excepció pròpia myException, la qual ens permet enviar excepcions amb el missatge desitjat, de controlar tots els errors de format possibles. Seguidament, s'envia aquesta matriu al controlador de kenkens on es crearà el kenken sense dificultats, recorrent la matriu d'enters correctament. Per acabar, la mateixa funció del control domini rebria el kenken i l'enviaria a una altra funció del controlador de kenken on es validaria abans de retornar-lo a presentació.

Anàlisi del cost

Aquest algorisme té cost Θ(n^2), sent n la mida del kenken. Això és així perquè, tot i estar recorrent-ho de forma diferent i dues vegades, els bucles del codi només depenen del nombre de caselles del kenken, que en total són la mida al quadrat.

Validació del KenKen

Per a la validació d'un kenken s'ha seguit una estratègia de backtracking, on es troba tots els possibles valors per a totes les caselles (si no és vàlid). Combinem un bucle que itera amb tots els possibles valors ([1..tamany]) i comprova a cada iteració si el valor és vàlid al tauler i a la zona, guardant-lo a la Casella en cas afirmatiu i avançant a la següent casella. Si no hi ha següent casella s'avançarà a la primera casella de la següent fila, en cas de no haver-hi vol dir que el KenKen és vàlid i ja hem guardat una solució. Per al cas de tornar enrere perquè cap valor és vàlid es posa la casella a 0 per a evitar errors en la validació.

Anàlisi del cost

Aquest és el mètode més costos del programa, amb un cost variable de *O*(n^5), sent n la mida del kenken. Per a entendre aquest cost, mirem primer el cost del bucle, el qual iterarà un màxim de n cops i el seu contingut té cost per la funció K.posicioValida, per tant ens queda que el bucle, sense les crides recursives té un cost de *O*(n^2). Seguidament, veiem que recursivament es podria cridar per a cada

casella (n^2 caselles) n cops (1 cop per a cada valor), llavors ens quedaria un cost total màxim de *O*(n^2)**O*(n^2)**O*(n) = *O*(n^5).

Demanar pista

Per a aquesta segona entrega, hem decidit millorar la funcionalitat de demanar una pista. A diferència de la primera entrega, on se li donava una posició d'una casella i retornava un valor vàlid en aquell context si n'hi havia, o altrament retornava un avís dient que el kenken era erroni, ara s'ha afegit que a part de comprovar el mencionat anteriorment, la casella de la pista serà aleatòria entre les que encara no tenen un valor assignat i es comprovarà si el valor és vàlid a futur per a la zona, és a dir, si la zona tindrà una possible solució a l'introduir aquell valor a la casella. En cas afirmatiu retorna el valor vàlid per a la casella i en negatiu avisa que en aquell moment de la partida el kenken no té solució (òbviament es comproven tots els valors, no només un).

Anàlisi del cost

El cost de l'algorisme és de *O*(n^(3+m)), on n és la mida del kenken i m és el nombre de caselles de la zona de la casella a comprovar. Segueix una justificació similar a la validació del kenken, però el nombre de caselles de la zona quasi sempre sol ser molt inferior al nombre total.

El cost de la validació de la zona és de *O*(n^(2+m)), ja que és un backtracking, però en comptes de totes les caselles només visitarà les de la zona, seguidament aquest backtracking es realitzarà com a molt un cop per cada possible valor, és a dir n cops, per tant, el cost és *O*(n^(2+m))**O*(n) = *O*(n^(3+m)).