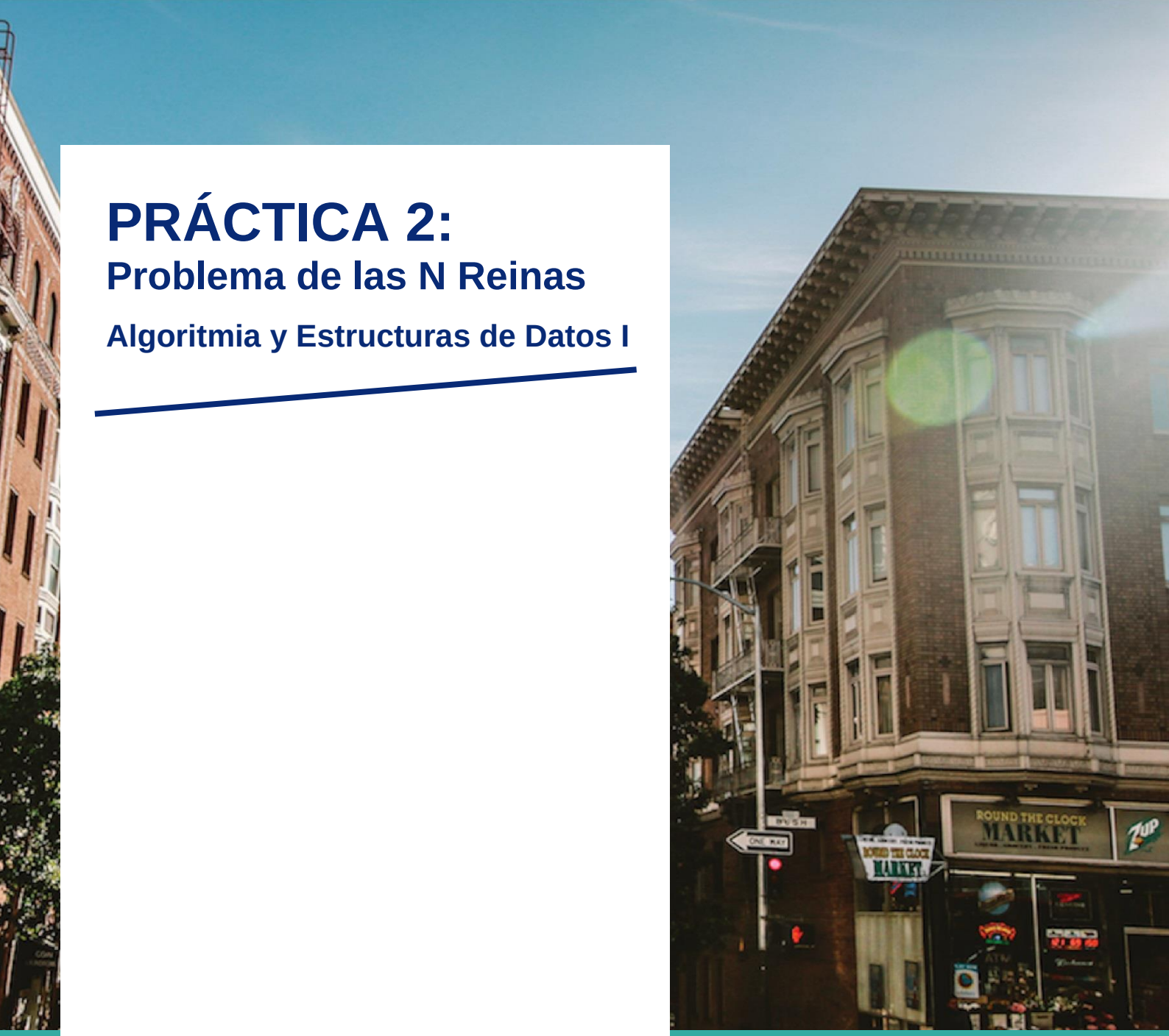
PRÁCTICA 2:
Problema de las N Reinas

Algoritmia y Estructuras de Datos I



27 GENER

Mario Ventura Burgos (43223476J)
Luis Miguel Vargas Durán (43214929E)
Felip Antoni Font Vicens (43224685A)

ENUNCIADO DE LA PRÁCTICA

ENUNCIADOS DE LOS EJERCICIOS

La práctica plantea una serie de ejercicios ciertamente parecidos, que deben ser resueltos mediante la implementación de un código recursivo en Java, es decir, usando backtracking.

Cada uno de estos ejercicios planteados está puntuado sobre 10 puntos, a excepción del primero, que se puntúa sobre 20 puntos al ser más complejo (este se puede interpretar como la implementación conjunta de varios de los ejercicios más simples puntuados sobre 10).

El enunciado propone que se entreguen ejercicios por un valor mínimo de 10 puntos, y sobre un máximo de 20 puntos. En nuestro caso, hemos optado por entregar 2 ejercicios simples, llegando así al valor máximo de 20 puntos.

Estos ejercicios son:

1. Dado un tablero de ajedrez de tamaño ***n***x***n*** (el tamaño, es decir, el valor de ***n***, lo introduce el usuario que ejecuta el programa), colocar en él ***n*** reinas de forma en que estas no se "maten" unas a otras. Es decir, dada ***n***, colocar ***n*** reinas de forma en que ninguna esté en la misma fila, la misma columna, o mismas diagonales que otra; y que, por tanto, puedan estar en el mismo tablero sin que ninguna esté en posición de matar o eliminarse unas a otras.
   Se deberán mostrar todas las soluciones posibles, si es que las hay, para el tamaño y número de reinas dado.
2. Dado un tablero de ajedrez de tamaño ***n***x***n*** (el tamaño, nuevamente lo introduce el usuario que ejecuta el programa), colocar una primera reina en una posición del tablero a elección del usuario y, en base a esta posición inicial, hallar una forma de colocar las ***n*** reinas de forma que no se maten entre sí.
   En este caso basta con mostrar una solución, si es que la hay.

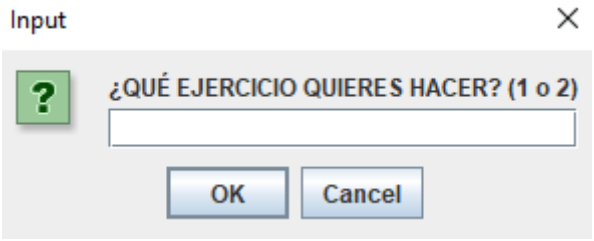
Cabe destacar que a pesar de que la solución debe ser encontrada mediante un algoritmo en Java que haga uso de backtracking, también se deberá implementar este código en pseudo PASCAL/pseudo código.

Presentados y planteados los dos ejercicios a resolver, y las condiciones que estos conllevan, empecemos con la implementación del código que resolverá estos problemas.

DESARROLLO DE CÓDIGO

CÓDIGO QUE RESUELVE LOS PROBLEMAS PLANTEADOS

El programa comienza su ejecución preguntando al usuario cuál de los dos ejercicios quiere hacer.



Obviamente, en función de lo que escoja el usuario se ejecutará el algoritmo que encuentra la solución para el ejercicio 1 o el ejercicio 2. Además, el programa tiene en cuenta que el usuario se equivoque y no escriba un 1 o un 2. En este caso, se informaría al usuario de que el número introducido no es válido y se volvería a preguntar por el ejercicio que se quiere hacer. Este proceso se repetirá hasta que el usuario introduzca un 1 o un 2, es decir, escoja un ejercicio/opción válido.

Pasemos pues, a la explicación del código que resuelve cada ejercicio, comenzando por el primero de los dos planteados.

·EJERCICIO 1 (Proyecto 4 del enunciado):

Recordemos que este ejercicio pedía hallar, mediante pseudo código y código en Java, una solución al problema de las n reinas en un tablero de ajedrez, pero con las condiciones siguientes:

1. El tamaño del tablero será introducido por el usuario que ejecute el programa.
2. Si el tablero es de tamaño *nxn,* se deberán colocar sobre el tablero un total de *n* reinas.
3. Dado el valor de *n*, se deberán mostrar todas las soluciones encontradas para este tamaño de tablero.
4. Dado que se pretende mostrar todas las soluciones posibles, la posición inicial sobre el tablero de la primera reina, no tiene relevancia.
5. Si dado un tamaño de tablero no se encuentra ninguna solución, se deberá avisar al usuario debidamente.

En primer lugar, por tal de mantener un orden creciente respecto a la complejidad de los algoritmos planteados, mostraremos y explicaremos primero la solución encontrada mediante un algoritmo en pseudo código, y posteriormente, la solución encontrada mediante el planteamiento de un algoritmo en un lenguaje de programación de alto nivel (en nuestro caso, Java).

1. PSEUDO CÓDIGO

Por tal de facilitar la comprensión del código, y no extender innecesariamente el informe, se mostrará el algoritmo recursivo sin entrar en excesivos detalles. Por ejemplo, en cuanto se mencione el código que hace las comprobaciones, este no se mostrará ya que es prácticamente idéntico al código en Java, y este podrá ser visto posteriormente.

Así pues, para evitar redundancias e informaciones innecesarias, se mostrará solo la parte del algoritmo en pseudo código que resuelve el problema planteado. Este algoritmo, es el siguiente:

```
func nReinas(t: tablero, etapa: entero) returns (soluciones: tableros)
{
BEGIN
    Tipus tableros = array[1..numSoluciones] of tablero;
    Var soluciones: tableros;
    Var ind,numSoluciones: entero;
    Var haySolucion: booleano
    ind:=0;
    numSoluciones:=0

    FOR i:=1 to numReinas DO BEGIN
        IF t.isValido THEN BEGIN          //Estado actual aceptable
            t.colocarReina(fila,columna); //Anotar solucion
            IF etapa<numReinas THEN //Solucion incompleta?
                nReinas(t,etapa+1);   //Siguiente etapa
            ELSE BEGIN
                soluciones[ind]:=t;    //Solucion encontrada, se guarda y se sigue
                numSoluciones:=numSoluciones+1;
                ind:=ind+1;           //Volver al estado anterior
                t.eliminarReina(fila,columna);
            END IF
        END IF
    END FOR
    IF numSoluciones > 0 THEN
        haySolucion:= true;
        Var n: entero
        n:=preguntarSolucion;
        imprimirSolucion(soluciones[n]);
    ELSE THEN
        haySolucion:=false;
END
}
```

Tal y como se ve en la página anterior, el funcionamiento del algoritmo anterior se basa en recorrer todo el tablero encontrando todas las posibles formas de colocar n reinas de manera que no se maten entre ellas.

Esto se hace de forma en que se busca una primera solución, y al encontrarla, la almacenamos en un array de tableros. Esto se debe a que el objetivo de la función es encontrar todas las soluciones al problema y dar al usuario la posterior posibilidad de escoger cuál de ellas quiere ver. Así pues, se encuentra una solución para el tamaño de tablero dado, se almacena la solución, y se elimina la última reina colocada. Haciendo esto último, podemos comprobar si existe alguna otra solución para N reinas que tenga la misma configuración inicial para N-1 reinas.

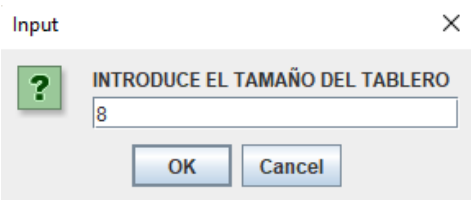
Por ejemplo, si N=8, puede ser que, colocadas las 7 primeras reinas, haya varias formas de encontrar una solución final. Si esto no es así, debido a que la función es recursiva, se eliminará la séptima reina y se comprobarán posibles soluciones dada la configuración de las primeras 6 reinas, y así sucesivamente. Esto hace que el rendimiento del algoritmo sea mucho mayor y le permita encontrar soluciones mucho más rápido.

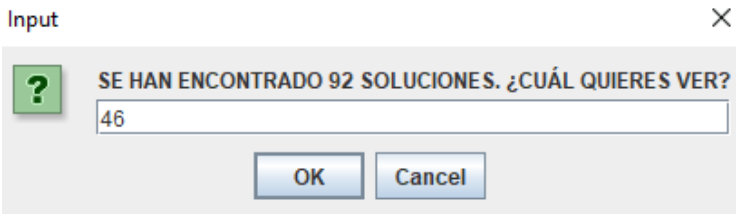
Una vez hemos acabado de buscar todas las posibles soluciones, si se ha encontrado alguna, se informa al usuario y se le pregunta cuál de ellas quiere ver.

2. ALGORITMO EN JAVA

En Java, la solución encontrada para el ejercicio consta de varias partes (métodos), por tanto, es vital entender cómo funciona el algoritmo, con qué partes cuenta, cuál es la función de cada una, y cómo se ha llegado a ella.

En primer lugar, se debe iniciar un tablero. Debido a que Java es un lenguaje de alto nivel orientado al objeto, podemos crear un objeto Tablero. En nuestro caso, debido a que el objetivo del algoritmo es mostrar todas las posibles soluciones a este problema, y teniendo en cuenta que puede haber muchas soluciones (mismamente, para un tablero de ajedrez estándar de tamaño 8x8, hay 92 soluciones), no tiene sentido mostrar la solución de forma gráfica en un primer momento, ya que esto implicaría crear 92 ventanas (en el caso del tablero 8x8) que muestren un tablero de ajedrez con una solución diferente cada uno; y esto es algo ilógico y absurdo por razones obvias. La solución planteada, por tanto, encuentra todas las soluciones posibles del ejercicio para un tablero de tamaño nxn, y almacena cada una de las soluciones en un array de objetos tablero. Hecho esto, se le preguntará al usuario cuál de todas las soluciones quiere ver y se mostrará la solución escogida por el usuario.





Imágenes que ilustran la explicación anterior

Debido a que trabajar con gráficos hace que el rendimiento del algoritmo sea más lento, además de añadir una dificultad innecesaria al algoritmo, se trabaja con gráficos cuando es totalmente necesario.

Encontraremos todas las soluciones, las almacenaremos en un array de matrices de Strings, y al preguntar al usuario que solución quiere ver, mostraremos gráficamente **únicamente la solución escogida por el usuario**. De esta forma, todo el proceso gráfico se ejecuta una sola vez: Si se encuentra solución, y transformando de matriz a gráficos solo una de las muchas soluciones encontradas.

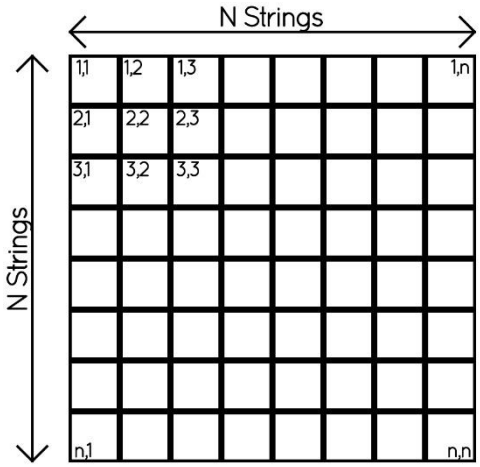
Nuestro algoritmo comienza, por tanto, creando un objeto tablero, que será, en un primer momento, un array bidimensional o matriz de Strings (String[][]).
Hecho esto lo próximo será preguntar al usuario por el tamaño del tablero. Una vez este haya introducido un valor para n, instanciamos el tablero creado, dándole dicho tamaño, y procedemos a darle un valor a todas las casillas de este (strings) iterando de forma en que se recorran todos los Strings que componen la matriz. Esto lo haremos mediante los siguientes métodos:

```
//Variables
String[][] tablero;
int tamaño;
//Preguntar al usuario por el tamaño
tamaño = Integer.parseInt(JOptionPane.showInputDialog("INTRODUCE EL TAMAÑO"));
//Instanciar tablero pasando el tamaño como parámetro
tablero = generarTablero(tamaño)*;
```

*

El método **generarTablero(int tamaño)** inicializa un tablero inicial previo a encontrar ningún tipo de solución. En él, simplemente hay una iteración por filas y columnas, que recorre toda la matriz dando un valor inicial a cada casilla de esta (el valor “ “).

Ahora que tenemos un tablero sobre el que operar, podemos desarrollar un algoritmo con backtracking que actúe sobre él, encontrando todas las soluciones posibles.



Representación gráfica del tablero de Strings.

Para ello, se ejecuta el método ***ejercicio1(int tamaño)***. Este método establece inicialmente a 0 el contador de soluciones encontradas, y crea una variable booleana "*solucionado=false*" que se usará para darle el valor *true* únicamente si se encuentra una solución. De esta forma, al acabar el algoritmo, si *solucionado == false*, se sabe que el algoritmo no tiene solución, y se avisa al usuario.

```
//RESTAURAR Nº DE SOLUCIONES
numSoluciones = 0;

//PARA SABER SI SE HA ENCONTRADO UNA SOLUCIÓN
solucionado = false;
```

Después de buscar soluciones al ejercicio:

```
//SI NO SE ENCUENTRA NINGUNA SOLUCIÓN, AVISAR AL USUARIO
if (solucionado == false) {
    aviso("------< NO SE HA ENCONTRADO SOLUCIÓN AL PROBLEMA >------","AVISO");
} else {
```

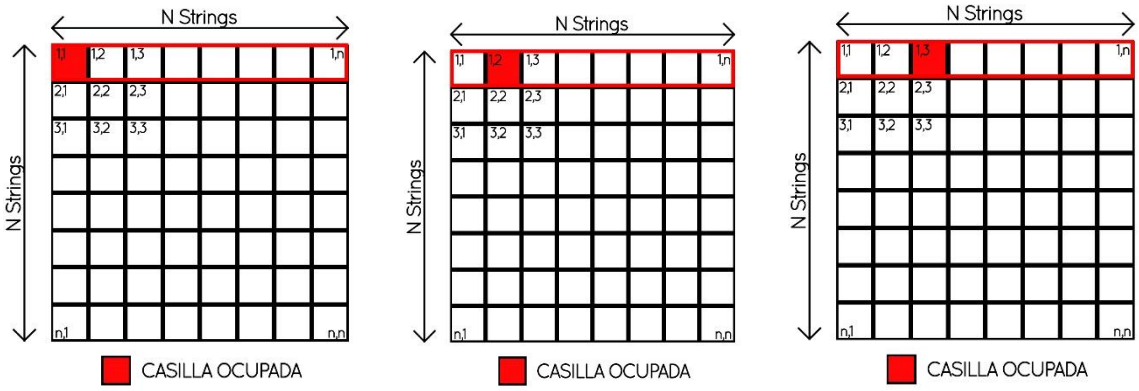
Hecho esto, procedemos a ejecutar el algoritmo recursivo. Este es un método que recibe el nombre ***ubicarReina(String[][] tablero, int etapa).***

Este método coge el tablero que recibe como parámetro y busca sobre él todas las posibles soluciones. Mediante una iteración que recorre casillas de la matriz, comprueba si en ellas se puede colocar una reina, y si se puede, establece con el valor "D" el String de la casilla de la matriz en cuestión. Cada vez que hace esto, el número de soluciones aumenta en 1, y repite el proceso en la próxima iteración, en la que se invoca de nuevo al método ***ubicarReina*** (ahí la recursividad).
Si en algún momento se obtiene que el número de soluciones es igual al tamaño del tablero (***numSoluciones==tablero.length***), querrá decir que hemos encontrado una forma de colocar n reinas en un tablero de tamaño nxn sin que estas se maten mutuamente. Habremos encontrado, por tanto, una solución.

Cuando esto sucede, almacenamos la información de la matriz en un array de matrices, aumentamos el tamaño de este en 1 para poder almacenar una posible próxima solución y repetimos el proceso **CON BACKTRACKING**.

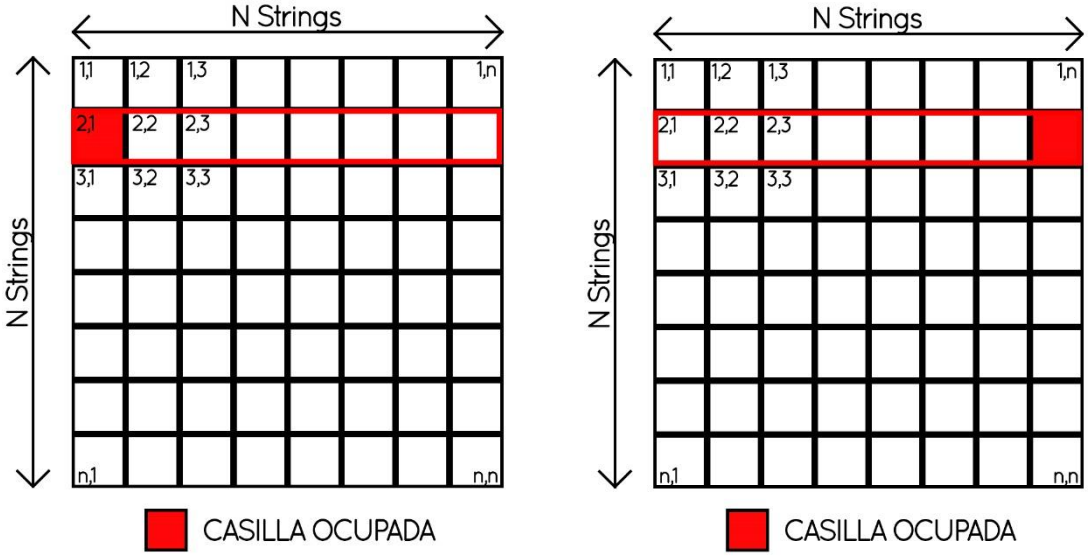
Esto quiere decir que, si se han colocado, por ejemplo, 8 reinas; se guarda dicha solución, se elimina la última reina, y se comprueba si con la configuración actual de las primeras 7 reinas colocadas existe otra solución. Si es así se crea, guarda y se vuelve a hacer esto; de lo contrario, se elimina la séptima reina y se comprueba si con la configuración actual de las primeras 6 reinas existe otra solución (si fuese así, se colocaría la reina y tendríamos una nueva configuración con 7 reinas a partir de la cuál seguir comprobando), etc… Es decir, ***se vuelve atrás (Backtracking)***.
Cada vez que se vuelve atrás, la casilla de la matriz que anteriormente tenía almacenado el valor "D", ahora volverá a tener el valor " " (es decir, se restaura/limpia su valor).

Las siguientes imágenes ilustran mejor como se realiza la iteración:



Previamente, las primeras iteraciones sobre la primera fila.

A continuación, la primera y última iteración sobre la siguiente fila, respectivamente:



El algoritmo en cuestión es el siguiente:

```
//MÉTODO PARA COLOCAR LA FICHA
  public static void ubicarReina(String[][] tablero, int etapa) {
    for (int i = 0; i < tablero.length; i++) {
      //Comprobar si se puede colocar una reina
      if (isValido(tablero, i, etapa)[1]) {
        //Se puede, colocar la reina
        tablero[i][etapa] = "D";
        //Comprobar si se ha llegado a una solución
        if (etapa < tablero.length - 1) {
          //No se ha llegado, colocar próxima reina
          ubicarReina(tablero, etapa + 1);[2]
        } else {
          //---------SOLUCIÓN ENCONTRADA---------
          //Crear tablero gráfico para almacenar solución
          tab = new Tablero(tamaño);
          //Aumenta el número de soluciones
          numSoluciones++;
          //La lista de soluciones auxiliar aumenta
          auxiliar = new Tablero[numSoluciones];
          //Copiamos las soluciones a la nueva lista auxiliar
          for (int idx=0; idx<soluciones.length; idx++) {
            auxiliar[idx] = soluciones[idx];
          }
          tab = copiarTablero(tablero, tab);
          auxiliar[numSoluciones-1] = tab;
          //Aumentamos el tamaño de la lista de soluciones
          soluciones = new Tablero[numSoluciones];
          //Copiamos las soluciones a la nueva lista auxiliar
          for (int idx=0; idx<soluciones.length; idx++) {
            soluciones[idx] = auxiliar[idx];
          }
        }
        //Volver atrás
        tablero[i][etapa] = " ";
      }
    }
  }
```

[1] y [2]: Mirar las aclaraciones de la siguiente página

Respecto al algoritmo mostrado en la página anterior, se deben aclarar algunas cosas:

1

ARRAY AUXILIAR:

Para almacenar la solución en un array y posteriormente aumentar en 1 el tamaño de este para almacenar una posible próxima solución, se necesita un array auxiliar. Este array auxiliar tiene un tamaño igual al del array de soluciones y se usa de la siguiente manera:

Se crea el array auxiliar con tamaño dado, se copia el array de soluciones en el auxiliar, se aumenta en 1 el tamaño del array de soluciones, y se copia el array auxiliar en el de soluciones. De esta forma, tenemos el mismo array de soluciones inicial, pero con un espacio más, que estará vacío y listo para ser usado en caso de que sea necesario.

Cabe mencionar, además, que el array auxiliar no se define como una variable global del programa (a diferencia del array de soluciones), ya que no tiene que almacenar nada y sólo tiene la función descrita anteriormente. De esta forma se evita que el programa use memoria para almacenar variables o listas de ellas que realmente no necesita.

2

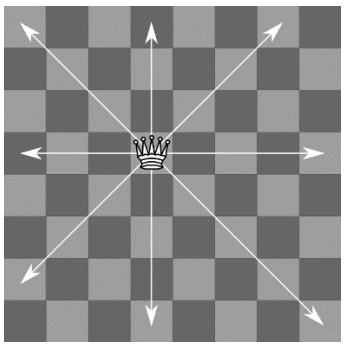
COMPROBACIONES:

En el algoritmo mostrado en la página anterior, una parte vital es aquella en la que se comprueba si una reina se puede colocar en el tablero. Para ello, se usa un método por separado, que devuelve true si se puede colocar una reina, y false en caso contrario. Este es el método ***isValido(tablero, i, etapa).***

La comprobación de si se puede colocar una reina en el tablero se basa en asegurarse de:

1. No hay ninguna reina colocada en la fila actual
2. No hay ninguna reina colocada en la columna actual
3. No hay ninguna reina colocada en ninguna de las diagonales respectivas a la casilla que se está comprobando en cualquier momento.

En cuanto alguna de estas condiciones se cumple, el método retorna ***false***; pero si en caso contrario, ninguna de estas condiciones se da, esto significa que es posible colocar una reina en la casilla actual, y, por tanto, el método retorna el valor ***true***.



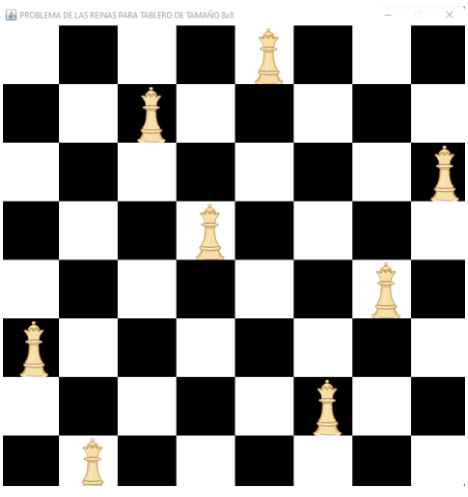
El código del algoritmo explicando anteriormente, que comprueba si se puede colocar una reina en una casilla o no, es el siguiente:

```
public static boolean isValido(String[][] tablero, int i, int etapa) {
    for (int x = 0; x < etapa; x++) {
      if (tablero[i][x].equals("D")) {
        return false;
      }
    }
    for (int j = 0; j < tablero.length && (i - j) >= 0 && (etapa - j) >= 0; j++) {
      if (tablero[i - j][etapa - j].equals("D")) {
        return false;
      }
    }
    for (int j = 0; j < tablero.length && (i + j) < tablero.length && etapa - j >= 0; j++) {
      if (tablero[i + j][etapa - j].equals("D")) {
        return false;
      }
    }
    return true;
  }
```

Hecho esto, ya sabemos como funciona el algoritmo recursivo que encuentra todas las posibles soluciones al problema. También conocemos cuáles son los métodos que usa, cómo funcionan y qué rol desempeñan.

Una vez termine la ejecución de este algoritmo, si se han encontrado soluciones dado un tamaño n, estas se habrán almacenado en un array de objetos tablero.

El último paso será preguntar al usuario cuál de todas las soluciones encontradas quiere ver, y cuando escoja una, mostrarla de forma gráfica mediante una ventana que se abrirá mostrando el resultado de la solución encontrada y la configuración de las reinas sobre el tablero.



·EJERCICIO 2 (Proyecto 3 del enunciado):

Recordemos que este ejercicio pedía hallar, mediante pseudo código y código en Java, una solución al problema de las n reinas en un tablero de ajedrez, pero con las condiciones siguientes:

1. El tamaño del tablero será introducido por el usuario que ejecute el programa.
2. Si el tablero es de tamaño *nxn,* se deberán colocar sobre el tablero un total de *n* reinas.
3. Dado el valor de *n*, se deberá mostrar una única solución, si es que esta existe
4. La posición inicial sobre el tablero de la primera reina será introducida por el usuario.
5. Si dado un tamaño de tablero no se encuentra ninguna solución al problema, se deberá avisar al usuario debidamente.

Al igual que en el caso anterior, por tal de mantener un orden, se mostrará primero el algoritmo mediante pseudo código, y posteriormente el código en Java.

1. PSEUDO CÓDIGO

El algoritmo que resuelve el problema descrito anteriormente es el siguiente:

```
func nReinasPI(t: tablero, etapa: entero) returns (t': tablero)
{
BEGIN
    Var f,c: entero;
    Var haySolucion: booleano
    f:=preguntarFila();
    c:=preguntarColumna();
    //Colocar reina en posición inicial
    t.colocarReina(f,c);
    etapa:=etapa+1;
    //Llenar el resto del tablero en búsqueda de una solución
    FOR i:=1 to (numReinas-1) DO BEGIN
        IF t.isValido THEN BEGIN              //Estado actual aceptable
            t.colocarReina(fila,columna);     //Anotar solucion
            IF etapa<numReinas THEN           //Solucion incompleta?
                nReinasPI(t,etapa+1);         //Siguiente etapa
            ELSE BEGIN
                //Solucion encontrada
                haySolucion:= true;
            END IF
        END IF
    END FOR
```

```
    IF haySolucion = true THEN
        imprimirTablero(t);
    ELSE THEN
        aviso("NO SE HA ENCONTRADO SOLUCIÓN");
END
}
```

El funcionamiento de este algoritmo es razonablemente parecido al del ejercicio 1. En este caso, sin embargo, se pregunta al usuario por la fila y columna inicial de la primera reina antes de empezar a iterar sobre el tablero buscando una solución.
Así pues, lo único que cambia es esto, además de que el número de iteraciones buscando una posible solución, se hará sobre el valor numReinas-1, ya que una de las N reinas ya ha sido colocada inicialmente, y por tanto, buscamos una forma de colocar las N-1 reinas restantes.
Si se encuentra una solución, se esta se mostrará al usuario, y en caso contrario, se le mandará un aviso al usuario informándole de que no existe una forma de colocar n reinas en un tablero nxn dada la posición inicial introducida.

2. ALGORITMO EN JAVA

Al igual que en el caso del pseudo código, el algoritmo en Java que resuelve este ejercicio es esencialmente el mismo. De hecho, por tal de evitar código redundante, los métodos que se usan son los mismos ya que las funcionalidades que implementan ambos ejercicios y la dinámica que siguen son prácticamente las mismas.
Este es el caso del método ***isValido(tablero, i, etapa)***, que también es el método que se usa en el ejercicio 2 para comprobar si se puede colocar una reina en una casilla concreta del ordenador. Lo mismo sucede con el método que inicializa la matriz de Strings.
De esta manera, lo único que en realidad cambia es el algoritmo recursivo, que itera con el mismo criterio y la misma dinámica, pero haciendo una iteración menos.

Este método (cuyo código no será mostrado debido a que es, esencialmente, una copia traducida del algoritmo pseudo código a Java), tiene ciertos cambios que hacen que su funcionamiento tenga ciertas diferencias al código del ejercicio 1. Estas diferencias son:

En primer lugar, debido a que la posición inicial de la primera reina sobre el tablero debe ser introducida por el usuario, esto será lo primero que haremos. Una de las primeras cosas que deberá hacer este método será declarar dos variables (o usar dos variables globales que ya hayan sido declaradas previamente) de tipo entero, que almacenen la fila y columna en la que el usuario quiere colocar la primera reina sobre el tablero.

Para darles valor, se abrirá una ventana emergente de tipo ***JOptionPane***, que contenga un espacio para que el usuario pueda escribir un número para la fila, y posteriormente, otra ventana con el mismo funcionamiento para la columna.
Una vez almacenados los valores de fila y columna, accederemos a la casilla (*fila,columna*) del objeto tablero que tengamos (que hasta este momento debe estar totalmente vacío y sin ninguna reina situada sobre él) y colocaremos una reina. A partir de este momento será cuando empiecen las iteraciones que recorrerán todo el tablero en busca de configurar una solución para las **N-1 reinas** que restan.
Las iteraciones, sin embargo, también presentan un cambio respecto al código planteado en el ejercicio 1.
En el ejercicio 1 buscábamos todas las posibles soluciones para el problema de las N reinas, y por eso iterábamos sobre un máximo de N veces. Mediante una sentencia condicional "*if*", comprobábamos si el número de reinas colocadas sobre el tablero era igual al del tamaño del tablero (para n filas y n columnas, deben ser colocadas n reinas). De ser así, esto significaba que habíamos encontrado una solución y la guardábamos en un array de soluciones ya que el objetivo era almacenar todas ellas para posteriormente escoger cuál queríamos ver.
En el ejercicio 2, sin embargo, no iteraremos n veces y tampoco almacenaremos las soluciones en un array. La explicación del primero de estos cambios se debe a que en el ejercicio 2 no empezamos a iterar sobre un tablero vacío buscando en él una solución, sino que empezamos a iterar sobre un tablero que ya tiene colocado una reina. Esto hace que el estado inicial del tablero sea diferente, y que por tanto no iteremos sobre n reinas, sino sobre n-1, que son las que en realidad nos quedan por colocar. Cada iteración, sin embargo, hace prácticamente lo mismo, a excepción del segundo de los cambios mencionados anteriormente.
Debido a que este ejercicio pretende encontrar una única solución al problema de colocar n reinas sobre un tablero, ya no hace falta almacenar las soluciones en un array y seguir iterando. Esto ya no es una necesidad ya que de nada sirve un array si no queremos almacenar una lista de variables u objetos de un mismo tipo. En nuestro caso, lo único que queremos ver es una solución sobre tablero, con lo cual una variable será suficiente.
Una variable global de tipo tablero guardará la configuración de las reinas si se llega a una solución.
En caso de que esta exista, se mostrará el tablero gráficamente por pantalla, y en caso contrario, se mostrará un aviso al usuario de que no existe ninguna forma de colocar n reinas sobre el tablero de tamaño dado tomando la posición inicial que se ha introducido.