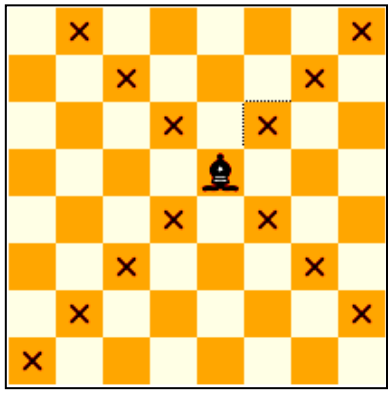
Problema 2



Para un tablero de NxN casillas se desea obtener el máximo número de alfiles que pueden situarse en dicho tablero sin que se amenacen entre ellos (sólo se puede situar un alfil en cada casilla). En la figura adjunta se muestra un alfil y las casillas amenazadas por él. Aunque puede existir un algoritmo voraz que resuelva el problema, se pide:

(a) Implementar un algoritmo basado en ***backtracking*** que devuelva el número de alfiles y las posiciones de éstos de una de las posibles soluciones óptimas.
- Indicar el esquema usado, incluyendo el código del mismo
- Documentar claramente las variables utilizadas
- Dibujar un esquema con el árbol de expansión correspondiente a la propuesta (al menos dos niveles aparte del nodo raíz)
- Codificar los métodos abstractos del esquema y otro en el que aparezca la llamada inicial al algoritmo de *backtracking*

(b) (EXCEPTO ITIS) Utilizar una función de cota que evite la exploración de nodos atendiendo a la función objetivo a optimizar (número máximo de alfiles que pueden ponerse)
- Definir dicha función de cota (formalmente o mediante lenguaje natural)
- Codificar dicha función
- Incorporar su uso en el algoritmo de *backtracking*

Solución

```
Clase EsquemaBtOptimo
   proc btOptimo(x: Etapa)
   var
       xsig: Etapa
       cand: Candidatos
   alg
       si esSolucion(x):
           si esMejor():
               actualizaSolucion()
           fsi
       fsi
       cand := calculaCandidatos(x)
       mientras quedanCandidatos(cand)
           xsig = seleccionaCandidato(cand, x)
           si esPrometedor(cand, x, xsig):
               anotaSolucion(cand, x, xsig)
               btOptimo(xsig)
               cancelaAnotacion(cand, x, xsig)
           fsi
       fmientras
   fin
   …
fClase
```

Un enfoque sencillo es considerar una etapa por cada casilla, y los candidatos los posibles valores asociados con poner o no un alfil en cada casilla.

```
Clase Alfiles hereda de EsquemaBTOptimo
   N: Entero    // el tablero tiene dimensiones NxN
   sol, solOpt: Solucion
   xini: Etapa

   Clase Posicion
      i, j: Entero
   fClase

   Clase Solucion
      alfColocados: array [1..2*N-1] de Posicion    // posiciones de los alfiles
                                 // que han sido colocados (como mucho son 2*N-1)
      numAlfiles: Entero    // número de alfiles colocados
   fClase


   Clase Etapa
      k: Entero   // número de casillas consideradas
      pos: Entero   // posición de la última casilla considerada
   fClase


   Clase Candidatos
      i: Entero    // 0..1 (no poner o poner un alfil)
   fClase


   proc colocaAlfiles()
   alg
      xini.k := 0    // es irrelevante la posición para la etapa inicial
      sol.numAlfiles := 0
      solOpt.numAlfiles := 0
      btOptimo(xini)
      < en solOpt está una de las posibles soluciones óptimas >
   fin


   func esSolucion(x: Etapa) dev (b: Logico)
   alg
      b := (x.k = N*N)
   fin


   func esMejor() dev (b: Logico)
   alg
      b:= (sol.numAlfiles > solOpt.numAlfiles)
   fin


   proc actualizaSolucion()
   alg
      solOpt := < copia sol >
   fin
```

```
   func calculaCandidatos(x: Etapa) dev (cand: Candidatos)
   alg
      si x.k = N*N:    // acabar
            cand.i := 1
      | otras:
            cand.i := -1;    // los valores candidatos válidos son 0 y 1
      fsi
   fin


   func quedanCandidatos(cand: Candidatos) dev (b: Logico)
   alg
      b := (cand.i < 1)
   fin


   func seleccionaCandidato(cand: Candidatos; x: Etapa) dev (xsig: Etapa)
   alg
      cand.i := cand.i + 1
      xsig.k := x.k + 1
      xsig.pos.i := ⌊(xsig.k - 1) / N⌋ + 1
      xsig.pos.j := ((xsig.k - 1) % N) + 1
   fin


   func esPrometedor(cand: Candidatos; x, xsig: Etapa) dev (b: Logico)
   var
      i: Entero
   alg
      b := cierto
      i := 1
      mientras b Y i <= sol.numAlfiles
      b := (abs((sol.alfColocados[i]).i – xsig.pos.i) <>
            abs((sol.alfColocados[i]).j – xsig.pos.j))
      i := i + 1
      fmientras
   fin


   proc anotaSolucion(cand: Candidatos; x, xsig: Etapa)
   alg
      si cand.i = 1:
            sol.numAlfiles := sol.numAlfiles + 1
            sol.alfColocados[sol.numAlfiles] := < copia xsig.pos >
      fsi
   fin


   proc cancelaAnotacion(cand: Candidatos, xsig: Etapa)
   alg
      si cand.i = 1:
            sol.numAlfiles := sol.numAlfiles – 1
      fsi
   fin

fClase
```

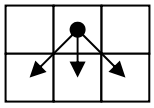
(b)

Una función de cota muy sencilla sería considerando que en todas las casillas que aún no se han considerado (N*N-xsig.k) puede colocarse un alfil, por lo que la mejor solución que podría obtenerse a partir del nodo actual (xsig) tendría como mucho sol.numAlfiles + cand.i + (N*N-xsig.k) alfiles. Si esta cantidad no supera al valor de solOpt.numAlfiles, no sería necesario continuar la búsqueda por ese nodo. Dicha condición puede establecerse en la función esPrometedor (la fórmula anterior es válida en el momento de ejecutar dicha función), que quedaría:

```
func esPrometedor(cand: Candidatos; x, xsig: Etapa) dev (b: Logico)
var
   i: Entero
alg
   b := ((sol.numAlfiles + cand.i + (N*N-xsig.k)) > solOpt.numAlfiles)
   i := 1
   mientras b Y i <= sol.numAlfiles
   b := (abs((sol.alfColocados[i]).i – xsig.pos.i) ≠
         abs((sol.alfColocados[i]).j – xsig.pos.j))
   i := i + 1
   fmientras
fin
```

Se pueden obtener funciones de cota mejores, considerando las casillas restantes no amenazadas por los alfiles ya colocados. Un recorrido del tablero por diagonales también permitiría definir una función de cota muy efectiva: dado que sólo puede colocarse un alfil por diagonal, el número máximo de alfiles que puede colocarse es el número de diagonales que no han sido recorridas.

Tenemos un juego que se desarrolla sobre un tablero de n x n casillas, donde cada una de las casillas tiene un valor entero positivo asociado, representado mediante el array bidimensional t de n x n enteros. El juego consiste en partir de una casilla de la fila superior (que asumimos que es la fila 0) y llegar hasta la fila inferior (la n – 1), **maximizando la suma de los valores asociados con las casillas por las que pasamos**. Los movimientos que podemos hacer son siempre para pasar de una casilla a otra de la fila inmediatamente inferior, y son posibles tres: o bien bajar en vertical (↓), o bien bajar por cualquiera de las dos diagonales (↙ o ↘), siempre que el movimiento no haga que nos salgamos del tablero; es decir, de la casilla (i, j) únicamente podemos pasar a la (i + 1, j – 1), a la (i + 1, j) o a la (i + 1, j + 1), aunque si estamos en la primera columna (j = 0) sólo tiene sentido el segundo y tercer término y si estamos en la última columna (j = n – 1) sólo tienen sentido el primero y el segundo. La casilla de la que se parte en la primera fila puede ser cualquiera, al igual que a la que se llega en la última fila. Los movimientos válidos se representan en la siguiente figura:



Movimientos válidos

Queremos resolver este problema mediante Programación Dinámica. Para ello definimos previamente gm_{ij} como la ganancia máxima que se puede obtener partiendo de la casilla (i, j) y llegando a cualquier punto de la fila inferior. Los casos base se dan, por tanto, cuando i = n – 1 (ya se está en la última fila), puesto que la ganancia en ese caso es el valor de la casilla (i, j), t[i][j]. La expresión recursiva que nos permite calcular cualquier otro término cuando i < n – 1 es

$$gm_{ij} = t[i][j] + \max(gm_{i+1,j-1}, gm_{i+1,j}, gm_{i+1,j+1})$$

siempre que los tres términos tengan sentido, es decir, que 0 < j < n – 1; en caso contrario (primera o última columna) sólo se considerarán los términos con sentido.

Por ejemplo, para el tablero

	0	1	2	n-1
0	3	**5**	2	1
1	8	2	**7**	9
2	1	5	2	**10**
n-1	1	4	**8**	3

la ganancia máxima es 30, que se obtiene partiendo de la casilla (0, 1) y llegando por el camino marcado en negritas ((0, 1) → (1, 2) → (2, 3) → (3, 2)) hasta la casilla (3, 2).

Con las ideas precedentes, rellénense los huecos del código de la clase JuegoTesorosPD, de modo que se resuelva el problema según la estructura de las prácticas vistas en clase; ha de respetarse el código que se da y hay que escribir en el sentido que indican los comentarios.

Notas:

- Suponemos definidas dos funciones max, con dos y tres argumentos enteros, respectivamente, que devuelven el mayor de los argumentos.
- El problema se modela mediante una clase que se denomina ProblemaTesoros y la solución a desarrollar pertenece al subpaquete soluciones.tesoros.
- La solución debe escribirse, a lápiz o bolígrafo, exclusivamente dentro de los huecos del código que se adjunta.
- La solución debe calcular únicamente la ganancia máxima, no el camino necesario para obtenerla.

Tiempo: 45 minutos **Puntuación: 5 puntos**

Apellidos:_____Nombre:_____*

```
package soluciones.tesoros;

import problemas.ProblemaTesoros;
import soluciones.EstrategiaSolucion;
```

(0.5)

```
public class JuegoTesorosPD implements EstrategiaSolucion {
```

(0.5)

```
    /** El valor del tesoro de cada casilla */
    private int[][] t;

    /** El tamaño del tablero */
    private int n;

    /** La ganacia máxima obtenida */
    private int gananciaMaxima;

    public JuegoTesorosPD(ProblemaTesoros pt) {
        t = pt.getT();
        n = pt.getN();
    }

    public void procesamientoInicial() {
    }

    public void procesamientoFinal() {
    }

    public void solucion() {
        juegoTesoroPD();
```

(0.5)

```
    }

    private void juegoTesoroPD() {
        /* La matriz en la que se va almacenando la ganancia máxima */
        int[][] gm = new int[n][n];

        /* Casos base, (i = n - 1) */
        for (int j = 0; j < n; j++) {
            gm[n - 1][j] = t[n - 1][j];
        }
```

(1.5)

```
        /* Casos recursivos (i < n - 1) */
        for (int i = n - 2; i >= 0; i--) {
            /*
             * Caso j = 0; sólo tienen sentido los dos últimos
             * términos
             */
            gm[i][0] = t[i][0] + max(gm[i + 1][0], gm[i + 1][1]);
```

(1.5)

```
            /* Casos j = 1..n - 2; tienen sentido los tres términos */
            for (int j = 1; j < n - 1; j++) {
                gm[i][j] =
                        t[i][j]
                                + max(gm[i + 1][j - 1], gm[i + 1][j],
                                        gm[i + 1][j + 1]);
            }
```

(2.0)

```
            /*
             * Caso j = n - 1; sólo tienen sentido los dos primeros
             * términos
             */
            gm[i][n - 1] =
                    t[i][n - 1]
                            + max(gm[i + 1][n - 2], gm[i + 1][n - 1]);
```

(1.5)

```
        }

        /* La máxima ganancia es el mayor valor de la fila 0 */
        gananciaMaxima = Integer.MIN_VALUE;
        for (int j = 0; j <= n - 1; j++) {
            if (gm[0][j] > gananciaMaxima) {
                gananciaMaxima = gm[0][j];
            }
        }
```

(2.0)

```
    }

    public String toString() {
        String s = "Ganancia juego Progamación Dinámica: "
                + gananciaMaxima;
        return s;
    }

}
```

La puntuación asignada a cada apartado aparece a su derecha.

---

* Recuerda que tienes que resolver el ejercicio en estas mismas hojas. No olvides poner tus apellidos y tu nombre. Suerte