Examen de Algorítmica y Complejidad (Plan 2014)

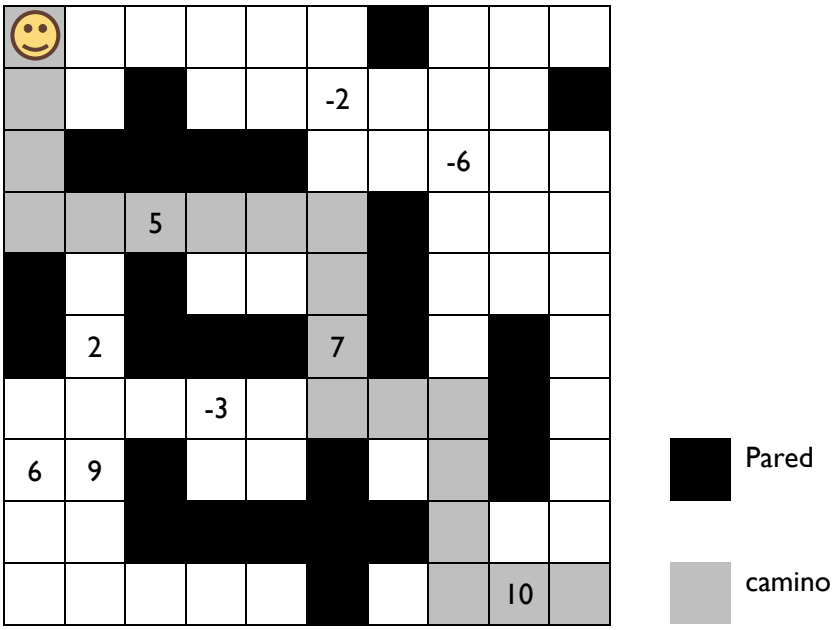
16 de diciembre de 2019

Nº matrícula:_____ **Nombre:** _____

Apellidos: _____

Problema. Consideremos el siguiente juego: disponemos de un tablero representado por una matriz de tamaño *NxN*. Inicialmente nos encontramos en la casilla de coordenadas (0,0), y nuestro objetivo es alcanzar la de coordenadas (*N*-1, *N*-1). Tenemos la opción de movernos a la casilla adyacente de **la derecha o a la de abajo**, pero nunca a la de la izquierda o a la de arriba. Algunas casillas están marcadas con un importe monetario que percibimos por transitar por ellas (un valor positivo si cobramos una recompensa o un valor negativo si tenemos que pagar un peaje). Además, algunas casillas del tablero son paredes por las que no es posible pasar. Deseamos conocer el beneficio máximo que nos reporta el mejor camino que transita desde la casilla de coordenadas (0,0) a la de coordenadas (*N*-1, *N*-1).

Ejemplo: Dado el siguiente tablero, el camino marcado daría el máximo beneficio 22.



Para resolver este problema se pide implementar un algoritmo en Java basado en **programación dinámica** con complejidad O(N^2) en tiempo y O(N) en memoria con la siguiente cabecera:

```
int maxBeneficio(int[][] tablero)
```

donde:

- `int[][] tablero` representa el tablero. El beneficio obtenido por pasar por una casilla (i,j) está representado por el valor `tablero[i][j]`. Una pared en el tablero está representada por el valor `Integer.MIN_VALUE`.
- La función deberá devolver el máximo beneficio obtenido.

a) Define la **entrada**, la **salida** y la **semántica** de la función sobre la que estará basado el algoritmo de programación dinámica.

Beneficio(i,j): máximo beneficio obtenido para llegar a la casilla (N-1, N-1) partiendo de la casilla (i,j).

Entrada: i≥0,j≥0 (2 enteros). Los valores i=0, j=0 representan la casilla (0,0)

Salida: valor entero (indica el beneficio máximo)

b) Define recursivamente la función sobre la que estará basado el algoritmo de programación dinámica. Expresa la semántica de dicha función.

Caso base:
Beneficio(N-1, N-1) = tablero[N-1][N-1];

Caso Recursivo: (0≤f≤N-1 y 0≤c≤N-1)
Si tablero[f][c]= -∞: -> Beneficio(f, c) = -∞ (la celda actual es pared)

Si f=N-1 y 0≤c<N-1: (celdas de la última fila del tablero)
Si Beneficio(f, c+1) = -∞ -> Beneficio(f, c) = -∞
En otro caso -> Beneficio(f, c) = tablero[f][c]+ Beneficio(f, c+1)

Si 0≤f<N-1 y c=N-1: (celdas de la última columna del tablero)
Si Beneficio(f+1, c) = -∞ -> Beneficio(f,c) = -∞
En otro caso -> Beneficio(f, c) = tablero[f][c]+ Beneficio(f+1, c)

Si Beneficio(f+1, c)= Beneficio(f, c+1) = -∞: (el camino desde la celda inferior a la (N-1,N-1) y el camino desde la celda de la derecha a la (N-1,N-1) tienen ambos beneficio -∞)
-> Beneficio(f, c) = -∞

En otro caso: -> Beneficio(f, c) = tablero[f][c]+ max{ Beneficio(f+1, c), Beneficio(f, c+1)}

c) Basándote en el anterior apartado implementa un algoritmo en Java basado en programación dinámica que tenga complejidad[1] en tiempo O(N^2) y en memoria O(N).

[1] No cumplir con los requisitos de complejidad pedidos conlleva **una puntuación de 0** en este apartado.

```
int maxBeneficio(int[][]tablero){
    int N=tablero.length;
    int M=tablero[0].length;
    int[][] beneficio= new int[2][M];
    beneficio[(N-1)%2][M-1]=tablero[N-1][M-1];
    for(int c=M-2; c>=0; c--) { // celdas de la última fila
        if ((tablero[N-1][c] == Integer.MIN_VALUE)||
            (beneficio[(N-1)%2][c+1] == Integer.MIN_VALUE))
            beneficio[(N-1)%2][c] = Integer.MIN_VALUE;
        else
            beneficio[(N-1)%2][c] = tablero[N-1][c] + beneficio[(N-1)%2][c+1];
    }
    for(int f=N-2; f>=0; f--) {
        for (int c = M-1; c >= 0; c--) {
            if (tablero[f][c] == Integer.MIN_VALUE) // la celda actual es pared
                beneficio[f%2][c] = Integer.MIN_VALUE;
            else if (c==M-1){  // celda de la última columna
                if (beneficio[(f+1)%2][c]== Integer.MIN_VALUE)
                    beneficio[f%2][c] = Integer.MIN_VALUE;
                else beneficio[f%2][c] = tablero[f][c] + beneficio[(f+1)%2][c];
            }
            else if (beneficio[(f+1)%2][c]==Integer.MIN_VALUE &&
                     beneficio[f%2][c+1]==Integer.MIN_VALUE)
                beneficio[f%2][c] = Integer.MIN_VALUE;
            else
                beneficio[f%2][c] = tablero[f][c] +
                        Math.max(beneficio[(f+1)%2][c],beneficio[f%2][c+1]);
        }
    }
    return beneficio[0][0];
}
```