Ejercicio 4

Para saber si podemos recorrer todo un tablero de ajedrez de tamaño m x n desde una posición inicial concreta aplicaremos el mismo método utilizado que para encontrar la salida de un laberinto.
Si hemos llegado a la última casilla del tablero finalizamos la recursión y notificamos que el problema tiene solución. En este caso adicionalmente devolveremos todos los pasos que es necesario dar para llegar a la solución para lo que simplemente almacenamos todos los valores de las posiciones por las que la recursión se corta cuando hemos encontrado la solución.
En el caso de que la casilla no sea solucion deberemos evaluar si es válida para ser visitada y en caso de serlo visitarla. Permaneceremos en un nodo de la recursión hasta que hayamos visitado todas las casillas a las que desde él en el estado concreto de la recursión podamos ir.
Para las ramas de recursión que nazcan de una casilla dicha casilla habrá sido visitada no siendo así cuando todavia no se hayan pasado por ella.

```
//Almacena el resultado
var answer = [(Int, Int)]()
/**
 Función que indica si se puede recorrer un tablero del tamaño de la matriz visited empezando en la posisión (x,y)
 En caso de poderse recorrer nos proporcionará el camino realizado
 */
func chees (startx x : Int, starty y : Int, visited : inout [[Bool]], count : Int = 0) -> Bool{
    if (count >= max_count){//La casilla es valida y además la última
        answer.append((x,y))
        return true;//Indimacos que todas las casillas que llevaron a llegar a esta sean añadidas a la recursión, además esta finalizará
    } else {
        visited[x][y] = true //Marca la casilla actual como visitada para todas las llamadas recursivas a partir de esta rama
        for i in -2...2 {
            for j in -2...2 {
                if (((abs(i)==1)&&(abs(j)==2))||((abs(i)==2)&&(abs(j)==1)))//Es un movimiento de caballo valido
                    && ((y+j) >= 0) && ((x+i) >= 0) && ((x+i) < visited.count) && ((y+j) < visited.first!.count)//Dentro del tablero
                    && (!visited[x+i][y+j])//Hacia una casilla no visitada anteriormente
                    && chees (startx : x+i, starty : y+j, visited : &visited, count : count + 1) { //Que permite llegar a una solución
                        answer.append((x,y))
                        return true
                }
            }
        }
        visited[x][y] = false //Hace que para el resto de las llamadas recursivas que no estén en la rama puedan visitar la casilla
        return false
    }
}
```

Evaluar si la casilla es válida para ser visitada.

Ejercicio 6

Para resolver el problema se evalúan todas las posibles formas de transformar la cadena de entrada según la tabla de traducción. Para hacerlo se toma una pareja de caracteres y se realiza la traducción introduciendo esta de forma recursiva de nuevo a la función.
Se prueban por tanto todas las combinaciones de dos parejas de caracteres para traducir en cada nodo del árbol de recursión.
Si en algún momento la cadena de entrada coincide con el resultado deseado se interrumpe la recursión guardando el estado de todos los nodos que llevaron al resultado para indicar cómo llegar al objetivo.
Se determina que no es posible realizar la traducción cuando ya se han probado todas las combinaciones

Función principal

```
/**
 Función que indica si una cadena de caracteres es converitble en otra
 En caso de ser convertible nos proporcionará cómo realizar la conversión
 */
func convertir (_ cadena : String, en target : String) -> [String]?{
    if (cadena == target) { //CASO BASE: Se puede hacer la conversión
        return [String]()
    }
    var first_index = cadena.startIndex
    var second_index = cadena.index(after: first_index)
    while second_index != cadena.endIndex {
        var nueva_cadena = cadena //Duplicamos la cadena para no perder la original
        //Realizamos la conversión que toque según los índices de lectura
        nueva_cadena.replaceSubrange(first_index...second_index, with: sustitucion(cadena[first_index], cadena[second_index]))
        guard var acciones = convertir(nueva_cadena, en: target) else { //No se pudo realizar la conversión
            //Vamos al siguiente elemento
            first_index = second_index
            second_index = cadena.index(after: second_index)
            continue
        }
        acciones.append("\(cadena) -> \(nueva_cadena), " +
            "(\(cadena[first_index])\(cadena[second_index]) -> \(sustitucion(cadena[first_index], cadena[second_index])))")
        return acciones //La conversión se pudo realizar, devolvemos la lista de pasos necesarios para hacerla
    }
    return nil //CASO BASE: NO se puede hacer la conversión
}
```

En este caso se ha decidido duplicar la cadena de entrada y crear una nueva cadena con la traducción para cada nodo de la recursión.

Función auxiliar para realizar las traducciones

```swift
//Estas dos variables almacenan los datos a partir de los que las sustituciones se realizan
let matriz_sustitucion : [[Character]] = [["b","b","a","d"],
                                          ["c","a","d","a"],
                                          ["b","a","c","c"],
                                          ["d","c","d","b"]]
let index_of  : [Character:Int] = ["a":0,"b":1,"c":2,"d":3]

/**
 Nos permite converitir un conjunto de dos letras en una sola aplicando las reglas proporcionadas
 */
func sustitucion (_ letra1 : Character, _ letra2 : Character) -> String{
    return String(matriz_sustitucion[index_of[letra1]!][index_of[letra2]!])
}
```