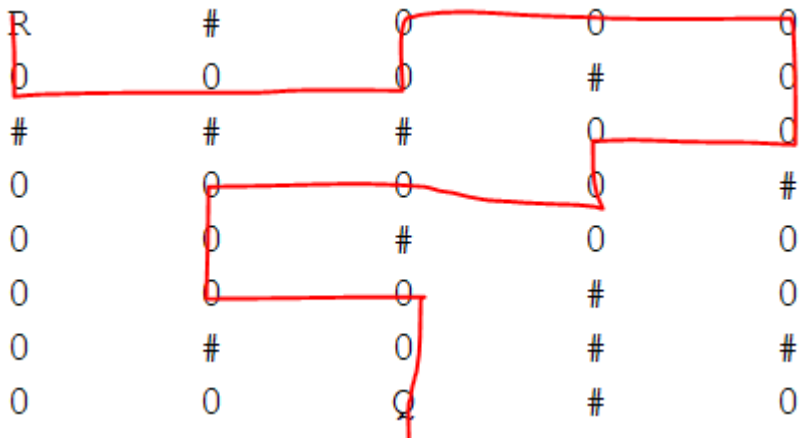
Encontrar un camino en un laberinto

Tal y como vamos a resolver este problema, tenemos que combinar los conceptos matrices, celdas adyacentes y recursividad.

Se trata de escribir un método recursivo que explora caminos en una matriz. El ejemplo clásico de laberinto es el de "el ratón y el queso". La matriz describe el laberinto, por ejemplo:

```
char[][] laberinto={
                    {'R','#','0','0','0'},
                    {'0','0','0','#','0'},
                    {'#','#','#','0','0'},
                    {'0','0','0','0','#'},
                    {'0','0','#','0','0'},
                    {'0','0','0','#','0'},
                    {'0','#','0','#','#'},
                    {'0','0','Q','#','0'},

   };
```

se trata de encontrar una solución (un camino) desde R(Ratón) hasta Q(queso), teniendo en cuenta que # es una pared y 0 representa una celda transitable. Ejemplo de una solución en el laberinto anterior(pero hay más soluciones):
(0,0),(1,0),(1,1)(1,2)(0,2)(0,3)(0,4)(1,4),(2,4)(2,3)(3,3)(4,3)(5,3)(5,2)(6,2)(7,2)



Vamos a escribir un método con la firma
boolean haySolucion(i,j)
que devuelve true si desde la celda[i][j] hay solución, es decir, se puede alcanzar el queso.

buscando "la frasecita"
Usar recursividad con claridad y elegancia no consiste sólo en invocar recursivamente a una función, ya que podemos hacer esto último para generar código que realmente es un "bucle encubierto". El resultado del bucle encubierto es un código ilegible con variables que se incrementan y decrementan como en un for java. Para evitar pensar "en modo bucle", como punto de partida para meterme en la esencia de la solución intento enunciar una frase que describa haySolucion() utilizando haySolución() en la propia descripción

"Desde una celda haySolución, si dicha celda contiene el queso o bien si puedo alcanzar desde ella el queso ya que es cierto que haySolucion por el NORTE,SUR,ESTE u OESTE"

metiendo ahora los índices en la frasecita

"Desde una celda i,j haySolución si la celda es el propio queso o bien si es true haySolucion(i-1,j) o haySolucion(i+1,j) o haySolucion(i,j+1) o haySolucion(i,j-1) "

Tengo dos tipos de celdas: terminal y transitable:

- Las celdas terminales o no transitables implican paradas de la recursividad y son el queso Q y la pared #.
- Las celdas transitables son celdas con 0.

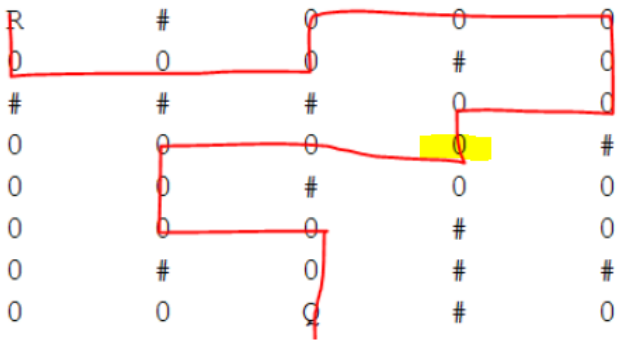
Un primer pseudo código

Caso base: estoy en una celda terminal
caso base 1: true si celda terminal es queso
caso base 2: false si celda terminal es pared
Caso recursivo: estoy en una celda transitable y
haySolucion(NORTE) || haySolución(SUR) ||haySolucion(ESTE) || haySolución(OESTE)

ahora con índices para acercarnos otro poco a la solución

actualmente estoy en el punto i,j del laberinto
Caso base: estoy en una celda terminal
caso base 1: return true si celda[i][j]=='Q'
caso base 2: return false si celda[i][j]=='#'
Caso recursivo: estoy en una celda transitable
haySolucion(i-1,j) || haySolución(i+1,j) ||haySolucion(i,j+1) || haySolución(i,j-1)

Suponemos por ejemplo que invocamos haySolucion(3,3) la celda[3][3] está marcada en amarilla en el dibujo. La función haySolucion(3,3) devuelve true ya que haySolucion() por el Oeste, o sea, haySolucion(i,j-1), o sea haySolucion(3,2) es true.



La definición anterior hay que tenerla clara, ya que es la esencia, pero además hay que tener en cuenta detalles extra muy importantes:

- Llamamos *camino* a una serie de celdas que se encadenan a través de llamadas recursivas, finalmente ese camino acabará en una llamada que ejecuta un caso base y devuelve true(solución) o false(sin solución). Si alcanzamos true a ese camino lo llamamos "camino solución" o simplemente "solución", si alcanzamos false se descarta ese camino y se intenta otro

- pueden llegar a la función haySolución() unos i,j fuera de rango y estos valores deben tratarse también como un caso base que detiene la recursividad y debe

devolver false ya que evidentemente ya sabemos que por ahí no hay camino solución

- en las llamadas recursivas puede haber ciclos, por ejemplo desde la celda ***a*** voy a la celda ***b,*** pero resulta que desde ***b*** se busca un camino por ***a*** y vuelta a empezar de ***a*** a ***b*** de ***b*** a ***a*** … y se queda colgado el algoritmo. Para evitar ciclos, hay que marcar las celdas por las que se pasa. Vamos a utilizar ". " para marcar una celda como visitada para no volver a pasar por ella. En el dibujo de arriba si estoy en [3][3] y previamente vengo [2][3], desde [3][3] tengo que evitar ir para atrás en el camino haciendo haySolucion(2,3)
- puede haber muchos caminos para llegar al queso, es decir, muchas soluciones, para simplificar nos quedamos con la primera que se encuentra. al encontrar la primera solución damos por terminado el algoritmo.

Las pueden tener por tanto los siguientes valores

- R es ratón, la primera celda a visitar obligatoriamente, aunque puede estar en cualquier celda del laberinto
- # es pared
- 0 es celda transitable
- . es celda visitada y parte del camino que se está explorando.

pseudocódigo más completo de hayCamino(fila,columna,laberinto)

```
//casos base

    si i y/o j fuera de rango return false
    si laberinto[fila][columna]=='#'   return false
    si laberinto[fila][columna]=='.'    return false (para evitar ciclos y por tanto stackoverflow)
    si laberinto[fila][columna]  es el queso (la meta) return true

//si no ocurre nada de lo anterior es que llegué a una celda laberinto[fila][columna] que vale '0' (o 'R' en el primer paso) y asigno a esta celda  el caracter  "." para marcarla como  visitada

    laberinto[i][j]='.'

//casos recursivos para seguir buscando subcaminos

//después de marcar i,j como visitada  tengo que intentar seguir completando el camino hasta llegar al queso y exploro recursivamente

    if(hayCamino(i-1,j,laberinto)) return true;//por el norte

     if(hayCamino(i+1,j,laberinto)) return true;//por el sur

     if(hayCamino(i,j+1,laberinto) ) return true;//por el oeste

      if(hayCamino(i,j-1,laberinto)) return true;// por el este

    // si todos los if anteriores tiene su condición false llego hasta aquí porque no se hizo ningún return y por lo tanto se que desde (i,j) es falso que haya solución.
        return false.
```

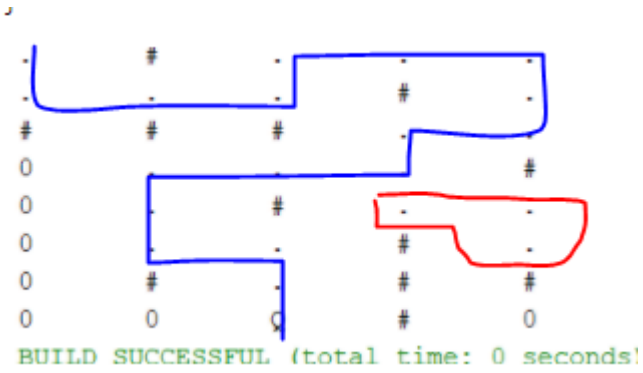
una implementación.

```
public class Laberinto {

   static boolean hayCamino(int i, int j, char[][] laberinto){

     if(i<0 || i>laberinto.length-1 || j<0 || j>laberinto[i].length -1 ) return false;

     if(laberinto[i][j]=='#')  return false;
     if(laberinto[i][j]=='.'){//ya fue visitado. evitar ciclos dentro de un camino
        return false;
     }
     if(laberinto[i][j]=='Q' ){
```

```
            return true;
        }

        //es '0'  o 'R' y desde aquí puede haber  camino
        laberinto[i][j]='.'; //marco como visitado y continuo la exploración recursiva
        //miramos caminos a través de sus posibles 4 vecinos

        if(hayCamino(i-1,j,laberinto)) return true; //vecino norte
        if(hayCamino(i+1,j,laberinto)) return true; //vecino sur
        if(hayCamino(i,j+1,laberinto) ) return true; //vecino oeste
        if(hayCamino(i,j-1,laberinto)) return true;//vecino este

        //si llego aquí es porque desde ningún vecino hay solución
        //y por tanto es falso que pueda haber solución desde (i,j)

         return false;
    }
    public static void main(String[] args) {

        char[][] laberinto={
            {'R','#','0','0','0'},
            {'0','0','0','#','0'},
            {'#','#','#','0','0'},
            {'0','0','0','0','#'},
            {'0','0','#','0','0'},
            {'0','0','0','#','0'},
            {'0','#','0','#','#'},
            {'0','0','Q','#','0'},

};
        if(!hayCamino(0,0, laberinto)){
            System.out.println("no hay solucion");
        }else{
             for(char[] x:laberinto){
                for(char c:x){
                    System.out.print(c+"\t");//un tab para mejorar presentación
                }
                 System.out.println("");
            }
        }
    }
}
```

```
.       #       .       .       .
.       .       .       #       .
#       #       #       .       .
0       .       .       .       #
0       .       #       .       .
0       .       .       #       .
0       #       .       #       #
0       0       Q       #       0
BUILD SUCCESSFUL (total time: 0 seconds)
```

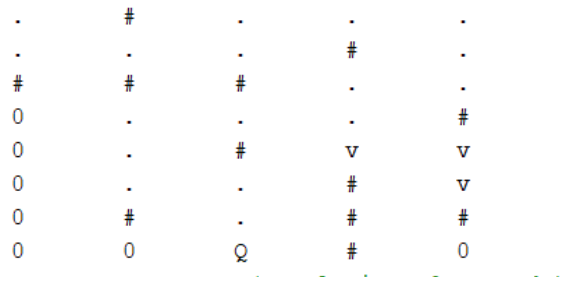
En la impresión de arriba, no se aprecia muy bien cuál es la solución … Observa entonces el siguiente gráfico en el que las celdas rodeadas en rojo fueron exploradas(visitadas) pero no forman parte del camino solución y la raya azul nos indica la solución encontrada.



pequeña mejora: marcar las celdas desde las que no se encontró solución

Cuando imprimo la matriz, para visualizar más claramente el camino solución puedo poner a 'v', aquellas celdas que previamente coloqué con valor '.' para explorar desde ellas pero finalmente no conseguí solución desde ellas

```
public class Laberinto {
   static boolean hayCamino(int i, int j, char[][] laberinto){

     if(i<0 || i>laberinto.length-1 || j<0 || j>laberinto[i].length -1 ) return false;

     if(laberinto[i][j]=='#')  return false;
     if(laberinto[i][j]=='.'){//ya fue visitado. evitar ciclos dentro de un camino
        return false;
     }
     if(laberinto[i][j]=='Q' ){

        return true;
     }

     //es '0'  o 'R' y desde aquí puede haber  camino
     laberinto[i][j]='.'; //marco como visitado y continuo la exploración recursiva
     //miramos caminos a través de sus posibles 4 vecinos

     if(hayCamino(i-1,j,laberinto)) return true; //vecino norte
     if(hayCamino(i+1,j,laberinto)) return true; //vecino sur
     if(hayCamino(i,j+1,laberinto) ) return true; //vecino oeste
     if(hayCamino(i,j-1,laberinto)) return true;//vecino este

      //si llego aquí es porque desde ningún vecino hay solución
      //y por tanto es falso que pueda haber solución desde (i,j)
      //la marco como 'v' de visitada para evitar ciclos
      //y observar  al imprimir matriz que se pasaron por puntos que no eran de camino solución
      laberinto[i][j]='v';
      return false;
  }
  public static void main(String[] args) {

     char[][] laberinto={
        {'R','#','0','0','0'},
        {'0','0','0','#','0'},
        {'#','#','#','0','0'},
        {'0','0','0','0','0','#'},
        {'0','0','#','0','0'},
        {'0','0','0','#','0'},
        {'0','#','0','#','#'},
        {'0','0','Q','#','0'},

};
     if(!hayCamino(0,0, laberinto)){
        System.out.println("no hay solucion");
     }else{
         for(char[] x:laberinto){
            for(char c:x){
                System.out.print(c+"\t");//un tab para mejorar presentación
            }
             System.out.println("");
        }
     }
  }
}
```



las celdas con 'v' tuvieron un '.' pero se les retiró al ver que desde ahí no había solución. Ahora además de las '.' hay que tener en cuenta las 'v' para evitar ciclos.

Por lo tanto:
'v'=> celda visitada desde la que no se pudo encontrar camino solución
'.' => celda también visitada pero que actualmente forma parte de un camino que puede tener solución ya que estamos explorando desde ahí. Ya se verá si hay solución o no desde aquí

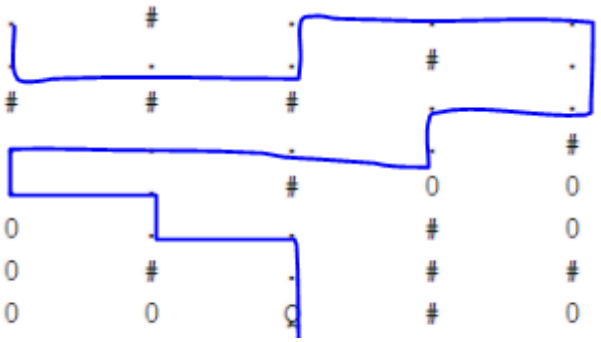
Vamos a razonar porque la celda (4,3) es 'v'

cuando estamos en (4,3) se intentan exploraciones por NORTE,SUR,ESTE y OESTE. fíjate sobre todo ahora en el primero que se intenta que es NORTE. Tendrá arriba un '.' y por tanto por ahí devolverá false ya que '.' indica que por ahí pasamos y estamos explorando y tenemos que evitar ciclos. Por SUR,ESTE y OESTE tarde o temprano encontrará paredes y también obtendrá un false.

El orden de las reglas importa
Un laberinto puede tener muchas soluciones. Cuando hay muchas soluciones y sólo nos quedamos con la primera que encontramos, la solución que se va a obtener depende del orden de las reglas recursivas. Por ejemplo, en nuestro laberinto, con el siguiente orden obtengo otra solución

```
if(hayCamino(i-1,j,laberinto)) return true; //vecino norte
if(hayCamino(i,j+1,laberinto) ) return true; //vecino oeste
if(hayCamino(i,j-1,laberinto)) return true;//vecino este
if(hayCamino(i+1,j,laberinto)) return true; //vecino sur
```

obtenemos la siguiente solución



Observamos que la solución es otro camino, y además que no se generó ninguna ‘v’

El algoritmo anterior simplemente encuentra una solución, pero puede haber varias y es posible que sean mejores(más cortas). Nuestro algoritmo es sencillo y no encuentra la mejor solución, simplemente encuentra una solución.

Observando paso a paso el progreso de la obtención de la solución

La matriz se modifica en dos sitios:
después de asignar a una celda un ‘.’ por considerarla parte de posible solución

```
laberinto[i][j]='.';
```

y al observar que no hay solución desde una celda que habíamos calificado como ‘.’ marcándola con ‘v’.

```
laberinto[i][j]='v';
```

Vamos a añadir un método auxiliar imprimirLab() para invocarlo después de estas dos instrucciones

Para que en el siguiente ejemplo haya celdas con ‘v’ cambiamos el orden de las reglas de nuevo.

Ejecuta el siguiente código y observa cómo en principio el ratón intenta alcanzar una solución avanzando por el oeste y como no la encuentra, la rata “vuelve hacia atrás” sustituyendo los ‘.’ por ‘v’ para no volver a pasar por ahí. Por tanto:

- ‘v’ signfica visitada sin solución
- ‘.’ significa simplemente visitada y puede mutar a ‘v’ si no tiene solución

```
public class Laberinto {

   static void imprimirLab(char[][] laberinto){
      System.out.println("---------------------------------");
      for(char[] x:laberinto){
            for(char c:x){
               System.out.print(c+"\t");
            }
             System.out.println("");
         }
   }


   static boolean hayCamino(int i, int j, char[][] laberinto){

      if(i<0 || i>laberinto.length-1 || j<0 || j>laberinto[i].length -1 ) return false;

      if(laberinto[i][j]=='#')  return false;
      if(laberinto[i][j]=='.'){//ya fue visitado. evitar ciclos dentro de un camino
         return false;
      }
      if(laberinto[i][j]=='Q' ){

         return true;
```

```
        }

        //es '0'  o 'R' y desde aquí puede haber  camino
        laberinto[i][j]='.'; //marco como visitado y continuo la exploración recursiva
        System.out.println("un pasito palante");
        imprimirLab(laberinto);

        //miramos caminos a través de sus posibles 4 vecinos

        if(hayCamino(i-1,j,laberinto)) return true; //vecino norte
        if(hayCamino(i,j+1,laberinto) ) return true; //vecino oeste
        if(hayCamino(i+1,j,laberinto)) return true; //vecino sur
        if(hayCamino(i,j-1,laberinto)) return true;//vecino este




         //si llego aquí es porque desde ningún vecino hay solución
        //y por tanto es falso que pueda haber solución desde (i,j)
        //la marco como 'v' de visitada para evitar ciclos
        //y observar  al imprimir matriz que se pasaron por puntos que no eran de camino solución
        laberinto[i][j]='v';
        System.out.println("Un  pasito patrás: mal camino, dejo marca de visitada");
        imprimirLab(laberinto);
        return false;

    }
    public static void main(String[] args) {

        char[][] laberinto={
            {'R','#','0','0','0'},
            {'0','0','0','#','0'},
            {'#','#','#','0','0'},
            {'0','0','0','0','#'},
            {'0','0','#','0','0'},
            {'0','0','0','#','0'},
            {'0','#','0','#','#'},
            {'0','0','Q','#','0'},

};
         //suponemos que el ratón está en 0,0
         if(!hayCamino(0,0,laberinto)){
            System.out.println("no hay solución");
        }else{
            System.out.println("iHay solución!");
        }

    }
}
```

Imprimir con colores

Nada importante, es simplemente para mejorar la visualización del paso a paso
Se puede configurar el color de impresión enviando como primer caracter al print() un caracter unicode que códifica el color deseado. Prueba el siguiente código para aclarar esto. Si buscas por internet podrás consultar un listado completo de colores. La constante RESET se refiere a que el print() utilice el color por defecto que tiene configurada la consola en que estoy trabajando, normalmente, el blanco.

```
public class Unidad4 {
    public static void main(String[] args) {
        final String GREEN = "\u001B[32m";
        final String RESET = "\u001B[0m";
        final String WHITE = "\u001B[37m";
        final String RED = "\u001B[31m";
        final String YELLOW = "\u001B[33m";
        final String BLUE = "\u001B[34m";
        final String PURPLE = "\u001B[35m";
        final String CYAN = "\u001B[36m";

        System.out.println(GREEN+"verde");
```

```
        System.out.println(RESET+"por defecto");
        System.out.println(RED+"rojo");
        System.out.println(RESET+"por defecto");
        System.out.println(WHITE+"aquí blanco es por defecto");
        System.out.println(YELLOW+"Amarillo");
        System.out.println(BLUE+"Azul");

    }

}
```

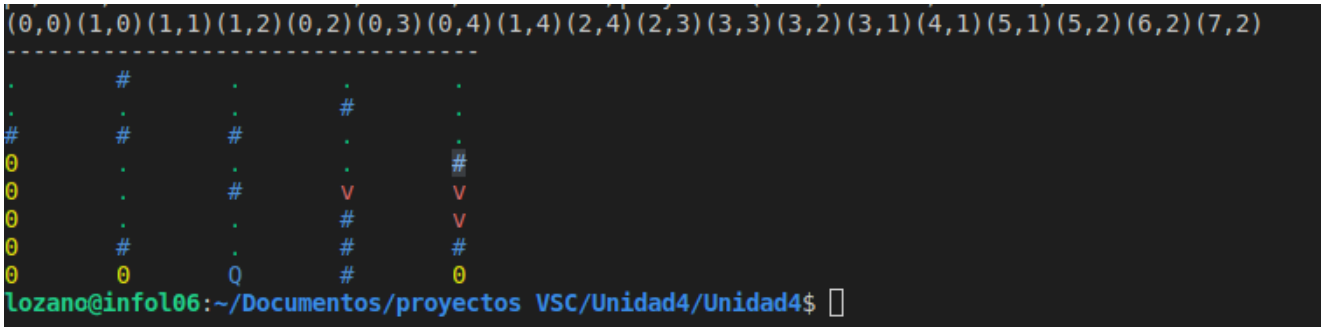
Entonces con un nuevo imprimirLab() con colores

```
public class Laberinto {

   static void imprimirLab(char[][] laberinto){
      final String GREEN = "\u001B[32m";
      final String RESET = "\u001B[0m";
      final String WHITE = "\u001B[37m";
      final String RED = "\u001B[31m";
      final String BLUE = "\u001B[34m";
      final String YELLOW = "\u001B[33m";


      System.out.println("-----------------------------------");
      for(char[] x:laberinto){
            for(char c:x){
               switch(c){
                  case '.': System.out.print(GREEN+c+"\t");break;
                  case 'v': System.out.print(RED+c+"\t");break;
                  case '0': System.out.print(YELLOW+c+"\t");break;
                  default: System.out.print(BLUE+c+"\t");
               }
            }
            System.out.println(RESET+"");
         }
   }




   static boolean hayCamino(int i, int j, char[][] laberinto){

      if(i<0 || i>laberinto.length-1 || j<0 || j>laberinto[i].length -1 ) return false;

      if(laberinto[i][j]=='#')  return false;
      if(laberinto[i][j]=='.'){//ya fue visitado. evitar ciclos dentro de un camino
         return false;
      }
      if(laberinto[i][j]=='Q' ){

         return true;
      }

      //es '0'  o 'R' y desde aquí puede haber  camino
      laberinto[i][j]='.'; //marco como visitado y continuo la exploración recursiva
      System.out.println("un pasito palante");
      imprimirLab(laberinto);

      //miramos caminos a través de sus posibles 4 vecinos

      if(hayCamino(i-1,j,laberinto)) return true; //vecino norte
      if(hayCamino(i,j+1,laberinto) ) return true; //vecino oeste
      if(hayCamino(i+1,j,laberinto)) return true; //vecino sur
      if(hayCamino(i,j-1,laberinto)) return true;//vecino este
```

```
        //si llego aquí es porque desde ningún vecino hay solución
        //y por tanto es falso que pueda haber solución desde (i,j)
        //la marco como 'v' de visitada para evitar ciclos
        //y observar  al imprimir matriz que se pasaron por puntos que no eran de camino solución
        laberinto[i][j]='v';
        System.out.println("Un  pasito patrás: mal camino, dejo marca de visitada");
        imprimirLab(laberinto);
        return false;

    }
    public static void main(String[] args) {

        char[][] laberinto={
            {'R','#','0','0','0'},
            {'0','0','0','#','0'},
            {'#','#','#','0','0'},
            {'0','0','0','0','#'},
            {'0','0','#','0','0'},
            {'0','0','0','#','0'},
            {'0','#','0','#','#'},
            {'0','0','Q','#','0'},

};
        //suponemos que el ratón está en 0,0
        if(!hayCamino(0,0,laberinto)){
            System.out.println("no hay solución");
        }else{
            System.out.println("¡Hay solución!");
        }

    }
}
```

Ejercicio U4_B4E_E1: Ahora queremos que el método recursivo nos devuelva un string que contenga las coordenadas del camino, de forma que en lugar de devolver true o false devuelva un String. Si no tuviera solución devolvería el String vacío "".

Por ejemplo, con el laberinto que venimos usando hasta ahora, y con el orden de reglas NORTE,SUR, OESTE, ESTE:



Se trata de ajustar un poco el código anterior, el algoritmo es el mismo. En tu solución, no añadas variables static(que desvirtúan la naturaleza recursiva) ni cambies el número de parámetros. Utiliza variables locales y haz que que la función recursiva devuelva

String. Observa que no es una solución sobre la solución ya hecha obtener la matriz y recorrer los ".". Los datos coinciden pero no es la solución que se pide.

Ahora el main será el siguiente

```
public static void main(String[] args) {

    char[][] laberinto = {
            { 'R', '#', '0', '0', '0' },
            { '0', '0', '0', '#', '0' },
            { '#', '#', '#', '0', '0' },
            { '0', '0', '0', '0', '#' },
            { '0', '0', '#', '0', '0' },
            { '0', '0', '0', '#', '0' },
            { '0', '#', '0', '#', '#' },
            { '0', '0', 'Q', '#', '0' },

    };
    // suponemos que el ratón esá en 0,0
    System.out.println(hayCamino(0, 0, laberinto));
    imprimirLab(laberinto);

}
```

Limitaciones del algoritmo estudiado.
Se contenta con una solución y no tiene porque ser la mejor, es simplemente una solución y se llegó a esa condicionada por el orden de las reglas recursivas. Si quisiéramos encontrar por ejemplo todas las posibles soluciones para por ejemplo quedarnos con la mejor, la más corta, no vale este algoritmo.

Ejercicio U4_B4E_E2: Escapando de las fuerzas imperiales (CodeRunner)