# 1. Componentes básicas del juego

El Sudoku Hidato es un tablero con varios números ubicados en él y algunas casillas en blanco. El menor número siempre será 1 y el mayor coincidirá con la cantidad de casillas del tablero, ambos siempre estarán presentes en el tablero. El objetivo es rellenarlo con números consecutivos que se conectan horizontal, vertical o diagonalmente, formando así una secuencia de números que empieza con 1 que están inmediatamente conectados. Para cada número, su antecesor y su sucesor estará en alguna casilla adyacente.

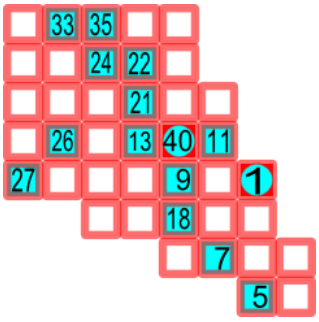
Se representa el tablero como un tipo compuesto de 3 elementos: una lista compuesta por listas para representar las casillas, el menor número del tablero y el mayor. Las casillas están representadas con números enteros, usando la clase `Integer` de Haskell. La definición del tablero es:

```haskell
data Board = Board {
    cells :: [[Integer]],
    minNum :: Integer,
    maxNum :: Integer
} | Empty deriving (Show, Eq)
```

Figura 1: Definición de un tablero de Hidato

Donde `Empty` es un valor especial que se usa para representar cuando un tablero es inválido. A pesar de que se tiene el conocimiento de que existen otras estructuras en Haskell para representar la ausencia de valor, como `Maybe`, que tiene `Just a` para representar un valor del tipo genérico `a` y `Nothing` para la ausencia de este, se decidió modificar la definición `Board` con el objetivo de ganar en simplicidad y flexibilidad con los métodos definidos. Así, por ejemplo un tablero sabe como dibujarse incluso cuando toma el valor de `Empty`.

Asumiendo que los números usados en el juego son naturales mayores que 1 se usa el 0 para representar las casillas que están vacías. Como un tablero no es necesariamente rectangular se usa además -1 para representar las casillas en las que no se pueden poner ningún número. Por ejemplo, en la Figura 2 se muestra un tablero de Hidato y como este se representaría en el código Haskell.




```haskell
Board [
    [0, 33, 35,  0,  0],
    [0,  0, 24, 22,  0],
    [0,  0,  0, 21,  0, 0],
    [0, 26,  0, 13, 40, 11],
    [27,  0,  0,  0,  9,  0, 1],
    [-1, -1,  0,  0, 18,  0, 0],
    [-1, -1, -1, -1,  0,  7, 0, 0],
    [-1, -1, -1, -1, -1, -1, 5, 0]]
    1 40
```

Figura 2: Un tablero de Hidato y su representación en Haskell

# 2. Resolver Hidatos

Uno de los objetivos principales del proyecto es, teniendo como entrada un Hidato de cualquier forma, con algunas casillas anotadas y el menor y el mayor número señalados, ejecutar un programa que permitirá dar solución al tablero.

La función principal para dar solución a un Hidato se llama `solve` y recibe como parámetro un `Board` y como salida imprime el tablero solucionado.

La estrategia llevada a cabo para la generación de la solución del tablero fue un procedimiento recursivo, que a pesar de poder ser ineficiente garantiza la solución del mismo.

Se comienza buscando la posición del menor número, a partir de él se comenzará a generar la secuencia de números que se posicionarán en el tablero hasta llegar a poner el último. Se llega a la solución cuando

todos los espacios vacíos son ubicados. Para comprobar esta condición se calcula inicialmente la cantidad de casillas vacías, y para cada nueva asignación esta cantidad se disminuye en 1. Cuando la cantidad de casillas vacías sea 0 entonces todas los números fueron puestos en el tablero y el método termina su ejecución.

Para cada número se analiza cada posición que puede tomar, se comprueban si se pueden poner arriba, abajo, a la derecha, a la izquierda o en alguna de las diagonales del número que fue su antecesor y se busca una solución en profundidad después de tomar una decisión con respecto a la ubicación del número. En el momento en que se encuentre una solución válida se dejan de comprobar otros caminos. A su vez, se intenta determinar lo más pronto posible cuando una solución deja de ser válida.

En el momento en que se descubre que una solución no es correcta se retorna el valor especial `Empty`. Las situaciones en las que se invalida un tablero son:

- Cuando se pone un número que no es el último en el tablero y todas las casillas alrededor están ocupadas. En este caso no es posible poner el siguiente número, lo que imposibilita la continuidad de la partida.
- Al ponerse un número se descubre que su sucesor está en el tablero y no está en ninguna de sus casillas adyacentes.

El método recursivo recibe además del tablero otros argumentos como un número y una posición, con el objetivo de que en ese llamado el número se ponga en la posición propuesta. Luego, el algoritmo general para resolver el Hidato es:

1. Si se llenan todas las casillas vacías, entonces ya se tiene una solución válida.
2. Si no se invalida la solución por uno de los casos expuestos anteriormente, entonces se continúa con la generación del tablero resuelto.
3. Se pone el número actual en el tablero en la posición prefijada.
4. Se determina cuál es el siguiente número. Si el sucesor del último número no está en el tablero entonces este es el siguiente número a ubicar. En otro caso, se busca cual es el siguiente número mayor que el actual que no esté todavía ubicado.
5. Se determina la posición del antecesor del número que se decidió poner.
6. Se comprueban las casillas adyacentes a esa posición en las que se puede poner un número (aquellas que no estén ocupadas o estén invalidadas por estar fuera del rango del tablero) y se genera una solución poniendo el número elegido en esta posición.
7. Este procedimiento se repite para todos los números hasta tener el tablero completo.

## 3. Generación de Hidatos

Otro de los objetivos del proyecto es crear un programa en Haskell que genere Hidatos, de varias formas, que tendrán solución para ser utilizados por el programa del objetivo anterior.

La función principal para generar un Hidato se llama `generate` y recibe como parámetro dos números `n` y `m` que serán las dimensiones del tablero. Como salida se genera devuelve un `Boarl`

Este algoritmo se lleva a cabo de manera cuasi-aleatoria.

En un inicio se definen tres formas principales:

- `cloudBoard`, que construye cuadrados cuya arista va disminuyendo desde la esquina superior izquierda hasta la esquina inferior derecha (se tomó como guía el Hidato brindado en el ejemplo, que sería un `cloudBoard` de dimensiones de 8x8)
- `mirrorBoard`, que deja disponibles aquellas casillas que no están en la diagonal principal.
- `squareBoard`, que es la forma estándar de un cuadrado o rectángulo.

La forma que toma el tablero se decide de forma aleatoria usando la función `randomRIO`, que también se usa para decidir de la misma manera la posición que toma el primer valor a insertar. Se genera un tablero en blanco (todas las casillas en 0) con las dimensiones deseadas, el número de casillas determina el mayor valor a colocar en el tablero.

Con estas condiciones comienza el algoritmo para completar el tablero, este recibe un tablero vacío y comenzando por la posición inicial, recursivamente toma cada casilla y recorre sus adyacentes decidiendo donde colocar el próximo número, chequea en profundidad si la decisión es factible y en caso contrario, toma otra decisión. El algoritmo se detiene antes de lo debido en caso de que se ponga un número que no es el último en el tablero y todas las casillas alrededor están ocupadas. En este caso, no es posible poner el siguiente número, y se devuelve el valor `Empty`, invalidando el tablero actual. Este algoritmo es bastante similar al explicado en la sección 2

Uno de los principales problemas en la ejecución de este algoritmo, es que resultaba muy lento, ya que colocar todos los números en el tablero le añade complejidad. A diferencia de lo que pasaba en la sección 2, no existían números que sirvieran para guiar la validez de la solución actual. Por lo tanto, una nueva poda fue realizada. Una de las cosas que se notó es que cuando se está resolviendo un tablero si los ceros dejan de estar conectados como se muestra en la Figura 3 entonces no será posible completar el tablero, ya que una parte de este quedará inaccesible. Para determinar cuando esto ocurre se calculan a cuantos ceros se pueden llegar desde una posición determinada en el tablero. Si la cantidad de ceros es igual a la cantidad de casillas vacías que quedan, entonces la solución es válida, en otro caso todos los ceros no están accesibles y la solución se invalida retornando `Empty`.

```
0 0 0 0
4 3 2 1
0 0 0 0 0
0 0 0 0 0
    0 0 0 0
      0 0
```

Figura 3: Un ejemplo de una solución inválida porque los ceros están desconectados

Esta poda puede resultar compleja de analizar (en el peor caso se recorren todas las casillas del tablero), por lo que una condición adicional es chequeada analizando solamente las casillas adyacentes a la casilla actual. Si alguna casilla adyacente deja de ser alcanzable por alguna otra adyacente (comprobando solo las casillas adyacentes), entonces es cuando único es posible que ocupando la casilla actual ocurra una desconexión. Solo en ese caso se comprueban las restantes casillas del tablero para comprobar si es cierto que ya no son alcanzables.

Si se ejecuta de forma exitosa el método, el tablero tiene todas las casillas llenas, lo que constituye una solución. En este punto se "eliminan" (se reemplazan con 0) algunas casillas en el tablero. La idea principal del algoritmo consiste en crear una lista de valores aleatorios entre 0 y la cantidad de casillas del tablero (sin contar las que contienen al primer y último valor), que se hace corresponder con la lista de dichas casillas para asignarle un "índice" a cada una, así, al ordenar la primera lista, la segunda queda en orden aleatorio. Esta lista resultante entra en un algoritmo iterativo que va reemplazando cada casilla con 0. El tablero final tiene n casillas de las cuales (n/2) están vacías.

## 4. Inicio del juego

Para iniciar el juego se debe abrir el compilador de haskell (en nuestro caso se usó ghci) en el directorio del proyecto y ejecutar `:load main`, ya que `main.hs` es el archivo principal. Este archivo importa los métodos `generate` y `solve` que fueron mencionados anteriormente.

Para llamar a `generate` se hace: `board <- generate n m`, donde n y m son 2 números cualesquiera que representan las dimensiones del tablero. Como se trabaja con números de `randomRIO`, el verdadero valor de retorno de este método es `Board IO`, por lo que es necesario guardar el valor en una variable antes de su ejecución para acceder al valor de `Board`.

Luego, una vez se tiene el board se ejecuta: `solve board`, siendo `board` el resultado obtenido mediante la ejecución del generador.

Este procedimiento se puede ejecutar repetidas veces.

La complejidad del algoritmo depende no solo de las dimensiones del tablero, sino de otros parámetros como la ubicación del menor número del tablero y lo cercanos que estén los números que se encuentran inicialmente en el tablero. Sin embargo, para dimensiones menores que 7 se ejecuta bastante bien.