Sudoku Solver

Hoje vamos tentar utilizar recursão para resolver um jogo de sudoku.

Caso não conheça sobre veja aqui: `https://pt.wikipedia.org/wiki/Sudoku`

Basicamente, o jogo consiste em preencher um tabuleiro 9 x 9 com os números de 1 a 9 de tal forma que não haja números repetidos em cada linha, coluna e grupos 3 x 3.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Exibindo nosso tabuleiro na tela:

Receberemos como entrada uma matriz 9 x 9, com o seguinte formato:

```
let board = [[5, 3, 0, 0, 7, 0, 0, 0, 0],
             [6, 0, 0, 1, 9, 5, 0, 0, 0],
             [0, 9, 8, 0, 0, 0, 0, 6, 0],
             [8, 0, 0, 0, 6, 0, 0, 0, 3],
             [4, 0, 0, 8, 0, 3, 0, 0, 1],
             [7, 0, 0, 0, 2, 0, 0, 0, 6],
             [0, 6, 0, 0, 0, 0, 2, 8, 0],
             [0, 0, 0, 4, 1, 9, 0, 0, 5],
             [0, 0, 0, 0, 8, 0, 0, 7, 9]];
```

(Lembrando que 0 representa uma casa vazia)

Começaremos criando uma função para exibir nosso tabuleiro na tela, basicamente, iremos iterar sobre cada linha do tabuleiro, adicionaremos cada número da linha em uma string e ao final do último iremos exibir a string:

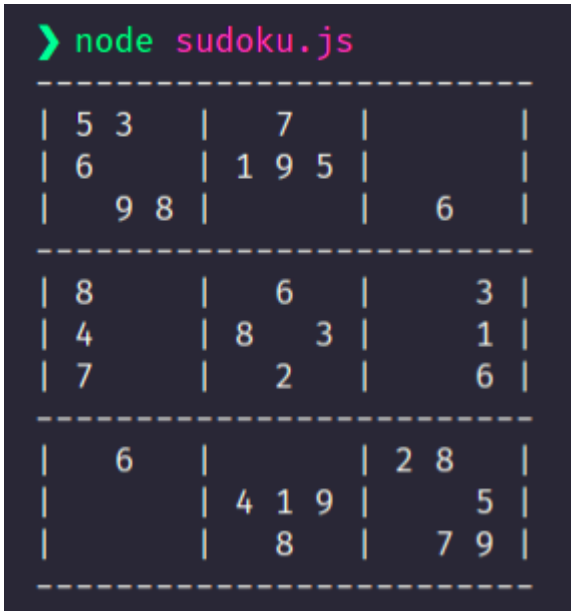
```
function showBoard(board) {
  for (let row = 0; row < board.length; row++) {
    let line = '';
    for (let column = 0; column < board[row].length; column++) {
      line += board[row][column] + ' ';
    }

    console.log(line);
  }
}
```

Certo, agora conseguimos imprimir nosso tabuleiro de forma mais decente no console.

Adicionei uma outra forma para deixar ainda melhor:

```
function showBoard(board) {
  console.log("-------------------------");
  for (let row = 0; row < board.length; row++) {
    let line = "";
    for (let column = 0; column < board[row].length; column++) {
      let space = board[row][column] === 0 ? " " : board[row][column];

      if (column % 3 === 0) line += "| " + space + " ";
      else if (column === 8) line += space + " |";
      else {
        line += space + " ";
      }
    }

    console.log(line);
    if (row === 2 || row === 5) {
      console.log("-------------------------");
    }
  }
  console.log("-------------------------");
}
```

Rodando ficará assim:



Encontrando uma casa vazia

Precisamos agora buscar em nosso tabuleiro por valores iguais a zero, isso significa que buscaremos por casas vazias, para que assim possamos tentar preenchê-las. A solução assim como a função anterior será bastante simples: Basicamente iteraremos por cada linha e cada coluna, caso o valor presente na linha x coluna atual seja igual a 0, retornaremos um array no formato `[linha, coluna]` ou seja, retornaremos a posição da casa vazia, para que assim possamos ter o endereço de onde preenchê-la.

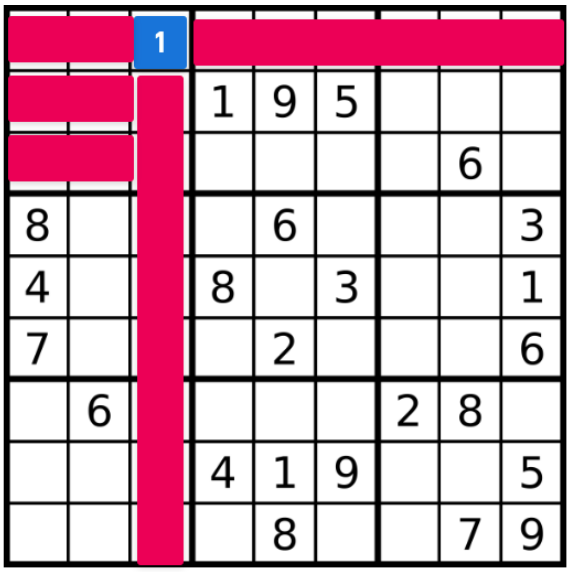
```
function findZero(board) {
  for (let row = 0; row < board.length; row++) {
    for (let column = 0; column < board[row].length; column++) {
      if (board[row][column] === 0) return [row, column];
    }
  }
}
```

Preenchendo corretamente

Chegamos agora na parte mais difícil da nossa implementação (eu achei kk). Agora que temos como saber qual a casa que está vazia no tabuleiro (com a ajuda da função `findZero`) precisamos preenchê-la com um número de 1 a 9, porém, precisamos encontrar uma forma de validar esse número. A validação

consistirá basicamente em verificar se não há nenhum número igual ao número que estamos inserindo na mesma linha, coluna ou grupo 3 x 3.

Por exemplo, supondo que para o 0 na posição `[0, 2]` queiramos colocar o número 1, para isso devemos verificar se existe o número 1 no conjunto das casas em vermelho:



Para isso há algumas maneiras, eu mostrarei agora a minha implementação, porém fique a vontade para melhorar o código que está na versão inicial.

Primeiro, criaremos uma função chamada `isValid` , essa função receberá como parâmetros o tabuleiro, a linha e a coluna onde gostaríamos de inserir um número, e por último o número que gostaríamos de inserir. A função deve apenas retornar se é possível a inserção apenas com `true` ou `false`

- Validação linha x coluna:

  Para validar se há algum número na mesma linha ou coluna é bem fácil, nossa função já recebe a linha e coluna desejada, então tudo que faremos é iterar por 9 vezes, e verificar:

  - Se para a mesma linha há um número igual ao desejado na coluna 1 a 9
  - Se para a mesma coluna há um número igual ao desejado na linha 1 a 9

```
function isValid(board, row, column, number) {
  for (let index = 0; index < board.length; index++) {
    if (board[row][index] === number || board[index][column] === number) {
```

```
      return false;
    }
  }
}
```

Essa parte foi fácil não?

Agora vem a parte em que eu tive realmente que pensar um pouco, minha solução me pareceu bem inocente, então sinta-se a vontade para deixar isso mais eficiente. Vamos lá

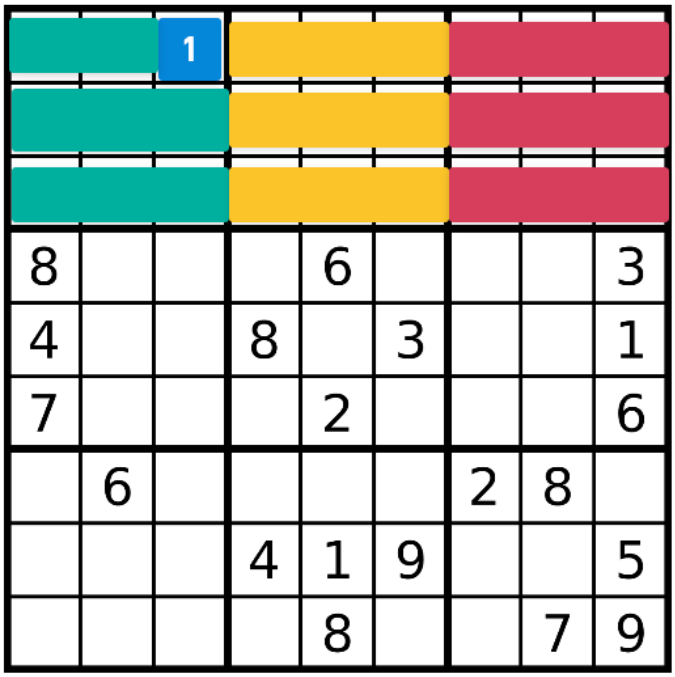
- Validação para grupo 3 x 3

  Para essa validação, inicialmente checaremos o intervalo em que a linha está inserida, se está entre as três primeiras, entre a quarta e a sexta, ou entre a sétima e a nona linha. Da mesma forma com as colunas. (Parece uma solução burra para você? Para mim também 😅)

  Mostrarei aqui como ficaria a verificação para os grupos 3×3 presentes nas três primeiras linhas:

```
function find3x3Group(board, startRow, startColumn) {
  if (startRow <= 2) {
    if (startColumn <= 2) {
      // verificar grupo no intervalo verde
    }

    else if (startColumn >= 2 and startColumn <= 5) {
      // verificar grupo no intervalo amarelo
    }

    else if (startColumn >= 6 and startColumn <= 8) {
      // verificar grupo no intervalo vermelho
    }
  }
}
```

  Ou seja nessa função até o momento estamos fazendo a busca neste intervalo:



Uma vez dentro de um intervalo 3×3, utilizaremos uma função para iterar sobre o intervalo e verificar se o número que queremos adicionar está contido nesse intervalo:

```
function verifyGrid(board, startRow, endRow, startColumn, endColumn) {
  let gridNumbers = [];
  for (let row = startRow; row < endRow; row++) {
    let gridLine = '';
    for (let column = startColumn; column < endColumn; column++) {
      gridLine += board[row][column] + ' ';
      gridNumbers.push(board[row][column]);
    }
    // console.log(gridLine);
  }

  return gridNumbers;
}
```

Agora utilizaremos nossa função `verifyGrid` em nossas verificações de intervalos:

```
function find3x3Group(board, startRow, startColumn) {
  if (startRow <= 2) {
    if (startColumn <= 2) {
      return verifyGrid(board, 0, 3, 0, 3);
```

```
    }

    else if (startColumn >= 2 and startColumn <= 5) {
      return verifyGrid(board, 0, 3, 3, 6);
    }

    else if (startColumn >= 6 and startColumn <= 8) {
      return verifyGrid(board, 0, 3, 6, 9);
    }
  }
}
```

No entanto, note que esse código só está verificando para as primeiras três linhas, vamos adicionar condições para as outras:

```
function find3x3Group(board, startRow, startColumn) {
  if (startRow <= 2) {
    if (startColumn <= 2) {
      return verifyGrid(board, 0, 3, 0, 3);
    }

    else if (startColumn >= 2 && startColumn <= 5) {
      return verifyGrid(board, 0, 3, 3, 6);
    }

    else if (startColumn >= 6 && startColumn <= 8) {
      return verifyGrid(board, 0, 3, 6, 9);
    }
  }

  else if (startRow >= 3 && startRow <= 5) {
    if (startColumn <= 2) {
      return verifyGrid(board, 3, 6, 0, 3);
    }

    else if (startColumn >= 2 && startColumn <= 5) {
      return verifyGrid(board, 3, 6, 3, 6);
    }

    else if (startColumn >= 6 && startColumn <= 8) {
      return verifyGrid(board, 3, 6, 6, 9);
    }
  }

  else if (startRow >= 6 && startRow <= 8) {
    if (startColumn <= 2) {
      return verifyGrid(board, 6, 9, 0, 3);
    }

    else if (startColumn >= 2 && startColumn <= 5) {
      return verifyGrid(board, 6, 9, 3, 6);
    }
```

```
    else if (startColumn >= 6 && startColumn <= 8) {
      return verifyGrid(board, 6, 9, 6, 9);
    }
  }
}
```

Eu sei eu sei, mas como eu estava para o crime neste dia eu resolvi desonrar ainda mais meus ancestrais, e adicionei um ternário lazarento, desculpem:

```
function find3x3Group(board, startRow, startColumn) {
  if (startRow <= 2) {
    return startColumn <= 2 ?
           verifyGrid(board, 0, 3, 0, 3) :
           (startColumn >= 2 && startColumn <= 5) ?
           verifyGrid(board, 0, 3, 3, 6) :
           verifyGrid(board, 0, 3, 6, 9);
  }

  else if (startRow >= 3 && startRow <= 5) {
    return startColumn <= 2 ?
           verifyGrid(board, 3, 6, 0, 3) :
           (startColumn >= 2 && startColumn <= 5) ?
           verifyGrid(board, 3, 6, 3, 6) :
           verifyGrid(board, 3, 6, 6, 9);
  }

  else if (startRow >= 6 && startRow <= 8) {
    return startColumn <= 2 ?
           verifyGrid(board, 6, 9, 0, 3) :
           (startColumn >= 2 && startColumn <= 5) ?
           verifyGrid(board, 6, 9, 3, 6) :
           verifyGrid(board, 6, 9, 6, 9);
  }
}
```

Agora adicionamos essa verificação na função `isValid` :

```
function isValid(board, row, column, number) {
  const currentGrid = find3x3Group(board, row, column);
  if (currentGrid.includes(number)) return false;

  for (let index = 0; index < board.length; index++) {
    if (board[row][index] === number || board[index][column] === number) {
      return false;
    }
```

```
    }

    return true;
  }
```

Com isso, finalizamos a etapa de validação.

Resolvendo sudoku (com recursão ^^)

Agora finalmente utilizaremos todas as funções anteriores para resolver nosso tabuleiro sudoku.

▼ Encontrando uma casa vazia

A primeira coisa a se fazer é percorrer nosso tabuleiro em busca de uma casa vazia, caso encontremos, vamos tentar preencher com um número de 1 a 9 e verificar se é uma jogada válida. Caso não haja nenhuma casa vazia, podemos supor que o tabuleiro está completo, portanto não há o que fazer e só resta retornar o próprio tabuleiro:

```
function solveSudoku(board) {
  let freePosition = findZero(board);

  if (freePosition === undefined) return board;

  let freeRow = freePosition[0];
  let freeColumn = freePosition[1];


}
```

Até o momento verificamos se há uma cada vazia e uma vez que haja, buscamos sua linha e coluna.

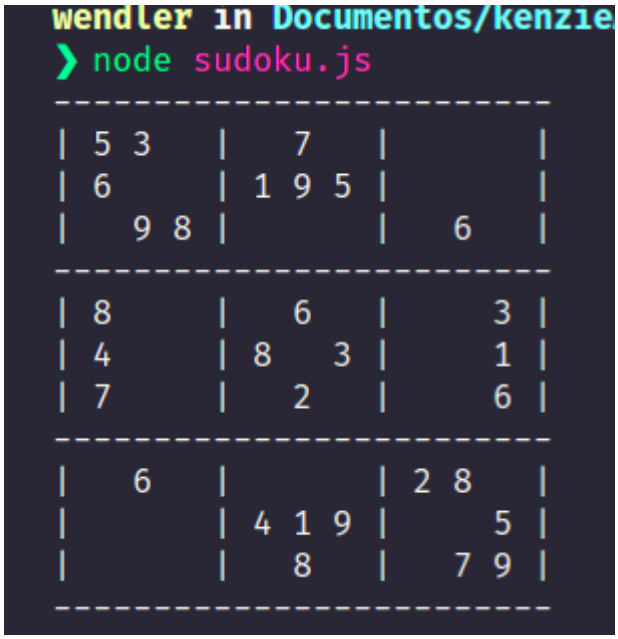
Agora a inserção de um número:

```
function solveSudoku(board) {
  let freePosition = findZero(board);
```

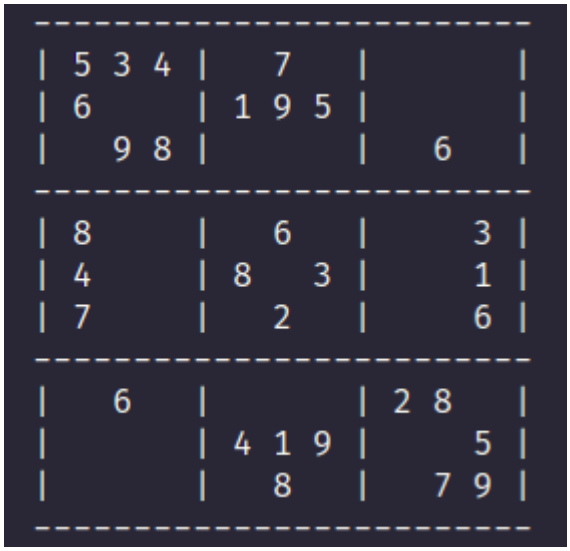
```
  if (freePosition === undefined) return board;

  let freeRow = freePosition[0];
  let freeColumn = freePosition[1];

  for (let number = 1; number < 10; number++) {
    if (isValid(board, freeRow, freeColumn, number)) {
      board[freeRow][freeColumn] = number;
    }
  }
  showBoard(board);
}
```

Se rodarmos a função do jeito em que está teremos:

Tabuleiro após a função:



Tabuleiro após a tentativa de inserção:



Note que no primeiro quadrante foi adicionado o número 4. Ótimo, nossa função insere números. No entanto, ainda restam mais espaços vazios no tabuleiro, teremos que executar a inserção correta para cada um deles para cada um deles.

Mas espere, após verificar isso percebi um possível erro, porque foi adicionado o número 4 e não o número 1? Se você prestou atenção viu que nosso loop vai tentar sempre substituir o número pelo contador `number` atual. Como o 4 foi o último número válido do loop possível, então ele permaneceu lá.

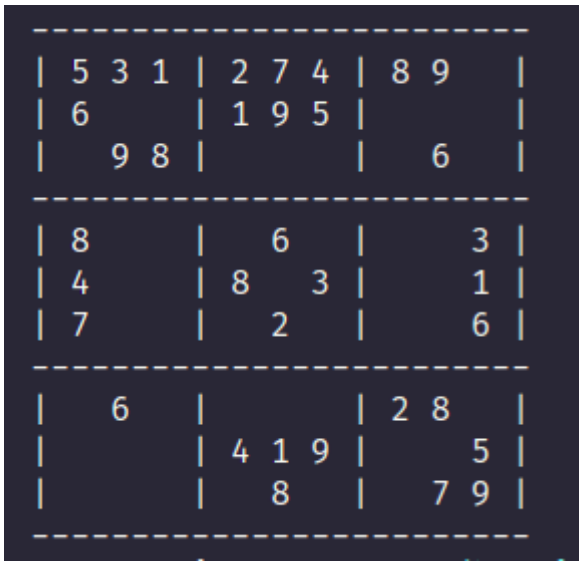
Precisamos testar se o primeiro caso é suficiente, e caso seja, continuar inserindo os próximos números. Como fazer isso? Bom, é aí que entra a tal da recursão.

Após inserir o número, vamos chamar novamente nossa função solveSudoku passando como parâmetro desta vez o tabuleiro atualizado com o último número inserido, a pilha de chamada será encerada no momento em que não haja mais espaços vazios no tabuleiro, afinal colocamos essa condição logo no início da função lembra? Então vamos fazer isso:

```
function solveSudoku(board) {
  let freePosition = findZero(board);

  if (freePosition === undefined) return board;

  let freeRow = freePosition[0];
  let freeColumn = freePosition[1];
```

```
  for (let number = 1; number < 10; number++) {
    if (isValid(board, freeRow, freeColumn, number)) {
      board[freeRow][freeColumn] = number;
      return solveSudoku(board);
    }
  }

  showBoard(board);
}
```

O resultado foi:



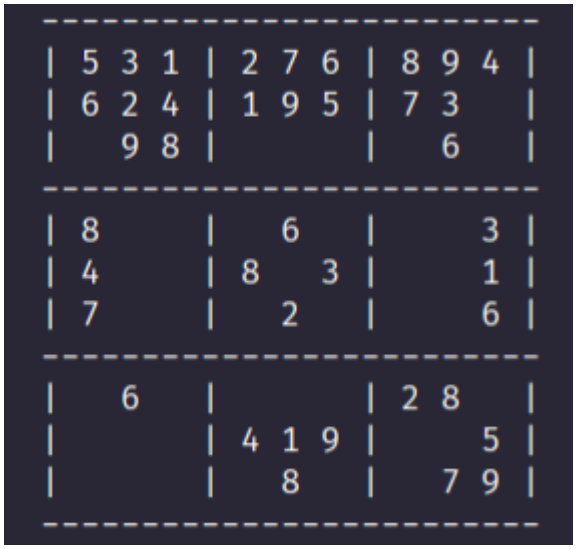
O que houve? Nosso último tabuleiro exibido no console foi este. Porém, assim como aconteceria caso você tivesse tentado resolver este tabuleiro na mão ou intuição, ele foi preenchendo com os números que dava, até chegar um momento em que na última coluna não era mais possível inserir nenhum número sem quebrar as regras do jogo. Uma vez que isso aconteceu a condição de nosso `if(isValid)` não foi atendida e a função não se chamou novamente. Interrompendo assim o ciclo de chamadas.

Precisamos adicionar uma alternativa para o caso da solução atual não funcionar. pense se você estivesse resolvendo este tabuleiro na mão e percebesse que a sequência de números que inseriu não funciona? Provavelmente você apagaria o que fez e tentaria de novo com outros números certo? E é exatamente isso que tentaremos fazer nosso código executar.

Primeiro, após inserir um número, vamos armazenar o resultado da próxima chamada em uma variável que chamaremos de `currentSolution` caso durante essa próxima chamada não tenha sido possível inserir um número, vamos retornar `null` e isso significa que o número inserido no momento fez com que o próximo não fosse possível

```
function solveSudoku(board) {
  let freePosition = findZero(board);

  if (freePosition === undefined) return board;

  let freeRow = freePosition[0];
  let freeColumn = freePosition[1];

  for (let number = 1; number < 10; number++) {
    if (isValid(board, freeRow, freeColumn, number)) {
      board[freeRow][freeColumn] = number;
      let currentSolution = solveBoard(board);
      if (currentSolution !== null) {
        return currentSolution;
      }
    }
  }

  showBoard(board);
  return null;
}
```

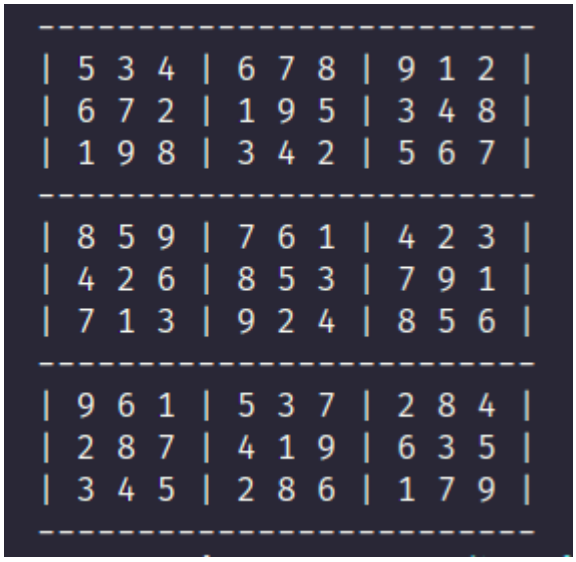
No entanto se rodarmos assim, o que teremos:



Dessa vez conseguimos antecipar alguns movimentos e verificar "Se eu colocar esse número, na próxima chamada ainda terei opções?" Se sim ele adicionava o número presente no loop. Porém não foi suficiente, e sabe porquê? Porque não não adicionamos o comportamento de "apagar" o número caso alguma jogada futura não seja possível graças a ele. E como faríamos para apagar um número? Bom basta igualar ele a 0:

```
function solveSudoku(board) {
  let freePosition = findZero(board);

  if (freePosition === undefined) return board;

  let freeRow = freePosition[0];
  let freeColumn = freePosition[1];

  for (let number = 1; number < 10; number++) {
    if (isValid(board, freeRow, freeColumn, number)) {
      board[freeRow][freeColumn] = number;
      let currentSolution = solveSudoku(board);
      if (currentSolution !== null) {
        showBoard(board);
        return currentSolution;
      } else {
        board[freeRow][freeColumn] = 0;
      }
    }
  }
  showBoard(board);
  return null;
}
```

Com a simples adição dessa linha foi possível encontrar:



Basicamente, o fluxo seguia: "Posso adicionar esse número? Pode", "Posso adicionar esse? Não" - Apaga, "Caso não seja possível para nenhum outro, apaga mais um e tenta de novo".

Esse desafio me tomou alguns dias e umas boas horas estudando alguns tópicos de fluxo de código, se você entendeu tudo de primeira e já consegue até encontrar melhorias, show de bola, caso não, não tem problema, essa foi apenas uma demostração de como seria para resolver um problema dessa natureza.

É isso, qualquer coisa estamos aí, valeu! 😀