1. Configuração inicial

A primeira coisa de que precisamos é uma configuração na qual possamos jogar o jogo e ver como está o progresso.

Isso nos dará todas as construções de que precisamos para jogar o jogo e implementar totalmente o jogador do computador - que só coloca peças aleatórias de qualquer maneira. Isso nos dá a oportunidade de implementar um jogador “humano” para jogar.

1.1. Tabuleiro de Jogo

Antes de mais nada, precisamos de um tabuleiro de jogo. Esta é uma grade de células nas quais os números podem ser colocados.

Para tornar algumas coisas um pouco mais fáceis de trabalhar, vamos começar com uma representação simples da localização de uma célula . Isso é literalmente apenas um invólucro em torno de um par de coordenadas:

```
public class Cell {
    private final int x;
    private final int y;

    // constructor, getters, and toString
}
```

Agora podemos escrever uma classe para representar o próprio conselho . Isso armazenará os valores em uma matriz bidimensional simples, mas nos permitirá acessá-los por meio da classe *Cell* acima :

```
public class Board {
    private final int[][] board;
    private final int score;

    public Board(int size) {
        this.board = new int[size][];
        this.score = 0;

        for (int x = 0; x < size; ++x) {
            this.board[x] = new int[size];
            for (int y = 0; y < size; ++y) {
                board[x][y] = 0;
            }
        }
    }

    public int getSize() {
        return board.length;
    }

    public int getScore() {
```

```
        return score;
    }

    public int getCell(Cell cell) {
        return board[cell.getX()][cell.getY()];
    }

    public boolean isEmpty(Cell cell) {
        return getCell(cell) == 0;
    }

    public List<Cell> emptyCells() {
        List<Cell> result = new ArrayList<>();
        for (int x = 0; x < board.length; ++x) {
            for (int y = 0; y < board[x].length; ++y) {
                Cell cell = new Cell(x, y);
                if (isEmpty(cell)) {
                    result.add(cell);
                }
            }
        }
        return result;
    }
}
```

Esta é uma classe imutável que representa um tabuleiro e nos permite interrogá-lo para descobrir o estado atual. Ele também mantém o controle de uma pontuação atual, que veremos mais tarde.

1.2. Um reprodutor de computador e colocação de peças

Agora que temos um tabuleiro de jogo, queremos poder brincar com ele. A primeira coisa que queremos é o reprodutor do computador, porque este é um reprodutor puramente aleatório e será exatamente o que for necessário mais tarde.

O jogador do computador não faz nada mais do que colocar uma peça em uma célula, então precisamos de alguma maneira de fazer isso em nosso tabuleiro. Queremos manter isso como sendo imutável, portanto, colocar um ladrilho irá gerar uma nova placa no novo estado.

Primeiro, queremos um construtor que assumirá o estado real da placa , ao contrário de nosso anterior, que apenas construiu uma placa em branco:

```
private Board(int[][] board, int score) {
    this.score = score;
    this.board = new int[board.length][];

    for (int x = 0; x < board.length; ++x) {
        this.board[x] = Arrays.copyOf(board[x], board[x].length);
```

```
    }
}
```

Isso é *privado, de* forma que só pode ser usado por outros métodos da mesma classe. Isso ajuda com nosso encapsulamento da placa.

A seguir, vamos adicionar um método para colocar um ladrilho. Isso retorna uma placa totalmente nova que é idêntica à atual, exceto que tem o número fornecido na célula fornecida:

```
public Board placeTile(Cell cell, int number) {
    if (!isEmpty(cell)) {
        throw new IllegalArgumentException("That cell is not empty");
    }

    Board result = new Board(this.board, this.score);
    result.board[cell.getX()][cell.getY()] = number;
    return result;
}
```

Finalmente, escreveremos uma nova classe representando um jogador de computador. Isso terá um único método que pegará a placa atual e retornará a nova:

```
public class Computer {
    private final SecureRandom rng = new SecureRandom();

    public Board makeMove(Board input) {
        List<Cell> emptyCells = input.emptyCells();

        double numberToPlace = rng.nextDouble();
        int indexToPlace = rng.nextInt(emptyCells.size());
        Cell cellToPlace = emptyCells.get(indexToPlace);

        return input.placeTile(cellToPlace, numberToPlace >= 0.9 ? 4 :
2);
    }
}
```

Isso obtém a lista de todas as células vazias do tabuleiro, escolhe uma célula aleatória e, em seguida, coloca um número nela. Decidiremos aleatoriamente colocar um “4” na célula 10% das vezes e um “2” nos outros 90%.

1.2. Um jogador “humano” e peças móveis

A próxima coisa de que precisamos é um jogador “humano”. Este não será o objetivo final, mas um jogador puramente aleatório que escolhe uma direção aleatória para mudar as peças toda vez que faz um movimento. Isso funcionará como um lugar sobre o qual podemos construir para fazer o nosso jogador ideal.

Em primeiro lugar, precisamos definir uma enumeração dos movimentos possíveis que podem ser feitos:

```
public enum Move {
```

```
    UP,
    DOWN,
    LEFT,
    RIGHT
}
```

Em seguida, precisamos aumentar a classe *Board* para apoiar a realização de movimentos, deslocando os ladrilhos em uma dessas direções. Para reduzir a complexidade aqui, queremos girar o tabuleiro de modo que estejamos sempre deslocando as peças na mesma direção.

Isso significa que precisamos de um meio para transpor e inverter o quadro:

```
private static int[][] transpose(int[][] input) {
    int[][] result = new int[input.length][];

    for (int x = 0; x < input.length; ++x) {
        result[x] = new int[input[0].length];
        for (int y = 0; y < input[0].length; ++y) {
            result[x][y] = input[y][x];
        }
    }

    return result;
}

private static int[][] reverse(int[][] input) {
    int[][] result = new int[input.length][];

    for (int x = 0; x < input.length; ++x) {
        result[x] = new int[input[0].length];
        for (int y = 0; y < input[0].length; ++y) {
            result[x][y] = input[x][input.length - y - 1];
        }
    }

    return result;
}
```

Transpor o tabuleiro trocará todas as linhas e colunas ao redor, de modo que a borda superior se torne a borda esquerda. Inverter o tabuleiro simplesmente o espelha de forma que a borda esquerda se torne a borda direita.

Em seguida, adicionamos um método ao *Conselho* para fazer um movimento em uma determinada direção e retornar um novo *Conselho* no novo estado.

Começamos fazendo uma cópia do estado da placa com a qual podemos trabalhar:

```
public Board move(Move move) {
    int newScore = 0;

    // Clone the board
    int[][] tiles = new int[this.board.length][];
    for (int x = 0; x < this.board.length; ++x) {
```

```
        tiles[x] = Arrays.copyOf(this.board[x], this.board[x].length);
    }
```

Em seguida, manipulamos nossa cópia de modo que estaremos sempre deslocando os blocos para cima:

```
if (move == Move.LEFT || move == Move.RIGHT) {
    tiles = transpose(tiles);

}
if (move == Move.DOWN || move == Move.RIGHT) {
    tiles = reverse(tiles);
}
```

Precisamos de outro conjunto de blocos - desta vez aquele em que construiremos o resultado final - e um rastreador para a nova pontuação obtida para este movimento:

```
int[][] result = new int[tiles.length][];
int newScore = 0;
```

Agora que estamos prontos para começar a mudar os ladrilhos e manipulamos as coisas para que estejamos sempre trabalhando na mesma direção, podemos começar.

Podemos mudar cada coluna independentemente das outras. Precisamos apenas iterar sobre as colunas e repetir, começando com a construção de outra cópia dos blocos que estamos mudando.

Desta vez, nós os incorporamos em uma *LinkedList* porque queremos ser capazes de extrair valores dela facilmente. Também adicionamos apenas os ladrilhos reais que têm números e pulamos os ladrilhos vazios.

Isso atinge a nossa mudança, mas ainda não a fusão dos blocos:

```
for (int x = 0; x < tiles.length; ++x) {
    LinkedList<Integer> thisRow = new LinkedList<>();
    for (int y = 0; y < tiles[0].length; ++y) {
        if (tiles[x][y] > 0) {
            thisRow.add(tiles[x][y]);
        }
    }
```

Em seguida, precisamos mesclar os blocos. Precisamos fazer isso separadamente do acima; caso contrário, corremos o risco de mesclar o mesmo bloco várias vezes.

Isso é conseguido construindo outra *LinkedList* dos tiles acima, mas desta vez mesclando conforme avançamos:

```
LinkedList<Integer> newRow = new LinkedList<>();
while (thisRow.size() >= 2) {
    int first = thisRow.pop();
    int second = thisRow.peek();
    if (second == first) {
        int newNumber = first * 2;
        newRow.add(newNumber);
        newScore += newNumber;
```

```
            thisRow.pop();
        } else {
            newRow.add(first);
        }
    }
    newRow.addAll(thisRow);
```

Aqui também estamos calculando a nova pontuação para este movimento. Esta é a soma dos blocos criados como resultado das mesclagens.

Agora podemos construir isso na matriz de resultados. Quando acabamos os blocos de nossa lista, o restante é preenchido com o valor “0” para indicar que estão em branco:

```
        result[x] = new int[tiles[0].length];
        for (int y = 0; y < tiles[0].length; ++y) {
            if (newRow.isEmpty()) {
                result[x][y] = 0;
            } else {
                result[x][y] = newRow.pop();
            }
        }
    }
```

Assim que terminarmos de mudar os ladrilhos, precisamos manipulá-los novamente para a rotação correta. Este é exatamente o oposto do que fizemos antes:

```
    if (move == Move.DOWN || move == Move.RIGHT) {
        result = reverse(result);
    }
    if (move == Move.LEFT || move == Move.RIGHT) {
        result = transpose(result);
    }
```

E, finalmente, podemos construir e retornar um novo tabuleiro com este novo conjunto de peças e a pontuação recém-calculada:

```
        return new Board(result, this.score + newScore);
    }
```

Agora estamos em uma posição em que podemos escrever nosso jogador “humano” aleatório. Isso não faz nada mais do que gerar um movimento aleatório e chamar o método acima para fazer esse movimento:

```
public class Human {
    private SecureRandom rng = new SecureRandom();

    public Board makeMove(Board input) {
        Move move = Move.values()[rng.nextInt(4)];
        return input.move(move);
    }
}
```

1.3. Jogando o jogo

Temos componentes suficientes para jogar o jogo, embora não com muito sucesso. Porém, em breve estaremos aprimorando a forma como a classe *Humana* joga, e isso nos permitirá ver as diferenças facilmente.

Primeiro, precisamos imprimir o tabuleiro do jogo.

Para este exemplo, vamos apenas imprimir no console, então *System.out.print* é bom o suficiente. Para um jogo real, gostaríamos de fazer gráficos melhores:

```
private static void printBoard(Board board) {
    StringBuilder topLines = new StringBuilder();
    StringBuilder midLines = new StringBuilder();
    for (int x = 0; x < board.getSize(); ++x) {
        topLines.append("+--------");
        midLines.append("|        ");
    }
    topLines.append("+");
    midLines.append("|");

    for (int y = 0; y < board.getSize(); ++y) {
        System.out.println(topLines);
        System.out.println(midLines);
        for (int x = 0; x < board.getSize(); ++x) {
            Cell cell = new Cell(x, y);
            System.out.print("|");
            if (board.isEmpty(cell)) {
                System.out.print("        ");
            } else {
                StringBuilder output = new
StringBuilder(Integer.toString(board.getCell(cell)));
                while (output.length() < 8) {
                    output.append(" ");
                    if (output.length() < 8) {
                        output.insert(0, " ");
                    }
                }
                System.out.print(output);
            }
        }
        System.out.println("|");
        System.out.println(midLines);
    }
    System.out.println(topLines);
    System.out.println("Score: " + board.getScore());
}
```

Estamos quase prontos para partir. Só precisamos configurar as coisas.

Isso significa criar o tabuleiro, os dois jogadores e fazer com que o computador faça dois movimentos iniciais - ou seja, colocar dois números aleatórios no tabuleiro:

```
Board board = new Board(4);
```

```
Computer computer = new Computer();
Human human = new Human();
for (int i = 0; i < 2; ++i) {
    board = computer.makeMove(board);
}
```

E agora temos o loop de jogo real. Esta será uma repetição dos jogadores humanos e do computador se revezando e parando apenas quando não houver mais células vazias:

```
printBoard(board);
do {
    System.out.println("Human move");
    System.out.println("==========");
    board = human.makeMove(board);
    printBoard(board);

    System.out.println("Computer move");
    System.out.println("=============");
    board = computer.makeMove(board);
    printBoard(board);
} while (!board.emptyCells().isEmpty());

System.out.println("Final Score: " + board.getScore());
```

Nesse ponto, se executássemos o programa, veríamos um jogo aleatório de 2048 sendo jogado.

2. Implementando o 2048 Player

Assim que tivermos uma base para jogar, podemos começar a implementar o jogador “humano” e jogar um jogo melhor do que apenas escolher uma direção aleatória.

2.1. Simulando movimentos

O algoritmo que estamos implementando aqui é baseado no algoritmo Expectimax . Como tal, o núcleo do algoritmo é simular todos os movimentos possíveis, atribuir uma pontuação a cada um e selecionar aquele que se sai melhor.

Faremos uso intenso do Java 8 Streams para ajudar a estruturar este código, por motivos que veremos mais tarde.

Começaremos reescrevendo o método *makeMove ()* de dentro de nossa classe *Human* :

```
public Board makeMove(Board input) {
    return Arrays.stream(Move.values())
      .map(input::move)
      .max(Comparator.comparingInt(board -> generateScore(board, 0)))
      .orElse(input);
}
```

Para cada direção possível em que podemos nos mover, geramos o novo tabuleiro e então iniciamos o algoritmo de pontuação - passando neste tabuleiro e uma profundidade de 0. Em seguida, selecionamos o movimento com a melhor pontuação.

Nosso método *generateScore ()* então simula todos os movimentos possíveis do computador - isto é, colocando um "2" ou um "4" em cada célula vazia - e então vê o que pode acontecer a seguir:

```
private int generateScore(Board board, int depth) {
    if (depth >= 3) {
        return calculateFinalScore(board);
    }
    return board.emptyCells().stream()
      .flatMap(cell -> Stream.of(new Pair<>(cell, 2), new Pair<>(cell,
4)))
      .mapToInt(move -> {
          Board newBoard = board.placeTile(move.getFirst(),
move.getSecond());
          int boardScore = calculateScore(newBoard, depth + 1);
          return (int) (boardScore * (move.getSecond() == 2 ? 0.9 :
0.1));
      })
      .sum();
}
```

Se atingirmos nosso limite de profundidade, pararemos imediatamente e calcularemos uma pontuação final de quão boa é esta prancha; caso contrário, continuamos com nossa simulação.

Nosso método *calculScore ()* é então a continuação de nossa simulação, executando o lado do movimento humano da equação.

Isso é muito semelhante ao método *makeMove ()* acima, mas estamos retornando a pontuação contínua em vez do quadro real:

```
private int calculateScore(Board board, int depth) {
    return Arrays.stream(Move.values())
      .map(board::move)
      .mapToInt(newBoard -> generateScore(newBoard, depth))
      .max()
      .orElse(0);
}
```

2.2. Pontuação das Tabelas Finais

Estamos agora em uma situação em que podemos simular movimentos para frente e para trás dos jogadores humanos e do computador, parando quando tivermos simulado o suficiente deles. Precisamos ser capazes de gerar uma pontuação para o tabuleiro final em cada ramificação da simulação, para que possamos ver qual ramificação é a que desejamos seguir.

Nossa pontuação é uma combinação de fatores, cada um dos quais iremos aplicar a cada linha e cada coluna do tabuleiro. Todos eles são somados e o total é retornado.

Como tal, precisamos gerar uma lista de linhas e colunas para pontuar:

```
List<List<Integer>> rowsToScore = new ArrayList<>();
for (int i = 0; i < board.getSize(); ++i) {
    List<Integer> row = new ArrayList<>();
    List<Integer> col = new ArrayList<>();

    for (int j = 0; j < board.getSize(); ++j) {
        row.add(board.getCell(new Cell(i, j)));
        col.add(board.getCell(new Cell(j, i)));
    }

    rowsToScore.add(row);
    rowsToScore.add(col);
}
```

Em seguida, pegamos a lista que construímos, pontuamos cada um deles e somamos as pontuações. Este é um espaço reservado que estamos prestes a preencher:

```
return rowsToScore.stream()
    .mapToInt(row -> {
        int score = 0;
        return score;
    })
    .sum();
```

Finalmente, precisamos realmente gerar nossas pontuações. Isso vai dentro do lambda acima, e há vários fatores diferentes que contribuem :

- Uma pontuação fixa para cada linha
- A soma de todos os números da linha
- Cada fusão possível na linha
- Cada célula vazia na linha
- A monotonicidade da linha. Isso representa a quantidade em que a linha é organizada em ordem numérica crescente.

Antes de podermos calcular as pontuações, precisamos construir alguns dados extras.

Primeiro, queremos uma lista dos números com células em branco removidas:

```
List<Integer> preMerged = row.stream()
  .filter(value -> value != 0)
  .collect(Collectors.toList());
```

Podemos então fazer algumas contagens a partir dessa nova lista, dando o número de células adjacentes com o mesmo número, com

números estritamente crescentes e números estritamente decrescentes:

```
int numMerges = 0;
int monotonicityLeft = 0;
int monotonicityRight = 0;
for (int i = 0; i < preMerged.size() - 1; ++i) {
    Integer first = preMerged.get(i);
    Integer second = preMerged.get(i + 1);
    if (first.equals(second)) {
        ++numMerges;
    } else if (first > second) {
        monotonicityLeft += first - second;
    } else {
        monotonicityRight += second - first;
    }
}
```

Agora podemos calcular nossa pontuação para esta linha:

```
int score = 1000;
score += 250 * row.stream().filter(value -> value == 0).count();
score += 750 * numMerges;
score -= 10 * row.stream().mapToInt(value -> value).sum();
score -= 50 * Math.min(monotonicityLeft, monotonicityRight);
return score;
```

Os números selecionados aqui são relativamente arbitrários. Números diferentes terão um impacto sobre a qualidade do jogo, priorizando diferentes fatores em como jogamos.

3. Melhorias no algoritmo

O que temos até agora funciona, e podemos ver que joga um bom jogo, mas é lento. Demora cerca de 1 minuto por movimento humano. Nós podemos fazer melhor que isso.

3.1. Processamento paralelo

A coisa óbvia que podemos fazer é trabalhar em paralelo. Este é um grande benefício de trabalhar com Java Streams - podemos fazer isso funcionar em paralelo apenas adicionando uma única instrução a cada stream.

Essa mudança por si só nos reduz para cerca de 20 segundos por movimento.

3.2. Podando ramos não jogáveis

A próxima coisa que podemos fazer é podar todos os ramos que não podem ser reproduzidos. Ou seja, sempre que um movimento humano resultar em um tabuleiro inalterado. É quase certo que essas ramificações irão resultar em resultados piores - elas estão efetivamente dando ao computador um movimento grátis - mas elas nos custam tempo de processamento para buscá-las.

Para fazer isso, precisamos implementar um método equals em nosso *Conselho* para que possamos compará-los:

```
@Override
public boolean equals(Object o) {
    if (this == o) {
        return true;
    }
    if (o == null || getClass() != o.getClass()) {
        return false;
    }
    Board board1 = (Board) o;
    return Arrays.deepEquals(board, board1.board);
}
```

Podemos então adicionar alguns filtros aos nossos pipelines de fluxo para interromper o processamento de qualquer coisa que não tenha mudado.

```
return Arrays.stream(Move.values())
    .parallel()
    .map(board::move)
    .filter(moved -> !moved.equals(board))
    ........
```

Isso tem impacto mínimo nas partes iniciais do jogo - quando há muito poucas células preenchidas, há muito poucos movimentos que podem ser cortados. Porém, mais tarde, isso começa a ter um impacto muito maior, reduzindo o tempo de movimentação para apenas alguns segundos.