**Report for Programming Problem 1 - 2048**

**Team:** Student ID: 2018280762 Name: Mariana Loreto
Student ID: 2018288500 Name: Mariana Lança

## 1. Algorithm description

O algoritmo começa por verificar se a completude do tabuleiro é possível, na função *isPossible*, fazendo $log_2$da soma de todos os valores; se tal der um resultado inteiro, significa que o tabuleiro tem pares suficientes para avançar.

Se a tabela for possível, é inicializada a variável que armazenará o, *best*, a infinito. De seguida, é feita a chamada da função *countMoves*. Esta recebe como parâmetros uma referência a um objeto do tipo Board, um contador, uma referência a *best*, e *previous_moves*. *Best* representa o valor da solução com menor número de jogadas até ao momento e *previous_moves* indica se a jogada anterior surtiu algum efeito e se, seguidamente, é relevante repeti-la. Assim, evita-se entrar numa sequência de jogadas desnecessária.

```
CountMoves (&board, counter, best, previous_moves)
1    if isCompleted then
2         if counter < best then
3              best = counter
4         return

5    if !(counter < best and counter < max_moves and !(best -
counter < 2 and occ > 2)) then
6         return

8    counter = counter + 1

9    if previous_move != 1 then
10        aux = board
11        if board.shiftRight(previous_move) then
12             countMoves(aux, counter, best, previous_move)

13   if previous_move != 2 then
14        aux = board
15        if board.shiftTop(previous_move) then
16             countMoves(aux, counter, best, previous_move)

17   if previous_move != 1 then
18        aux = board
19        if board.shiftLeft(previous_move) then
20             countMoves(aux, counter, best, previous_move)

21   if previous_move != 2 then
22        aux = board
23        if board.shiftBottom(previous_move) then
24             countMoves(aux, counter, best, previous_move)
```

Aquando da primeira chamada, são passados, como argumentos, o tabuleiro recebido como input, *counter* a 0, *best* a infinito (max_moves + 1) e *previous_moves* a -1.

A função *countMoves* começa por testar a condição de aceitação na **linha 1**, i.e , por verificar se o tabuleiro já se encontra completo (se apenas houver um bloco preenchido), e a de negação, na **linha 5**.

Nesta primeira condição, o valor de *best* é atualizado e a função termina. Já na segunda condição faz três verificações sendo que a última verifica apenas se, sendo a próxima jogada a última válida (por *counter* estar quase a atingir o valor de *best*) o número de casas ocupadas é superior a 2. Se alguma destas condições falhar, a função termina.

Nas **linhas 11**, **15**, **19** e **23** é feito o *shift* do tabuleiro nas várias direções e, se tiverem ocorrido alterações no tabuleiro, é feita a chamada recursiva na linha seguinte, passando o tabuleiro auxiliar onde foi realizado o movimento.

Para fazer os movimentos criamos quatro funções de shift, que seguem todas o mesmo algoritmo. Assim, para demonstração, vamos explicar apenas a função *shiftLeft*.

Começamos por percorrer uma linha (da esquerda para a direita) e selecionamos o primeiro número não nulo, guardando-o na variável *value*. De seguida, procuramos o próximo valor não nulo nessa linha. Se ambos os valores forem iguais, fazemos o *merge*, e arrastamos para a posição correta. Se forem diferentes, arrastamos o primeiro número, atualizando a variável *value* com o segundo e procuramos o número não nulo seguinte. Caso não exista um valor para fazer *merge*, arrastar-se-á o número para a posição correta. Passamos para a linha seguinte e o algoritmo repete-se.

Ainda temos duas variáveis auxiliares, *isUpdated* e *shifted*, que nos permitem saber que operações ocorreram na matriz (*merge* e *shift,* respetivamente). Se apenas tiver ocorrido um *shift*, o valor de *previous_move* é alterado para 1 (visto ser uma operação horizontal; no caso de ser vertical, este valor será 2). A função retorna *isUpdated*.

## 2. Data structures

Criamos uma classe denominada **Board**, que contém uma matriz de inteiros que guarda o tabuleiro e duas variáveis inteiras: **occ** (que indica o números de casas ocupadas) e **sum** (a soma total dos valores no tabuleiro). Criamos várias funções que permitem manipular a classe.

## 3. Correctness

De forma a minimizar o número de movimentos,é feita uma sequência de verificações ao longo do código. Começamos por analisar se à partida existem pares suficientes para fazer *merge* e atingir uma solução, na função *isPossible*. Isto permite, logo desde o início, filtrar alguns dos casos impossíveis.

Já dentro da função recursiva, na condição de rejeição, verifica-se se o *counter* é menor do que *best* e *max_moves*, assim, não se permite que sejam feitas chamadas recursivas desnecessárias. Além disto, tal como referido antes, é verificado se, no caso da próxima jogada ser a última válida (i.e, o counter está a

duas jogadas de diferença do *best*) e há, no tabuleiro, mais de duas casas ocupadas. Isto significa que, na última jogada, não será possível atingir a completude do tabuleiro. Apesar de não ser uma restrição bastante significativa, esta permite um aumento de performance temporal do programa por bloquear chamadas que não afetarão o resultado final.

A utilização da variável *previous_moves*, cujo valor é alterado na função *shift*, previne que se movimente numa orientação quando, na jogada anterior, apenas tiver ocorrido mudança na disposição do tabuleiro, sem que tenha havido convergência. Posteriormente, aquando do shift, apenas se o tabuleiro tiver sofrido algum tipo de alteração, é feita a chamada recursiva.

Ao fazer uma alternação da ordem de invocação das funções de *shift*, estamos, também, a fazer melhoramentos na performance, visto que a probabilidade de ocorrer uma jogada significativa quando se muda de direção em duas jogadas, é maior do que ao se manter a mesma.

## 4. Algorithm Analysis

A complexidade temporal deste algoritmo é :

$$
\begin{aligned}
T(n) &= 4(T(n) + S) + c \\
&= 4\,T(n) + 4S + c \\
&= 4\,(4T(n-1) + S) + c) + c \\
&= 16T(n-1) + 20S + 5c \\
&= 16\,(4(T(n-2) + S) + c) + 20S + 5c \\
&= 64\,T(n-2) + 84\,S + 21c \\
&= 4^k T(n-k) + \sum_{i=1}^{k} 4^i S + \sum_{i=0}^{k-1} 4^i c \\
&= 4^k T(0) + \sum_{i=1}^{k} 4^i S + \sum_{i=0}^{k-1} 4^i c \\
&= 4^k + \sum_{i=1}^{k} 4^i S + \sum_{i=0}^{k-1} 4^i
\end{aligned}
$$

em que S representa as funções Shift, cuja complexidade temporal é O($n^2$). Abrindo esta expressão, chegamos a que a complexidade deste algoritmo é $\sum_{i=1}^{k} 4^k\, n^{\,2} + \sum_{i=0}^{k} 4^k \in O(4^k)$ , sendo que **k** é o número máximo de movimentos e **n** é a dimensão do tabuleiro.

A complexidade espacial é O(n²), sendo **n** a dimensão do tabuleiro.

## 5. References

Cormen et al, Introduction to Algorithms (Section 2.1)