Report for Programming Problem 1

Team: 2019219068_2019214567

Student ID: 2019214567 **Name** Miguel Antunes Carvalho Portugal Ferreira
Student ID: 2019219068 **Name** Tiago Miguel Gomes Marques Oliveira

1. Algorithm description

Para facilitar a explicação, consideremos os números que formam a metade direita de uma peça como *Above* e *Below*, e os números que formam a metade inferior de uma peça como *Left* e *Right*.

Inicialmente todas as peças são percorridas de forma a formar um dicionário de pares, em que a chave indica um par de números e o valor dessa chave indica quais peças contêm esse par. O tabuleiro é preenchido linha a linha, da esquerda para a direita através da função *tree*. Esta função é auxiliada na recursão por 3 outras: **addPrimeiraLinha**, usada apenas na primeira linha do tabuleiro, compara o *Above* e o *Below* da peça da esquerda com o dicionário de pares e, de seguida, com as peças que contêm esse par. Antes desta comparação, é verificado se a potencial peça já está a ser utilizada ou não; **addPrimeiraColuna**, usada na primeira coluna do tabuleiro, funciona da mesma forma que a função anterior, mas compara o *Left* e *Right* da peça de cima com o dicionário, em vez do Above e *Below*; **addPeca**, usada para adicionar peças que estejam fora da primeira linha e da primeira coluna. É formado um trio através da reunião do *Below* e *Above* da peça da esquerda e do *Left* e *Right* da peça de cima. Após a formação do trio, este é usado para verificar quais das possíveis peças que encaixam à esquerda também encaixam em cima.

Após ser selecionada a peça que vai ser inserida, a mesma irá ser rodada, usando os métodos *Rotate90*, *Rotate180* e *Rotate270*, tendo em atenção em que posições do array da peça se encontra o par ou o trio, selecionando assim a rotação que coloca o par ou trio no local pretendido. De seguida, é chamada a função *tree* para a posição seguinte do tabuleiro. Se esta função chegar à última posição do tabuleiro é retornado *True* e o puzzle é dado como resolvido. Se durante a recursão se chegar a um ponto em que não há peças que encaixem, a função devolve *False* e a recursão retrocede, removendo as peças das posições anteriores até haver uma que encaixe. No caso em que a peça tem 3 números iguais, antes de ser testado outro possível *match* para aquela posição, a peça será rodada novamente (apenas uma rotação de 90 graus, no sentido horário) e é verificado se o puzzle termina. Caso o puzzle não termine a recursão irá retroceder e continuará a verificar os possíveis *matches* para aquela posição.

Foram ainda usados dois tipos de pré-processamento para melhorar a eficiência do algoritmo. Inicialmente foi criado um dicionário (*map* em *cpp*) contendo como *key* um par de números consecutivos existente na peça e como *values* um *array* com todas as peças que possuem esse par. O segundo tipo de pré-processamento consiste na identificação de *impossible puzzles* sem introduzir qualquer peça no tabuleiro. Para essa identificação foi necessário verificar as ocorrências de cada número existente nas peças do puzzle, sendo que se existirem um mínimo de 5 números com um número de ocorrências ímpar o puzzle é considerado impossível.

2. Data structures

Neste projeto foram utilizadas duas classes principais: **Piece** e **Board**.

A classe **Piece** representa uma peça do puzzle e contém um vetor *num* que guarda os números que compõem a peça, dois inteiros *row* e *col* que representam, respetivamente, a linha e a coluna em que a peça foi colocada, um booleano *used* que indica se a peça já foi colocada no puzzle e um booleano *three*, que indica se a peça é composta por três números iguais. Esta classe também contém os métodos *Rotate90*, *Rotate180* e *Rotate270*, que servem para rodar a peça, e os métodos *getBellowAbove*, que devolve o par de números que forma a metade direita da peça, e *getLeftRight*, que devolve o par de números que forma a parte inferior da peça.

A classe **Board** representa o tabuleiro onde as peças são colocadas. Contém a variável *n_pieces*, que indica o número total de peças, as variáveis *rows* e *columns*, que indicam, respetivamente, o número total de linhas e colunas do tabuleiro. O vetor *pieces*, que contém todas as peças a serem colocadas e o vetor *board*, que inicialmente está vazia, mas que vai sendo preenchido à medida que o tabuleiro é completado. O método *printBoard* constrói o output quando é encontrada uma solução para o puzzle e o *removeToALimit* elimina todas as peças introduzidas na board até à peça em que não foi possível encontrar correspondência, inclusivamente.

3. Correctness

O algoritmo apenas teve uma pontuação de 175 pontos no *Mooshak*, sendo os restantes 25 pontos relativos a um caso de teste em que era obtido uma resposta errada. Esse caso de teste não pôde ser resolvido devido à dificuldade em introduzir corretamente, no tabuleiro, as peças com três números iguais. Esse problema poderia ser resolvido se em vez de rodar, uma segunda vez, as peças desse tipo apenas 90 graus, no sentido horário, ser verificada a rotação que melhor se enquadrava no problema. Assim, é pressuposto que o problema ficaria resolvido e seria possível obter 200 pontos, no entanto esta teoria não foi testada, não podendo ser assumida como certa.

Para melhorar a eficiência do algoritmo poderia ser usado unordered_map em vez de map devido a terem uma complexidade de inserção e procura de O(1), enquanto que o map tem complexidade de O(log n) para os dois casos.

Chegar aos 175 pontos apenas foi possível devido ao pré-processamento aplicado (desenvolvido no primeiro ponto do relatório) e ao elevado número de testes realizados, com diferentes *inputs*, na constante tentativa de aperfeiçoar o algoritmo e de encontrar diversos erros e *bugs* no código.

4. Algorithm Analysis

Em termos de complexidade espacial, esta vai ser O(n), sendo n o número de peças, porque tanto a estrutura usada para guardar as peças como o tabuleiro têm tamanho n.

Usando o *Master Theorem* para calcular a complexidade temporal do algoritmo obtivemos os seguintes resultados:

n = número de peças

a = *n* - *m*, sendo m o número de peças já colocadas no tabuleiro

n/b = *n*-1, porque no pior dos casos experimentamos encaixar todas as peças

d = 1, custo de encaixar uma peça

n^c = log(n), custo de percorrer o map para encontrar o par igual ao da peça com que se quer dar match

Como b = n/n-1 e c = log(log(n)), verificamos através do seguinte gráfico que $\log_b(a) > c$.

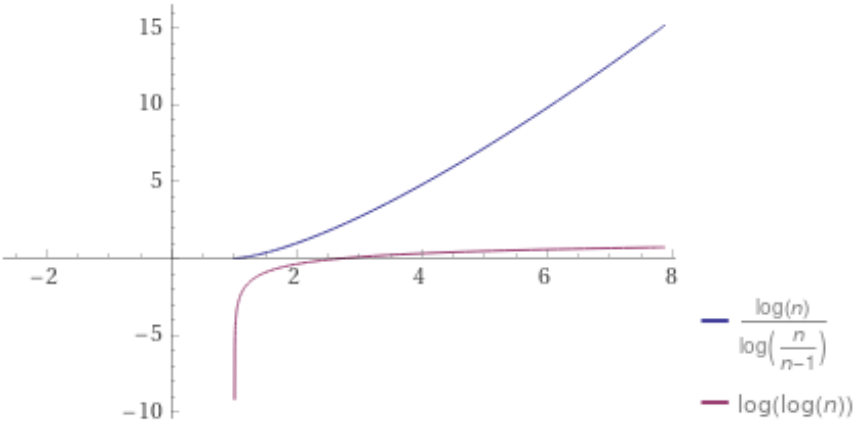

Figura 1 - Gráfico para cálculo do *Master Theorem*

Assim, concluímos que $T(n) = O(n^{\log_b(a)})$.

No melhor caso, em que as peças estão com a ordem correta, esta vai ser O(n), porque para cada posição do tabuleiro é verificada apenas uma peça, e não há retrocessos na recursão.

5. References

AfterAcademy, A. (24 de Dezembro de 2019). *Analysis of Recursion in Programming*. Obtido de AfterAcademy: https://afteracademy.com/blog/analysis-of-recursion-in-programming

lokeshpotta20, a. m. (13 de Dezembro de 2021). *Analysis of time and space complexity of C++ STL containers*. Obtido de GeeksforGeeks: https://www.geeksforgeeks.org/analysis-of-time-and-space-complexity-of-stl-containers/

Parewa Labs Pvt. Ltd. (s.d.). *Master Theorem*. Obtido de Programiz: https://www.programiz.com/dsa/master-theorem

Standard C++ Library reference. (2000-2021). Obtido de cplusplus.com: https://www.cplusplus.com/reference/

Wolfram Alpha LLC. (2022). Obtido de WolframAlpha: https://www.wolframalpha.com/input?i2d=true&i=Log%5BDivide%5Bn%2Cn-1%5D%2Cn%5D+%3D+log%5C%2840%29log%5C%2840%29n%5C%2841%29%5C%2841%29