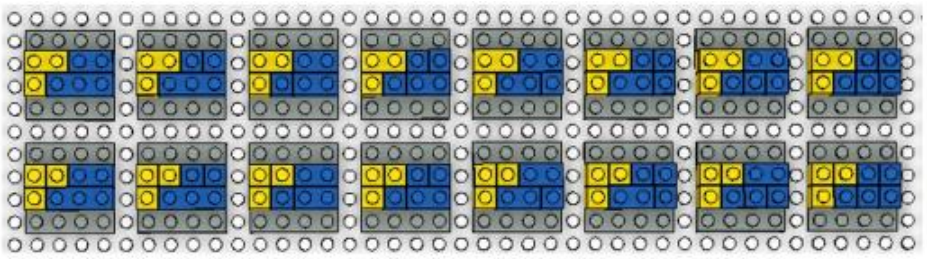
Relatório do Trabalho

Disciplina de Estruturas de Dados e Algoritmos II

- Problem: Mosaics –

```
....
YYBB
YBBB
....
```



Membros do grupo (g119):

- Diogo Carreiro nº 48729
- Rodrigo Alves nº 48681

Universidade de Évora, 23 de março de 2022

UNIVERSIDADE
DE ÉVORA

Introdução/Descrição

A montagem de imagens a partir de mosaicos, utilizando diferentes peças de lego, relativamente ao seu tamanho como também à sua cor, podemos concluir que com as peças de lego disponíveis se apresenta flexibilidade e versatilidade na criação de mosaicos. Mas qual será o número de maneiras diferentes de criar um certo mosaico a partir das peças de lego disponíveis?

Este trabalho tem como objetivo criar um algoritmo eficiente, que permita indicar o número de maneiras diferentes de construir um dado mosaico com um certo conjunto de peças de lego. A implementação deste algoritmo foi escrita em JAVA. O programa desenvolvido pelo grupo tem como base os conselhos do professor responsável pela cadeira neste semestre, que foram fundamentais para percebermos tanto o problema como também a dissociação do mesmo em problemas mais pequenos, que nos ajudaria a desenvolver a função recursiva que estava por detrás, como também na percetibilidade do algoritmo.

Em primeiro lugar, começámos a tratar do problema a partir de esquemas no papel, isto é, para um certo mosaico tentamos decifrar todas as possibilidades. A partir da realização de vários exemplos criados aleatoriamente, chegámos a uma conclusão acerca do algoritmo/função a ser utilizada.

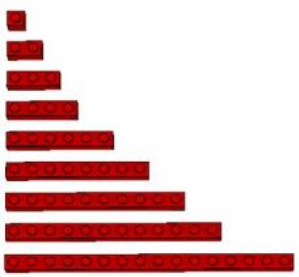
$$w(t) = \begin{cases} 1 & , se\ t = 0 \\ soma\{w(t - p_i)\} & , se\ t > 0\ \cap\ p_i \leq t \end{cases}$$

$$t = número\ de\ peças\ consecutivas\ iguais;$$

$$p = array\ que\ vai\ conter\ o\ tamanho\ das\ peças\ de\ lego\ disponiveis;$$

O caso base desta função é quando não existe nenhuma peça numa certa posição do mosaico, para este caso a função vai retornar um (1), pois só existe uma maneira de colocar zero peças no mosaico. Caso contrário, a função vai retornar o somatório da própria função usando como parâmetro o tamanho de uma peça encontrada no mosaico menos o tamanho de uma das peças de lego disponíveis (p_i) que seja menor ou igual ao tamanho encontrado (t).

Neste algoritmo o tamanho do mosaico é limitado, isto é, as colunas e linhas tem de ser menores ou iguais do que mil (1000) e maiores ou iguais que um (1). As peças de lego que permitem construir o nosso mosaico são colocadas horizontalmente, por isso as dimensões disponíveis são as seguintes: 1×1, 1×2, 1×3, 1×4, 1×6, 1×8, 1×10, 1×12 e 1×16.



Descrição do Programa

Leitura dos dados e controlo da execução do programa (*main*)

Em primeiro lugar é pedido ao utilizador o número de linhas (*lines*) e de colunas (*cols*) que o mosaico possui, sendo que cada número tem de estar separado por um espaço. Após serem introduzidas estas informações na primeira linha do terminal, decidimos guardar num *array* de *strings* (*mosaic_size*) os respetivos valores, sendo que a primeira posição do *array* (índice=0) vai possuir o número de linhas, e a segunda posição (índice=1) vai possuir o número de colunas em formato de *string*, respetivamente.

Ao serem introduzidos os dados anteriormente referidos, criamos um mosaico (*m*), a partir da classe *Mosaic*, que possui um construtor com um parâmetro *cols*, que representa o número de colunas do mosaico. De seguida, é inicializada a variável *result* que vai guardar o número total de maneiras diferentes de construir o mosaico introduzido no terminal a partir das peças de lego que estão disponíveis, esta variável vai ser igualada inicialmente a 1 (um), pois o número um é o elemento neutro da multiplicação, e como as linhas de código seguintes vão estar a multiplicar várias vezes esta variável, e posteriormente atualiza-la. Esta variável é do tipo *long* pois poderemos obter para alguns casos valores que não são abrangidos pelo tipo *int*, isto deve-se à possibilidade de haver sucessivas multiplicações de números muito grandes. O *result* corresponde ao nosso *output*.

O grupo decidiu que cada linha do mosaico ia ser tratada separadamente, ou seja, cada vez que é inserida uma linha, esta irá ser guardada num *array* de *strings* em que cada posição contem um ponto ou uma letra representativa de uma cor. Posteriormente é chamado o método *line_ways* da classe *Mosaic* (*c.line_ways*)que cujo o parâmetro da mesma é o *array* de *strings* que possui uma dada linha. Este método vai retornar o número de maneiras diferentes de criar a linha averiguada, e depois este valor é multiplicado pelo valor que está armazenado na variável *result* e a seguir esta variável é atualizada. O mesmo acontece para as restantes linhas, no final é apresentado o resultado do output.

Classe *Mosaic*

A classe Mosaic possui os dois métodos responsáveis que foram devolvidos pelo o algoritmo já anterior mencionado, por isso é necessário explicar cada um dos métodos individualmente:

line_ways – São inicializadas duas variáveis, a *count_equal_bricks* que vai guardar o número de posições seguidas que são preenchidas com a mesma cor, esta é igualada a zero (0) para começar a contagem. A segunda variável é o *res* que vai guardar o número de maneiras diferentes de criar a linha do mosaico que está no parâmetro do método, *res* é igualada a um (1) pois este é o elemento neutro da multiplicação. Neste método começamos por averiguar cada posição da linha, de coluna em coluna, por isso utilizamos um ciclo *for* que vai começar com a variável *col* igual a zero (0), que corresponde à primeira posição/coluna da linha, este ciclo vai ser executado enquanto o mesmo for menor que a variável privada da classe Mosaic (*this.c*), que foi definida pelo construtor. Dentro do ciclo avaliamos em primeiro lugar se a posição da linha recebida é diferente que um ponto “.” (representa um espaço vazio), se a condição for verdadeira vamos avaliar se a posição é a primeira (*col*=0), se for a primeira a variável *count_equal_bricks* vai ser igualado a 1 pois começamos uma contagem, caso contrário

vamos comparar a cor da peça da posição atual com a da posição anterior, se ambas forem iguais o *count_equal_bricks* vai ser atualizado, isto é, o valor que estava nesta variável vai ser incrementado. Caso a condição anterior não se verifique, a variável *res* vai ser multiplicada pelo valor que será retornado pelo método number_of_ways, esta operação é guardada na variável *res* e o *count_equal_bricks* é igualado a 1 para recomeçar uma nova contagem de posições consecutivas que possuem a mesma cor. Este processo anteriormente descrito é efetuado até a variável *col* ser igual ao último índice da linha. Quando chega ao fim, o programa sai do ciclo, mas como não foi realizado o número de maneiras diferentes de criar a última contagem de posições iguais com as peças de lego disponíveis, a variável *res* vai ser multiplicada pelo valor que será retornado pelo método number_of_ways, esta operação é guardada na variável *res* e no final do método é retornada.

number_of_ways – Este método começa por inicializar um *array* do tipo *long* (*b*) que vai possuir um tamanho igual ao valor que está no parâmetro (*bricks*) mais um (1), pois neste *array* vamos calcular para cada posição, o número de maneiras diferentes de construi-lo a partir das peças de lego disponíveis, em que cada posição deste *array* representa um certo tamanho de posições da linha do mosaico com a mesma cor. Com base no que foi definido na função recursiva, o nosso *array* no índice igual a zero (0) vai ser igualado a um (1), sendo este o caso base da nossa função recursiva. Para calcularmos o número de maneiras diferentes de contruir uma linha de tamanho *t*, este diferente de zero e menor ou igual à variável *bricks*, para isso vamos percorrer o nosso array b com o primeiro ciclo for, e para cada posição deste array a variável soma é inicializada a zero (0), pois a mesma vai conter o número total de maneiras de criar um "sub_mosaico" com tamanho igual a *t*. De forma a ver todos os casos possíveis de contruir o mosaico vamos percorrer com um ciclo for o *array this.values*_bricks, que contem todos os tamanhos de peças de lego disponíveis, e caso a peça *array this.values*_bricks[i] (uma peça do *array*), for menor ou igual ao *t*, a variável soma vai ser somada ao valor encontrado no *array b* na posição *t* menos a peça de lego utilizada. Quando o programa sai do segundo ciclo *for*, o valor da variável soma vai ser introduzido na posição t do *array b*, o processo anteriormente referido vai ser executado até à última posição do *array b*. No final do método é retornado o valor do *array b* na posição *bricks*.

Complexidade Temporal e Espacial

A complexidade temporal será *O(this.c)*, sendo este o tempo que o método *line_ways* demora a ser executado, pois *this.c é* o número de colunas de uma dada linha. Como a inicialização das variáveis e a atualização da variável *res* tem como complexidade *O(1)*, e o ciclo for percorre todas as posições da nossa linha do mosaico, a complexidade vai depender do número de colunas que a linha do mosaico possui, tornando-a a maior das complexidades. Neste método a complexidade espacial será *O(this.c)*, pois é o espaço de memoria que o programa necessita para chegar ao resultado final.

Em relação ao método *number_of_ways* a complexidade temporal será *O(N*M)*, sendo *N* o tamanho do *array b* menos um (1), e M o número de peças de lego disponíveis. Para cada tamanho superior a 0 vai ser averiguado quais as peças de lego disponíveis que possam ser usadas num dado tamanho, por isso é necessário percorrer o *array b* com um ciclo *for*, este ciclo vai ser executado N-1 vezes, o segundo ciclo como vai percorrer o *array values_bricks*, o ciclo for vai ser executado M vezes. Como o último ciclo descrito está dentro do primeiro ciclo for a complexidade irá ser igual a *O(N*M).* A complexidade espacial deste método será igual a *O(N),* sendo *N* o tamanho do *array b* menos um (1), pois o tamanho do array b depende do bricks, e como o array que contem o tamanho das peças de lego disponíveis já se encontra definido na classe Mosaic, não é contabilizado para o calculo da complexidade espacial.