Relatório para o problema de programação 2 - ARChitecture

Equipa:

N.º estudante: 2018295474 Nome Bruno Ricardo Leitão Faria

N.º estudante: 2018282583 Nome Diogo Alves Almeida Flórido

1. Descrição do Algoritmo

Para resolver este problema consideramos o cenário como sendo uma **matriz** com os números de formas possíveis de, peça a peça, se chegar desde a posição inicial (0,0) até a cada posição da matriz. Desta forma podemos determinar, através das posições "**menos complexas**" mais perto da origem, quantas possibilidades existem para uma determinada posição "**mais complexa**" sem nos termos de preocupar com o percurso exato de cada arco. Ao contrário duma abordagem ingénua que consistiria em formar uma "árvore" com o número de peças novas que se poderiam colar a uma peça anterior e registar cada combinação uma a uma, nesta abordagem partimos só dum arco que **temos a certeza** que existe se a solução **não fôr zero arcos**: o mais pequeno de todos.

Se (H <= h ou n < 3) então, **zero arcos** podem ser formados.

Com este método conseguimos ir facilmente obtendo novos valores a partir dos anteriores **à medida que o arco vai subindo**:

```
if x >= y :
    if up[x-1][y] != 0:
        up[x,y] <- up[x - 1][y] + up[x - 1][y - 1]
        if x >= h:
            up[x,y] <- up[x,y] - up[x-h][y-1]
    else:
        up[x,y] <- 1
```

Também conseguimos facilmente obter novos valores à **medida que o arco vai descendo**:

```
down[x,y] <- down[x - 1][y] + down[x - 1][y + 1]
if x >= h
    down[x,y] <- down[x,y] - down[x-h][y+1]
```

Mas **não é assim tão simples** obter resultados quando temos uma mudança de direção numa posição que não é previsível. Por isto mesmo é que dividimos, novamente, o problema em dois **subproblemas** e em vez de termos só uma matriz para o arco todo, temos uma matriz para a fase de subida e outra matriz para a fase de descida. Para saber o número de arcos com pico numa determinada posição **basta multiplicar** os valores das duas matrizes para essa mesma posição:

counter <- counter + up[x,y]*down[x,y]

Percorremos as posições das matrizes só uma vez visto que no mesmo ciclo podemos inserir valores nas duas matrizes ao mesmo tempo para a mesma posição e assim podemos continuar fiéis à ideia que tínhamos de resolver o problema sem "andar nenhuma vez para trás" (linearidade).

No ciclo no qual percorremos as matrizes também **não percorremos todas as colunas** pois sabemos onde é que os arcos começam (0,0) e a coluna mais longe onde podem acabar (min((H-h)*2,n-1)) e, por sua vez, as **primeira e última** colunas nas quais uma linha pode ter peças:

for x = 2 to H-h:

for y = ceil(x / (h - 1)) to start - ceil(x / (h - 1))

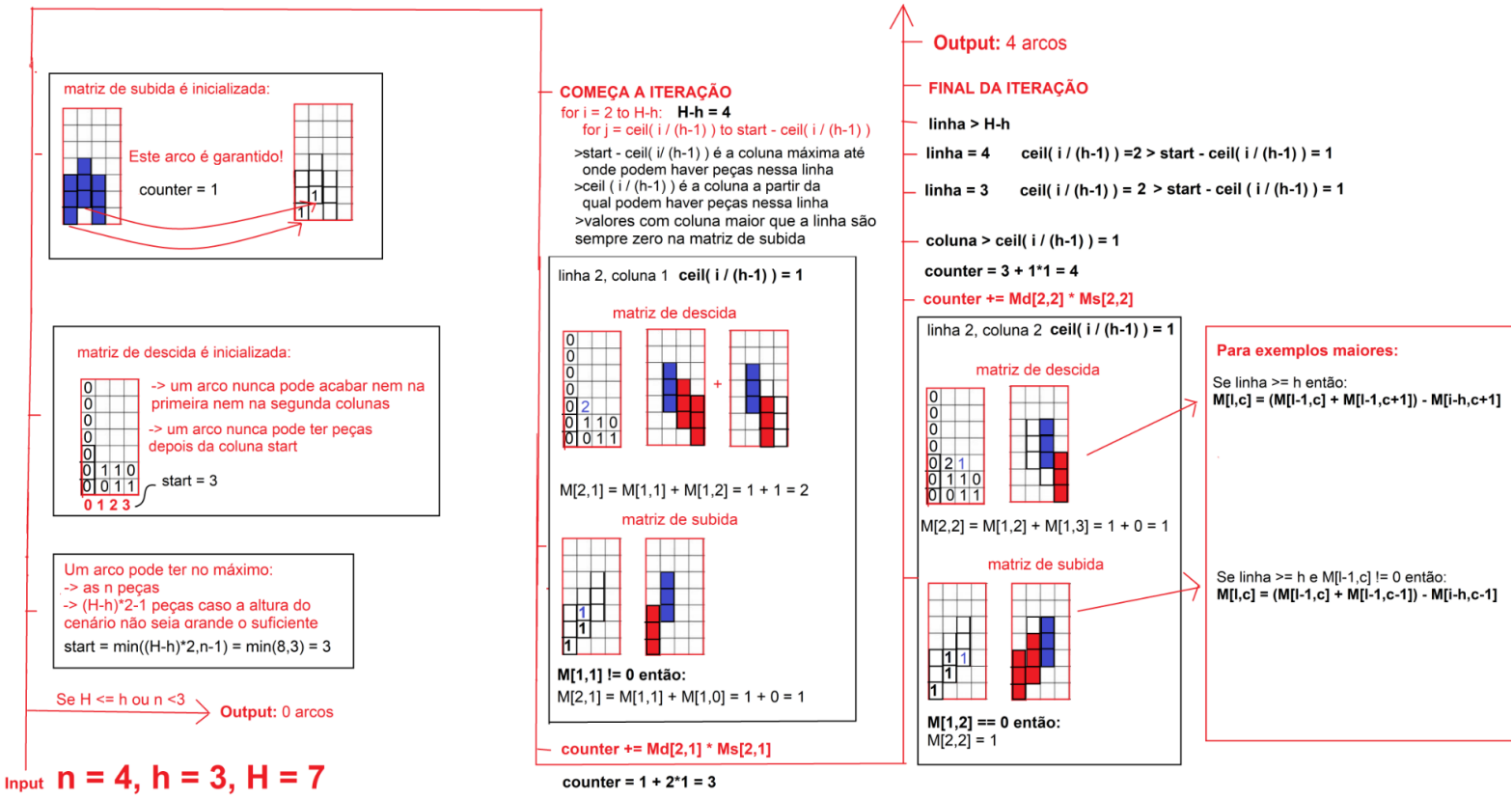

Figura 1. Explicação visual do funcionamento do algoritmo para um input n=4, h=3, H=7.

2. Estruturas de Dados

O nosso algoritmo consiste essencialmente na manipulação de dois vetores de vetores de inteiros (“matrizes dinâmicas”). Para isso utilizamos a linguagem C++ e criámos uma classe **ARC**:

Atributos: public int n, h, H -> Os parâmetros dados pelo input (n.º de peças, altura das peças e altura do cenário, respetivamente); **public int** counter -> variável onde é armazenado o número de arcos diferentes que se podem construir no cenário; **public vector<vector<int>>** up, down -> matrizes para as fases de subida e descida dos arcos cada uma com H linhas e n colunas e os números de possibilidades para alcançar cada posição; **private int** mod -> 10^9+7, módulo sobre o qual as respostas são apresentadas.

Métodos: public ARC(**int** n, **int** h, **int** H) -> o construtor: define o tamanho das matrizes e inicializa-as a zeros; **public void** build() -> trata de preencher as duas matrizes down e up; **private int** mod_abs(**int** a) -> converte um inteiro para o equivalente positivo no módulo dado; **private int** mod_add(**int** a, **int** b) -> operação de soma no módulo dado; **private int** mod_sub(**int** a, **int** b) -> operação de subtração no módulo dado; **private int** mod_mul(**int** a, **int** b) -> operação de multiplicação no módulo dado.

3. Correção

Pensamos que obtivemos os 200 pontos no Mooshak porque:

-Respondemos corretamente ao problema;

-Apresentámos os resultados em módulo e utilizámos operações em módulo eficientes: foi o uso de uma função de multiplicação que ainda podia ser otimizada para este caso que nos causou Time Limit Exceeded durante muito tempo;

-Dividimos o problema em subproblemas menos complexos e realizámos operações sobre os resultados para chegar à solução final;

-Reduzimos o ciclo pela matriz removendo as posições que não podem ter peças.

4. Análise do Algoritmo

Complexidade temporal: **O(n*H),** sendo n o número de peças e H a altura do cenário.

Complexidade espacial: **O(n*H),** sendo n o número de peças e H a altura do cenário.

Ambos devido às dimensões das matrizes.