* **Descrição do Problema e da Solução**

Ao processar o input, criamos o nosso grafo , representado por um vetor de vetores de adjacências de cada vértice. Ou seja, em cada posição do vetor, temos um vetor composto por todos os vértices tal que e se encontram ligados por uma aresta em . Passando agora para o algoritmo em si, a ideia é a de encontrar as componentes fortemente ligadas, SCC, e calcular o maior caminho.

Ao executarmos uma DFS no grafo, a ordem de tempos de fim corresponde a uma “ordem topológica” (a menos de ciclos). Portanto, para encontrarmos o número máximo de saltos possíveis num grafo, aproveitamos essa mesma ordem e ignoramos eventuais SCC. Deste modo, escusamos de calcular o grafo das SCC. Temos então duas versões da DFS, uma clássica, em que nos interessa apenas armazenar a ordem de tempos de fim, e uma outra versão, em que percorremos o grafo para dar conta de saltos de vizinhos e unificar SCC, aproveitando evidentemente o maior dos possíveis saltos para a solução final.

* **Análise Teórica**

Função recursiva da solução proposta:

(,)

De notar que utilizámos os limites superiores /2+1 e/2+1, dado que o resultado de cortar a chapa [, ] considerando = em [, ] e [, - ] é igual a cortar a mesma chapa considerando *= j -* em[*, j -* ] e [*, j - (j - )*].

Pseudocódigo:

parseDimensions():

read , from input

= create 2 vector of size ( + 1) by (+ 1) initialized with zeros

parsePrice():

read from input

for from 0 to do

read ,, from input

if <= and <= then

[][] =

if j <= and <= then

[][] =

computeMaxValue():

for from 1 to do

for from 1 to do

for from 1 to /2 + 1 do

[][] = max([][], [][] + [ - ][])

for from 1 to /2 + 1 do

[][] = max([][j], [][] + [][ - ])

1. Leitura dos dados de entrada: Simples leitura do input, com um ciclo a depender linearmente de (número de arestas). Logo, O().
2. Processamento da instância: Inserir o vértice no vetor de vetor de adjacências. Logo, O(1).
3. Aplicação da DFS para o grafo: A versão da DFS que utilizámos é iterativa. Para tal, o algoritmo utiliza uma pilha para acompanhar os vértices a serem visitados. Ele explora o máximo possível ao longo de cada vértice antes de retroceder. A complexidade é então determinada pelo número de vértices e arestas no grafo, dado que cada aresta e cada vértice são visitados uma vez. Logo, O(+).
4. Apresentar o resultado final: É feito um print do valor que se encontra armazenado na variável . Corresponde a uma complexidade O(1).
5. Complexidade global da solução: O() + O(1) + 2O(+) + O(1) = O(+).

* **Avaliação Experimental dos Resultados**

Neste gráfico, apresentamos o tempo de execução do algoritmo em função do tamanho (+) da entrada. Para tal, utilizámos 16 instâncias espaçadas 100 de tamanho entre si.

A graph with blue dots

Description automatically generated

O tempo de execução não é linear nas dimensões da chapa. Assim, para determinar se a previsão pela análise teórica é correta, vamos pôr o eixo dos XX a variar com a quantidade prevista pela análise teórica.

A graph with blue dots

Description automatically generated

Ao mudarmos o eixo dos XX para (, ) = O((+)), vemos que temos uma relação linear com os tempos no eixo dos YY, confirmando que a nossa implementação está de acordo com a análise teórica de O((, )).